The Neo4j Graph Data Science Library Manual v1.7

[[graph-data-science]]

Table of Contents

Introduction . . . 3
- Algorithms . . . 3
- Graph Catalog . . . 4
- Editions . . . 4

Installation . . . 5
- Supported Neo4j versions . . . 5
- Neo4j Desktop . . . 6
- Neo4j Server . . . 6
- Enterprise Edition Configuration . . . 7
- Neo4j Docker . . . 8
- Neo4j Causal Cluster . . . 8
- Additional configuration options . . . 9
- System Requirements . . . 9

Common usage . . . 12
- Memory Estimation . . . 13
- Creating graphs . . . 17
- Running algorithms . . . 18
- Logging . . . 20

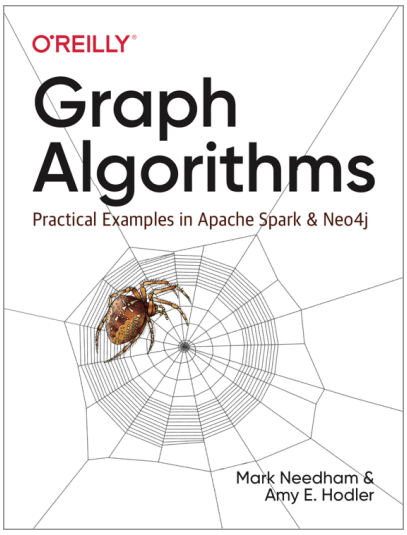
This is the manual for Neo4j Graph Data Science library version 1.7.

The manual covers the following areas:

- Introduction — An introduction to the Neo4j Graph Data Science library.
- Installation — Instructions for how to install and use the Neo4j Graph Data Science library.
- Common usage — General usage patterns and recommendations for getting the most out of the Neo4j Graph Data Science library.
- Graph management — A detailed guide to the graph catalog and utility procedures included in the Neo4j Graph Data Science library.
- Model catalog — A detailed guide to the model catalog and utility procedures included in the Neo4j Graph Data Science library.
- Algorithms — A detailed guide to each of the algorithms in their respective categories, including use-cases and examples.
- Production deployment — This chapter explains advanced details with regards to common Neo4j components.
- Operations reference — Reference of all procedures contained in the Neo4j Graph Data Science library.
- Migration from Graph Algorithms v3.5 — Additional resources - migration guide, books, etc - to help using the Neo4j Graph Data Science library.

The source code of the library is available at GitHub. If you have a suggestion on how we can improve the library or want to report a problem, you can create a new issue.

For further reading resources, we recommend studying the free Graph Algorithms book.



Graph Algorithms: Practical Examples in Apache Spark and Neo4j, by Mark Needham & Amy E. Hodler and published by O'Reilly Media is available now.

Download it for free at neo4j.com/graph-algorithms-book/.

Introduction

This chapter provides a brief introduction of the main concepts in the Neo4j Graph Data Science library.

This library provides efficiently implemented, parallel versions of common graph algorithms for Neo4j, exposed as Cypher procedures.

Algorithms

Graph algorithms are used to compute metrics for graphs, nodes, or relationships.

They can provide insights on relevant entities in the graph (centralities, ranking), or inherent structures like communities (community-detection, graph-partitioning, clustering).

Many graph algorithms are iterative approaches that frequently traverse the graph for the computation using random walks, breadth-first or depth-first searches, or pattern matching.

Due to the exponential growth of possible paths with increasing distance, many of the approaches also have high algorithmic complexity.

Fortunately, optimized algorithms exist that utilize certain structures of the graph, memoize already explored parts, and parallelize operations. Whenever possible, we've applied these optimizations.

The Neo4j Graph Data Science library contains a large number of algorithms, which are detailed in the Algorithms chapter.

Algorithm traits

Algorithms in GDS have specific ways to make use of various aspects of its input graph(s). We call these *algorithm traits*. When an algorithm supports an algorithm trait this indicates that the algorithm has been implemented to produce well-defined results in accordance with the trait. The following algorithm traits exist:

Directed

The algorithm is well-defined on a directed graph.

Undirected

The algorithm is well-defined on an undirected graph.

Homogeneous

The algorithm will treat all nodes and relationships in its input graph(s) similarly, as if they were all of the same type. If multiple types of nodes or relationships exist in the graph, this must be taken into account when analysing the results of the algorithm.

Heterogeneous

The algorithm has the ability to distinguish between nodes and/or relationships of different types.

Weighted

The algorithm supports configuration to set node and/or relationship properties to use as weights. These values can represent *cost*, *time*, *capacity* or some other domain-specific property, specified via the nodeWeightProperty and relationshipWeightProperty configuration parameters. The algorithm will by default consider each node and/or relationship as equally important.

Graph Catalog

In order to run the algorithms as efficiently as possible, the Neo4j Graph Data Science library uses a specialized in-memory graph format to represent the graph data. It is therefore necessary to load the graph data from the Neo4j database into an in memory graph catalog. The amount of data loaded can be controlled by so called graph projections, which also allow, for example, filtering on node labels and relationship types, among other options.

For more information see Graph Management.

Editions

The Neo4j Graph Data Science library is available in two editions.

- The open source Community Edition includes all algorithms and features, but is limited to four CPU cores.
- The Neo4j Graph Data Science library Enterprise Edition:
  - Can run on an unlimited amount of CPU cores.
  - Supports the role-based access control system (RBAC) from Neo4j Enterprise Edition.
  - Supports various additional model catalog features
    - Storing unlimited amounts of models in the model catalog
    - Publishing a stored model
    - Persisting a stored model to disk
  - Supports an optimized in-memory graph implementation

For more information see System Requirements - CPU.

Installation

> *This chapter provides instructions for installation and basic usage of the Neo4j Graph Data Science library.*

The Neo4j Graph Data Science (GDS) library is delivered as a plugin to the Neo4j Graph Database. The plugin needs to be installed into the database and added to the allowlist in the Neo4j configuration. There are two main ways of achieving this, which we will detail in this chapter.

This chapter is divided into the following sections:

1. Supported Neo4j versions
2. Neo4j Desktop
3. Neo4j Server
4. Enterprise Edition Configuration
5. Neo4j Docker
6. Neo4j Causal Cluster
7. Additional configuration options
8. System Requirements

Supported Neo4j versions

Below is the compatibility matrix for The GDS library vs Neo4j. In general, you can count on the latest version of GDS supporting the latest version of Neo4j and vice versa, and we recommend you always upgrade to that combination.

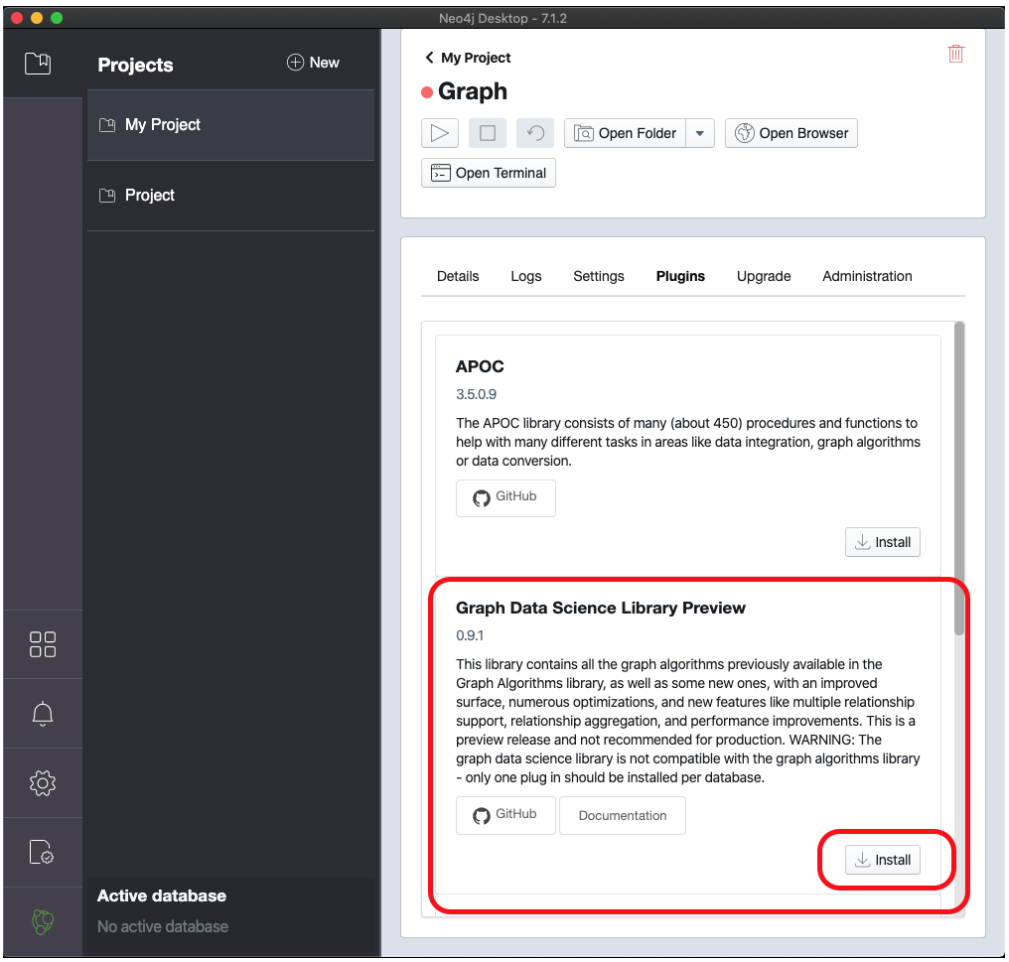
We list software with major and minor version only, e.g. GDS library 1.5. You should read that as any patch version of that major+minor version, but again, do upgrade to the latest patch always, to ensure you get all bug fixes included.

Not finding your version of GDS or Neo4j listed? Time to upgrade!

Neo4j Graph Data Science	Neo4j version
1.7	4.3
	4.2
	4.1 [1]
1.6	4.3
	4.2
	4.1 [2]
	4.0
1.1	3.5

Neo4j Desktop

The most convenient way of installing the GDS library is through the Neo4j Desktop plugin called Neo4j Graph Data Science. The plugin can be found in the 'Plugins' tab of a database.



The installer will download the GDS library and install it in the 'plugins' directory of the database. It will also add the following entry to the settings file:

```
dbms.security.procedures.unrestricted=gds.*
```

This configuration entry is necessary because the GDS library accesses low-level components of Neo4j to maximise performance.

If the procedure allowlist is configured, make sure to also include procedures from the GDS library:

```
dbms.security.procedures.allowlist=gds.*
```

Before `Neo4j 4.2`, the configuration setting is called `dbms.security.procedures.whitelist`

Neo4j Server

The GDS library is intended to be used on a standalone Neo4j server.

Running the GDS library in a Neo4j Causal Cluster is not supported. Read more about how to use GDS in conjunction with Neo4j Causal Cluster deployment below.

On a standalone Neo4j Server, the library will need to be installed and configured manually.

1. Download `neo4j-graph-data-science-[version].jar` from the Neo4j Download Center and copy it into the `$NEO4J_HOME/plugins` directory.
2. Add the following to your `$NEO4J_HOME/conf/neo4j.conf` file:

   ```
   dbms.security.procedures.unrestricted=gds.*
   ```

   This configuration entry is necessary because the GDS library accesses low-level components of Neo4j to maximise performance.

3. Check if the procedure allowlist is enabled in the `$NEO4J_HOME/conf/neo4j.conf` file and add the GDS library if necessary:

   ```
   dbms.security.procedures.allowlist=gds.*
   ```

   Before `Neo4j 4.2`, the configuration setting is called `dbms.security.procedures.whitelist`

4. Restart Neo4j

Verifying installation

To verify your installation, the library version can be printed by entering into the browser in Neo4j Desktop and calling the `gds.version()` function:

```
RETURN gds.version()
```

To list all installed algorithms, run the `gds.list()` procedure:

```
CALL gds.list()
```

Enterprise Edition Configuration

Unlocking the Enterprise Edition of the Neo4j Graph Data Science library requires a valid license key. To register for a license, please contact Neo4j at https://neo4j.com/contact-us/?ref=graph-analytics.

The license is issued in the form of a license key file, which needs to be placed in a directory accessible by the Neo4j server. You can configure the location of the license key file by setting the `gds.enterprise.license_file` option in the `neo4j.conf` configuration file of your Neo4j installation. The location must be specified using an absolute path. It is necessary to restart the database when configuring the license key for the first time and every time the license key is changed, e.g., when a new license key is added or the location of the key file changes.

Example configuration for the license key file:

```
gds.enterprise.license_file=/path/to/my/license/keyfile
```

If the `gds.enterprise.license_file` setting is set to a non-empty value, the Neo4j Graph Data Science library will verify that the license key file is accessible and contains a valid license key. When a valid license key is configured, all Enterprise Edition features are unlocked. In case of a problem, e.g, when the license key file is inaccessible, the license has expired or is invalid for any other reason, all calls to the Neo4j Graph Data Science Library will result in an error, stating the problem with the license key.

Neo4j Docker

The Neo4j Graph Data Science library is available as a plugin for Neo4j on Docker. The plugins guide for Docker is found at the operations manual.

To run a Neo4j Container with GDS available, you can run

```
docker run -it --rm \
  --publish=7474:7474 --publish=7687:7687 \
  --user="$(id -u):$(id -g)" \
  -e NEO4J_AUTH=none \
  --env NEO4JLABS_PLUGINS='["graph-data-science"]' \
  neo4j:4.2
```

Neo4j Causal Cluster

A Neo4j Causal Cluster consists of multiple machines that together support a highly available database management system. The GDS library uses main memory on a single machine for hosting graphs in the graph catalog and computing algorithms over these. These two architectures are not compatible and should not be used in conjunction. A GDS workload will attempt to consume most of the system resources of the machine during runtime, which may make the machine unresponsive for extended periods of time. For these reasons, we strongly advise against running GDS in a cluster as this potentially leads to data corruption or cluster outage.

To make use of GDS on graphs hosted by a Neo4j Causal Cluster deployment, these graphs should be detached from the running cluster. This can be accomplished in several ways, including:

1. Dumping a snapshot of the Neo4j store and importing it in a separate standalone Neo4j server.
2. Adding a Read Replica to the Neo4j Causal Cluster and then detaching it to safely operate GDS on a snapshot in separation from the Neo4j Causal Cluster.
3. Adding a Read Replica to the Neo4j Causal Cluster and configuring it for GDS workloads. Be aware that the in-memory graph and the underlying database will eventually become out of sync due to updates to the Read Replica. Since GDS can consume all available resources, responsiveness of the Read Replica might decrease and its state might fall behind the cluster. Using GDS in this scenario requires:
    - installing GDS on the Read Replica
    - using mutate or stream invocation modes

- consuming results from GDS workloads directly via Cypher (see Utility functions)
- **not using GDS write-back features** (writing triggers many large transactions and will potentially terminate the cluster)

After the GDS workload has finished on a detached machine (for cases 1. and 2.) it now contains out-of-sync results written to its copied version of the graph from the Neo4j Causal Cluster. To integrate these results back to the cluster, custom programs are necessary.

Additional configuration options

In order to make use of certain features of the GDS library, additional configuration is necessary. Configuration is done in the *neo4j.conf* configuration file before starting the DBMS. The following features require such additional configuration:

Graph export

Exporting graphs to CSV files requires the configuration parameter `gds.export.location` to be set to the absolut path to the folder in which exported graphs will be stored. This directory has to be writable by the Neo4j process.

Model persistence

The model persistence feature requires the configuration parameter `gds.model.store_location` to be set to the absolut path to the folder in which the models will be stored. This directory has to be writable by the Neo4j process.

System Requirements

Main Memory

The GDS library runs within a Neo4j instance and is therefore subject to the general Neo4j memory configuration.

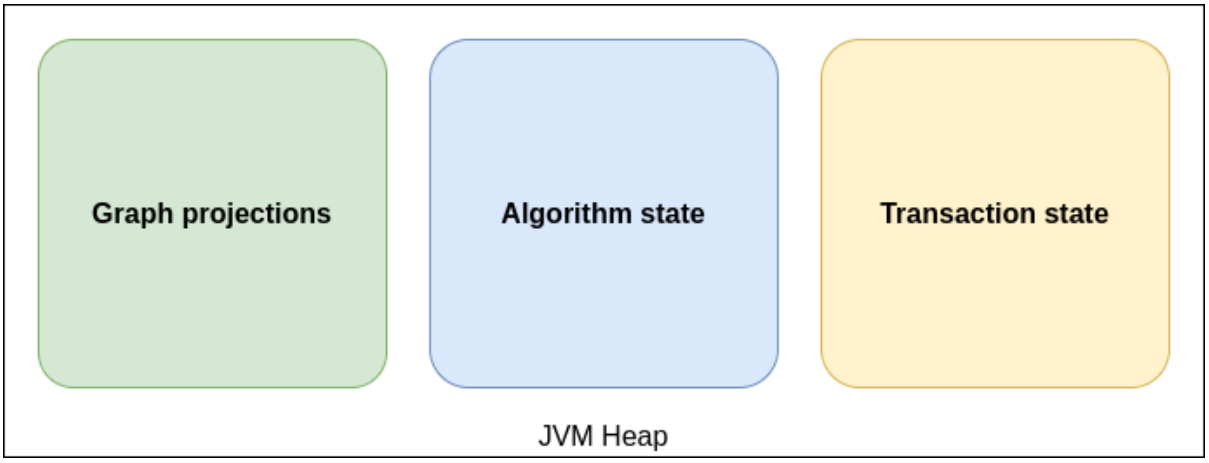


Figure 1. GDS heap memory usage

Heap size

The heap space is used for storing graph projections in the graph catalog and algorithm state. When writing algorithm results back to Neo4j, heap space is also used for handling transaction state (see dbms.tx_state.memory_allocation). For purely analytical workloads, a general recommendation is to set the heap space to about 90% of the available main memory. This can be done via dbms.memory.heap.initial_size and dbms.memory.heap.max_size.

To better estimate the heap space required to create in-memory graphs and run algorithms, consider the Memory Estimation feature. The feature estimates the memory consumption of all involved data structures using information about number of nodes and relationships from the Neo4j count store.

Page cache

The page cache is used to cache the Neo4j data and will help to avoid costly disk access.

For purely analytical workloads including native projections, it is recommended to decrease dbms.memory.pagecache.size in favor of an increased heap size. However, setting a minimum page cache size is still important while creating in-memory graphs:

- For native projections, the minimum page cache size for creating the in-memory graph can be roughly estimated by `8KB * 100 * readConcurrency`.
- For Cypher projections, a higher page cache is required depending on the query complexity.

However, if it is required to write algorithm results back to Neo4j, the write performance is highly depended on store fragmentation as well as the number of properties and relationships to write. We recommend starting with a page cache size of roughly `250MB * writeConcurrency` and evaluate write performance and adapt accordingly. Ideally, if the memory estimation feature has been used to find a good heap size, the remaining memory can be used for page cache and OS.

Decreasing the page cache size in favor of heap size is **not** recommended if the Neo4j instance runs both, operational and analytical workloads at the same time. See Neo4j memory configuration for general information about page cache sizing.

CPU

The library uses multiple CPU cores for graph projections, algorithm computation, and results writing. Configuring the workloads to make best use of the available CPU cores in your system is important to achieve maximum performance. The concurrency used for the stages of projection, computation and writing is configured per algorithm execution, see Common Configuration parameters

The default concurrency used for most operations in the Graph Data Science library is 4.

The maximum concurrency that can be used is limited depending on the license under which the library is being used:

- Neo4j Graph Data Science Library - Community Edition (GDS CE)
    - The maximum concurrency in the library is limited to 4.

- Neo4j Graph Data Science Library - Enterprise Edition (GDS EE)
    - The maximum concurrency in the library is unlimited. To register for a license, please contact Neo4j at https://neo4j.com/contact-us/?ref=graph-data-science.

Concurrency limits are determined based on whether you have a GDS EE license, or if you are using GDS CE. The maximum concurrency limit in the graph data science library is not set based on your edition of the Neo4j database.

[1] There is a bug in Neo4j 4.1.1 that can lead to an exception when using Cypher projection. If possible, use the lastest patch version.
[2] There is a bug in Neo4j 4.1.1 that can lead to an exception when using Cypher projection. If possible, use the lastest patch version.

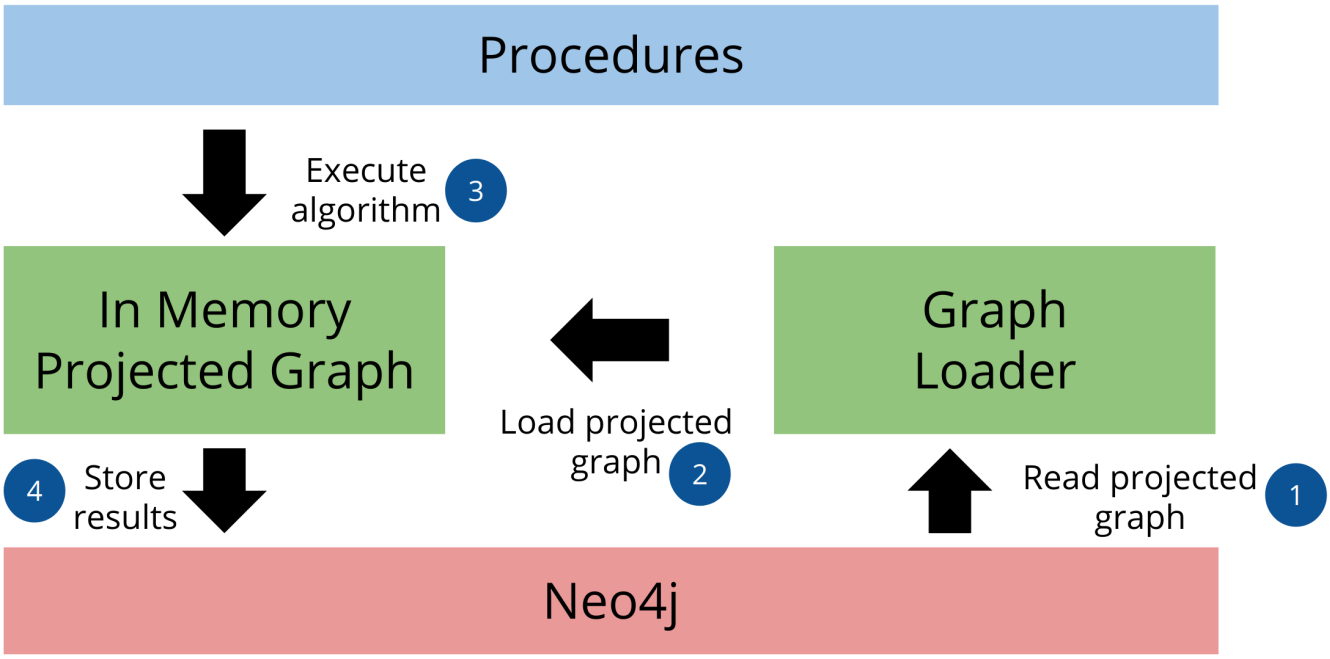
Common usage

This chapter explains the common usage patterns and operations that constitute the core of the Neo4j Graph Data Science library.

The GDS library usage pattern is typically split in two phases: development and production. In the development phase the goal is to establish a workflow of useful algorithms. In order to do this, the system must be configured, graph projections must be defined, and algorithms must be selected. It is typical to make use of the memory estimation features of the library. This enables you to successfully configure your system to handle the amount of data to be processed. There are two kinds of resources to keep in mind: the in-memory graph and the algorithm data structures.

In the production phase, the system would be configured appropriately to successfully run the desired algorithms. The sequence of operations would normally be to create a graph, run one or more algorithms on it, and consume results.

The below image illustrates an overview of standard operation of the GDS library:



The GDS library runs its procedures greedily in terms of system resources. That means that each procedure will try to use:

- as much memory as it needs (see Memory estimation)
- as many CPU cores as it needs (not exceeding the limits of the `concurrency` it's configured to run with)

Concurrently running procedures share the resources of the system hosting the DBMS and as such may affect each other's performance. To get an overview of the status of the system you can use the System monitor procedure.

The more detail on each individual operation, see the corresponding section:

1. Graph Catalog
2. Creating graphs
3. Running algorithms

In this chapter, we will go through these aspects and guide you towards the most useful operations.

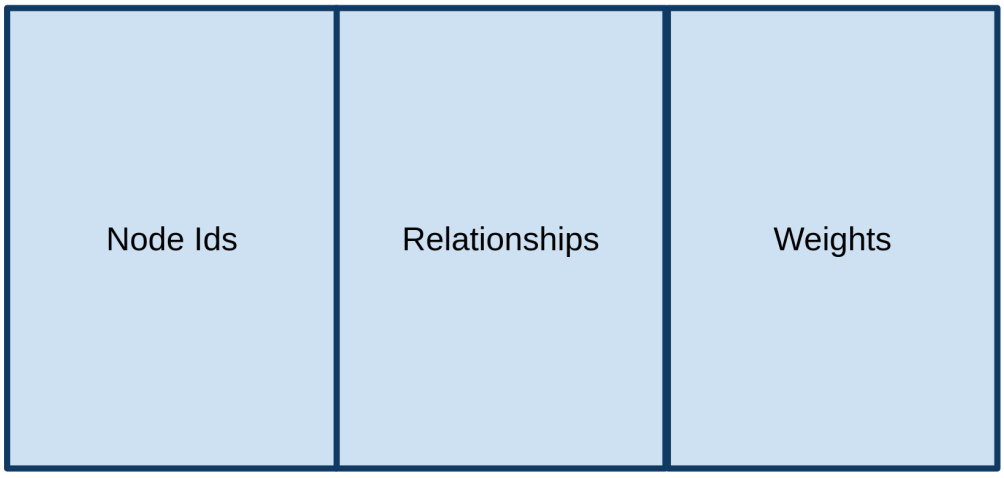
This chapter is divided into the following sections:

- Memory Estimation
- Creating graphs
- Running algorithms
- Logging
- Monitoring system

Memory Estimation

> *This section describes how to estimate memory requirements for the projected graph model used by the Neo4j Graph Data Science library.*

The graph algorithms library operates completely on the heap, which means we'll need to configure our Neo4j Server with a much larger heap size than we would for transactional workloads. The diagram belows shows how memory is used by the projected graph model:



In Memory Graph Model

The model contains three types of data:

- Node ids - up to 2^{45} ("35 trillion")
- Relationships - pairs of node ids. Relationships are stored twice if `orientation: "UNDIRECTED"` is used.
- Weights - stored as doubles (8 bytes per node) in an array-like data structure next to the relationships

Memory configuration depends on the graph projection that we're using.

Estimating memory requirements for algorithms

In many use cases it will be useful to estimate the required memory of projecting a graph and running an algorithm *before* running it in order to make sure that the workload can run on the available free memory. To do this the `.estimate` mode can be used, which returns an estimate of the amount of memory required to run graph algorithms. Note that only algorithms in the production-ready tier are guaranteed to have an `.estimate` mode. For more details please refer to Syntax overview.

Syntax outline:

```
CALL gds[.<tier>].<algorithm>.<execution-mode>.estimate(
  graphNameOrConfig: String or Map, configuration: Map
) YIELD
  nodeCount: Integer,
  relationshipCount: Integer,
  requiredMemory: String,
  treeView: String,
  mapView: Map,
  bytesMin: Integer,
  bytesMax: Integer,
  heapPercentageMin: Float,
  heapPercentageMax: Float
```

Table 1. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	-	no	The name of the projected graph or the algorithm configuration in the case of an anonymous graph.
configuration	Map	{}	yes	If the first parameter is the name of a projected graph, this parameter is the algorithm config, otherwise it needs to be null or an empty map.

The configuration map accepts the same configuration parameters as the estimated algorithm. See the specific algorithm documentation for more information.

Table 2. Results

Name	Type	Description
`nodeCount`	Integer	The number of nodes in the graph.
`relationshipCount`	Integer	The number of relationships in the graph.
`requiredMemory`	String	An estimation of the required memory in a human readable format.
`treeView`	String	A more detailed representation of the required memory, including estimates of the different components in human readable format.
`mapView`	Map	A more detailed representation of the required memory, including estimates of the different components in structured format.
`bytesMin`	Integer	The minimum number of bytes required.
`bytesMax`	Integer	The maximum number of bytes required.

Name	Type	Description
heapPercentageMin	Float	The minimum percentage of the configured maximum heap required.
heapPercentageMax	Float	The maximum percentage of the configured maximum heap required.

Estimating memory requirements for graphs

The gds.graph.create procedures also support .estimate to estimate memory usage for just the graph. Those procedures don't accept the graph name as the first argument, as they don't actually create the graph.

Syntax

```
CALL gds.graph.create.estimate(nodeProjection: String|List|Map, relationshipProjection: String|List|Map,
configuration: Map)
YIELD requiredMemory, treeView, mapView, bytesMin, bytesMax, heapPercentageMin, heapPercentageMax,
nodeCount, relationshipCount
```

The nodeProjection and relationshipProjection parameters follow the same syntax as in gds.graph.create.

Table 3. Parameters

Name	Type	Default	Optional	Description
nodeProjection	String or List or Map	-	no	The node projection to estimate for.
relationshipProjection	String or List or Map	-	no	The relationship projection to estimate for.
configuration	Map	{}	yes	Additional configuration, such as concurrency.

The result of running gds.graph.create.estimate has the same form as the algorithm memory estimation results above.

It is also possible to estimate the memory of a fictive graph, by explicitly specifying its node and relationship count. Using this feature, one can estimate the memory consumption of an arbitrarily sized graph.

To achieve this, use the following configuration options:

Table 4. Configuration

Name	Type	Default	Optional	Description
nodeCount	Integer	0	yes	The number of nodes in a fictive graph.
relationshipCount	Integer	0	yes	The number of relationships in a fictive graph.

When estimating a fictive graph, syntactically valid nodeProjection and relationshipProjection must be specified. However, it is recommended to specify '*' for both in the fictive graph case as this does not

interfere with the specified values above.

The query below is an example of estimating a fictive graph with 100 nodes and 1000 relationships.

Example

```
CALL gds.graph.create.estimate('*', '*', {
  nodeCount: 100,
  relationshipCount: 1000,
  nodeProperties: 'foo',
  relationshipProperties: 'bar'
})
YIELD requiredMemory, treeView, mapView, bytesMin, bytesMax, nodeCount, relationshipCount
```

Table 5. Results

requiredMemory	bytesMin	bytesMax	nodeCount	relationshipCount
"593 KiB"	607576	607576	100	1000

The `gds.graph.create.cypher` procedure has to execute both, the `nodeQuery` and `relationshipQuery`, in order to count the number of nodes and relationships of the graph.

Syntax

```
CALL gds.graph.create.cypher.estimate(nodeQuery: String, relationshipQuery: String, configuration: Map)
YIELD requiredMemory, treeView, mapView, bytesMin, bytesMax, heapPercentageMin, heapPercentageMax,
nodeCount, relationshipCount
```

Table 6. Parameters

Name	Type	Default	Optional	Description
nodeQuery	String	-	no	The node query to estimate for.
relationshipQuery	String	-	no	The relationship query to estimate for.
configuration	Map	{}	yes	Additional configuration, such as concurrency.

Automatic estimation and execution blocking

All procedures in the GDS library that support estimation, including graph creation, will do an estimation check at the beginning of their execution. This includes all execution modes, but not the `estimate` procedures themselves.

If the estimation check can determine that the current amount of free memory is insufficient to carry through the operation, the operation will be aborted and an error will be reported. The error will contain details of the estimation and the free memory at the time of estimation.

This heap control logic is restrictive in the sense that it only blocks executions that are certain to not fit into memory. It does not guarantee that an execution that passed the heap control will succeed without depleting memory. Thus, it is still useful to first run the estimation mode before running an algorithm or graph creation on a large data set, in order to view all details of the estimation.

The free memory taken into consideration is based on the Java runtime system information. The amount of

free memory can be increased by either dropping unused graphs from the catalog, or by increasing the maximum heap size prior to starting the Neo4j instance.

Bypassing heap control

Occasionally you will want the ability to bypass heap control if it is too restrictive. You might have insights into how your particular procedure call will behave, memory-wise; or you might just want to take a chance e.g. because the memory estimate you received is very close to system limits.

For that use case we have *sudo mode* which allows you to manually skip heap control and run your procedure regardless. Sudo mode is off by default to protect users - we fail fast if we can see your potentially long-running procedure would not be able to complete successfully.

To enable sudo mode, add the sudo parameter when calling a procedure. Here is an example of calling the popular Louvain community detection algorithm in sudo mode:

Run Louvain in sudo mode:

```
CALL gds.louvain.write('myGraph', { writeProperty: 'community', sudo: true })
YIELD communityCount, modularity, modularities
```

Accidentally enabling sudo mode when calling a procedure, causing it to run out of memory, will not significantly damage your installation, but it will waste your time.

Creating graphs

> *This section discusses creating named and anonymous graphs to be used for algorithm computation in the Neo4j Graph Data Science library.*

In order for any algorithm in the GDS library to run, we must first create a graph to run on. The graph is created as either an *anonymous graph* or a *named graph*. An anonymous graph is created for just a single algorithm and will be lost after its execution has finished. A named graph is given a name and stored in the graph catalog. For a detailed guide on all graph catalog operations, see Graph Catalog.

Creating a named graph has several advantages:

- it can be used by multiple algorithms
- the creation is cleanly separated from the algorithm execution
- the algorithm runtime can be measured in isolation
- the configuration for creating the graph may be retrieved from the graph catalog

Using an anonymous graph has the advantage that a single query may be used for an entire algorithm computation. This can be especially useful in the development phase when the workflow is being set up and the graph projections are experimented with.

Running algorithms

> *This section describes the common execution modes for algorithms:* `stream`, `stats`, `mutate` *and* `write`.

All algorithms are exposed as Neo4j procedures. They can be called directly from Cypher using Neo4j Browser, `cypher-shell`, or from your client code using a Neo4j Driver in the language of your choice.

For a detailed guide on the syntax to run algorithms, please see the Syntax overview section. In short, algorithms are run using one of the execution modes `stream`, `stats`, `mutate` or `write`, which we cover in this chapter.

The execution of any algorithm can be canceled by terminating the Cypher transaction that is executing the procedure call. For more on how transactions are used, see Transaction Handling.

Stream

The `stream` mode will return the results of the algorithm computation as Cypher result rows. This is similar to how standard Cypher reading queries operate.

The returned data can be a node ID and a computed value for the node (such as a Page Rank score, or WCC componentId), or two node IDs and a computed value for the node pair (such as a Node Similarity similarity score).

If the graph is very large, the result of a `stream` mode computation will also be very large. Using the `ORDER BY` and `LIMIT` subclauses in the Cypher query could be useful to support 'top N'-style use cases.

Stats

The `stats` mode returns statistical results for the algorithm computation like counts or percentile distributions. A statistical summary of the computation is returned as a single Cypher result row. The direct results of the algorithm are not available when using the `stats` mode. This mode forms the basis of the `mutate` and `write` execution modes but does not attempt to make any modifications or updates anywhere.

Mutate

The `mutate` mode will write the results of the algorithm computation back to the in-memory graph. Note that the specified `mutateProperty` value must not exist in the in-memory graph beforehand. This enables running multiple algorithms on the same in-memory graph without writing results to Neo4j in-between algorithm executions.

This execution mode is especially useful in three scenarios:

- Algorithms can depend on the results of previous algorithms without the need to write to Neo4j.
- Algorithm results can be written altogether (see write node properties and write relationships).
- Algorithm results can be queried via Cypher without the need to write to Neo4j at all (see

`gds.util.nodeProperty`).

A statistical summary of the computation is returned similar to the `stats` mode. Mutated data can be node properties (such as Page Rank scores), new relationships (such as Node Similarity similarities), or relationship properties.

Write

The `write` mode will write the results of the algorithm computation back to the Neo4j database. This is similar to how standard Cypher writing queries operate. A statistical summary of the computation is returned similar to the `stats` mode. This is the only execution mode that will attempt to make modifications to the Neo4j database.

The written data can be node properties (such as Page Rank scores), new relationships (such as Node Similarity similarities), or relationship properties. The `write` mode can be very useful for use cases where the algorithm results would be inspected multiple times by separate queries since the computational results are handled entirely by the library.

In order for the results from a `write` mode computation to be used by another algorithm, a new graph must be created from the Neo4j database with the updated graph.

Common Configuration parameters

All algorithms allow adjustment of their runtime characteristics through a set of configuration parameters. Although some of the parameters are algorithm-specific, many are shared between algorithms and execution modes.

To learn more about algorithm specific parameters and to find out if an algorithm supports a certain parameter, please consult the algorithm-specific documentation page.

List of the most commonly accepted configuration parameters

concurrency - Integer

Controls the parallelism with which the algorithm is executed. By default this value is set to 4. For more details on the concurrency settings and limitations please see the CPU section of the System Requirements.

nodeLabels - List of String

If the graph, on which the algorithm is run, was created with multiple node label projections, this parameter can be used to select only a subset of the projected labels. The algorithm will only consider nodes with the selected labels.

relationshipTypes - List of String

If the graph, on which the algorithm is run, was created with multiple relationship type projections, this parameter can be used to select only a subset of the projected types. The algorithm will only consider relationships with the selected types.

nodeWeightProperty - String

In algorithms that support node weights this parameter defines the node property that contains the

weights.

relationshipWeightProperty - String

In algorithms that support relationship weights this parameter defines the relationship property that contains the weights. The specified property is required to exist in the specified graph on all specified relationship types. The values must be numeric, and some algorithms may have additional value restrictions, such as requiring only positive weights.

maxIterations - Integer

For iterative algorithms this parameter controls the maximum number of iterations.

tolerance - Float

Many iterative algorithms accept the tolerance parameter. It controls the minimum delta between two iterations. If the delta is less than the tolerance value, the algorithm is considered converged and stops.

seedProperty - String

Some algorithms can be calculated incrementally. This means that results from a previous execution can be taken into account, even though the graph has changed. The `seedProperty` parameter defines the node property that contains the seed value. Seeding can speed up computation and write times.

writeProperty - String

In `write` mode this parameter sets the name of the node or relationship property to which results are written. If the property already exists, existing values will be overwritten.

writeConcurrency - Integer

In `write` mode this parameter controls the parallelism of write operations. The Default is `concurrency`

Logging

This section describes logging features in the Neo4j Graph Data Science library.

In the GDS library there are two types of logging: debug logging and progress logging.

Debug logging provides information about events in the system. For example, when an algorithm computation completes, the amount of memory used and the total runtime may be logged. Exceptional events, when an operation fails to complete normally, are also logged. The debug log information is useful for understanding events in the system, especially when troubleshooting a problem.

Progress logging is performed to track the progress of operations that are expected to take a long time. This includes graph projections, algorithm computation, and result writing.

All log entries are written to the log files configured for the Neo4j database. For more information on configuring Neo4j logs, please refer to the Neo4j Operations Manual.

Progress-logging procedure

Progress is also tracked by the GDS library itself. This makes it possible to inspect progress via Cypher, in

addition to looking in the log files. To access progress information for currently running tasks (also referred to as jobs), we can make use of the list progress procedure: `gds.beta.listProgress`. A task in the GDS library is defined as a running procedure, such as an algorithm or a graph load procedure.

The list progress procedure has two modes, depending on whether a `jobId` parameter was set: First, if `jobId` is not set, the procedure will produce a single row for each task currently running. This can be seen as the summary of those tasks, displaying the overall progress of a particular task for example. Second, if the `jobId` parameter is set it will show a detailed view for the given running job. The detailed view will produce a row for each step or task that job will perform during execution. It will also show how tasks are structured as a tree and print progress for each individual task.

Syntax

Getting the progress of tasks:

```
CALL gds.beta.listProgress(jobId: String)
YIELD
  jobId,
  taskName,
  progress,
  progressBar,
  status,
  timeStarted,
  elapsedTime
```

Table 7. Parameters

Name	Type	Default	Optional	Description
jobId	String	""	yes	The jobId of a running task. This will trigger a detailed overview for that particular task.

Table 8. Results

Name	Type	Description
`jobId`	String	A generated identifier of the running task.
`taskName`	String	The name of the running task, i.e. `Node2Vec`.
`progress`	String	The progress of the job shown as a percentage value.
`progressBar`	String	The progress of the job shown as an ASCII progress bar.
`status`	String	The current status of the job, i.e. `RUNNING` or `CANCELED`.
`timeStarted`	LocalTime	The local wall clock time when the task has been started.
`elapsedTime`	Duration	The duration from `timeStarted` to now. === Examples Assuming we just started `gds.beta.node2vec.stream` procedure. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.listProgress() YIELD jobId, taskName, progress ---- .Results [opts="header"]

| jobId | taskName | progress | "d21bb4ca-e1e9-4a31-a487-42ac8c9c1a0d" | "Node2Vec" | "42%"

:leveloffset!:

:leveloffset: +2

:description: This section describes features for monitoring a system's capacity and analytics workload using the Neo4j Graph Data Science library. = Monitoring system

[abstract] — This section describes features for monitoring a system's capacity and analytics workload using the Neo4j Graph Data Science library. — GDS supports multiple users concurrently working on the same system. Typically, GDS procedures are resource heavy in the sense that they may use a lot of memory and/or many CPU cores to do their computation. To know whether it is a reasonable time for a user to run a GDS procedure it is useful to know the current capacity of the system hosting Neo4j and GDS, as well as the current GDS workload on the system. Graphs and models are not shared between non-admin users by default, however GDS users on the same system will share its capacity.

[.alpha] == System monitor procedure

To be able to get an overview of the system's current capacity and its analytics workload one can use the procedure `gds.alpha.systemMonitor`. It will give you information on the capacity of the DBMS's JVM instance in terms of memory and CPU cores, and an overview of the resources consumed by the GDS procedures currently being run on the system.

=== Syntax

[.system-monitor-syntax] — .Monitor the system capacity and analytics workload: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.systemMonitor() YIELD freeHeap, totalHeap, maxHeap, jvmAvailableCpuCores, availableCpuCoresNotRequested, jvmHeapStatus, ongoingGdsProcedures ----

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | freeHeap | Integer | The amount of currently free memory in bytes in the Java Virtual Machine hosting the Neo4j instance. | totalHeap | Integer | The total amount of memory in bytes in the Java virtual machine hosting the Neo4j instance. This value may vary over time, depending on the host environment. | maxHeap | Integer | The maximum amount of memory in bytes that the Java virtual machine hosting the Neo4j instance will attempt to use. | jvmAvailableCpuCores | Integer | The number of logical CPU cores currently available to the Java virtual machine. This value may change vary over the lifetime of the DBMS. | availableCpuCoresNotRequested | Integer | The number of logical CPU cores currently available to the Java virtual machine that are not requested for use by currently running GDS procedures. Note that this number may be negative in case there are fewer available cores to the JVM than there are cores being requested by ongoing GDS procedures. | jvmHeapStatus | Map | The above-mentioned heap metrics in human-readable form. | ongoingGdsProcedures | List of Map | A list of maps containing resource usage and progress information for all GDS procedures (of all users) currently running on the Neo4j instance. Each map contains the name of the procedure, how far it has progressed, its estimated memory usage as well as how many CPU cores it will try to use at most.

— [NOTE] ==== `freeHeap` is influenced by ongoing GDS procedures, graphs stored the Graph catalog and the underlying Neo4j DBMS. Stored graphs can take up a significant amount of heap memory. To inspect the graphs in the graph catalog you can use the Graph list procedure. ====

=== Example

First let us assume that we just started `gds.beta.node2vec.stream` procedure with some arbitrary parameters.

We can have a look at the status of the JVM heap.

.Monitor JVM heap status: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.systemMonitor() YIELD freeHeap, totalHeap, maxHeap ----

.Results [opts="header"]

| freeHeap | totalHeap | maxHeap | 1234567 | 2345678 | 3456789

We can see that there currently is around `1.23 MB` free heap memory in the JVM instance running our Neo4j DBMS. This may increase independently of any procedures finishing their execution as `totalHeap` is currently smaller than `maxHeap`. We can also inspect CPU core usage as well as the status of currently running GDS procedures on the system.

.Monitor CPU core usage and ongoing GDS procedures: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.systemMonitor() YIELD availableCpuCoresNotRequested, jvmAvailableCpuCores, ongoingGdsProcedures ----

.Results [opts="header",cols='2,3,5']

| jvmAvailableCpuCores | availableCpuCoresNotRequested | ongoingGdsProcedures | 100 | 84 | [{ procedure: "Node2Vec", progress: "33.33%", estimatedMemoryRange: "[123 kB ... 234 kB]", requestedNumberOfCpuCores: "16" }]

Here we can note that there is only one GDS procedure currently running, namely the `Node2Vec` procedure we just started. It has finished around `33.33%` of its execution already. We also see that it may use up to an estimated `234 kB` of memory. Note that it may not currently be using that much memory and so it may require more memory later in its execution, thus possible lowering our current `freeHeap`. Apparently it wants to use up to `16` CPU cores, leaving us with a total of `84` currently available cores in the system not requested by any GDS procedures.

:leveloffset: 2

:leveloffset: +1

:description: This chapter explains the graph catalog, the different graph projection variants and utility functions in the Neo4j Graph Data Science library. = Graph management

[abstract] — This chapter explains the graph catalog, the different graph projection variants and utility functions in the Neo4j Graph Data Science library. — A central concept in the GDS library is the management of in-memory graphs.

This chapter is divided into the following sections:

* Graph Catalog * Anonymous graphs * Node Properties * Utility functions * Cypher on GDS graph * Administration

:leveloffset: 2

:leveloffset: +2

:description: This section details the graph catalog operations available to manage named graph projections within the Neo4j Graph Data Science library. = Graph Catalog

[abstract] — This section details the graph catalog operations available to manage named graph projections within the Neo4j Graph Data Science library. — Graph algorithms run on a graph data model which is a *projection* of the Neo4j property graph data model. A graph projection can be seen as a materialized view over the stored graph, containing only analytically relevant, potentially aggregated, topological and property information. Graph projections are stored entirely in-memory using compressed data structures optimized for topology and property lookup operations.

The graph catalog is a concept within the GDS library that allows managing multiple graph projections by name. Using its name, a created graph can be used many times in the analytical workflow. Named graphs can be created using either a Native projection or a Cypher projection. After usage, named graphs can be removed from the catalog to free up main memory.

Graphs can also be created when running an algorithm without placing them in the catalog. We refer to such graphs as anonymous graphs.

[NOTE] ==== The graph catalog exists as long as the Neo4j instance is running. When Neo4j is restarted, graphs stored in the catalog are lost and need to be re-created. ====

This chapter explains the available graph catalog operations.

[opts=header,cols="1m,1"]

| Name | Description | gds.graph.create | Creates a graph in the catalog using Native projection. | gds.graph.create.cypher | Creates a graph in the catalog using Cypher projection. | gds.beta.graph.create.subgraph | Creates a graph in the catalog by filtering an existing graph using node and relationship predicates. | gds.graph.list | Prints information about graphs that are currently stored in the catalog. | gds.graph.exists | Checks if a named graph is stored in the catalog. | gds.graph.removeNodeProperties | Removes node properties from a named graph. | gds.graph.deleteRelationships | Deletes relationships of a given relationship type from a named graph. | gds.graph.drop | Drops a named graph from the catalog. | gds.graph.streamNodeProperty | Streams a

single node property stored in a named graph. | gds.graph.streamNodeProperties | Streams node properties stored in a named graph. | gds.graph.streamRelationshipProperty | Streams a single relationship property stored in a named graph. | gds.graph.streamRelationshipProperties | Streams relationship properties stored in a named graph. | gds.graph.writeNodeProperties | Writes node properties stored in a named graph to Neo4j. | gds.graph.writeRelationship | Writes relationships stored in a named graph to Neo4j. | gds.graph.export | Exports a named graph into a new offline Neo4j database. | gds.beta.graph.export.csv | Exports a named graph into CSV files.

[NOTE] ==== Creating, using, listing, and dropping named graphs are management operations bound to a Neo4j user. Graphs created by a different Neo4j user are not accessible at any time. ====

:leveloffset: 2

:leveloffset: +3

:description: This section details projecting GDS graphs using `native` projections. = Creating graphs

[abstract] — This section details projecting GDS graphs using `native` projections. — A projected graph can be stored in the catalog under a user-defined name. Using that name, the graph can be referred to by any algorithm in the library. This allows multiple algorithms to use the same graph without having to re-create it on each algorithm run.

Native projections provide the best performance by reading from the Neo4j store files. Recommended for both the development, and the production phase.

[NOTE] — There is also a way to generate a random graph, see Graph Generation documentation for more details. — [NOTE] — The projected graphs will reside in the catalog until:

- the graph is dropped using gds.graph.drop - the Neo4j database from which to graph was projected is stopped or dropped - the Neo4j database management system is stopped. —

== Syntax

A native projection takes three mandatory arguments: `graphName`, `nodeProjection` and `relationshipProjection`. In addition, the optional `configuration` parameter allows us to further configure the graph creation.

[.graph-create-syntax] — [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create(graphName: String, nodeProjection: String or List or Map, relationshipProjection: String or List or Map, configuration: Map) YIELD graphName: String, nodeProjection: Map, nodeCount: Integer, relationshipProjection: Map, relationshipCount: Integer, createMillis: Integer ----

NOTE: To get information about a stored graph, such as its schema, one can use gds.graph.list.

.Parameters [opts="header",cols="1,1,1, 4"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | nodeProjection | String, List or Map | no | One or more node projections. | relationshipProjection| String, List or Map | no | One or more relationship projections. | configuration | Map | yes | Additional parameters to configure the native projection.

.Configuration [opts="header",cols="1,1,1,4"]

| Name | Type | Default | Description | readConcurrency | Integer | 4 | The number of concurrent threads used for creating the graph. | nodeProperties | String, List or Map | {} | The node properties to load for *all* node projections. | relationshipProperties | String, List or Map | {} | The relationship properties to load for *all* relationship projections. | validateRelationships | Boolean | false | Whether to throw an error if the

`relationshipProjection` includes relationships between nodes not part of the `nodeProjection`.

.Results [opts="header",cols="2,1,4"]

| Name | Type | Description | graphName | String | The name under which the graph is stored in the catalog. | nodeProjection | Map | The node projections used to project the graph. | nodeCount | Integer | The number of nodes stored in the projected graph. | relationshipProjection | Map | The relationship projections used to project the graph. | relationshipCount | Integer | The number of relationships stored in the projected graph. | createMillis | Integer | Milliseconds for creating the graph.

—

=== Node Projection

.Short-hand String-syntax for `nodeProjection`. The projected graph will contain the given `neo4j-label`. ---- <neo4j-label> ----

.Short-hand List-syntax for `nodeProjection`. The projected graph will contain the given `neo4j-label`s. ---- [<neo4j-label>, …, <neo4j-label>] ----

.Extended Map-syntax for `nodeProjection`. ---- { <projected-label>: { label: <neo4j-label>, properties: <neo4j-property-key> }, <projected-label>: { label: <neo4j-label>, properties: [<neo4j-property-key>, <neo4j-property-key>, ...] }, ... <projected-label>: { label: <neo4j-label>, properties: { <projected-property-key>: { property: <neo4j-property-key>, defaultValue: <fallback-value> }, ... <projected-property-key>: { property: <neo4j-property-key>, defaultValue: <fallback-value> } } } } ----

.Node Projection fields [opts="header",cols="1,1,1,2,4"]

| Name | Type | Optional | Default | Description | <projected-label> | String | no | n/a | The node label in the projected graph. | label | String | yes | `projected-label` | The node label in the Neo4j graph. If not set, uses the `projected-label`. | properties | Map, List or String | yes | {} | The projected node properties for the specified `projected-label`. | <projected-property-key> | String | no | n/a | The key for the node property in the projected graph. | property | String | yes | `projected-property-key` | The node property key in the Neo4j graph. If not set, uses the `projected-property-key`. .4+| defaultValue | Float .4+| yes | `Double.NaN` .4+| The default value if the property is not defined for a node. |Float[] | null |Integer |`Integer.MIN_VALUE` |Integer[] | null

=== Relationship Projection

.Short-hand String-syntax for `relationshipProjection`. The projected graph will contain the given `neo4j-type`. ---- <neo4j-type> ----

.Short-hand List-syntax for `relationshipProjection`. The projected graph will contain the given `neo4j-type`s. ---- [<neo4j-type>, …, <neo4j-type>] ----

.Extended Map-syntax for `relationshipProjection`. ---- { <projected-type>: { type: <neo4j-type>, orientation: <orientation>, aggregation: <aggregation-type>, properties: <neo4j-property-key> }, <projected-type>: { type: <neo4j-type>, orientation: <orientation>, aggregation: <aggregation-type>, properties: [<neo4j-property-key>, <neo4j-property-key>] }, ... <projected-type>: { type: <neo4j-type>, orientation: <orientation>, aggregation: <aggregation-type>, properties: { <projected-property-key>: { property: <neo4j-property-key>, defaultValue: <fallback-value>, aggregation: <aggregation-type> }, ... <projected-property-key>: { property: <neo4j-property-key>, defaultValue: <fallback-value>, aggregation: <aggregation-type> } } } } ----

.Relationship Projection fields [opts="header",cols="2,1,1,3,4"]

| Name | Type | Optional | Default | Description | <projected-type> | String | no | n/a | The name of the relationship type in the projected graph. | type | String | yes | `projected-type` | The relationship type in the Neo4j graph. | orientation | String | yes | `NATURAL` | Denotes how Neo4j relationships are represented in the projected graph. Allowed values are `NATURAL`, `UNDIRECTED`, `REVERSE`. | aggregation | String | no | `NONE` | Handling of parallel relationships. Allowed values are `NONE`, `MIN`, `MAX`, `SUM`, `SINGLE`, `COUNT`. | properties | Map, List or String | yes | {} | The projected relationship properties for the specified `projected-type`. | <projected-property-key> | String | no | n/a | The key for the relationship property in the projected graph. | property | String | yes | `projected-property-key` | The node property key in the Neo4j graph. If not set, uses the `projected-property-key`. | defaultValue | Float or Integer | yes | `Double.NaN` | The default value if the property is not defined for a node.

== Examples

In order to demonstrate the GDS Graph Create capabilities we are going to create a small social network graph in Neo4j. The example graph looks like this:

image::example-graphs/graph-create-example.svg[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (florentin:Person { name: 'Florentin', age: 16 }), (adam:Person { name: 'Adam', age: 18 }), (veselin:Person { name: 'Veselin', age: 20, ratings: [5.0] }), (hobbit:Book { name: 'The Hobbit', isbn: 1234, numberOfPages: 310, ratings: [1.0, 2.0, 3.0, 4.5] }), (frankenstein:Book { name: 'Frankenstein', isbn: 4242, price: 19.99 }),

(florentin)-[:KNOWS { since: 2010 }]→(adam), (florentin)-[:KNOWS { since: 2018 }]→(veselin), (florentin)-[:READ { numberOfPages: 4 }]→(hobbit), (florentin)-[:READ { numberOfPages: 42 }]→(hobbit), (adam)-[:READ { numberOfPages: 30 }]→(hobbit), (veselin)-[:READ]→(frankenstein) ----

=== Simple graph

A simple graph is a graph with only one node label and relationship type, i.e., a monopartite graph. We are going to start with demonstrating how to load a simple graph by projecting only the `Person` node label and `KNOWS` relationship type.

[role=query-example] — .Project `Person` nodes and `KNOWS` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('persons', // <1> 'Person', // <2> 'KNOWS' // <3>) YIELD graphName AS graph, nodeProjection, nodeCount AS nodes, relationshipProjection, relationshipCount AS rels ---- <1> The name of the graph. Afterwards, `persons` can be used to run algorithms or manage the graph. <2> The nodes to be projected. In this example, the nodes with the `Person` label. <3> The relationships to be projected. In this example, the relationships of type `KNOWS`.

.Results [opts="header", cols="1,3m,1,3m,1m"]

| graph | nodeProjection | nodes | relationshipProjection | rels | "persons" | {Person={label=Person, properties={}}} | 3 | {KNOWS={orientation=NATURAL, aggregation=DEFAULT, type=KNOWS, properties={}}} | 2

— In the example above, we used a short-hand syntax for the node and relationship projection. The used projections are internally expanded to the full `Map` syntax as shown in the `Results` table. In addition, we can see the projected in-memory graph contains three `Person` nodes, and the two `KNOWS` relationships.

=== Multi-graph

A multi-graph is a graph with multiple node labels and relationship types.

To project multiple node labels and relationship types, we can adjust the projections as follows:

[role=query-example] — .Project `Person` and `Book` nodes and `KNOWS` and `READ` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('personsAndBooks', // <1> ['Person', 'Book'], // <2> ['KNOWS', 'READ'] // <3>) YIELD graphName AS graph, nodeProjection, nodeCount AS nodes, relationshipCount AS rels ---- <1> Projects a graph under the name `personsAndBooks`. <2> The nodes to be projected. In this example, the nodes with a `Person` or `Book` label. <3> The relationships to be projected. In this example, the relationships of type `KNOWS` or `READ`.

.Results [opts="header", cols="1,3m,1m,1m"]

| graph | nodeProjection | nodes | rels | "personsAndBooks" | {Book={label=Book, properties={}}, Person={label=Person, properties={}}} | 5 | 6

— In the example above, we used a short-hand syntax for the node and relationship projection. The used projections are internally expanded to the full `Map` syntax as shown for the `nodeProjection` in the Results table. In addition, we can see the projected in-memory graph contains five nodes, and the two relationships.

=== Relationship orientation

By default, relationships are loaded in the same orientation as stored in the Neo4j db. In GDS, we call this the `NATURAL` orientation. Additionally, we provide the functionality to load the relationships in the `REVERSE` or even `UNDIRECTED` orientation.

[role=query-example] — .Project `Person` nodes and undirected `KNOWS` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('undirectedKnows', // <1> 'Person', // <2> {KNOWS: {orientation: 'UNDIRECTED'}} // <3>) YIELD graphName AS graph, relationshipProjection AS knowsProjection, nodeCount AS nodes, relationshipCount AS rels ---- <1> Projects a graph under the name `undirectedKnows`. <2> The nodes to be projected. In this example, the nodes with the Person label. <3> Projects relationships with type `KNOWS` and specifies that they should be `UNDIRECTED` by using the `orientation` parameter.

.Results [opts="header", cols="1,3m,1m,1m"]

| graph | knowsProjection | nodes | rels | "undirectedKnows" | {KNOWS={orientation=UNDIRECTED, aggregation=DEFAULT, type=KNOWS, properties={}}} | 3 | 4

— To specify the orientation, we need to write the `relationshipProjection` with the extended Map-syntax. Projecting the `KNOWS` relationships `UNDIRECTED`, loads each relationship in both directions. Thus, the `undirectedKnows` graph contains four relationships, twice as many as the `persons` graph in Simple graph.

=== Node properties

To project node properties, we can either use the `nodeProperties` configuration parameter for shared properties, or extend an individual `nodeProjection` for a specific label.

[role=query-example, group=node-properties] — .Project `Person` and `Book` nodes and `KNOWS` and `READ` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('graphWithProperties', // <1> { // <2> Person: {properties: 'age'}, // <3> Book: {properties: {price: {defaultValue: 5.0}}} // <4> }, ['KNOWS', 'READ'], // <5> {nodeProperties: 'ratings'} // <6>) YIELD graphName, nodeProjection, nodeCount AS nodes, relationshipCount AS rels RETURN graphName, nodeProjection.Book AS bookProjection, nodes, rels ---- <1> Projects a graph under the name `graphWithProperties`. <2> Use the expanded node projection syntax. <3> Projects nodes with the `Person` label and their `age` property. <4> Projects nodes with the `Book` label and their `price` property. Each `Book` that doesn't have the `price` property will get the `defaultValue` of `5.0`. <5> The relationships to be projected. In this example, the relationships of type `KNOWS` or `READ`. <6> The global configuration, projects node property `rating` on each of the specified labels.

.Results [opts="header", cols="1,3m,1m,1m"]

| graphName | bookProjection | nodes | rels | "graphWithProperties" | {label=Book, properties={price={defaultValue=5.0, property=price}, ratings={defaultValue=null, property=ratings}}} | 5 | 6

— The projected `graphWithProperties` graph contains five nodes and six relationships. In the returned `bookProjection` we can observe, the node properties `price` and `ratings` are loaded for `Books`.

NOTE: GDS currently only supports loading numeric properties.

Further, the `price` property has a default value of `5.0`. Not every book has a price specified in the example graph. In the following we check if the price was correctly projected:

[role=query-example, group=node-properties] — .Verify the ratings property of Adam in the projected graph: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Book) RETURN n.name AS name, gds.util.nodeProperty('graphWithProperties', id(n), 'price') as price ORDER BY price ----

.Results [opts="header", cols="1,1"]

| name | price | "The Hobbit" | 5.0 | "Frankenstein"| 19.99

— We can see, that the price was projected with the Hobbit having the default price of 5.0.

=== Relationship properties

Analogous to node properties, we can either use the `relationshipProperties` configuration parameter or extend an individual `relationshipProjection` for a specific type.

[role=query-example, group=rel-properties] — .Project `Person` and `Book` nodes and `READ` relationships with `numberOfPages` property: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('readWithProperties', // <1> ['Person', 'Book'], // <2> { // <3> READ: { properties: "numberOfPages" } // <4> }) YIELD graphName AS graph, relationshipProjection AS readProjection, nodeCount AS nodes, relationshipCount AS rels ---- <1> Projects a graph under the name `readWithProperties`. <2> The nodes to be projected. In this example, the nodes with a `Person` or `Book` label. <3> Use the expanded relationship projection syntax. <4> Project relationships of type `READ` and their `numberOfPages` property.

.Results [opts="header", cols="1,3m,1m,1m"]

| graph | readProjection | nodes | rels | "readWithProperties" | {READ={orientation=NATURAL, aggregation=DEFAULT, type=READ, properties={numberOfPages={defaultValue=null, property=numberOfPages, aggregation=DEFAULT}}}} | 5 | 4

— Next, we will verify that the relationship property `numberOfPages` were correctly loaded.

[role=query-example, group=rel-properties] — .Stream the relationship property `numberOfPages` of the projected graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('readWithProperties', 'numberOfPages') YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfPages RETURN gds.util.asNode(sourceNodeId).name AS person, gds.util.asNode(targetNodeId).name AS book, numberOfPages ORDER BY person ASC, numberOfPages DESC ----

.Results [opts="header", cols="1,1,1"]

| person | book | numberOfPages | "Adam" | "The Hobbit" | 30.0 | "Florentin" | "The Hobbit" | 42.0 | "Florentin" | "The Hobbit" | 4.0 | "Veselin" | "Frankenstein" | NaN

— We can see, that the `numberOfPages` property is loaded. The default property value is `Double.NaN` and could be changed using the Map-Syntax the same as for node properties in Node properties.

=== Parallel relationships

Neo4j supports parallel relationships, i.e., multiple relationships between two nodes. By default, GDS preserves parallel relationships. For some algorithms, we want the projected graph to contain at most one relationship between two nodes.

We can specify how parallel relationships should be aggregated into a single relationship via the `aggregation` parameter in a relationship projection.

For graphs without relationship properties, we can use the `COUNT` aggregation. If we do not need the count, we could use the `SINGLE` aggregation.

[role=query-example, group=count-aggregate] — .Project `Person` and `Book` nodes and `COUNT` aggregated `READ` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('readCount', // <1> ['Person', 'Book'], // <2> { READ: { // <3> properties: { numberOfReads: { // <4> property: '*', // <5> aggregation: 'COUNT' // <6> } } } }) YIELD graphName AS graph, relationshipProjection AS readProjection, nodeCount AS nodes, relationshipCount AS rels ---- <1> Projects a graph under the name `readCount`. <2> The nodes to be projected. In this example, the nodes with a `Person` or `Book` label. <3> Project relationships of type `READ`. <4> Project relationship property `numberOfReads`. <5> A placeholder, signaling that the value of the relationship property is derived and not based on Neo4j property. <6> The aggregation type. In this example, `COUNT` results in the value of the property being the number of parallel relationships.

.Results [opts="header", cols="1,3m,1m,1m"]

| graph | readProjection | nodes | rels | "readCount" | {READ={orientation=NATURAL, aggregation=DEFAULT, type=READ, properties={numberOfReads={defaultValue=null, property=*, aggregation=COUNT}}}} | 5 | 3

— Next, we will verify that the `READ` relationships were correctly aggregated.

[role=query-example, group=count-aggregate] — .Stream the relationship property `numberOfReads` of the projected graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('readCount', 'numberOfReads') YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfReads RETURN gds.util.asNode(sourceNodeId).name AS person, gds.util.asNode(targetNodeId).name AS book, numberOfReads ORDER BY numberOfReads DESC, person ----

.Results [opts="header", cols="1,1,1"]

| person | book | numberOfReads | "Florentin" | "The Hobbit" | 2.0 | "Adam" | "The Hobbit" | 1.0 | "Veselin" | "Frankenstein" | 1.0

— We can see, that the two READ relationships between Florentin, and the Hobbit result in `2` numberOfReads.

=== Parallel relationships with properties

For graphs with relationship properties we can also use other aggregations.

[role=query-example, group=sum-aggregate] — .Project `Person` and `Book` nodes and aggregated `READ` relationships by summing the `numberOfPages`: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('readSums', // <1> ['Person', 'Book'], // <2> {READ: {properties: {numberOfPages: {aggregation: 'SUM'}}}} // <3>) YIELD graphName AS graph, relationshipProjection AS readProjection, nodeCount AS nodes, relationshipCount AS rels ---- <1> Projects a graph under the name `readSums`. <2> The nodes to be projected. In this example, the nodes with a `Person` or `Book` label. <3> Project relationships of type `READ`. Aggregation type `SUM` results in a projected `numberOfPages` property with its value being the sum of the `numberOfPages` properties of the parallel relationships.

.Results [opts="header", cols="1,3m,1m,1m"]

| graph | readProjection | nodes | rels | "readSums" | {READ={orientation=NATURAL, aggregation=DEFAULT, type=READ, properties={numberOfPages={defaultValue=null, property=numberOfPages, aggregation=SUM}}}} | 5 | 3

— Next, we will verify that the relationship property `numberOfPages` was correctly aggregated.

[role=query-example, group=sum-aggregate] — .Stream the relationship property `numberOfPages` of the projected graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('readSums', 'numberOfPages') YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfPages RETURN gds.util.asNode(sourceNodeId).name AS person, gds.util.asNode(targetNodeId).name AS book, numberOfPages ORDER BY numberOfPages DESC, person ----

.Results [opts="header", cols="1,1,1"]

| person | book | numberOfPages | "Florentin" | "The Hobbit" | 46.0 | "Adam" | "The Hobbit" | 30.0 | "Veselin" | "Frankenstein" | 0.0

— We can see, that the two READ relationships between Florentin and the Hobbit sum up to `46` numberOfReads.

=== Validate relationships flag

As mentioned in the syntax section, the `validateRelationships` flag controls whether an error will be raised when attempting to create a relationship where either the source or target node is not present in the node projection. Note that even if the flag is set to `false` such a relationship will still not be created but the loading process will not be aborted.

We can simulate such a case with the graph present in the Neo4j database: — .Project `READ` and `KNOWS` relationships but only `Person` nodes, with `validateRelationships` set to true: [source, cypher, indent=0] ---- CALL gds.graph.create('danglingRelationships', 'Person', ['READ', 'KNOWS'], { validateRelationships: true }) YIELD graphName AS graph, relationshipProjection AS readProjection, nodeCount AS nodes, relationshipCount AS rels ----

.Results ---- org.neo4j.graphdb.QueryExecutionException: Failed to invoke procedure `gds.graph.create`: Caused by: java.lang.IllegalArgumentException: Failed to load a relationship because its target-node with id 3 is not part of the node query or projection. To ignore the relationship, set the configuration parameter `validateRelationships` to false. ---- — We can see that the above query resulted in an exception being thrown. The exception message will provide information about the specific node id that was missing, which will help debugging underlying problems.

:leveloffset: 2

:leveloffset: +3

:description: This section details projecting GDS graphs using `Cypher` projections. = Creating graphs using Cypher

[abstract] — This section details projecting GDS graphs using `Cypher` projections. —

A projected graph can be stored in the catalog under a user-defined name. Using that name, the graph can be referred to by any algorithm in the library. This allows multiple algorithms to use the same graph without having to re-create it on each algorithm run.

Using Cypher projections is a more flexible and expressive approach with diminished focus on performance compared to the native projections. Cypher projections are primarily recommended for the development phase (see Common usage).

[NOTE] — There is also a way to generate a random graph, see Graph Generation documentation for more details. — [NOTE] — The projected graph will reside in the catalog until:

- the graph is dropped using gds.graph.drop - the Neo4j database from which the graph was projected is stopped or dropped - the Neo4j database management system is stopped. —

== Syntax

A Cypher projection takes three mandatory arguments: `graphName`, `nodeQuery` and `relationshipQuery`. In addition, the optional `configuration` parameter allows us to further configure graph creation.

[.graph-create-cypher-syntax] — [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher(graphName: String, nodeQuery: String, relationshipQuery: String, configuration: Map) YIELD graphName: String, nodeQuery: String, nodeCount: Integer, relationshipQuery: String, relationshipCount: Integer, createMillis: Integer ----

.Parameters [opts="header",cols="1,1,8"]

| Name | Optional | Description | graphName | no | The name under which the graph is stored in the catalog. | nodeQuery | no | Cypher query to project nodes. The query result must contain an `id` column. Optionally, a `labels` column can be specified to represent node labels. Additional columns are interpreted as properties. | relationshipQuery | no | Cypher query to project relationships. The query result must contain `source` and `target` columns. Optionally, a `type` column can be specified to represent relationship type. Additional columns are interpreted as properties. | configuration | yes | Additional parameters to

configure the Cypher projection.

.Configuration [opts="header",cols="1,1,1,4"]

| Name | Type | Default | Description | readConcurrency | Integer | 4 | The number of concurrent threads used for creating the graph. | validateRelationships | Boolean | true | Whether to throw an error if the `relationshipQuery` returns relationships between nodes not returned by the `nodeQuery`. | parameters | Map | {} | A map of user-defined query parameters that are passed into the node and relationship queries.

.Results [opts="header",cols="2,1,4"]

| Name | Type | Description | graphName | String | The name under which the graph is stored in the catalog. | nodeQuery | String | The Cypher query used to project the nodes in the graph. | nodeCount | Integer | The number of nodes stored in the projected graph. | relationshipQuery | String | The Cypher query used to project the relationships in the graph. | relationshipCount | Integer | The number of relationships stored in the projected graph. | createMillis | Integer | Milliseconds for creating the graph.

— NOTE: To get information about a stored graph, such as its schema, one can use gds.graph.list.

== Examples

In order to demonstrate the GDS Graph Create capabilities we are going to create a small social network graph in Neo4j. The example graph looks like this:

image::example-graphs/graph-create-example.svg[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (florentin:Person { name: 'Florentin', age: 16 }), (adam:Person { name: 'Adam', age: 18 }), (veselin:Person { name: 'Veselin', age: 20, ratings: [5.0] }), (hobbit:Book { name: 'The Hobbit', isbn: 1234, numberOfPages: 310, ratings: [1.0, 2.0, 3.0, 4.5] }), (frankenstein:Book { name: 'Frankenstein', isbn: 4242, price: 19.99 }),

(florentin)-[:KNOWS { since: 2010 }]→(adam), (florentin)-[:KNOWS { since: 2018 }]→(veselin), (florentin)-[:READ { numberOfPages: 4 }]→(hobbit), (florentin)-[:READ { numberOfPages: 42 }]→(hobbit), (adam)-[:READ { numberOfPages: 30 }]→(hobbit), (veselin)-[:READ]→(frankenstein) ----

=== Simple graph

A simple graph is a graph with only one node label and relationship type, i.e., a monopartite graph. We are going to start with demonstrating how to load a simple graph by projecting only the `Person` node label and `KNOWS` relationship type.

[role=query-example] — .Project `Person` nodes and `KNOWS` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher('persons', 'MATCH (n:Person) RETURN id(n) AS id', 'MATCH (n:Person)-[r:KNOWS]→(m:Person) RETURN id(n) AS source, id(m) AS target') YIELD graphName AS graph, nodeQuery, nodeCount AS nodes, relationshipQuery, relationshipCount AS rels ----

.Results [opts="header", cols="1,3m,1,3m,1m"]

| graph | nodeQuery | nodes | relationshipQuery | rels | "persons" | "MATCH (n:Person) RETURN id(n) AS id" | 3 a| "MATCH (n:Person)-[r:KNOWS]→(m:Person) RETURN id(n) AS source, id(m) AS target" | 2

—	READ]→(m) RETURN id(n) AS source, id(m) AS target, type(r) AS type') YIELD graphName AS graph, nodeQuery, nodeCount AS nodes, relationshipCount AS rels ---- .Results [opts="header", cols="1,3m,1m,1m"]
=== Multi-graph A multi-graph is a graph with multiple node labels and relationship types. To retain the label and type information when we load multiple node labels and relationship types, we can add a `labels` column to the node query and a `type` column to the relationship query. [role=query-example] — .Project `Person` and `Book` nodes and `KNOWS` and `READ` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher('personsAndBooks', 'MATCH (n) WHERE n:Person OR n:Book RETURN id(n) AS id, labels(n) AS labels', 'MATCH (n)-[r:KNOWS	graph | nodeQuery | nodes | rels | "personsAndBooks" | "MATCH (n) WHERE n:Person OR n:Book RETURN id(n) AS id, labels(n) AS labels" | 5 | 6	

—	READ]→(m) RETURN id(n) AS source, id(m) AS target, type(r) AS type') YIELD graphName, nodeCount AS nodes, relationshipCount AS rels RETURN graphName, nodes, rels ---- .Results [opts="header", cols="1,1,1"]
=== Relationship orientation The native projection supports specifying an orientation per relationship type. The Cypher projection will treat every relationship returned by the relationship query as if it was in `NATURAL` orientation. It is thus not possible to project graphs in `UNDIRECTED` or `REVERSE` orientation when Cypher projections are used. [NOTE] — Some algorithms require that the graph was loaded with `UNDIRECTED` orientation. These algorithms can not be used with a graph created by a Cypher projection. — === Node properties To load node properties, we add a column to the result of the node query for each property. Thereby, we use the Cypher function coalesce() function to specify the default value, if the node does not have the property. [role=query-example, group=cypher-node-properties] — .Project `Person` and `Book` nodes and `KNOWS` and `READ` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher('graphWithProperties', 'MATCH (n) WHERE n:Book OR n:Person RETURN id(n) AS id, labels(n) AS labels, coalesce(n.age, 18) AS age, coalesce(n.price, 5.0) AS price, n.ratings AS ratings', 'MATCH (n)-[r:KNOWS	graphName | nodes | rels | "graphWithProperties" | 5 | 6	

— The projected `graphWithProperties` graph contains five nodes and six relationships. In a Cypher projection every node from the `nodeQuery` gets the same node properties, which means you can't have label-specific properties. For instance in the example above the `Person` nodes will also get `ratings` and `price` properties, while `Book` nodes get the `age` property.

Further, the `price` property has a default value of `5.0`. Not every book has a price specified in the example graph. In the following we check if the price was correctly projected:

[role=query-example, group=cypher-node-properties] — .Verify the ratings property of Adam in the projected graph: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Book) RETURN n.name AS name, gds.util.nodeProperty('graphWithProperties', id(n), 'price') AS price ORDER BY price ----

.Results [opts="header", cols="1,1"]

| name | price | "The Hobbit" | 5.0 | "Frankenstein"| 19.99

— We can see, that the price was projected with the Hobbit having the default price of 5.0.

=== Relationship properties

Analogous to node properties, we can project relationship properties using the `relationshipQuery`.

[role=query-example, group=cypher-rel-properties] — .Project `Person` and `Book` nodes and `READ` relationships with `numberOfPages` property: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher('readWithProperties', 'MATCH (n) RETURN id(n) AS id, labels(n) AS labels', 'MATCH (n)-[r:READ]→(m) RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages') YIELD graphName AS graph, nodeCount AS nodes, relationshipCount AS rels ----

.Results [opts="header", cols="1,1,1"]

| graph | nodes | rels | "readWithProperties" | 5 | 4

— Next, we will verify that the relationship property `numberOfPages` was correctly loaded.

[role=query-example, group=cypher-rel-properties] — .Stream the relationship property `numberOfPages` from the projected graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('readWithProperties', 'numberOfPages') YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfPages RETURN gds.util.asNode(sourceNodeId).name AS person, gds.util.asNode(targetNodeId).name AS book, numberOfPages ORDER BY person ASC, numberOfPages DESC ----

.Results [opts="header", cols="1,1,1"]

| person | book | numberOfPages | "Adam" | "The Hobbit" | 30.0 | "Florentin" | "The Hobbit" | 42.0 | "Florentin" | "The Hobbit" | 4.0 | "Veselin" | "Frankenstein" | NaN

— We can see, that the `numberOfPages` are loaded. The default property value is `Double.Nan` and can be changed as in the previous example Node properties by using the Cypher function coalesce().

=== Parallel relationships

The Property Graph Model in Neo4j supports parallel relationships, i.e., multiple relationships between two nodes. By default, GDS preserves the parallel relationships. For some algorithms, we want the projected graph to contain at most one relationship between two nodes.

The simplest way to achieve relationship deduplication is to use the `DISTINCT` operator in the relationship query. Alternatively, we can aggregate the parallel relationship by using the count() function and store the count as a relationship property.

[role=query-example, group=cypher-count-aggregate] — .Project `Person` and `Book` nodes and `COUNT` aggregated `READ` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher('readCount', 'MATCH (n) RETURN id(n) AS id, labels(n) AS labels', 'MATCH (n)-[r:READ]→(m) RETURN id(n) AS source, id(m) AS target, type(r) AS type, count(r) AS numberOfReads') YIELD graphName AS graph, nodeCount AS nodes, relationshipCount AS rels ----

.Results [opts="header", cols="1,1,1"]

| graph | nodes | rels | "readCount" | 5 | 3

— Next, we will verify that the `READ` relationships were correctly aggregated.

[role=query-example, group=cypher-count-aggregate] — .Stream the relationship property `numberOfReads` of the projected graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('readCount', 'numberOfReads') YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfReads RETURN gds.util.asNode(sourceNodeId).name AS person, gds.util.asNode(targetNodeId).name AS book, numberOfReads ORDER BY numberOfReads DESC, person ----

.Results [opts="header", cols="1,1,1"]

| person | book | numberOfReads | "Florentin" | "The Hobbit" | 2.0 | "Adam" | "The Hobbit" | 1.0 | "Veselin" | "Frankenstein" | 1.0

— We can see, that the two READ relationships between Florentin and the Hobbit result in `2` numberOfReads.

=== Parallel relationships with properties

For graphs with relationship properties we can also use other aggregations documented in the Cypher Manual.

[role=query-example, group=cypher-sum-aggregate] — .Project `Person` and `Book` nodes and aggregated `READ` relationships by summing the `numberOfPages`: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher('readSums', 'MATCH (n) RETURN id(n) AS id, labels(n) AS labels', 'MATCH (n)-[r:READ]→(m) RETURN id(n) AS source, id(m) AS target, type(r) AS type, sum(r.numberOfPages) AS numberOfPages') YIELD graphName AS graph, nodeCount AS nodes, relationshipCount AS rels ----

.Results [opts="header", cols="1,1,1"]

| graph | nodes | rels | "readSums" | 5 | 3

— Next, we will verify that the relationship property `numberOfPages` were correctly aggregated.

[role=query-example, group=cypher-sum-aggregate] — .Stream the relationship property `numberOfPages` of the projected graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('readSums', 'numberOfPages') YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfPages RETURN gds.util.asNode(sourceNodeId).name AS person, gds.util.asNode(targetNodeId).name AS book, numberOfPages ORDER BY numberOfPages DESC, person ----

.Results [opts="header", cols="1,1,1"]

| person | book | numberOfPages | "Florentin" | "The Hobbit" | 46.0 | "Adam" | "The Hobbit" | 30.0 | "Veselin" | "Frankenstein" | 0.0

— We can see, that the two `READ` relationships between Florentin and the Hobbit sum up to `46` numberOfPages.

=== Projecting filtered Neo4j graphs

Cypher-projections allow us to specify the graph to project in a more fine-grained way. The following examples will demonstrate how we to filter out `READ` relationship if they do not have a `numberOfPages` property.

[role=query-example, group=cypher-rel-filtering-properties] — .Project `Person` and `Book` nodes and `READ` relationships where `numberOfPages` is present: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher('existingNumberOfPages', 'MATCH (n) RETURN id(n) AS id, labels(n) AS labels', 'MATCH (n)-[r:READ]→(m) WHERE r.numberOfPages IS NOT NULL RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages') YIELD graphName AS graph, nodeCount AS nodes, relationshipCount AS rels ----

.Results [opts="header", cols="1,1,1"]

| graph | nodes | rels | "existingNumberOfPages" | 5 | 3

— Next, we will verify that the relationship property `numberOfPages` was correctly loaded.

[role=query-example, group=cypher-rel-filtering-properties] — .Stream the relationship property `numberOfPages` from the projected graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('existingNumberOfPages', 'numberOfPages') YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfPages RETURN gds.util.asNode(sourceNodeId).name AS person, gds.util.asNode(targetNodeId).name AS book, numberOfPages ORDER BY person ASC, numberOfPages DESC ----

.Results [opts="header", cols="1,1,1"]

| person | book | numberOfPages | "Adam" | "The Hobbit" | 30.0 | "Florentin" | "The Hobbit" | 42.0 | "Florentin" | "The Hobbit" | 4.0

— If we compare the results to the ones from Relationship properties, we can see that using `IS NOT NULL` is filtering out the relationship from Veselin to the book Frankenstein. This functionality is only expressible with native projections by creating a subraph.

=== Using query parameters

Similar to Cypher, it is also possible to set query parameters. In the following example we supply a list of strings to limit the cities we want to project.

.Project `Person` and `Book` nodes and `READ` relationships where `numberOfPages` is greater than 9: [role=query-example] — [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher('existingNumberOfPages', 'MATCH (n) RETURN id(n) AS id, labels(n) AS labels', 'MATCH (n)-[r:READ]→(m) WHERE r.numberOfPages > $minNumberOfPages RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages', { parameters: { minNumberOfPages: 9} }) YIELD graphName AS graph, nodeCount AS nodes, relationshipCount AS rels

----

.Results [opts="header", cols="1,1,1"]

| graph | nodes | rels | "existingNumberOfPages" | 5 | 2

— === Further usage of parameters

The parameters can also be used to directly pass in a list of nodes or a list of relationships. For example, pre-computing the list of nodes can be useful if the node filter is expensive.

.Project `Person` nodes younger than 17 and their name not beginning with V, and `KNOWS` relationships: [role=query-example] — [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create.cypher('personSubset', 'MATCH (n) WHERE n.age < 20 AND NOT n.name STARTS WITH "V" RETURN id(n) AS id, labels(n) AS labels', 'MATCH (n)-[r:KNOWS]→(m) WHERE (n.age < 20 AND NOT n.name STARTS WITH "V") AND (m.age < 20 AND NOT m.name STARTS WITH "V") RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages') YIELD graphName, nodeCount AS nodes, relationshipCount AS rels ----

.Results [opts="header", cols="1,1,1"]

| graphName | nodes | rels | "personSubset" | 2 | 1

— By passing the relevant Persons as a parameter, the above query can be transformed into the following:

.Project `Person` nodes younger than 20 and their name not beginning with V, and `KNOWS` relationships by using parameters: [role=query-example] — [source, cypher, role=noplay, indent=0] ---- MATCH (n) WHERE n.age < 20 AND NOT n.name STARTS WITH "V" WITH collect(n) AS olderPersons CALL gds.graph.create.cypher('personSubsetViaParameters', 'UNWIND $nodes AS n RETURN id(n) AS id, labels(n) AS labels', 'MATCH (n)-[r:KNOWS]→(m) WHERE (n IN $nodes) AND (m IN $nodes) RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages', { parameters: { nodes: olderPersons} }) YIELD graphName, nodeCount AS nodes, relationshipCount AS rels RETURN graphName, nodes, rels ----

.Results [opts="header", cols="1,1,1"]

| graphName | nodes | rels | "personSubsetViaParameters" | 2 | 1

—

:leveloffset: 2

:leveloffset: +3

:description: This section details how to list graphs stored in the graph catalog of the Neo4j Graph Data Science library. = Listing graphs

[abstract] — This section details how to list graphs stored in the graph catalog of the Neo4j Graph Data Science library. — Information about graphs in the catalog can be retrieved using the `gds.graph.list()` procedure.

== Syntax

[.graph-list-syntax] — .List information about graphs in the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.list(graphName: String) YIELD graphName: String, database: String, nodeProjection: Map, relationshipProjection: Map, nodeQuery: String, relationshipQuery: String, nodeFilter: String, relationshipFilter: String, nodeCount: Integer, relationshipCount: Integer, schema: Map, degreeDistribution: Map, density: Float, creationTime: Datetime, modificationTime: Datetime, sizeInBytes: Integer, memoryUsage: String ----

.Parameters [opts="header",cols="1,1,1,4"]

| Name | Type | Optional | Description | graphName | String | yes | The name under which the graph is stored in the catalog. If no graph name is given, information about all graphs will be listed. If a graph name is given but not found in the catalog, an empty list will be returned.

.Results [opts="header",cols="3m,1,6"]

| Name | Type | Description | graphName | String | Name of the graph. | database | String | Name of the database in which the graph has been created. | nodeProjection | Map | Node projection used to create the graph. If a Cypher projection was used, this will be a derived node projection. | relationshipProjection | Map | Relationship projection used to create the graph. If a Cypher projection was used, this will be a derived relationship projection. | nodeQuery | String | Node query used to create the graph. If a native projection was used, this will be `null`. | relationshipQuery | String | Relationship query used to create the graph. If a native projection was used, this will be `null`. | nodeFilter | String | The node filter used when creating this subgraph from another in-memory graph. If the graph has been created from Neo4j, this will be `null`. | relationshipFilter | String | The relationship filter used when creating this subgraph from another in-memory graph. If the graph has been created from Neo4j, this will be `null`. | nodeCount | Integer | Number of nodes in the graph. | relationshipCount | Integer | Number of relationships in the graph. | schema | Map | Node labels, Relationship types and properties contained in the in-memory graph. | degreeDistribution | Map | Histogram of degrees in the graph. | density | Float | Density of the graph. | creationTime | Datetime | Time when the graph was created. | modificationTime | Datetime | Time when the graph was last modified. | sizeInBytes | Integer | Number of bytes used in the Java heap to store the graph. | memoryUsage | String | Human readable description of `sizeInBytes`.

— The information contains basic statistics about the graph, e.g., the node and relationship count. The result field `creationTime` indicates when the graph was created in memory. The result field `modificationTime` indicates when the graph was updated by an algorithm running in `mutate` mode.

The `database` column refers to the name of the database the corresponding graph has been created on. Referring to a named graph in a procedure is only allowed on the database it has been created on.

The `schema` consists of information about the nodes and relationships stored in the graph. For each node label, the schema maps the label to its property keys and their corresponding property types. Similarly, the schema maps the relationship types to their property keys and property types. The property type is either `Integer`, `Float`, `List of Integer` or `List of Float`.

The `degreeDistribution` field can be fairly time-consuming to compute for larger graphs. Its computation is cached per graph, so subsequent listing for the same graph will be fast. To avoid computing the degree distribution, specify a `YIELD` clause that omits it. Note that not specifying a `YIELD` clause is the same as requesting all possible return fields to be returned.

The `density` is the result of `relationshipCount` divided by the maximal number of relationships for a simple graph with the given `nodeCount`.

== Examples

In order to demonstrate the GDS Graph List capabilities we are going to create a small social network graph in Neo4j.

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (florentin:Person { name: 'Florentin', age: 16 }), (adam:Person { name: 'Adam', age: 18 }), (veselin:Person { name: 'Veselin', age: 20 }), (florentin)-[:KNOWS { since: 2010 }]→(adam), (florentin)-[:KNOWS { since: 2018 }]→(veselin) ----

Additionally we will project a few graphs to the graph catalog, for more details see native projections and Cypher projections.

.Project `Person` nodes and `KNOWS` relationships using native projections: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('personsNative', 'Person', 'KNOWS') ----

.Project `Person` nodes and `KNOWS` relationships using Cypher projections: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create.cypher('personsCypher', 'MATCH (n:Person) RETURN id(n) AS id, labels(n) as labels', 'MATCH (n:Person)-[r:KNOWS]→(m:Person) RETURN id(n) AS source, id(m) AS target, type(r) as type') ----

.Project `Person` nodes with property `age` and `KNOWS` relationships using Native projections: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('personsWithAgeNative', { Person: {properties: 'age'} }, 'KNOWS') ----

=== List basic information about all graphs in the catalog

[role=query-example] — .List basic information about all graphs in the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.list() YIELD graphName, nodeCount, relationshipCount RETURN graphName, nodeCount, relationshipCount ORDER BY graphName ASC ----

.Results [opts="header",cols="1,1,1"]

| graphName | nodeCount | relationshipCount | "personsCypher" | 3 | 2 | "personsNative" | 3 | 2 | "personsWithAgeNative" | 3 | 2

—

=== List extended information about a specific named graph in the catalog

[role=query-example] — .List extended information about a specific Cypher named graph in the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.list('personsCypher') YIELD graphName, nodeProjection, nodeQuery ----

.Results [opts="header",cols="1,1,3"]

| graphName | nodeProjection | nodeQuery | "personsCypher" | null | "MATCH (n:Person) RETURN id(n) AS id, labels(n) as labels"

— [role=query-example] — .List extended information about a specific native named graph in the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.list('personsNative') YIELD graphName, nodeProjection, nodeQuery ----

.Results [opts="header",cols="1,3,1"]

| graphName | nodeProjection | nodeQuery | "personsNative" | {Person={label=Person, properties={}}} | null

— The above examples demonstrate that `nodeQuery` only has value when the graph is projected using Cypher projection while `nodeProjection` is present when we have a native graph. This is also true for `relationshipQuery` and relationshipProjection` respectively.

Despite different result columns being present for the different projections that we can use the Graph Schemas are the same, which is demonstrated in the example below.

[role=query-example] — .Cypher graph schema: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.list('personsCypher') YIELD graphName, schema ----

.Results [opts="header",cols="2,8"]

| graphName | schema | "personsCypher" | {relationships={KNOWS={}}, nodes={Person={}}}

— [role=query-example] — .Native graph schema: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.list('personsNative') YIELD graphName, schema ----

.Results [opts="header",cols="2,8"]

| graphName | schema | "personsNative" | {relationships={KNOWS={}}, nodes={Person={}}}

—

=== Degree distribution of a specific graph

[role=query-example] — .List information about the degree distribution of a specific graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.list('personsNative') YIELD graphName, degreeDistribution; ----

.Results [opts="header",cols="2,8"]

| graphName | degreeDistribution | "personsNative" | {p99=2, min=0, max=2, mean=0.6666666666666666, p90=2, p50=0, p999=2, p95=2, p75=0}

— :leveloffset: 2

:leveloffset: +3

= Check if a graph exists

We can check if a graph is stored in the catalog by looking up its name.

== Syntax

[.graph-exists-syntax] — .Check if a graph exists in the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.exists(graphName: String) YIELD graphName: String, exists: Boolean ----

.Parameters [opts="header",cols="1,1,1,4"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog.

.Results [opts="header",cols="3m,1,6"]

| Name | Type | Description | graphName | String | Name of the removed graph. | exists | Boolean | If the graph exists in the graph catalog.

— Additionally, to the procedure, we provide a function which directly returns the exists field from the procedure.

.Check if a graph exists in the catalog: [source, cypher, role=noplay, indent=0] ---- RETURN gds.graph.exists(graphName: String)::Boolean ----

== Examples

In order to demonstrate the GDS Graph Exists capabilities we are going to create a small social network graph in Neo4j and project it into our graph catalog.

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (florentin:Person { name: 'Florentin', age: 16 }), (adam:Person { name: 'Adam', age: 18 }), (veselin:Person { name: 'Veselin', age: 20 }), (florentin)-[:KNOWS { since: 2010 }]→(adam), (florentin)-[:KNOWS { since: 2018 }]→(veselin) ----

.Project `Person` nodes and `KNOWS` relationships: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('persons', 'Person', 'KNOWS') ----

== Procedure

[role=query-example] — .Check if graphs exist in the catalog: [source, cypher, role=noplay, indent=0] ---- UNWIND ['persons', 'books'] AS graph CALL gds.graph.exists(graph) YIELD graphName, exists RETURN graphName, exists ----

.Results [opts="header",cols="1,1"]

| graphName | exists | "persons" | true | "books" | false

— We can verify the projected `persons` graph exists while a `books` graph does not.

== Function

As an alternative to the procedure, we can also use the corresponding function. Unlike procedures, functions can be inlined in other cypher-statements such as `RETURN` or `WHERE`.

[role=query-example] — .Check if graphs exists in the catalog: [source, cypher, role=noplay, indent=0] ---- RETURN gds.graph.exists('persons') AS personsExists, gds.graph.exists('books') AS booksExists ----

.Results [opts="header",cols="1,1"]

| personsExists | booksExists | true | false

— As before, we can verify the projected `persons` graph exists while a `books` graph does not.

:leveloffset: 2

:leveloffset: +3

:description: This section details how to remove graphs stored in the graph catalog of the Neo4j Graph Data Science library. = Removing graphs

[abstract] — This section details how to remove graphs stored in the graph catalog of the Neo4j Graph Data Science library. — To free up memory, we can remove unused graphs. In order to do so, the `gds.graph.drop` procedure comes in handy.

== Syntax

[.graph-drop-syntax] — .Remove a graph from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.drop(graphName: String, failIfMissing: Boolean, dbName: String, username: String) YIELD graphName: String, database: String, nodeProjection: Map, relationshipProjection: Map, nodeQuery: String, relationshipQuery: String, nodeFilter: String, relationshipFilter: String, nodeCount: Integer, relationshipCount: Integer, schema: Map, density: Float, creationTime: Datetime, modificationTime: Datetime, sizeInBytes: Integer, memoryUsage: String ----

.Parameters [opts="header",cols="1,1,1,4"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | failIfMissing | Boolean | true | By default, the library will raise an error when trying to remove a non-existing graph. When set to `false`, the procedure returns empty result. | dbName | String | active database name | Then name of the database that was used to project the graph. When empty, the current database is used. | username | String | active user | The name of the user who projected the graph. Can only be used by GDS administrator.

.Results [opts="header",cols="3m,1,6"]

| Name | Type | Description | graphName | String | Name of the removed graph. | database | String | Name of the database in which the graph has been created. | nodeProjection | Map | Node projection used to create the graph. If a Cypher projection was used, this will be a derived node projection. | relationshipProjection | Map | Relationship projection used to create the graph. If a Cypher projection was used, this will be a derived relationship projection. | nodeQuery | String | Node query used to create the graph. If a native projection was used, this will be `null`. | relationshipQuery | String | Relationship query used to create the graph. If a native projection was used, this will be `null`. | nodeFilter | String | The node filter used when creating this subgraph from another in-memory graph. | relationshipFilter | String | The

relationship filter used when creating this subgraph from another in-memory graph. | nodeCount | Integer | Number of nodes in the graph. | relationshipCount | Integer | Number of relationships in the graph. | schema | Map | Node labels, Relationship types and properties contained in the in-memory graph. | density | Float | Density of the graph. | creationTime | Datetime | Time when the graph was created. | modificationTime | Datetime | Time when the graph was last modified. | sizeInBytes | Integer | Number of bytes used in the Java heap to store the graph. | memoryUsage | String | Human readable description of `sizeInBytes`.

— == Examples

In this section we are going to demonstrate the usage of `gds.graph.drop`. All the graph names used in these examples are fictive and should be replaced with real values.

=== Basic usage

.Remove a graph from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.drop('my-store-graph') YIELD graphName; ----

If we run the example above twice, the second time it will raise an error. If we want the procedure to fail silently on non-existing graphs, we can set a boolean flag as the second parameter to false. This will yield an empty result for non-existing graphs.

.Try removing a graph from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.drop('my-fictive-graph', false) YIELD graphName; ----

[.enterprise-edition] === Multi-database support

If we want to drop a graph created on another database, we can set the database name as the third parameter.

.Try removing a graph from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.drop('my-fictive-graph', true, 'my-other-db') YIELD graphName; ----

=== Multi-user support

If we are a GDS administrator and want to drop a graph that belongs to another user we can set the username as the fourth parameter to the procedure. This is useful if there are multiple users with graphs of the same name.

.Remove a graph from a specific user's graph catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.drop('my-fictive-graph', true, '', 'another-user') YIELD graphName; ----

See Administration for more details on this.

:leveloffset: 2

:leveloffset: +3

:description: This section details how to create subgraphs from existing graphs stored in the graph catalog of the Neo4j Graph Data Science library. [.beta] = Creating a subgraph

[abstract] — This section details how to create subgraphs from existing graphs stored in the graph catalog of the Neo4j Graph Data Science library. — In GDS, algorithms can be executed on a named graph that has been filtered based on its node labels and relationship types. However, that filtered graph only exists during the execution of the algorithm and it is not possible to filter on property values. If a filtered graph needs to be used multiple times, one can use the subgraph catalog procedure to create a new graph in the graph catalog.

The filter predicates in the subgraph procedure can take labels, relationship types as well as node and relationship properties into account. The new graph can be used in the same way as any other in-memory graph in the catalog. Creating subgraphs of subgraphs is also possible.

== Syntax

[.create-subgraph-syntax] — .A new graph can be created by using the `gds.beta.graph.create.subgraph()` procedure: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graph.create.subgraph(graphName: String, fromGraphName: String, nodeFilter: String, relationshipFilter: String, configuration: Map) YIELD graphName: String, fromGraphName: String, nodeFilter: String, relationshipFilter: String, nodeCount: Integer, relationshipCount: Integer, createMillis: Integer ----

.Parameters [opts="header",cols="1,1,4"]

| Name | Type | Description | graphName | String | The name of the new graph that is stored in the graph catalog. | fromGraphName | String | The name of the original graph in the graph catalog. | nodeFilter | String | A Cypher predicate for filtering nodes in the input graph. `can be used to allow all nodes. | relationshipFilter | String | A Cypher predicate for filtering relationships in the input graph.` can be used to allow all relationships. | configuration | Map | Additional parameters to configure subgraph creation.

.Subgraph specific configuration [opts="header",cols="1,1,1m,1,5"]

| Name | Type | Default | Optional | Description | concurrency | Integer | 4 | yes | The number of concurrent threads used for filtering the graph.

.Results [opts="header",cols="1,1,4"]

| Name | Type | Description | graphName | String | The name of the new graph that is stored in the graph catalog. | fromGraphName | String | The name of the original graph in the graph catalog. | nodeFilter | String | Filter predicate for nodes. | relationshipFilter | String | Filter predicate for relationships. | nodeCount | Integer | Number of nodes in the subgraph. | relationshipCount | Integer | Number of relationships in the subgraph. | createMillis | Integer | Milliseconds for creating the subgraph.

— The `nodeFilter` and `relationshipFilter` configuration keys can be used to express filter predicates. Filter predicates are Cypher predicates bound to a single entity. An entity is either a node or a relationship. The filter predicate always needs to evaluate to `true` or `false`. A node is contained in the subgraph if the node filter evaluates to `true`. A relationship is contained in the subgraph if the relationship filter evaluates to `true` and its source and target nodes are contained in the subgraph.

A predicate is a combination of expressions. The simplest form of expression is a literal. GDS currently supports the following literals:

* float literals, e.g., `13.37` * integer literals, e.g., `42` * boolean literals, i.e., `TRUE` and `FALSE`

Property, label and relationship type expressions are bound to an entity. The node entity is always identified by the variable `n`, the relationship entity is identified by `r`. Using the variable, we can refer to:

* node label expression, e.g., `n:Person` * relationship type expression, e.g., `r:KNOWS` * node property expression, e.g., `n.age` * relationship property expression, e.g., `r.since`

Boolean predicates combine two expressions and return either `true` or `false`. GDS supports the following boolean predicates:

* greater/lower than, such as `n.age > 42` or `r.since < 1984` * greater/lower than or equal, such as `n.age > 42` or `r.since < 1984` * equality, such as `n.age = 23` or `r.since = 2020` * logical operators, such as * `n.age > 23 AND n.age < 42` * `n.age = 23 OR n.age = 42` * `n.age = 23 XOR n.age = 42` * `n.age IS NOT 23`

Variable names that can be used within predicates are not arbitrary. A node predicate must refer to variable `n`. A relationship predicate must refer to variable `r`.

== Examples

In order to demonstrate the GDS create subgraph capabilities we are going to create a small social graph in Neo4j.

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (p0:Person { age: 16 }), (p1:Person { age: 18 }), (p2:Person { age: 20 }), (b0:Book { isbn: 1234 }), (b1:Book { isbn: 4242 }), (p0)-[:KNOWS { since: 2010 }]→(p1), (p0)-[:KNOWS { since: 2018 }]→(p2), (p0)-[:READS]→(b0), (p1)-[:READS]→(b0), (p2)-[:READS]→(b1) ----

.Project the social network graph: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('social-graph', { Person: { properties: 'age' }, // <1> Book: {} // <2> }, { KNOWS: { properties: 'since' }, // <3> READS: {} // <4> }) YIELD graphName, nodeCount, relationshipCount, createMillis ---- <1> Project `Person` nodes with their `age` property. <2> Project `Book` nodes without any of their properties. <3> Project `KNOWS` relationships with their `since` property. <4> Project `READS` relationships without any of their properties.

=== Node filtering

[role=query-example] — .Create a new graph containing only users of a certain age group: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graph.create.subgraph('teenagers', 'social-graph', 'n.age > 13 AND n.age ⇐ 18', '*') YIELD graphName, fromGraphName, nodeCount, relationshipCount ----

.Results [opts="header"]

| graphName | fromGraphName | nodeCount | relationshipCount | "teenagers" | "social-graph" | 2 | 1

—

=== Node and relationship filtering

[role=query-example] — .Create a new graph containing only users of a certain age group that know each other since a given point a time: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graph.create.subgraph('teenagers', 'social-graph', 'n.age > 13 AND n.age ⇐ 18', 'r.since >= 2012') YIELD graphName, fromGraphName, nodeCount, relationshipCount ---- .Results [opts="header"]

| graphName | fromGraphName | nodeCount | relationshipCount | "teenagers" | "social-graph" | 2 | 0

—

=== Bipartite subgraph

[role=query-example] — .Create a new bipartite graph between books and users connected by the `READS` relationship type: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graph.create.subgraph('teenagers-books', 'social-graph', 'n:Book OR n:Person', 'r:READS') YIELD graphName, fromGraphName, nodeCount, relationshipCount ----

.Results [opts="header"]

| graphName | fromGraphName | nodeCount | relationshipCount | "teenagers-books" | "social-graph" | 5 | 3

—

=== Bipartite graph node filtering

[role=query-example] — .The previous example can be extended with an additional filter applied only to persons: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graph.create.subgraph('teenagers-books', 'social-graph', 'n:Book OR (n:Person AND n.age > 18)', 'r:READS') YIELD graphName, fromGraphName, nodeCount, relationshipCount ----

.Results [opts="header"]

| graphName | fromGraphName | nodeCount | relationshipCount | "teenagers-books" | "social-graph" | 3 | 1

— :leveloffset: 2

:leveloffset: +3

:description: This section details the operations available over node-properties stored in projected graphs within the Neo4j Graph Data Science library. = Node operations

[abstract] — This section details the operations available over node-properties stored in projected graphs within the Neo4j Graph Data Science library. — The graphs in the Neo4j Graph Data Science Library support properties for nodes. We provide multiple operations to work with the stored node-properties in projected graphs. Node properties are either created during the graph creation or when using the `mutate` mode of our graph algorithms.

To inspect stored values, the `gds.graph.streamNodeProperties` procedure can be used. This is useful if we ran multiple algorithms in `mutate` mode and want to retrieve some or all of the results.

To persist the values in a Neo4j database, we can use `gds.graph.writeNodeProperties`. Similar to streaming node properties, it is also possible to write those back to Neo4j. This is similar to what an algorithm `write` execution mode does, but allows more fine-grained control over the operations.

We can also remove node properties from a named graph in the catalog. This is useful to free up main memory or to remove accidentally created node properties.

== Syntax

.Syntax descriptions of the different operations over node properties [.tabbed-example, caption =] ====

[.include-with-stream-single-property] ====== [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamNodeProperty(graphName: String, nodeProperties: String, nodeLabels: String or List of Strings, configuration: Map) YIELD nodeId: Integer, propertyValue: Integer or Float or List of Integer or List of Float ----

.Parameters [opts="header",cols="1,3,1,5"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | nodeProperties | String | no | The node property in the graph to stream. | nodeLabels | String or List of Strings | yes | The node labels to stream the node properties for graph. | configuration | Map | yes | Additional parameters to configure streamNodeProperties.

.Configuration [opts="header",cols="1,1,1,7"]

| Name | Type | Default | Description | concurrency | Integer | 4 | The number of concurrent threads. Note, this procedure is always running single-threaded.

.Results [opts="header",cols="2,3,5"]

| Name | Type | Description |nodeId | Integer | The id of the node. .^|propertyValue a| * Integer * Float * List of Integer * List of Float .^| The stored property value.

======

[.include-with-stream] ====== [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamNodeProperties(graphName: String, nodeProperties: String or List of Strings, nodeLabels: String or List of Strings, configuration: Map) YIELD nodeId: Integer, nodeProperty: String, propertyValue: Integer or Float or List of Integer or List of Float ----

.Parameters [opts="header",cols="1,3,1,5"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | nodeProperties | String or List of Strings | no | The node properties in the graph to stream. | nodeLabels | String or List of Strings | yes | The node labels to stream the node properties for graph. | configuration | Map | yes | Additional parameters to configure streamNodeProperties.

.Configuration [opts="header",cols="1,1,1,7"]

| Name | Type | Default | Description | concurrency | Integer | 4 | The number of concurrent threads. Note, this procedure is always running single-threaded.

.Results [opts="header",cols="2,3,5"]

| Name | Type | Description |nodeId | Integer | The id of the node. |nodeProperty | String | The name of the node property. .^|propertyValue a| * Integer * Float * List of Integer * List of Float .^| The stored property value.

======

[.include-with-write] ====== [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.writeNodeProperties(graphName: String, nodeProperties: String or List of Strings, nodeLabels: String or List of Strings, configuration: Map) YIELD writeMillis: Integer, propertiesWritten: Integer, graphName: String, nodeProperties: String or List of String ----

.Parameters [opts="header",cols="1,3,1,5"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | nodeProperties | String or List of Strings | no | The node properties in the graph to write back. | nodeLabels | String or List of Strings | yes | The node labels to write back their node properties. | configuration | Map | yes | Additional parameters to configure writeNodeProperties.

.Configuration [opts="header",cols="1,1,1,7"]

| Name | Type | Default | Description | concurrency | Integer | 4 | The number of concurrent threads used for running the procedure. Also provides the default value for `writeConcurrency` | writeConcurrency | Integer | 'concurrency' | The number of concurrent threads used for writing the node properties.

.Results [opts="header",cols="2,3,5"]

| Name | Type | Description | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | propertiesWritten | Integer | Number of properties written. | graphName | String | The name of a graph stored in the catalog. | nodeProperties | String or List of String | The written node properties.

======

[.include-with-remove] ====== [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.removeNodeProperties(graphName: String, nodeProperties: String or List of Strings, nodeLabels: String or List of Strings, configuration: Map) YIELD propertiesRemoved: Integer, graphName: String, nodeProperties: String or List of String ----

.Parameters [opts="header",cols="1,3,1,5"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is

stored in the catalog. | nodeProperties | String or List of Strings | no | The node properties in the graph to remove. | nodeLabels | String or List of Strings | yes | The node labels to remove the node properties from. | configuration | Map | yes | Additional parameters to configure removeNodeProperties.

.Configuration [opts="header",cols="1,1,1,7"]

| Name | Type | Default | Description | concurrency | Integer | 4 | The number of concurrent threads. Note, this procedure is always running single-threaded.

.Results [opts="header",cols="2,3,5"]

| Name | Type | Description | propertiesRemoved | Integer | Number of properties removed. | graphName | String | The name of a graph stored in the catalog. | nodeProperties | String or List of String | The removed node properties.

====== ====

== Examples

In order to demonstrate the GDS capabilities over node properties, we are going to create a small social network graph in Neo4j and project it into our graph catalog.

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (florentin:Person { name: 'Florentin', age: 16 }), (adam:Person { name: 'Adam', age: 18 }), (veselin:Person { name: 'Veselin', age: 20 }), (hobbit:Book { name: 'The Hobbit', numberOfPages: 310 }), (florentin)-[:KNOWS { since: 2010 }]→(adam), (florentin)-[:KNOWS { since: 2018 }]→(veselin), (adam)-[:READ]→(hobbit) ----

.Project the small social network graph: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('socialGraph', { Person: {properties: "age"}, Book: {} }, ['KNOWS', 'READ']) ----

.Compute the Degree Centrality in our social graph: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.degree.mutate('socialGraph', {mutateProperty: 'score'}) ----

=== Stream

We can stream node properties stored in a named in-memory graph back to the user. This is useful if we ran multiple algorithms in `mutate` mode and want to retrieve some or all of the results. This is similar to what an algorithm `stream` execution mode does, but allows more fine-grained control over the operations.

==== Single property

In the following, we stream the previously computed scores `score`.

[role=query-example] — .Stream the `score` node property: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamNodeProperty('socialGraph', 'score') YIELD nodeId, propertyValue RETURN gds.util.asNode(nodeId).name AS name, propertyValue AS score ORDER BY score DESC ----

.Results [opts="header"]

| name | score | "Florentin" | 2.0 | "Adam" | 1.0 | "Veselin" | 0.0 | "The Hobbit" | 0.0

— NOTE: The above example requires all given properties to be present on at least one node projection, and the properties will be streamed for all such projections.

==== NodeLabels

The procedure can be configured to stream just the properties for specific node labels.

[role=query-example] — .Stream the `score` property for `Person` nodes: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamNodeProperty('socialGraph', 'score', ['Person']) YIELD nodeId, propertyValue RETURN gds.util.asNode(nodeId).name AS name, propertyValue AS score ORDER BY score DESC ----

.Results [opts="header"]

| name | score | "Florentin" | 2.0 | "Adam" | 1.0 | "Veselin" | 0.0

— It is required, that all specified node labels have the node property.

==== Multiple Properties

We can also stream several properties at once.

[role=query-example] — .Stream multiple node properties: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamNodeProperties('socialGraph', ['score', 'age']) YIELD nodeId, nodeProperty, propertyValue RETURN gds.util.asNode(nodeId).name AS name, nodeProperty, propertyValue ORDER BY name, nodeProperty ----

.Results [opts="header"]

| name | nodeProperty | propertyValue | "Adam" | "age" | 18 | "Adam" | "score" | 1.0 | "Florentin" | "age" | 16 | "Florentin" | "score" | 2.0 | "Veselin" | "age" | 20 | "Veselin" | "score" | 0.0

— [NOTE] ==== When streaming multiple node properties, the name of each property is included in the result. This adds with some overhead, as each property name must be repeated for each node in the result, but is necessary in order to distinguish properties. ====

=== Write

To write the 'score' property for all node labels in the social graph, we use the following query:

[role=query-example] — .Write the `score` property back to Neo4j: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.writeNodeProperties('socialGraph', ['score']) YIELD propertiesWritten ----

.Results [opts="header"]

| propertiesWritten | 4

— The above example requires the `score` property to be present on *at least one* projected node label, and the properties will be written for all such labels.

==== NodeLabels

The procedure can be configured to write just the properties for some specific node labels. In the following example, we will only write back the scores of the `Person` nodes.

[role=query-example] — .Write node properties of a specific projected node label to Neo4j: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.writeNodeProperties('socialGraph', ['score'], ['Person']) YIELD propertiesWritten ----

.Results [opts="header"]

| propertiesWritten | 3

— [NOTE] — If the `nodeLabels` parameter is specified, it is required that *all* given node labels have *all* of the given properties. —

=== Remove

[role=query-example] — .Remove the `score` property from all projected nodes in the `socialGraph`: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.removeNodeProperties('socialGraph', ['score']) YIELD propertiesRemoved ----

.Results [opts="header"]

| propertiesRemoved | 4

— [NOTE] — The above example requires all given properties to be present on *at least one* projected node label. — ==== NodeLabels

Consider we compute the Degree Centrality only for a subset of the graph.

.Compute the Degree Centrality for only the `Book` nodes in our social graph: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.degree.mutate('socialGraph', {nodeLabels: ['Book'], mutateProperty: 'degree'}) ----

The procedure can be configured to remove just the properties for s. In the following example, we will only remove the scores from the `Book` nodes.

[role=query-example] — .Remove the `degree` property from the projected `Book` nodes: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.removeNodeProperties('socialGraph', ['degree'], ['Book']) YIELD propertiesRemoved ----

.Results [opts="header"]

| propertiesRemoved | 1

— [NOTE] — If the `nodeLabels` parameter is specified, it is required that *all* given node labels have *all* of the given properties. —

== Utility functions

Utility functions allow accessing specific nodes of in-memory graphs directly from a Cypher query.

.Catalog Functions [opts=header,cols="1m,1"]

| Name | Description | gds.util.nodeProperty | Allows accessing a node property stored in a named graph.

=== Syntax

[opts=header,cols="1m,1"]

| Name | Description | gds.util.nodeProperty(graphName: STRING, nodeId: INTEGER, propertyKey: STRING, nodeLabel: STRING?) | Named graph in the catalog, Neo4j node id, node property key and optional node label present in the named-graph.

If a node label is given, the property value for the corresponding projection and the given node is returned. If no label or `'*'` is given, the property value is retrieved and returned from an arbitrary projection that contains the given propertyKey. If the property value is missing for the given node, `null` is returned.

=== Examples

We use the `socialGraph` with the property `score` introduced above.

[role=query-example] — .Access a property node property for Florentin: [source, cypher, role=noplay, indent=0] ---- MATCH (florentin:Person {name: 'Florentin'}) RETURN florentin.name AS name, gds.util.nodeProperty('socialGraph', id(florentin), 'score') AS score ----

.Results [opts="header",cols="2"]

| name | score | "Florentin" | 2.0

— We can also specifically return the `score` property from the `Person` projection in case other projections also have a `score` property as follows.

[role=query-example] — .Access a property node property from Person for Florentin: [source, cypher, role=noplay, indent=0] ---- MATCH (florentin:Person {name: 'Florentin'}) RETURN florentin.name AS name, gds.util.nodeProperty('socialGraph', id(florentin), 'score', 'Person') AS score ----

.Results [opts="header",cols="2"]

| name | score | "Florentin" | 2.0

— :leveloffset: 2

:leveloffset: +3

:description: This section details the operations available over relationships and relationship properties stored in projected graphs within the Neo4j Graph Data Science library. = Relationship operations

[abstract] — This section details the operations available over relationships and relationship properties stored in projected graphs within the Neo4j Graph Data Science library. — The graphs in the Neo4j Graph Data Science Library support properties for relationships. We provide multiple operations to work with the stored relationship-properties in projected graphs. Relationship properties are either created during the graph creation or when using the `mutate` mode of our graph algorithms.

To inspect stored relationship property values, the `streamRelationshipProperties` procedure can be used. This is useful if we ran multiple algorithms in `mutate` mode and want to retrieve some or all of the results.

To persist relationship types in a Neo4j database, we can use `gds.graph.writeRelationship`. Similar to streaming relationship properties, it is also possible to write back to Neo4j. This is similar to what an algorithm `write` execution mode does, but allows more fine-grained control over the operations. By default, no relationship properties will be written. To write relationship properties, these have to be explicitly specified.

We can also remove relationships from a named graph in the catalog. This is useful to free up main memory or to remove accidentally created relationship types.

== Syntax

.Syntax descriptions of the different operations over relationship types [.tabbed-example, caption=] ====

[.include-with-stream-single-property] ====== [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty(graphName: String, relationshipProperties: List of String, relationshipTypes: List of Strings, configuration: Map) YIELD sourceNodeId: Integer, targetNodeId: Integer, relationshipType: String, propertyValue: Integer or Float ----

.Parameters [opts="header",cols="1,3,1,5"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | relationshipProperties | List of String | no | The relationship properties in the graph to stream. | relationshipTypes | List of Strings | yes | The relationship types to stream the relationship properties for graph. | configuration | Map | yes | Additional parameters to configure streamNodeProperties.

.Configuration [opts="header",cols="1,1,1,7"]

| Name | Type | Default | Description | concurrency | Integer | 4 | The number of concurrent threads. Note, this procedure is always running single-threaded.

.Results [opts="header",cols="2,3,5"]

| Name | Type | Description | sourceNodeId | Integer | The id of the source node for the relationship. | targetNodeId | Integer | The id of the target node for the relationship. | relationshipType | Integer | The type of the relationship. .^| propertyValue a| * Integer * Float .^| The stored property value.

```
======

[.include-with-stream] ====== [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperties( graphName: String, relationshipProperties: List of String, relationshipTypes: List of Strings, configuration: Map ) YIELD sourceNodeId: Integer, targetNodeId: Integer, relationshipType: String, relationshipProperty: String, propertyValue: Integer or Float ----

.Parameters [opts="header",cols="1,3,1,5"]
```

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | relationshipProperties | List of String | no | The relationship properties in the graph to stream. | relationshipTypes | List of Strings | yes | The relationship types to stream the relationship properties for graph. | configuration | Map | yes | Additional parameters to configure streamNodeProperties.

```
.Configuration [opts="header",cols="1,1,1,7"]
```

| Name | Type | Default | Description | concurrency | Integer | 4 | The number of concurrent threads. Note, this procedure is always running single-threaded.

```
.Results [opts="header",cols="2,3,5"]
```

| Name | Type | Description | sourceNodeId | Integer | The id of the source node for the relationship. | targetNodeId | Integer | The id of the target node for the relationship. | relationshipType | Integer | The type of the relationship. | relationshipProperty | Integer | The name of the relationship property. .^| propertyValue a| * Integer * Float .^| The stored property value.

```
======

[.include-with-write] ====== [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.writeRelationship( graphName: String, relationshipType: String, relationshipProperty: String, configuration: Map ) YIELD writeMillis: Integer, graphName: String, relationshipType: String, relationshipsWritten: Integer, relationshipProperty: String, propertiesWritten: Integer ----

.Parameters [opts="header",cols="1,3,1,5"]
```

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | relationshipType | String | no | The relationship type in the graph to write back. | relationshipProperty | String | yes | The relationship property to write back. | configuration | Map | yes | Additional parameters to configure writeRelationship.

```
.Configuration [opts="header",cols="1,1,1,7"]
```

| Name | Type | Default | Description | concurrency | Integer | 4 | The number of concurrent threads used for running the procedure. Also provides the default value for `writeConcurrency`. Note, this procedure is always running single-threaded. | writeConcurrency | Integer | 'concurrency' | The number of concurrent threads used for writing the relationship properties. Note, this procedure is always running single-threaded.

```
.Results [opts="header",cols="2,3,5"]
```

| Name | Type | Description | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | graphName | String | The name of a graph stored in the catalog. | relationshipType | String | The type of the relationship that was written. | relationshipsWritten | Integer | Number relationships written. | relationshipProperty | String | The name of the relationship property that was written. | propertiesWritten | Integer | Number relationships properties written.

```
======

[.include-with-delete-relationships] ====== [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.deleteRelationships( graphName: String, relationshipType: String ) YIELD graphName: String, relationshipType: String, deletedRelationships: Integer, deletedProperties: Map ----

.Parameters [opts="header",cols="1,3,1,5"]
```

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | relationshipType | String | no | The relationship type in the graph to remove.

```
.Results [opts="header",cols="2,3,5"]
```

| Name | Type | Description | graphName | String | The name of a graph stored in the catalog. | relationshipType | String | The type of the removed relationships. | deletedRelationships | Integer | Number of removed relationships from the in-memory graph. | deletedProperties | Integer | Map where the key is the name of the relationship property, and the value is the number of removed properties under that name.

====== ====

== Examples

In order to demonstrate the GDS capabilities over node properties, we are going to create a small graph in Neo4j and project it into our graph catalog.

image::example-graphs/node-similarity.svg[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:Person {name: 'Alice'}), (bob:Person {name: 'Bob'}), (carol:Person {name: 'Carol'}), (dave:Person {name: 'Dave'}), (eve:Person {name: 'Eve'}), (guitar:Instrument {name: 'Guitar'}), (synth:Instrument {name: 'Synthesizer'}), (bongos:Instrument {name: 'Bongos'}), (trumpet:Instrument {name: 'Trumpet'}),

(alice)-[:LIKES { score: 5 }]→(guitar), (alice)-[:LIKES { score: 4 }]→(synth), (alice)-[:LIKES { score: 3, strength: 0.5}]→(bongos), (bob)-[:LIKES { score: 4 }]→(guitar), (bob)-[:LIKES { score: 5 }]→(synth), (carol)-[:LIKES { score: 2 }]→(bongos), (dave)-[:LIKES { score: 3 }]→(guitar), (dave)-[:LIKES { score: 1 }]→(synth), (dave)-[:LIKES { score: 5 }]→(bongos) ----

.Project the graph: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('personsAndInstruments', ['Person', 'Instrument'], // <1> { LIKES: { type: 'LIKES', // <2> properties: { strength: { // <3> property: 'strength', defaultValue: 1.0 }, score: { property: 'score' // <4> } } } }) ---- <1> Project node labels `Person` and `Instrument`. <2> Project relationship type `LIKES`. <3> Project property `strength` of relationship type `LIKES` setting a default value of `1.0` because not all relationships have that property. <4> Project property `score` of relationship type `LIKES`.

.Compute the Node Similarity in our graph: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.nodeSimilarity.mutate('personsAndInstruments', { // <1> mutateRelationshipType: 'SIMILAR', // <2> mutateProperty: 'score' // <3> }) ---- <1> Run NodeSimilarity in `mutate` mode on `personsAndInstruments` projected graph. <2> The algorithm will create relationships of type `SIMILAR` in the projected graph. <3> The algorithm will create relationship property `score` for each created relationship.

=== Stream

==== Single property

The most basic case for streaming relationship information from a named graph is a single property. In the example below we stream the relationship property `score`.

[role=query-example] — .Stream a single relationship property: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('personsAndInstruments', // <1> 'score' // <2>) YIELD sourceNodeId, targetNodeId, relationshipType, propertyValue RETURN gds.util.asNode(sourceNodeId).name as source, gds.util.asNode(targetNodeId).name as target, relationshipType, propertyValue ORDER BY source ASC, target ASC ---- <1> The name of the projected graph. <2> The property we want to stream out.

.Results [opts="header"]

| source | target | relationshipType | propertyValue | "Alice" | "Bob" | "SIMILAR" | 0.6666666666666666 | "Alice" | "Bongos" | "LIKES" | 3.0 | "Alice" | "Carol" | "SIMILAR" | 0.3333333333333333 | "Alice" | "Dave" | "SIMILAR" | 1.0 | "Alice" | "Guitar" | "LIKES" | 5.0 | "Alice" | "Synthesizer" | "LIKES" | 4.0 | "Bob" | "Alice" | "SIMILAR" | 0.6666666666666666 | "Bob" | "Dave" | "SIMILAR" | 0.6666666666666666 | "Bob" | "Guitar" | "LIKES" | 4.0 | "Bob" | "Synthesizer" | "LIKES" | 5.0 | "Carol" | "Alice" | "SIMILAR" | 0.3333333333333333 | "Carol" | "Bongos" | "LIKES" | 2.0 | "Carol" | "Dave" | "SIMILAR" | 0.3333333333333333 | "Dave" | "Alice" | "SIMILAR" | 1.0 | "Dave" | "Bob" | "SIMILAR" | 0.6666666666666666 | "Dave" | "Bongos" | "LIKES" | 5.0 | "Dave" | "Carol" | "SIMILAR" | 0.3333333333333333 | "Dave" | "Guitar" | "LIKES" | 3.0 | "Dave" | "Synthesizer" | "LIKES" | 1.0

— As we can see from the results, we get two relationship types (`SIMILAR` and `LIKES`) that have the `score` relationship property. We can further on filter the relationship types we want to stream, this is demonstrated in the next example.

[role=query-example] — .Stream a single relationship property for specific relationship type: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('personsAndInstruments', // <1> 'score', // <2> ['SIMILAR'] // <3>) YIELD sourceNodeId, targetNodeId, relationshipType, propertyValue RETURN gds.util.asNode(sourceNodeId).name as source, gds.util.asNode(targetNodeId).name as target, relationshipType, propertyValue ORDER BY source ASC, target ASC ---- <1> The name of the projected graph. <2> The property we want to stream out. <3> List of relationship types we want to stream the property from, only use the ones we need.

.Results [opts="header"]

| source | target | relationshipType | propertyValue | "Alice" | "Bob" | "SIMILAR" | 0.6666666666666666 | "Alice" | "Carol" | "SIMILAR" | 0.3333333333333333 | "Alice" | "Dave" | "SIMILAR" | 1.0 | "Bob" | "Alice" | "SIMILAR" | 0.6666666666666666 | "Bob" | "Dave" | "SIMILAR" | 0.6666666666666666 | "Carol" | "Alice" | "SIMILAR" | 0.3333333333333333 | "Carol" | "Dave" | "SIMILAR" | 0.3333333333333333 | "Dave" | "Alice" | "SIMILAR" | 1.0 | "Dave" | "Bob" | "SIMILAR" | 0.6666666666666666 | "Dave" | "Carol" | "SIMILAR" | 0.3333333333333333

— ==== Multiple properties

It is also possible to stream multiple relationship properties.

[role=query-example] — .Stream multiple relationship properties: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperties('personsAndInstruments', // <1> ['score', 'strength'], // <2> ['LIKES'] // <3>) YIELD sourceNodeId, targetNodeId, relationshipType, relationshipProperty, propertyValue RETURN gds.util.asNode(sourceNodeId).name as source, gds.util.asNode(targetNodeId).name as target, relationshipType, relationshipProperty, propertyValue ORDER BY source ASC, target ASC ---- <1> The name of the projected graph. <2> List of properties we want to stream out, allows us to stream more than one property. <3> List of relationship types we want to stream the property from, only use the ones we need.

.Results [opts="header"]

| source | target | relationshipType | relationshipProperty | propertyValue | "Alice" | "Bongos" | "LIKES" | "score" | 3.0 | "Alice" | "Bongos" | "LIKES" | "strength" | 0.5 | "Alice" | "Guitar" | "LIKES" | "score" | 5.0 | "Alice" | "Guitar" | "LIKES" | "strength" | 1.0 | "Alice" | "Synthesizer" | "LIKES" | "score" | 4.0 | "Alice" | "Synthesizer" | "LIKES" | "strength" | 1.0 | "Bob" | "Guitar" | "LIKES" | "score" | 4.0 | "Bob" | "Guitar" | "LIKES" | "strength" | 1.0 | "Bob" | "Synthesizer" | "LIKES" | "score" | 5.0 | "Bob" | "Synthesizer" | "LIKES" | "strength" | 1.0 | "Carol" | "Bongos" | "LIKES" | "score" | 2.0 | "Carol" | "Bongos" | "LIKES" | "strength" | 1.0 | "Dave" | "Bongos" | "LIKES" | "score" | 5.0 | "Dave" | "Bongos" | "LIKES" | "strength" | 1.0 | "Dave" | "Guitar" | "LIKES" | "score" | 3.0 | "Dave" | "Guitar" | "LIKES" | "strength" | 1.0 | "Dave" | "Synthesizer" | "LIKES" | "score" | 1.0 | "Dave" | "Synthesizer" | "LIKES" | "strength" | 1.0

—

==== Multiple relationship types

Similar to the multiple relationship properties we can stream properties for multiple relationship types.

[role=query-example] — .Stream relationship properties of a multiple relationship projections: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperties('personsAndInstruments', // <1> ['score'], // <2> ['LIKES', 'SIMILAR'] // <3>) YIELD sourceNodeId, targetNodeId, relationshipType, relationshipProperty, propertyValue RETURN gds.util.asNode(sourceNodeId).name as source, // <4> gds.util.asNode(targetNodeId).name as target, // <5> relationshipType, relationshipProperty, propertyValue ORDER BY source ASC, target ASC ---- <1> The name of the projected graph. <2> List of properties we want to stream out, allows us to stream more than one property. <3> List of relationship types we want to stream the property from, only use the ones we need. <4> Return the `name` of the source node. <5> Return the `name` of the target node.

.Results [opts="header"]

| source | target | relationshipType | relationshipProperty | propertyValue | "Alice" | "Bob" | "SIMILAR" | "score" | 0.6666666666666666 | "Alice" | "Bongos" | "LIKES" | "score" | 3.0 | "Alice" | "Carol" | "SIMILAR" | "score" | 0.3333333333333333 | "Alice" | "Dave" | "SIMILAR" | "score" | 1.0 | "Alice" | "Guitar" | "LIKES" | "score" | 5.0 | "Alice" | "Synthesizer" | "LIKES" | "score" | 4.0 | "Bob" | "Alice" | "SIMILAR" | "score" | 0.6666666666666666 | "Bob" | "Dave" | "SIMILAR" | "score" | 0.6666666666666666 | "Bob" | "Guitar" | "LIKES" | "score" | 4.0 | "Bob" | "Synthesizer" | "LIKES" | "score" | 5.0 | "Carol" | "Alice" | "SIMILAR" | "score" | 0.3333333333333333 | "Carol" | "Bongos" | "LIKES" | "score" | 2.0 | "Carol" | "Dave" | "SIMILAR" | "score" | 0.3333333333333333 | "Dave" | "Alice" | "SIMILAR" | "score" | 1.0 | "Dave" | "Bob" | "SIMILAR" | "score" | 0.6666666666666666 | "Dave" | "Bongos" | "LIKES" | "score" | 5.0 | "Dave" | "Carol" | "SIMILAR" | "score" | 0.3333333333333333 | "Dave" | "Guitar" | "LIKES" | "score" | 3.0 | "Dave" | "Synthesizer" | "LIKES" | "score" | 1.0

— NOTE: The properties we want to stream must exist for each specified relationship type.

=== Write

We can write relationships stored in a named in-memory graph back to Neo4j. This can be used to write algorithm results (for example from Node Similarity) or relationships that have been aggregated during graph creation.

The relationships to write are specified by a relationship type.

NOTE: Relationships are always written using a single thread.

==== Relationship type

[role=query-example] — .Write relationships to Neo4j: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.writeRelationship('personsAndInstruments', // <1> 'SIMILAR' // <2>) YIELD graphName, relationshipType, relationshipProperty, relationshipsWritten, propertiesWritten ---- <1> The name of the projected graph. <2> The relationship type we want to write back to the Neo4j database.

.Results [opts="header"]

| graphName | relationshipType | relationshipProperty | relationshipsWritten | propertiesWritten | "personsAndInstruments" | "SIMILAR" | null | 10 | 0

— By default, no relationship properties will be written, as it can be seen from the results, the `relationshipProperty` value is `null` and `propertiesWritten` are `0`.

Here is an illustration of how the example graph looks in Neo4j after executing the example above.

image::write_relationships_graph.svg[Visualization of the example graph after writing relationships back,align="center"]

The `SIMILAR` relationships have been added to the underlying database and can be used in Cypher queries or for projecting to in-memory graph for running algorithms. The relationships in this example are undirected because we used Node Similarity to mutate the in-memory graph and this algorithm creates undirected relationships, this may not be the case if we use different algorithms.

==== Relationship type with property

To write relationship properties, these have to be explicitly specified.

[role=query-example] — .Write relationships and their properties to Neo4j: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.writeRelationship('personsAndInstruments', // <1> 'SIMILAR', // <2> 'score' // <3>) YIELD graphName, relationshipType, relationshipProperty, relationshipsWritten, propertiesWritten ---- <1> The name of the projected graph. <2> The relationship type we want to write back to the Neo4j database. <3> The property name of the relationship we want to write back to the Neo4j database.

.Results [opts="header"]

| graphName | relationshipType | relationshipProperty | relationshipsWritten | propertiesWritten | "personsAndInstruments" | "SIMILAR" | "score" | 10 | 10

—

=== Delete

We can delete all relationships of a given type from a named graph in the catalog. This is useful to free up main memory or to remove accidentally created relationship types.

[NOTE] ==== Deleting relationships of a given type is only possible if it is not the last relationship type present in the graph. If we still want to delete these relationships we need to drop the graph instead. ====

[role=query-example] — .Delete all relationships of type `SIMILAR` from a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.deleteRelationships('personsAndInstruments', // <1> 'SIMILAR' // <2>) YIELD graphName, relationshipType, deletedRelationships, deletedProperties ---- <1> The name of the projected graph. <2> The relationship type we want to delete from the projected graph.

.Results [opts="header"]

| graphName | relationshipType | deletedRelationships | deletedProperties | "personsAndInstruments" | "SIMILAR" | 10 | {score=10}

— :leveloffset: 2

:leveloffset: +3

= Export operations

:leveloffset: +1

= Create Neo4j databases from named graphs

We can create new Neo4j databases from named in-memory graphs stored in the graph catalog. All nodes, relationships and properties present in an in-memory graph are written to a new Neo4j database. This includes data that has been projected in `gds.graph.create` and data that has been added by running algorithms in `mutate` mode. The newly created database will be stored in the Neo4j `databases` directory using a given database name.

The feature is useful in the following, exemplary scenarios:

* Avoid heavy write load on the operational system by exporting the data instead of writing back. * Create an analytical view of the operational system that can be used as a basis for running algorithms. * Produce snapshots of analytical results and persistent them for archiving and inspection. * Share analytical results within the organization.

== Syntax

[.graph-export-syntax] — .Export an in-memory graph to a new database in the Neo4j databases directory: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.export(graphName: String, configuration: Map) YIELD dbName: String, graphName: String, nodeCount: Integer, nodePropertyCount: Integer, relationshipCount: Integer, relationshipTypeCount: Integer, relationshipPropertyCount: Integer, writeMillis: Integer ----

.Parameters [opts="header",cols="1,1,1, 4"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | configuration | Map | no | Additional parameters to configure the database export.

.Graph export configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dbName | String | none | No | The name of the exported Neo4j database. | writeConcurrency | Boolean | 4 | yes | The number of concurrent threads used for writing the database. | enableDebugLog | Boolean | false | yes | Prints debug information to Neo4j log files. | batchSize | Integer | 10000 | yes | Number of entities processed by one single thread at a time. | defaultRelationshipType | String | __ALL__ | yes | Relationship type used for * relationship projections. | additionalNodeProperties | String, List or Map | {} | yes | Allows for exporting additional node properties from the original graph backing the in-memory graph.

.Results [opts="header",cols="2,1,4"]

| Name | Type | Description | dbName | String | The name of the exported Neo4j database. | graphName | String | The name under which the graph is stored in the catalog. | nodeCount | Integer | The number of nodes exported. | nodePropertyCount | Integer | The number of node properties exported. | relationshipCount | Integer | The number of relationships exported. | relationshipTypeCount | Integer | The number of relationship types exported. | relationshipPropertyCount | Integer | The number of relationship properties exported. | writeMillis | Integer | Milliseconds for writing the graph into the new database.

—

== Example

.Export the `my-graph` from GDS into a Neo4j database called `mydatabase`: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.export('my-graph', { dbName: 'mydatabase' }) ----

The new database can be started using databases management commands.

[NOTE] ==== The database must not exist when using the export procedure. It needs to be created manually using the following commands. ====

.After running exporting the graph, we can start a new database and query the exported graph: [source, cypher, role=noplay, indent=0] ---- :use system CREATE DATABASE mydatabase; :use mydatabase MATCH (n) RETURN n; ----

== Example with additional node properties

Suppose we have a graph `my-db-graph` in the Neo4j database that has a string node property `myproperty`, and that we have a corresponding in-memory graph called `my-in-memory-graph` which does not have the `myproperty` node property. If we want to export `my-in-memory-graph` but additionally add the `myproperty` properties from `my-db-graph` we can use the `additionalProperties` configuration parameter.

.Export the `my-in-memory-graph` from GDS with `myproperty` from `my-db-graph` into a Neo4j database called `mydatabase`: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.export('my-graph', { dbName: 'mydatabase', additionalNodeProperties: ['myproperty']}) ----

The new database can be started using databases management commands.

[NOTE] ==== The original database (`my-db-graph`) must not have changed since loading the in-memory representation (`my-in-memory-graph`) that we export in order for the export to work correctly. ====

The `additionalNodeProperties` parameter uses the same syntax as `nodeProperties` of the graph create procedure. So we could for instance define a default value for our `myproperty`.

.Export the `my-in-memory-graph` from GDS with `myproperty` from `my-db-graph` with default value into a Neo4j database called `mydatabase`: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.export('my-graph', { dbName: 'mydatabase', additionalNodeProperties: [{ myproperty: {defaultValue: 'my-default-value'}}] }) ----

:leveloffset: 2

:leveloffset: +1

[.beta]

= Export a named graph to CSV

We can export named in-memory graphs stored in the graph catalog to a set of CSV files. All nodes, relationships and properties present in an in-memory graph are exported. This includes data that has been projected with `gds.graph.create` and data that has been added by running algorithms in `mutate` mode. The location of the exported CSV files can be configured via the configuration parameter `gds.export.location` in the `neo4j.conf`. All files will be stored in a subfolder using the specified export name. The export will fail if a folder with the given export name already exists.

[NOTE] ==== The `gds.export.location` parameter must be configured for this feature. ====

== Syntax

[.graph-export-syntax] — .Export a named graph to a set of CSV files: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graph.export.csv(graphName: String, configuration: Map) YIELD graphName: String, exportName: String, nodeCount: Integer, nodePropertyCount: Integer, relationshipCount: Integer, relationshipTypeCount: Integer,

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | configuration | Map | no | Additional parameters to configure the database export.

.Graph export configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | exportName | String | none | No | The name of the directory where the graph is exported to. The absolute path of the exported CSV files depends on the configuration parameter `gds.export.location` in the `neo4j.conf`. | writeConcurrency | Boolean | 4 | yes | The number of concurrent threads used for writing the database. | defaultRelationshipType | String | __ALL__ | yes | Relationship type used for `*` relationship projections. | additionalNodeProperties | String, List or Map | {} | yes | Allows for exporting additional node properties from the original graph backing the in-memory graph.

.Results [opts="header",cols="2,1,4"]

| Name | Type | Description | graphName | String | The name under which the graph is stored in the catalog. | exportName | String | The name of the directory where the graph is exported to. | nodeCount | Integer | The number of nodes exported. | nodePropertyCount | Integer | The number of node properties exported. | relationshipCount | Integer | The number of relationships exported. | relationshipTypeCount | Integer | The number of relationship types exported. | relationshipPropertyCount | Integer | The number of relationship properties exported. | writeMillis | Integer | Milliseconds for writing the graph into the new database.

— == Estimation

As many other procedures in GDS, export to csv has an estimation mode. For more details see Memory Estimation. Using the `gds.beta.graph.export.csv.estimate` procedure, it is possible to estimate the required disk space of the exported CSV files. The estimation uses sampling to generate a more accurate estimate.

[.estimate-syntax] — .Estimate the required disk space for exporting a named graph to CSV files.: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graph.export.csv.estimate(graphName:String, configuration: Map) YIELD nodeCount: Integer, relationshipCount: Integer, requiredMemory: String, treeView: String, mapView: Map, bytesMin: Integer, bytesMax: Integer, heapPercentageMin: Float, heapPercentageMax: Float; ----

.Parameters [opts="header",cols="1,1,1, 4"]

| Name | Type | Optional | Description | graphName | String | no | The name under which the graph is stored in the catalog. | configuration | Map | no | Additional parameters to configure the database export.

.Graph export estimate configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | exportName | String | none | no | Name of the folder the exported CSV files are saved at. | samplingFactor | Double | 0.001 | yes | The fraction of nodes and relationships to sample for the estimation. | writeConcurrency | Boolean | 4 | yes | The number of concurrent threads used for writing the database. | defaultRelationshipType | String | __ALL__ | yes | Relationship type used for `*` relationship projections.

.Results [opts="header",cols="1m,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes in the graph. | relationshipCount |

Integer | The number of relationships in the graph. | requiredMemory | String | An estimation of the required memory in a human readable format. | treeView | String | A more detailed representation of the required memory, including estimates of the different components in human readable format. | mapView | Map | A more detailed representation of the required memory, including estimates of the different components in structured format. | bytesMin | Integer | The minimum number of bytes required. | bytesMax | Integer | The maximum number of bytes required. | heapPercentageMin | Float | The minimum percentage of the configured maximum heap required. | heapPercentageMax | Float | The maximum percentage of the configured maximum heap required.

—

== Export format

The format of the exported CSV files is based on the format that is supported by the Neo4j Admin import command.

=== Nodes

Nodes are exported into files grouped by the nodes labels, i.e., for every label combination that exists in the graph a set of export files is created. The naming schema of the exported files is: nodes_LABELS_INDEX.csv, where:

- LABELS is the ordered list of labels joined by . - INDEX is a number between 0 and concurrency.

For each label combination one or more data files are created, as each exporter thread exports into a separate file.

Additionally, each label combination produces a single header file, which contains a single line describing the columns in the data files More information about the header files can be found here: CSV header format.

For example a Graph with the node combinations :A, :B and :A:B might create the following files

---- nodes_A_header.csv nodes_A_0.csv nodes_B_header.csv nodes_B_0.csv nodes_B_2.csv nodes_A_B_header.csv nodes_A_B_0.csv nodes_A_B_1.csv nodes_A_B_2.csv ----

=== Relationships

The format of the relationship files is similar to those of the nodes. Relationships are exported into files grouped by the relationship type. The naming schema of the exported files is: relationships_TYPE_INDEX.csv, where:

- TYPE is the relationship type - INDEX is a number between 0 and concurrency.

For each relationship type one or more data files are created, as each exporter thread exports into a separate file.

Additionally, each relationship type produces a single header file, which contains a single line describing the columns in the data files.

For example a Graph with the relationship types :KNOWS, :LIVES_IN might create the following files

---- relationships_KNOWS_header.csv relationships_KNOWS_0.csv relationships_LIVES_IN_header.csv relationships_LIVES_IN_0.csv relationships_LIVES_IN_2.csv ----

== Example

.Export the my-graph from GDS into a directory my-export: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graph.export.csv('my-graph', { exportName: 'my-export' }) ----

== Example with additional node properties

Suppose we have a graph my-db-graph in the Neo4j database that has a string node property myproperty, and that we have a corresponding in-memory graph called my-in-memory-graph which does not have the myproperty node property. If we want to export my-in-memory-graph but additionally add the myproperty properties from my-db-graph we can use the additionalProperties configuration parameter.

.Export the my-in-memory-graph from GDS with the myproperty from my-db-graph into a directory my-export: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.export('my-graph', { exportName: 'my-export', additionalNodeProperties: ['myproperty']}) ----

[NOTE] ==== The original database (my-db-graph) must not have changed since loading the in-memory representation (my-in-memory-graph) that we export in order for the export to work correctly. ====

The additionalNodeProperties parameter uses the same syntax as nodeProperties of the graph create procedure. So we could for instance define a default value for our myproperty.

.Export the my-in-memory-graph from GDS with myproperty from my-db-graph with default value into a directory called my-export: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.export('my-graph', { exportName: 'my-export', additionalNodeProperties: [{ myproperty: {defaultValue: 'my-default-value'}}] }) ----

| Name | Description | gds.version | Return the version of the installed Neo4j Graph Data Science library.

[role=query-example] — .Usage: [source, cypher, role=noplay, indent=0] ---- RETURN gds.version() AS version ----

.Results [opts="header"]

| version | "1.7.1"

—

== Numeric Functions

.Numeric Functions [opts=header,cols="1m,1"]

| Name | Description | gds.util.NaN | Returns NaN as a Cypher value. | gds.util.infinity | Return infinity as a Cypher value. | gds.util.isFinite | Return true iff the given argument is a finite value (not ±Infinity, NaN, or null). | gds.util.isInfinite | Return true iff the given argument is not a finite value (not ±Infinity, NaN, or null).

=== Syntax

[opts=header,cols="1m,1"]

| Name | Parameter | gds.util.NaN() | - | gds.util.infinity() | - | gds.util.isFinite(value: NUMBER) | value to be checked if it is finite | gds.util.isInfinite(value: NUMBER) | value to be checked if it is infinite.

=== Examples

[role=query-example] — .Example for gds.util.IsFinite: [source, cypher, role=noplay, indent=0] ---- UNWIND [1.0, gds.util.NaN(), gds.util.infinity()] AS value RETURN gds.util.isFinite(value) AS isFinite ----

.Results [opts="header"]

| isFinite | true | false | false

— [role=query-example] — .Example for gds.util.isInfinite(): [source, cypher, role=noplay, indent=0] ---- UNWIND [1.0, gds.util.NaN(), gds.util.infinity()] AS value RETURN gds.util.isInfinite(value) AS isInfinite ----

.Results [opts="header"]

| isInfinite | false | true | true

— The utility function `gds.util.NaN` can be used as an default value for input parameters, as shown in the examples of cosine similarity. A common usage of `gds.util.IsFinite` and `gds.util.IsInfinite` is for filtering streamed results, as for instance seen in the examples of `gds.alpha.allShortestPaths`.

== Node id functions

.Node id functions [opts=header,cols="1m,1m"]

| Name | Description | gds.util.asNode | Return the node object for the given node id or null if none exists. | gds.util.asNodes | Return the node objects for the given node ids or an empty list if none exists.

=== Syntax

[opts=header,cols="1m,1"]

| Name | Parameters | gds.util.asNode(nodeId: NUMBER) | nodeId of a node in the neo4j-graph | gds.util.asNodes(nodeIds: List of NUMBER) | list of nodeIds of nodes in the neo4j-graph

=== Examples

Consider the graph created by the following Cypher statement:

.Example graph: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (nAlice:User {name: 'Alice'}) CREATE (nBridget:User {name: 'Bridget'}) CREATE (nCharles:User {name: 'Charles'}) CREATE (nAlice)-[:LINK]→(nBridget) CREATE (nBridget)-[:LINK]→(nCharles) ----

[role=query-example] — .Example for gds.util.asNode: [source, cypher, role=noplay, indent=0] ---- MATCH (u:User{name: 'Alice'}) WITH id(u) AS nodeId RETURN gds.util.asNode(nodeId).name AS node ----

.Results [opts="header"]

| node | "Alice"

— [role=query-example] — .Example for gds.util.asNodes: [source, cypher, role=noplay, indent=0] ---- MATCH (u:User) WHERE NOT u.name = 'Charles' WITH collect(id(u)) AS nodeIds RETURN [x in gds.util.asNodes(nodeIds) | x.name] AS nodes ----

.Results [opts="header"]

| nodes | [Alice, Bridget]

— As many algorithms streaming mode only return the node id, `gds.util.asNode` and `gds.util.asNodes` can be used to retrieve the whole node from the neo4j database.

:leveloffset: 2

:leveloffset: +2

:description: This chapter explains how to execute Cypher queries on named graphs in the Neo4j Graph Data Science library. [.enterprise-edition] = Cypher on GDS graph

[abstract] — This chapter explains how to execute Cypher queries on named graphs in the Neo4j Graph Data Science library. — [.alpha-symbol] [.tier-note] This feature is in the alpha tier.

Exploring in-memory graphs after loading them and potentially executing algorithms in mutate mode can be tricky in the Neo4j Graph Data Science library. A natural way to achieve this in the Neo4j database is to use Cypher queries. Cypher queries allow for example to get a hold of which properties are present on a node among many other things. Executing Cypher queries on an in-memory graph can be achieved by leveraging the `gds.alpha.create.cypherdb` procedure. This procedure will create a new impermanent database which you can switch to. That database will then use data from the in-memory graph as compared to the store files for usual Neo4j databases.

== Limitations

Although it is possible to execute arbitrary Cypher queries on the database created by the `gds.alpha.create.cypherdb` procedure, not every aspect of Cypher is implemented yet. Some known limitations are listed below:

* Dropping the newly created database **Restarting the DBMS will remove the database instead * Writes** All queries that attempt to write things, such as nodes, properties or labels, will fail * Referencing relationships **Referencing variables bound to a relationship is not supported, this includes returning relationships** It is however possible to use relationship as traversal, i.e. `MATCH (n)-[:KNOWS]→(m)` will work as expected

== Syntax

[.create-cypher-db-syntax] — [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.create.cypherdb(dbName: String graphName: String) YIELD dbName: String, graphName: String, createMillis: Integer ----

.Parameters [opts="header",cols="1,1,1, 4"]

| Name | Type | Optional | Description | dbName | String | no | The name under which the new database is stored. | graphName | String | no | The name under which the graph is stored in the catalog.

.Results [opts="header",cols="2,1,4"]

| Name | Type | Description | dbName | String | The name under which the new database is stored. | graphName | String | The name under which the graph is stored in the catalog. | createMillis | Integer | Milliseconds for creating the database.

—

== Example

To demonstrate how to execute cypher statements on in-memory graphs we are going to create a simple social network graph. We will use this graph to create a new database which we will execute our statements on.

[source, cypher, role=noplay, indent=0] ---- CREATE (alice:Person { name: 'Alice', age: 23 }), (bob:Person { name: 'Bob', age: 42 }), (carl:Person { name: 'Carl', age: 31 }),

(alice)-[:KNOWS]→(bob), (bob)-[:KNOWS]→(alice), (alice)-[:KNOWS]→(carl) ----

We will now load a graph projection of the created graph via the graph create procedure:

.Project `Person` nodes and `KNOWS` relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('social_network', 'Person', 'KNOWS') YIELD graphName, nodeCount, relationshipCount ----

.Results [opts="header"]

| graph | nodeCont | relationshipCount | "social_network" | 3 | 3

With a named graph loaded into the Neo4j Graph Data Science library, we can proceed to create the new database using the loaded graph as underlying data.

.Create a new database `gdsDb` using our `social_network` graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.create.cypherdb('gdsDb', 'social_network') ----

In order to verify that the new database was created successfully we can use the Neo4j database administration commands.

[source, cypher, role=noplay, indent=0] ---- SHOW DATABASES ----

.Results [opts="header"]

| name | address | role | requestedStatus | currentStatus | error | default | home | "neo4j" | "localhost:7687" | "standalone" | "online" | "online" | "" | true | true | "system" | "localhost:7687" | "standalone" | "online" | "online" | "" | false | false | "gdsDb" | "localhost:7687" | "standalone" | "online" | "online" | "" | false | false

We can now switch to the newly created database.

[source, cypher, role=noplay, indent=0] ---- :use gdsDb ----

Finally, we are set up to execute cypher queries on our in-memory graph.

[source, cypher, role=noplay, indent=0] ---- MATCH (n:Person)-[:KNOWS]→(m:Person) RETURN n.age AS age1, m.age AS age2 ----

.Results [opts="header"]

| age1 | age2 | 23 | 42 | 42 | 23 | 23 | 31

We can see that the returned ages correspond to the structure of the original graph.

:leveloffset: 2

:leveloffset: +2

:description: This section explains administration capabilities in the Neo4j Graph Data Science library. [.enterprise-edition] = Administration

[abstract] — This section explains administration capabilities in the Neo4j Graph Data Science library. — The GDS catalog offers elevated access to administrator users. Any user granted a role with the name `admin` is considered an administrator by GDS.

A GDS administrator has access to graphs created by any other user. This includes the ability to list, drop and run algorithms over these graphs.

== Disambiguating identically named graphs

Sometimes, several users (including the admin user themselves) could have a graph with the same name. To disambiguate between these graphs, the `username` configuration parameter can be used.

== Examples

We will illustrate the administrator capabilities using a small example. In this example we have three users where one is an administrator. We create the users and set up the roles using the following Cypher commands:

[source, cypher, role=noplay, indent=0] ---- CREATE USER alice SET PASSWORD $alice_pw CHANGE NOT REQUIRED; CREATE USER bob SET PASSWORD $bob_pw CHANGE NOT REQUIRED; CREATE USER carol SET PASSWORD $carol_pw CHANGE NOT REQUIRED;

GRANT ROLE reader TO alice; GRANT ROLE reader TO bob; GRANT ROLE admin TO carol; ----

As we can see, `alice` and `bob` are standard users with read access to the database. `carol` is an administrator by virtue of being granted the `admin` role (for more information about this role see the Cypher manual).

Now `alice` and `bob` each create a few graphs. They both create a graph called `graphA` and `bob` also creates a graph called `graphB`.

=== Listing

To list all graphs from all users, `carol` simply uses the graph list procedure.

.Listing all graphs as administrator user: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.list() YIELD graphName ----

.Results [opts="header"]

| graphName | "graphA" | "graphA" | "graphB"

Notice that all graphs from all users are visible to carol since they are considered a GDS admin.

=== Running algorithms with other users' graphs

carol may use graphB by simply naming it.

.carol can run WCC on the graphB graph owned by bob: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.stats('graphB') YIELD componentCount ----

To use the graphA owned by alice, carol must use the username override.

.carol can run WCC on graphA owned by alice: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.stats('graphA', { username: 'alice' }) YIELD componentCount ----

=== Dropping other users' graphs

Unlike for listing, the full procedure signature must be used when using the username override to disambiguate. In the query below we have used the default values for the second and third parameter for the drop procedure. username is the fourth parameter. For more details see Dropping graphs.

.To drop graphA owned by bob, carol can run the following: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.drop('graphA', true, '', 'bob') YIELD graphName ----

:leveloffset: 2

:leveloffset: +1

:description: This section details the model catalog operations available to manage named trained models within the Neo4j Graph Data Science library. = Model catalog

[abstract] — This section details the model catalog operations available to manage named trained models within the Neo4j Graph Data Science library. —

Some graph algorithms use trained models in their computation. A model is generally a mathematical formula representing a real-world or fictitious entities. Each algorithm requiring a trained model provides the formulation and means to compute this model (see GraphSage train syntax).

The model catalog is a concept within the GDS library that allows storing and managing multiple trained models by name.

This chapter explains the available model catalog operations.

[opts=header,cols="1m,1"]

| Name | Description | gds.beta.model.list | Prints information about models that are currently available in the catalog. | gds.beta.model.exists | Checks if a named model is available in the catalog. | gds.beta.model.drop | Drops a named model from the catalog. | gds.alpha.model.store | Stores a names model from the catalog on disk. | gds.alpha.model.load | Loads a named and stored model from disk. | gds.alpha.model.delete | Removes a named and stored model from disk. | gds.alpha.model.publish | Makes a model accessible to all users.

[NOTE] ==== Training models is a responsibility of the corresponding algorithm and is provided by a procedure mode - `train`. Training, using, listing, and dropping named models are management operations bound to a Neo4j user. Models trained by a different Neo4j user are not accessible at any time. ====

:leveloffset: 2

:leveloffset: +2

[.beta] = Listing models

Information about models in the catalog can be retrieved using the `gds.beta.model.list()` procedure.

== Syntax

[.model-list-syntax] — .List models from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.model.list(modelName: String) YIELD modelInfo: Map, trainConfig: Map, graphSchema: Map, loaded: Boolean, stored: Boolean, creationTime: DateTime, shared: Boolean ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | yes | The name of a model. If not specified, all models in the catalog are listed.

.Results [opts="header",cols="1,1,4"]

| Name | Type | Description | modelInfo | Map | Detailed information about the trained model. Always includes the `modelName` and `modelType`, e.g., GraphSAGE. Dependent on the model type, there are additional fields. | trainConfig | Map | The configuration used for training the model. | graphSchema | Map | The schema of the graph on which the model was trained. | loaded | Boolean | True, if the model is loaded in the in-memory model catalog. | stored | Boolean | True, if the model is stored on disk. | creationTime | Datetime | Time when the model was created. | shared | Boolean | True, if the model is shared between users.

— == Examples

Once we have trained models in the catalog we can see information about either all of them or a single model using its name

=== Listing all models

[role=query-example] — .Listing detailed information about all models: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.model.list() YIELD modelInfo, loaded, shared, stored RETURN modelInfo.modelName AS modelName, loaded, shared, stored ----

.Results [opts="header",cols="1m,1m,1m,1m"]

| modelName | loaded | shared | stored | "my-model" | true | false | false

— === Listing a specific model

[role=query-example] — .Listing detailed information about specific model: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.model.list('my-model') YIELD modelInfo, loaded, shared, stored RETURN modelInfo.modelName AS modelName, loaded, shared, stored ----

.Results [opts="header",cols="1m,1m,1m,1m"]

| modelName | loaded | shared | stored | "my-model" | true | false | false

```
— :leveloffset: 2

:leveloffset: +2

[.beta]  = Checking if a model exists

We can check if a model is available in the catalog by looking up its name.

== Syntax

[.model-exists-syntax] — .Check if a model exists in the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.model.exists(modelName: String) YIELD modelName: String, modelType: String, exists: Boolean ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of a model.

```
.Results [opts="header",cols="1,1,4"]
```

| Name | Type | Description | modelName | String | The name of a model. | modelType | String | The type of the model. | exists | Boolean | True, if the model exists in the model catalog.

```
—

== Example

In this section we are going to demonstrate the usage of `gds.beta.model.exists`. Assume we trained a model by running `train` on one of our Machine learning algorithms.

[role=query-example] — .Check if a model exists in the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.model.exists('my-model'); ----

.Results [opts="header"]
```

| modelName | modelType | exists | "my-model" | "graphSage" | true

```
— :leveloffset: 2

:leveloffset: +2

[.beta]  = Removing models

If we no longer need a trained model and want to free up memory, we can remove the model from the catalog.

== Syntax

[.model-drop-syntax] — .Remove a model from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.model.drop(modelName: String) YIELD modelInfo: Map, trainConfig: Map, graphSchema: Map, loaded: Boolean, stored: Boolean, creationTime: DateTime, shared: Boolean ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of a model stored in the catalog.

.Results [opts="header",cols="1,1,4"]

| Name | Type | Description | modelInfo | Map | Detailed information about the trained model. Always includes the `modelName` and `modelType`, e.g., GraphSAGE. Dependent on the model type, there are additional fields. | trainConfig | Map | The configuration used for training the model. | graphSchema | Map | The schema of the graph on which the model was trained. | loaded | Boolean | True, if the model is loaded in the in-memory model catalog. | stored | Boolean | True, if the model is stored on disk. | creationTime | Datetime | Time when the model was created. | shared | Boolean | True, if the model is shared between users.

—

== Example

In this section we are going to demonstrate the usage of `gds.beta.model.drop`. Assume we trained a model by running `train` on one of our Machine learning algorithms.

[role=query-example] — .Remove a model from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.model.drop('my-model') YIELD modelInfo, loaded, shared, stored RETURN modelInfo.modelName AS modelName, loaded, shared, stored ----

.Results [opts="header",cols="1m,1m,1m,1m"]

| modelName | loaded | shared | stored | "my-model" | true | false | false

— In this example, the removed `my-model` was of the imaginary type `some-model-type`. The model was loaded in-memory, but neither stored on disk nor published.

NOTE: If the model name does not exist, an error will be raised.

:leveloffset: 2

:leveloffset: +2

[.enterprise-edition] = Storing models on disk

[.alpha-symbol] [.tier-note] The model store feature is in the alpha tier.

The model catalog exists as long as the Neo4j instance is running. When Neo4j is restarted, models are no longer available in the catalog and need to be trained again. This can be prevented by storing a model on disk.

The location of the stored models can be configured via the configuration parameter `gds.model.store_location` in the `neo4j.conf`. The location must be a directory and writable by the Neo4j process.

[NOTE] ==== The `gds.model.store_location` parameter must be configured for this feature. ====

[.alpha] == Storing models from the catalog on disk

Models that can be stored

* GraphSAGE model * Node Classification model * Link Prediction model

Models that cannot be stored

* Link prediction training pipeline * Link prediction pipeline

=== Syntax

[.model-store-syntax] — .Remove a model from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.model.store(modelName: String, failIfUnsupportedType: Boolean) YIELD modelName: String, storeMillis: Integer ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of a model. | failIfUnsupportedType | Boolean | true | yes | By default, the library will raise an error when trying to store a non-supported model. When set to `false`, the procedure returns an empty result.

.Results [opts="header",cols="1,1,4"]

| Name | Type | Description | modelName | String | The name of the stored model. | storeMillis | Integer | The number of milliseconds it took to store the model.

—

=== Example

[role=query-example, no-result=true] — .Store a model on disk: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.model.store('my-model') YIELD modelName, storeMillis ---- — [.alpha] == Loading models from disk

GDS will discover available models from the configured store location upon database startup. During discovery, only model metadata is loaded, not the actual model data. In order to use a stored model, it has to be explicitly loaded.

=== Syntax

[.model-load-syntax] — .Remove a model from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.model.load(modelName: String) YIELD modelName: String, loadMillis: Integer ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of a model.

.Results [opts="header",cols="1,1,4"]

| Name | Type | Description | modelName | String | The name of the loaded model. | loadMillis | Integer | The number of milliseconds it took to load the model.

— === Example

[role=query-example, no-result=true] — .Store a model on disk: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.model.load('my-model') YIELD modelName, loadMillis ---- — To verify if a model is loaded, we can use the `gds.beta.model.list` procedure. The procedure returns flags to indicate if the model is stored and if the model is loaded into memory. The operation is idempotent, and skips loading if the model is already loaded.

[.alpha] == Deleting models from disk

To remove a stored model from disk, it has to be deleted. This is different from dropping a model. Dropping a model will remove it from the in-memory model catalog, but not from disk. Deleting a model will remove it from disk, but keep it in the in-memory model catalog if it was already loaded.

=== Syntax

[.model-delete-syntax] — .Remove a model from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.model.delete(modelName: String) YIELD modelName: String, deleteMillis: Integer ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of a model.

.Results [opts="header",cols="1,1,4"]

| Name | Type | Description | modelName | String | The name of the loaded model. | deleteMillis | Integer | The number of milliseconds it took to delete the model.

— === Example

[role=query-example, no-result=true] — .Store a model on disk: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.model.delete('my-model') YIELD modelName, deleteMillis ---- — :leveloffset: 2

:leveloffset: +2

[.alpha] [.enterprise-edition] = Publishing models

[.alpha-symbol] [.tier-note] Publishing models is an alpha tier feature.

By default, a trained model is visible to the user that created it. Making a model accessible to other users can be achieved by publishing it.

== Syntax

[.model-publish-syntax] — .Publish a model from the catalog: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.model.publish(modelName: String) YIELD modelInfo: Map, trainConfig: Map, graphSchema: Map, loaded: Boolean, stored: Boolean, creationTime: DateTime, shared: Boolean ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of a model stored in the catalog.

.Results [opts="header",cols="1,1,4"]

| Name | Type | Description | modelInfo | Map | Detailed information about the trained model. Always includes the `modelName` and `modelType`, e.g., GraphSAGE. Dependent on the model type, there are additional fields. | trainConfig | Map | The configuration used for training the model. | graphSchema | Map | The schema of the graph on which the model was trained. | loaded | Boolean | True, if the model is loaded in the in-memory model catalog. | stored | Boolean | True, if the model is stored on disk. | creationTime | Datetime | Time when the model was created. | shared | Boolean | True, if the model is shared between users.

—

== Examples

[role=query-example] — .Publishing trained model: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.model.publish('my-model') YIELD modelInfo, loaded, shared, stored RETURN modelInfo.modelName AS modelName, shared ----

.Results [opts="header"]

| modelName | shared | "my-model_public" | true

— We can see that the model is now shared. The shared model has the public suffix.

```
:leveloffset: 2

:leveloffset: +1

:description: This chapter describes each of the algorithms in the Neo4j Graph Data Science library, including algorithm tiers, execution modes and general syntax.  = Algorithms

[abstract] — This chapter describes each of the algorithms in the Neo4j Graph Data Science library, including algorithm tiers, execution modes and general syntax. —  The Neo4j Graph Data Science (GDS) library contains many graph algorithms. The algorithms are divided into categories which represent different problem classes. The categories are listed in this chapter.

Algorithms exist in one of three tiers of maturity:

* Production-quality  Indicates that the algorithm has been tested with regards to stability and scalability. Algorithms in this tier are prefixed with gds.<algorithm>. * Beta  Indicates that the algorithm is a candidate for the production-quality tier.  Algorithms in this tier are prefixed with gds.beta.<algorithm>. * Alpha  Indicates that the algorithm is experimental and might be changed or removed at any time.  Algorithms in this tier are prefixed with gds.alpha.<algorithm>.

This chapter is divided into the following sections:

* Syntax overview * Centrality * Community detection * Similarity * Path finding * Topological link prediction
* Node embeddings * Machine learning models * Auxiliary procedures * Pregel API

:leveloffset: 2

:leveloffset: +2

:description: This section describes the general syntax for running algorithms in the Neo4j Graph Data Science library, including execution modes and common configuration parameters.  = Syntax overview

[abstract] — This section describes the general syntax for running algorithms in the Neo4j Graph Data Science library, including execution modes and common configuration parameters. —  The general algorithm syntax comes in two variants:

* Named graph variant  The graph to operate over will be read from the graph catalog. * Anonymous graph variant  The graph to operate over will be created and deleted as part of the algorithm execution.

Each syntax variant additionally provides different execution modes. These are the supported execution modes:

* stream  Returns the result of the algorithm as a stream of records. * stats  Returns a single record of summary statistics, but does not write to the Neo4j database. * mutate  Writes the results of the algorithm to the in-memory graph and returns a single record of summary statistics. This mode is designed for the named graph variant, as its effects will be invisible on an anonymous graph. * write  Writes the results of the algorithm to the Neo4j database and returns a single record of summary statistics.

Finally, an execution mode may be estimated by appending the command with estimate.

[NOTE] Only the production-quality tier guarantees availability of all execution modes and estimation procedures.

Including all of the above mentioned elements leads to the following syntax outlines:

.Syntax composition for the named graph variant: [source] ---- CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>]( graphName: String, configuration: Map ) ----

.Syntax composition for the anonymous graph variant: [source] ---- CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>]( configuration: Map ) ----

The detailed sections in this chapter include concrete syntax overviews and examples.

:leveloffset: 2

:leveloffset: +2

:description: This chapter provides explanations and examples for each of the centrality algorithms in the
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Page Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | score | Float | PageRank score.

======

[.include-with-stats] ======

.Run PageRank in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.stats(graphName: String, configuration: Map) YIELD ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, centralityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Page Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the `centralityDistribution`. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-mutate] ======

.Run PageRank in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.mutate(graphName: String, configuration: Map) YIELD nodePropertiesWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, centralityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Page Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes |

The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the `centralityDistribution`. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | nodePropertiesWritten | Integer | The number of properties that were written to the in-memory graph. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ======

.Run PageRank in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.write(graphName: String, configuration: Map) YIELD nodePropertiesWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, centralityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Page Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the `centralityDistribution`. | writeMillis | Integer | Milliseconds for writing result data back. | nodePropertiesWritten | Integer | The number of properties that were written to Neo4j. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run PageRank in write mode on an anonymous graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.write(configuration: Map) YIELD nodePropertiesWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, centralityDistribution: Map, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Page Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the

algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: web network :image-file: page-rank-graph.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (home:Page {name:'Home'}), (about:Page {name:'About'}), (product:Page {name:'Product'}), (links:Page {name:'Links'}), (a:Page {name:'Site A'}), (b:Page {name:'Site B'}), (c:Page {name:'Site C'}), (d:Page {name:'Site D'}),

(home)-[:LINKS {weight: 0.2}]→(about), (home)-[:LINKS {weight: 0.2}]→(links), (home)-[:LINKS {weight: 0.6}]→(product), (about)-[:LINKS {weight: 1.0}]→(home), (product)-[:LINKS {weight: 1.0}]→(home), (a)-[:LINKS {weight: 1.0}]→(home), (b)-[:LINKS {weight: 1.0}]→(home), (c)-[:LINKS {weight: 1.0}]→(home), (d)-[:LINKS {weight: 1.0}]→(home), (links)-[:LINKS {weight: 0.8}]→(home), (links)-[:LINKS {weight: 0.05}]→(a), (links)-[:LINKS {weight: 0.05}]→(b), (links)-[:LINKS {weight: 0.05}]→(c), (links)-[:LINKS {weight: 0.05}]→(d); ----

This graph represents eight pages, linking to one another. Each relationship has a property called `weight`, which describes the importance of the relationship.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Page', 'LINKS', { relationshipProperties: 'weight' }) ----

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.write.estimate('myGraph', { writeProperty: 'pageRank', maxIterations: 20, dampingFactor: 0.85 }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header",cols="1,1,1,1,1"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 8 | 14 | 696 | 696 | "696 Bytes"

—

=== Stream

:stream-details: For example, we can order the results to find the nodes with the highest PageRank score. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm in `stream` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.stream('myGraph') YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 3.215681999884452 | "About" | 1.0542700552146722 | "Links" | 1.0542700552146722 | "Product" | 1.0542700552146722 | "Site A" | 0.3278578964488539 | "Site B" | 0.3278578964488539 | "Site C" | 0.3278578964488539 | "Site D" | 0.3278578964488539

— The above query is running the algorithm in `stream` mode as `unweighted` and the returned scores are not normalized. Below, one can find an example for weighted graphs. Another example shows the application of a scaler to normalize the final scores.

NOTE: While we are using the `stream` mode to illustrate running the algorithm as `weighted` or `unweighted`, all the algorithm modes support this configuration parameter.

=== Stats

:stats-details: For example PageRank stats returns centrality histogram which can be used to monitor the distribution of PageRank score values across all computed nodes. :stats-syntax: algorithms-page-rank-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm and returns the result in form of statistical and measurement values [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.stats('myGraph', { maxIterations: 20, dampingFactor: 0.85 }) YIELD centralityDistribution RETURN centralityDistribution.max AS max ----

.Results [opts="header",cols="1"]

| max | 3.2156810760498047

— The centrality histogram can be useful for inspecting the computed scores or perform normalizations.

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.mutate('myGraph', { maxIterations: 20, dampingFactor: 0.85, mutateProperty: 'pagerank' }) YIELD nodePropertiesWritten, ranIterations ----

.Results [opts="header",cols="1m,1m"]

| nodePropertiesWritten | ranIterations | 8 | 20

—

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.write('myGraph', { maxIterations: 20, dampingFactor: 0.85, writeProperty: 'pagerank' }) YIELD nodePropertiesWritten, ranIterations ----

.Results [opts="header",cols="1m,1m"]

| nodePropertiesWritten | ranIterations | 8 | 20

—

=== Weighted

By default, the algorithm is considering the relationships of the graph to be `unweighted`, to change this behaviour we can use configuration parameter called `relationshipWeightProperty`. In the `weighted` case, the previous score of a node send to its neighbors, is multiplied by the relationship weight and then divided by the sum of the weights of its outgoing relationships. If the value of the relationship property is negative it will be ignored during computation. Below is an example of running the algorithm using the relationship property.

[role=query-example] — .The following will run the algorithm in `stream` mode using relationship weights: [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.stream('myGraph', { maxIterations: 20, dampingFactor: 0.85, relationshipWeightProperty: 'weight' }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 3.53751028396339 | "Product" | 1.9357838291651097 | "About" |

0.7452612763883698 | "Links" | 0.7452612763883698 | "Site A" | 0.18152677135466103 | "Site B" | 0.18152677135466103 | "Site C" | 0.18152677135466103 | "Site D" | 0.18152677135466103

— NOTE: We are using `stream` mode to illustrate running the algorithm as `weighted` or `unweighted`, all the algorithm modes support this configuration parameter.

=== Tolerance

The `tolerance` configuration parameter denotes the minimum change in scores between iterations. If all scores change less than the configured `tolerance` value the result stabilises, and the algorithm returns.

[role=query-example] — .The following will run the algorithm in `stream` mode using bigger `tolerance` value: [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.stream('myGraph', { maxIterations: 20, dampingFactor: 0.85, tolerance: 0.1 }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 1.5812450669583336 | "About" | 0.5980194356381945 | "Links" | 0.5980194356381945 | "Product" | 0.5980194356381945 | "Site A" | 0.23374955154166668 | "Site B" | 0.23374955154166668 | "Site C" | 0.23374955154166668 | "Site D" | 0.23374955154166668

— In this example we are using `tolerance: 0.1`, so the results are a bit different compared to the ones from stream example which is using the default value of `tolerance`. Note that the nodes 'About', 'Link' and 'Product' now have the same score, while with the default value of `tolerance` the node 'Product' has higher score than the other two.

=== Damping Factor

The damping factor configuration parameter accepts values between 0 (inclusive) and 1 (exclusive). If its value is too high then problems of sinks and spider traps may occur, and the values may oscillate so that the algorithm does not converge. If it's too low then all scores are pushed towards 1, and the result will not sufficiently reflect the structure of the graph.

[role=query-example] — .The following will run the algorithm in `stream` mode using smaller `dampingFactor` value: [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.stream('myGraph', { maxIterations: 20, dampingFactor: 0.05 }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 1.2487309425844906 | "About" | 0.9708121818724536 | "Links" | 0.9708121818724536 | "Product" | 0.9708121818724536 | "Site A" | 0.9597081216238426 | "Site B" | 0.9597081216238426 | "Site C" | 0.9597081216238426 | "Site D" | 0.9597081216238426

— Compared to the results from the stream example which is using the default value of `dampingFactor` the score values are closer to each other when using `dampingFactor: 0.05`. Also, note that the nodes 'About', 'Link' and 'Product' now have the same score, while with the default value of `dampingFactor` the node 'Product' has higher score than the other two.

=== Personalised PageRank

Personalized PageRank is a variation of PageRank which is biased towards a set of `sourceNodes`. This variant of PageRank is often used as part of recommender systems.

The following examples show how to run PageRank centered around 'Site A'.

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- MATCH (siteA:Page {name: 'Site A'}) CALL gds.pageRank.stream('myGraph', { maxIterations: 20, dampingFactor: 0.85, sourceNodes: [siteA] }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.39902290442518784 | "Site A" | 0.16890325301726694 | "About" | 0.11220151747374331 | "Links" | 0.11220151747374331 | "Product" | 0.11220151747374331 | "Site B" | 0.01890325301726691 | "Site C" | 0.01890325301726691 | "Site D" | 0.01890325301726691

— Comparing these results to the ones from the stream example (which is not using `sourceNodes` configuration parameter) shows that the 'Site A' node that we used in the `sourceNodes` list now scores second instead of fourth.

=== Scaling centrality scores

To normalize the final scores as part of the algorithm execution, one can use the `scaler` configuration parameter. A common scaler is the `L1Norm`, which normalizes each score to a value between 0 and 1. A description of all available scalers can be found in the documentation for the `scaleProperties` procedure.

[role=query-example] — .The following will run the algorithm in `stream` mode and returns normalized results: [source, cypher, role=noplay, indent=0] ---- CALL gds.pageRank.stream('myGraph', { scaler: "L1Norm" }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.4181682554824872 | "About" | 0.1370975954128506 | "Links" | 0.1370975954128506 | "Product" | 0.1370975954128506 | "Site A" | 0.04263473956974027 | "Site B" | 0.04263473956974027 | "Site C" | 0.04263473956974027 | "Site D" | 0.04263473956974027

— Comparing the results with the stream example, we can see that the relative order of scores is the same.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the {algorithm} algorithm in the Neo4j Graph Data Science library. = Article Rank :entity: node :result: score :algorithm: Article Rank

[abstract] — This section describes the {algorithm} algorithm in the Neo4j Graph Data Science library. — :directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

ArticleRank is a variant of the Page Rank algorithm, which measures the **transitive** influence of nodes.

Page Rank follows the assumption that relationships originating from low-degree nodes have a higher influence than relationships from high-degree nodes. Article Rank lowers the influence of low-degree nodes by lowering the scores being sent to their neighbors in each iteration.

The Article Rank of a node v at iteration *i* is defined as:

image::equations/articleRank.svg[]

where,

* $N_{in}(v)$ denotes incoming neighbors and $N_{out}(v)$ denotes outgoing neighbors of node *v*. * *d* is a damping factor in *[0, 1]*. * N_{out} is the average out-degree

For more information, see ArticleRank: a PageRank-based alternative to numbers of citations for analysing citation networks.

== Considerations

There are some things to be aware of when using the Article Rank algorithm:

* If there are no relationships from within a group of pages to outside the group, then the group is considered a spider trap. * Rank sink can occur when a network of pages is forming an infinite cycle. * Dead-ends occur when pages have no outgoing relationship.

Changing the damping factor can help with all the considerations above. It can be interpreted as a probability of a web surfer to sometimes jump to a random page and therefore not getting stuck in sinks.

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

.Article Rank syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ======

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Article Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | score | Float | Eigenvector score.

======

[.include-with-stats] ======

.Run Article Rank in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.stats(graphName: String, configuration: Map) YIELD ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, centralityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Article Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the `centralityDistribution`. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-mutate] ======

.Run Article Rank in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.mutate(graphName: String, configuration: Map) YIELD nodePropertiesWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, centralityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Article Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes |

The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the `centralityDistribution`. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | nodePropertiesWritten | Integer | The number of properties that were written to the in-memory graph. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ======

.Run Article Rank in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.write(graphName: String, configuration: Map) YIELD nodePropertiesWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, centralityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Article Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the `centralityDistribution`. | writeMillis | Integer | Milliseconds for writing result data back. | nodePropertiesWritten | Integer | The number of properties that were written to Neo4j. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run Article Rank in write mode on an anonymous graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.write(configuration: Map) YIELD nodePropertiesWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, centralityDistribution: Map, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | dampingFactor | Float | 0.85 | yes | The damping factor of the Page Rank calculation. Must be in [0, 1). | maxIterations | Integer | 20 | yes | The maximum number of iterations of Article Rank to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the

algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: web network :image-file: page-rank-graph.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (home:Page {name:'Home'}), (about:Page {name:'About'}), (product:Page {name:'Product'}), (links:Page {name:'Links'}), (a:Page {name:'Site A'}), (b:Page {name:'Site B'}), (c:Page {name:'Site C'}), (d:Page {name:'Site D'}),

(home)-[:LINKS {weight: 0.2}]→(about), (home)-[:LINKS {weight: 0.2}]→(links), (home)-[:LINKS {weight: 0.6}]→(product), (about)-[:LINKS {weight: 1.0}]→(home), (product)-[:LINKS {weight: 1.0}]→(home), (a)-[:LINKS {weight: 1.0}]→(home), (b)-[:LINKS {weight: 1.0}]→(home), (c)-[:LINKS {weight: 1.0}]→(home), (d)-[:LINKS {weight: 1.0}]→(home), (links)-[:LINKS {weight: 0.8}]→(home), (links)-[:LINKS {weight: 0.05}]→(a), (links)-[:LINKS {weight: 0.05}]→(b), (links)-[:LINKS {weight: 0.05}]→(c), (links)-[:LINKS {weight: 0.05}]→(d); ----

This graph represents eight pages, linking to one another. Each relationship has a property called `weight`, which describes the importance of the relationship.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Page', 'LINKS', { relationshipProperties: 'weight' }) ----

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.write.estimate('myGraph', { writeProperty: 'centrality', maxIterations: 20 }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header",cols="1,1,1,1,1"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 8 | 14 | 696 | 696 | "696 Bytes"

—

=== Stream

:stream-details: For example, we can order the results to find the nodes with the highest Eigenvector score. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm in `stream` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.stream('myGraph') YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.5607071761939444 | "About" | 0.250337073634706 | "Links" | 0.250337073634706 | "Product" | 0.250337073634706 | "Site A" | 0.18152391630760797 | "Site B" | 0.18152391630760797 | "Site C" | 0.18152391630760797 | "Site D" | 0.18152391630760797

— The above query is running the algorithm in `stream` mode as `unweighted`. Below, one can find an example for weighted graphs.

=== Stats

:stats-details: For example Eigenvector stats returns centrality histogram which can be used to monitor the distribution of centrality scores across all computed nodes. :stats-syntax: algorithms-article-rank-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm and return statistics about the centrality scores. [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.stats('myGraph') YIELD centralityDistribution RETURN centralityDistribution.max AS max ----

.Results [opts="header",cols="1"]

| max | 0.5607099533081055

—

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.mutate('myGraph', { mutateProperty: 'centrality' }) YIELD nodePropertiesWritten, ranIterations ----

.Results [opts="header",cols="1m,1m"]

| nodePropertiesWritten | ranIterations | 8 | 19

—

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.write('myGraph', { writeProperty: 'centrality' }) YIELD nodePropertiesWritten, ranIterations ----

.Results [opts="header",cols="1m,1m"]

| nodePropertiesWritten | ranIterations | 8 | 19

—

=== Weighted

By default, the algorithm considers the relationships of the graph to be unweighted. To change this behaviour, we can use the `relationshipWeightProperty` configuration parameter. If the parameter is set, the associated property value is used as relationship weight. In the `weighted` case, the previous score of a node sent to its neighbors is multiplied by the normalized relationship weight. Note, that negative relationship weights are ignored during the computation.

In the following example, we use the `weight` property of the input graph as relationship weight property.

[role=query-example] — .The following will run the algorithm in `stream` mode using relationship weights: [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.stream('myGraph', { relationshipWeightProperty: 'weight' }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.5160810726222141 | "Product" | 0.24570958074084706 | "About" | 0.18190319358024 | "Links" | 0.18190319358024 | "Site A" | 0.15281123078335393 | "Site B" | 0.15281123078335393 | "Site C" | 0.15281123078335393 | "Site D" | 0.15281123078335393

— As in the unweighted example, the "Home" node has the highest score. In contrast, the "Product" now has the second highest instead of the fourth highest score.

NOTE: We are using `stream` mode to illustrate running the algorithm as `weighted`, however, all the algorithm modes support the `relationshipWeightProperty` configuration parameter.

=== Tolerance

The `tolerance` configuration parameter denotes the minimum change in scores between iterations. If all scores change less than the configured tolerance, the iteration is aborted and considered converged. Note, that setting a higher tolerance leads to earlier convergence, but also to less accurate centrality scores.

[role=query-example] — .The following will run the algorithm in `stream` mode using a high `tolerance` value: [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.stream('myGraph', { tolerance: 0.1 }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.4470707070707072 | "About" | 0.23000212652844235 | "Links" | 0.23000212652844235 | "Product" | 0.23000212652844235 | "Site A" | 0.16888888888888892 | "Site B" | 0.16888888888888892 | "Site C" | 0.16888888888888892 | "Site D" | 0.16888888888888892

— We are using `tolerance: 0.1`, which leads to slightly different results compared to the stream example. However, the computation converges after four iterations, and we can already observe a trend in the resulting scores. === Personalised Article Rank Personalized Article Rank is a variation of Article Rank which is biased towards a set of `sourceNodes`. By default, the power iteration starts with the same value for all nodes: `1 /	V	`. For a given set of source nodes `S`, the initial value of each source node is set to `1 /	S	` and to `0` for all remaining nodes. The following examples show how to run Eigenvector centrality centered around 'Site A' and 'Site B'. [role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- MATCH (siteA:Page {name: 'Site A'}), (siteB:Page {name: 'Site B'}) CALL gds.articleRank.stream('myGraph', { maxIterations: 20, sourceNodes: [siteA, siteB] }) YIELD nodeId, score RETURN gds.util.asNode(nodeId) .name AS name, score ORDER BY score DESC, name ASC ---- .Results [opts="header",cols="1 ,1"]

| name | score | "Site A" | 0.15249052775314756 | "Site B" | 0.15249052775314756 | "Home" | 0.1105231342997017 | "About" | 0.019777824032578193 | "Links" | 0.019777824032578193 | "Product" | 0.019777824032578193 | "Site C" | 0.002490527753147571 | "Site D" | 0.002490527753147571

— Comparing these results to the ones from the stream example (which is not using `sourceNodes` configuration parameter) shows the 'Site A' and `Site B` nodes we used in the `sourceNodes` list now score second and third instead of fourth and fifth.

=== Scaling centrality scores

To normalize the final scores as part of the algorithm execution, one can use the `scaler` configuration parameter. A common scaler is the `L1Norm`, which normalizes each score to a value between 0 and 1. A description of all available scalers can be found in the documentation for the `scaleProperties` procedure.

[role=query-example] — .The following will run the algorithm in `stream` mode and returns normalized results: [source, cypher, role=noplay, indent=0] ---- CALL gds.articleRank.stream('myGraph', { scaler: "L1Norm" }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.275151294006312 | "About" | 0.12284588582564794 | "Links" | 0.12284588582564794 | "Product" | 0.12284588582564794 | "Site A" | 0.08907776212918608 | "Site B" | 0.08907776212918608 | "Site C" | 0.08907776212918608 | "Site D" | 0.08907776212918608

— Comparing the results with the stream example, we can see that the relative order of scores is the same.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the {algorithm} algorithm in the Neo4j Graph Data Science library. = Eigenvector Centrality :entity: node :result: score :algorithm: Eigenvector Centrality

[abstract] — This section describes the {algorithm} algorithm in the Neo4j Graph Data Science library. — :directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

Eigenvector Centrality is an algorithm that measures the **transitive** influence of nodes. Relationships originating from high-scoring nodes contribute more to the score of a node than connections from low-scoring nodes. A high eigenvector score means that a node is connected to many nodes who themselves have high scores.

The algorithm computes the eigenvector associated with the largest absolute eigenvalue. To compute that eigenvalue, the algorithm applies the power iteration approach. Within each iteration, the centrality score for each node is derived from the scores of its incoming neighbors. In the power iteration method, the eigenvector is L2-normalized after each iteration, leading to normalized results by default.

The PageRank algorithm is a variant of Eigenvector Centrality with an additional jump probability.

== Considerations

There are some things to be aware of when using the Eigenvector centrality algorithm:

* Centrality scores for nodes with no incoming relationships will converge to `0`. * Due to missing degree normalization, high-degree nodes have a very strong influence on their neighbors' score.

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

.Eigenvector Centrality syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ======

.Run Eigenvector Centrality in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, score: Float ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | 20 | yes | The maximum number of iterations of Eigenvector Centrality to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | score | Float | Eigenvector score.

======

[.include-with-stats] ======

.Run Eigenvector Centrality in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.stats(graphName: String, configuration: Map) YIELD ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, centralityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | 20 | yes | The maximum number of iterations of Eigenvector Centrality to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String |

None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the `centralityDistribution`. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-mutate] ======

.Run Eigenvector Centrality in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.mutate(graphName: String, configuration: Map) YIELD nodePropertiesWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, centralityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | 20 | yes | The maximum number of iterations of Eigenvector Centrality to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer |

Milliseconds for computing the `centralityDistribution`. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | nodePropertiesWritten | Integer | The number of properties that were written to the in-memory graph. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | configuration | Map | The configuration used for running the algorithm.

```
======

[.include-with-write] ======

.Run Eigenvector Centrality in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.write( graphName: String, configuration: Map ) YIELD nodePropertiesWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, centralityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | maxIterations | Integer | 20 | yes | The maximum number of iterations of Eigenvector Centrality to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

```
.Results [opts="header",cols="1,1,6"]
```

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the `centralityDistribution`. | writeMillis | Integer | Milliseconds for writing result data back. | nodePropertiesWritten | Integer | The number of properties that were written to Neo4j. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | configuration | Map | The configuration used for running the

algorithm.

```
======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the write mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run Eigenvector Centrality in write mode on an anonymous graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.write( configuration: Map ) YIELD nodePropertiesWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, centralityDistribution: Map, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | maxIterations | Integer | 20 | yes | The maximum number of iterations of Eigenvector Centrality to run. | tolerance | Float | 0.0000001 | yes | Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | sourceNodes | List or Node or Number | [] | yes | The nodes or node ids to use for computing Personalized Page Rank. | scaler | String | None | yes | The name of the scaler applied for the final scores. Supported values are `None`, `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: web network :image-file: page-rank-graph.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (home:Page {name:'Home'}), (about:Page {name:'About'}), (product:Page {name:'Product'}), (links:Page {name:'Links'}), (a:Page {name:'Site A'}), (b:Page {name:'Site B'}), (c:Page {name:'Site C'}), (d:Page {name:'Site D'}),

(home)-[:LINKS {weight: 0.2}]→(about), (home)-[:LINKS {weight: 0.2}]→(links), (home)-[:LINKS {weight: 0.6}]→(product), (about)-[:LINKS {weight: 1.0}]→(home), (product)-[:LINKS {weight: 1.0}]→(home), (a)-[:LINKS {weight: 1.0}]→(home), (b)-[:LINKS {weight: 1.0}]→(home), (c)-[:LINKS {weight: 1.0}]→(home), (d)-[:LINKS {weight: 1.0}]→(home), (links)-[:LINKS {weight: 0.8}]→(home), (links)-[:LINKS {weight: 0.05}]→(a), (links)-[:LINKS {weight: 0.05}]→(b), (links)-[:LINKS {weight: 0.05}]→(c), (links)-[:LINKS {weight: 0.05}]→(d); ----

This graph represents eight pages, linking to one another. Each relationship has a property called `weight`, which describes the importance of the relationship.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Page', 'LINKS', { relationshipProperties: 'weight' }) ----

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.write.estimate('myGraph', { writeProperty: 'centrality', maxIterations: 20 }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header",cols="1,1,1,1,1"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 8 | 14 | 696 | 696 | "696 Bytes"

—

=== Stream

:stream-details: For example, we can order the results to find the nodes with the highest Eigenvector score. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm in `stream` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.stream('myGraph') YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.7465574981728249 | "About" | 0.33997520529777137 | "Links" | 0.33997520529777137 | "Product" | 0.33997520529777137 | "Site A" | 0.15484062876886298 | "Site B" | 0.15484062876886298 | "Site C" | 0.15484062876886298 | "Site D" | 0.15484062876886298

— The above query is running the algorithm in `stream` mode as `unweighted`. Below, one can find an example for weighted graphs.

=== Stats

:stats-details: For example Eigenvector stats returns centrality histogram which can be used to monitor the distribution of centrality scores across all computed nodes. :stats-syntax: algorithms-eigenvector-centrality-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm and return statistics about the centrality scores. [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.stats('myGraph', { maxIterations: 20 }) YIELD centralityDistribution RETURN centralityDistribution.max AS max ----

.Results [opts="header",cols="1"]

| max | 0.7465581893920898

—

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.mutate('myGraph', { maxIterations: 20, mutateProperty: 'centrality' }) YIELD nodePropertiesWritten, ranIterations ----

.Results [opts="header",cols="1m,1m"]

| nodePropertiesWritten | ranIterations | 8 | 20

—

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.write('myGraph', { maxIterations: 20, writeProperty: 'centrality' }) YIELD nodePropertiesWritten, ranIterations ----

.Results [opts="header",cols="1m,1m"]

| nodePropertiesWritten | ranIterations | 8 | 20

—

=== Weighted

By default, the algorithm considers the relationships of the graph to be unweighted. To change this behaviour, we can use the `relationshipWeightProperty` configuration parameter. If the parameter is set, the associated property value is used as relationship weight. In the `weighted` case, the previous score of a node sent to its neighbors is multiplied by the normalized relationship weight. Note, that negative relationship weights are ignored during the computation.

In the following example, we use the `weight` property of the input graph as relationship weight property.

[role=query-example] — .The following will run the algorithm in `stream` mode using relationship weights: [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.stream('myGraph', { maxIterations: 20, relationshipWeightProperty: 'weight' }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.8328163407319487 | "Product" | 0.5004775834976313 | "About" |

0.1668258611658771 | "Links" | 0.1668258611658771 | "Site A" | 0.008327591469710233 | "Site B" | 0.008327591469710233 | "Site C" | 0.008327591469710233 | "Site D" | 0.008327591469710233

— As in the unweighted example, the "Home" node has the highest score. In contrast, the "Product" now has the second highest instead of the fourth highest score.

NOTE: We are using `stream` mode to illustrate running the algorithm as `weighted`, however, all the algorithm modes support the `relationshipWeightProperty` configuration parameter.

=== Tolerance

The `tolerance` configuration parameter denotes the minimum change in scores between iterations. If all scores change less than the configured tolerance, the iteration is aborted and considered converged. Note, that setting a higher tolerance leads to earlier convergence, but also to less accurate centrality scores.

[role=query-example] — .The following will run the algorithm in `stream` mode using a high `tolerance` value: [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.stream('myGraph', { maxIterations: 20, tolerance: 0.1 }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ----

.Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.7108273818583551 | "About" | 0.3719400001993262 | "Links" | 0.3719400001993262 | "Product" | 0.3719400001993262 | "Site A" | 0.14116155811301126 | "Site B" | 0.14116155811301126 | "Site C" | 0.14116155811301126 | "Site D" | 0.14116155811301126

— We are using `tolerance: 0.1`, which leads to slightly different results compared to the stream example. However, the computation converges after three iterations, and we can already observe a trend in the resulting scores. === Personalised Eigenvector Centrality Personalized Eigenvector Centrality is a variation of Eigenvector Centrality which is biased towards a set of `sourceNodes`. By default, the power iteration starts with the same value for all nodes: `1 /	V	`. For a given set of source nodes `S`, the initial value of each source node is set to `1 /	S	` and to `0` for all remaining nodes. The following examples show how to run Eigenvector centrality centered around 'Site A'. [role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- MATCH (siteA:Page {name: 'Site A'}), (siteB:Page {name: 'Site B'}) CALL gds.eigenvector.stream('myGraph', { maxIterations: 20, sourceNodes: [siteA, siteB] }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ---- .Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.7465645391567868 | "About" | 0.33997203172449453 | "Links" | 0.33997203172449453 | "Product" | 0.33997203172449453 | "Site A" | 0.15483736775159632 | "Site B" | 0.15483736775159632 | "Site C" | 0.15483736775159632 | "Site D" | 0.15483736775159632

— === Scaling centrality scores Internally, centrality scores are scaled after each iteration using L2 normalization. As a consequence, the final values are already normalized. This behavior cannot be changed as it is part of the power iteration method. However, to normalize the final scores as part of the algorithm execution, one can use the `scaler` configuration parameter. A common scaler is the `L1Norm`, which normalizes each score to a value between 0 and 1. A description of all available scalers can be found in the documentation for the `scaleProperties` procedure. [role=query-example] — .The following will run the algorithm in `stream` mode and returns normalized results: [source, cypher, role=noplay, indent=0] ---- CALL gds.eigenvector.stream('myGraph', { scaler: "L1Norm" }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC ---- .Results [opts="header",cols="1,1"]

| name | score | "Home" | 0.31291106560043064 | "About" | 0.1424967320371402 | "Links" |

0.1424967320371402 | "Product" | 0.1424967320371402 | "Site A" | 0.06489968457203725 | "Site B" | 0.06489968457203725 | "Site C" | 0.06489968457203725 | "Site D" | 0.06489968457203725

— Comparing the results with the stream example, we can see that the relative order of scores is the same.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Betweenness Centrality algorithm in the Neo4j Graph Data Science library. = Betweenness Centrality :entity: node :result: centrality :algorithm: Betweenness Centrality

[abstract] — This section describes the Betweenness Centrality algorithm in the Neo4j Graph Data Science library. — :directed: :undirected: :homogeneous: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

Betweenness centrality is a way of detecting the amount of influence a node has over the flow of information in a graph. It is often used to find nodes that serve as a bridge from one part of a graph to another.

The algorithm calculates unweighted shortest paths between all pairs of nodes in a graph. Each node receives a score, based on the number of shortest paths that pass through the node. Nodes that more frequently lie on shortest paths between other nodes will have higher betweenness centrality scores.

The GDS implementation is based on Brandes' approximate algorithm for unweighted graphs. The implementation requires $O(n + m)$ space and runs in $O(n * m)$ time, where n is the number of nodes and m the number of relationships in the graph.

For more information on this algorithm, see:

* A Faster Algorithm for Betweenness Centrality * Centrality Estimation in Large Networks * A Set of Measures of Centrality Based on Betweenness

[NOTE] ==== Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read Memory Estimation. ====

== Considerations and sampling

The Betweenness Centrality algorithm can be very resource-intensive to compute. Brandes' approximate algorithm computes single-source shortest paths (SSSP) for a set of source nodes. When all nodes are selected as source nodes, the algorithm produces an exact result. However, for large graphs this can potentially lead to very long runtimes. Thus, approximating the results by computing the SSSPs for only a subset of nodes can be useful. In GDS we refer to this technique as *sampling*, where the size of the source node set is the *sampling size*.

There are two things to consider when executing the algorithm on large graphs:

* A higher parallelism leads to higher memory consumption as each thread executes SSSPs for a subset of source nodes sequentially. ** In the worst case, a single SSSP requires the whole graph to be duplicated in memory. * A higher sampling size leads to more accurate results, but also to a potentially much longer execution time.

Changing the values of the configuration parameters `concurrency` and `samplingSize`, respectively, can help to manage these considerations.

=== Sampling strategies

Brandes defines several strategies for selecting source nodes. The GDS implementation is based on the random degree

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | samplingSize | Integer | node count | yes | The number of source nodes to consider for computing centrality scores. | samplingSeed | Integer | null | yes | The seed value for the random number generator that selects start nodes.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | score | Float | Betweenness Centrality score.

======

[.include-with-stats] ====== .Run Betweenness Centrality in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.stats(graphName: String, configuration: Map) YIELD centralityDistribution: Map, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | samplingSize | Integer | node count | yes | The number of source nodes to consider for computing centrality scores. | samplingSeed | Integer | null | yes | The seed value for the random number generator that selects start nodes.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | createMillis | Integer | Milliseconds for

creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the statistics. | configuration | Map | Configuration used for running the algorithm.

```
======

[.include-with-mutate] ====== .Run Betweenness Centrality in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.mutate( graphName: String, configuration: Map ) YIELD centralityDistribution: Map, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | samplingSize | Integer | node count | yes | The number of source nodes to consider for computing centrality scores. | samplingSeed | Integer | null | yes | The seed value for the random number generator that selects start nodes.

```
.Results [opts="header",cols="1,1,6"]
```

| Name | Type | Description | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the statistics. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | nodePropertiesWritten | Integer | Number of properties added to the in-memory graph. | configuration | Map | Configuration used for running the algorithm.

```
======

[.include-with-write] ====== .Run Betweenness Centrality in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.write( graphName: String, configuration: Map ) YIELD centralityDistribution: Map, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ["] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ["] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | samplingSize | Integer | node count | yes | The number of source nodes to consider for computing centrality scores. | samplingSeed | Integer | null | yes | The seed value for the random number generator that selects start nodes.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the statistics. | writeMillis | Integer | Milliseconds for writing result data back. | nodePropertiesWritten | Integer | Number of properties written to Neo4j. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run Betweenness Centrality in write mode on an anonymous graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.write(configuration: Map) YIELD centralityDistribution: Map, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. |

nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | samplingSize | Integer | node count | yes | The number of source nodes to consider for computing centrality scores. | samplingSeed | Integer | null | yes | The seed value for the random number generator that selects start nodes.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: betweenness_centrality.png In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:User {name: 'Alice'}), (bob:User {name: 'Bob'}), (carol:User {name: 'Carol'}), (dan:User {name: 'Dan'}), (eve:User {name: 'Eve'}), (frank:User {name: 'Frank'}), (gale:User {name: 'Gale'}),

(alice)-[:FOLLOWS]→(carol), (bob)-[:FOLLOWS]→(carol), (carol)-[:FOLLOWS]→(dan), (carol)-[:FOLLOWS]→(eve), (dan)-[:FOLLOWS]→(frank), (eve)-[:FOLLOWS]→(frank), (frank)-[:FOLLOWS]→(gale); ----

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `User` nodes and the `FOLLOWS` relationships.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'User', 'FOLLOWS') ----

In the following examples we will demonstrate using the Betweenness Centrality algorithm on this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.write.estimate('myGraph', { writeProperty: 'betweenness' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 7 | 7 | 2912 | 2912 | "2912 Bytes"

— As is discussed in Considerations and sampling we can configure the memory requirements using the `concurrency` configuration parameter.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.write.estimate('myGraph', { writeProperty: 'betweenness', concurrency: 1 }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 7 | 7 | 848 | 848 | "848 Bytes"

— Here we can note that the estimated memory requirements were lower than when running with the default concurrency setting. Similarly, using a higher value will increase the estimated memory requirements.

=== Stream

:stream-details: For example, we can order the results to find the nodes with the highest betweenness centrality. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm in `stream` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.stream('myGraph') YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY name ASC ----

.Results [opts="header"]

| name | score | "Alice" | 0.0 | "Bob" | 0.0 | "Carol" | 8.0 | "Dan" | 3.0 | "Eve" | 3.0 | "Frank" | 5.0 | "Gale" | 0.0

— We note that the 'Carol' node has the highest score, followed by the 'Frank' node. Studying the example graph we can see that these nodes are in bottleneck positions in the graph. The 'Carol' node connects the 'Alice' and 'Bob' nodes to all other nodes, which increases its score. In particular, the shortest path from 'Alice' or 'Bob' to any other reachable node passes through 'Carol'. Similarly, all shortest paths that lead to the 'Gale' node passes through the 'Frank' node. Since 'Gale' is reachable from each other node, this causes the score for 'Frank' to be high.

Conversely, there are no shortest paths that pass through either of the nodes 'Alice', 'Bob' or 'Gale' which causes their betweenness centrality score to be zero.

=== Stats

:stats-details: In particular, Betweenness Centrality returns the minimum, maximum and sum of all centrality scores. :stats-syntax: algorithms-betweenness-centrality-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm in `stats` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.stats('myGraph') YIELD centralityDistribution RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore ----

.Results [opts="header"]

| minimumScore | meanScore | 0.0 | 2.714292253766741

— Comparing this to the results we saw in the stream example, we can find our minimum and maximum values from the table. It is worth noting that unless the graph has a particular shape involving a directed cycle, the minimum score will almost always be zero.

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.mutate('myGraph', { mutateProperty: 'betweenness' }) YIELD centralityDistribution, nodePropertiesWritten RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore, nodePropertiesWritten ----

.Results [opts="header"]

| minimumScore | meanScore | nodePropertiesWritten | 0.0 | 2.714292253766741 | 7

— The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `betweenness` which stores the betweenness centrality score for each node. To find out how to inspect the new schema of the in-memory graph, see Listing graphs.

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.write('myGraph', { writeProperty: 'betweenness' }) YIELD centralityDistribution, nodePropertiesWritten RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore, nodePropertiesWritten ----

.Results [opts="header"]

| minimumScore | meanScore | nodePropertiesWritten | 0.0 | 2.714292253766741 | 7

— The returned result is the same as in the `stats` example. Additionally, each of the seven nodes now has a new property `betweenness` in the Neo4j database, containing the betweenness centrality score for that node.

=== Sampling

Betweenness Centrality can be very resource-intensive to compute. To help with this, it is possible to approximate the results using a sampling technique. The configuration parameters `samplingSize` and `samplingSeed` are used to control the sampling. We illustrate this on our example graph by approximating Betweenness Centrality with a sampling size of two. The seed value is an arbitrary integer, where using the same value will yield the same results between different runs of the procedure.

[role=query-example] — .The following will run the algorithm in `stream` mode with a sampling size of two: [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.stream('myGraph', {samplingSize: 2, samplingSeed: 0}) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY name ASC ----

.Results [opts="header"]

| name | score | "Alice" | 0.0 | "Bob" | 0.0 | "Carol" | 4.0 | "Dan" | 2.0 | "Eve" | 2.0 | "Frank" | 2.0 | "Gale" | 0.0

— Here we can see that the 'Carol' node has the highest score, followed by a three-way tie between the 'Dan', 'Eve', and 'Frank' nodes. We are only sampling from two nodes, where the probability of a node being picked for the sampling is proportional to its outgoing degree. The 'Carol' node has the maximum degree and is the most likely to be picked. The 'Gale' node has an outgoing degree of zero and is very unlikely to be picked. The other nodes all have the same probability to be picked.

With our selected sampling seed of 0, we seem to have selected either of the 'Alice' and 'Bob' nodes, as well as the 'Carol' node. We can see that because either of 'Alice' and 'Bob' would add four to the score of the 'Carol' node, and each of 'Alice', 'Bob', and 'Carol' adds one to all of 'Dan', 'Eve', and 'Frank'.

To increase the accuracy of our approximation, the sampling size could be increased. In fact, setting the `samplingSize` to the node count of the graph (seven, in our case) will produce exact results.

=== Undirected

Betweenness Centrality can also be run on undirected graphs. To illustrate this, we will project our example graph using the `UNDIRECTED` orientation.

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myUndirectedGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myUndirectedGraph', 'User', {FOLLOWS: {orientation: 'UNDIRECTED'}}) ----

Now we can run Betweenness Centrality on our undirected graph. The algorithm automatically figures out that the graph is undirected.

[WARNING] Running the algorithm on an undirected graph is about twice as computationally intensive compared to a directed graph.

[role=query-example] — .The following will run the algorithm in `stream` mode on the undirected graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.betweenness.stream('myUndirectedGraph') YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY name ASC ----

.Results [opts="header"]

| name | score | "Alice" | 0.0 | "Bob" | 0.0 | "Carol" | 9.5 | "Dan" | 3.0 | "Eve" | 3.0 | "Frank" | 5.5 | "Gale" | 0.0

— The central nodes now have slightly higher scores, due to the fact that there are more shortest paths in the graph, and these are more likely to pass through the central nodes. The 'Dan' and 'Eve' nodes retain the same centrality scores as in the directed case.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Degree Centrality algorithm in the Neo4j Graph Data Science library. = Degree Centrality :entity: node :result: degree centrality :algorithm: Degree Centrality

[abstract] — This section describes the Degree Centrality algorithm in the Neo4j Graph Data Science library. — :directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

The Degree Centrality algorithm can be used to find popular nodes within a graph. Degree centrality measures the number of incoming or outgoing (or both) relationships from a node, depending on the orientation of a relationship projection. For more information on relationship orientations, see the relationship projection syntax section. It can be applied to either weighted or unweighted graphs. In the weighted case the algorithm computes the sum of all positive weights of adjacent relationships of a node, for each node in the graph. Non-positive weights are ignored.

For more information on this algorithm, see:

* Linton C. Freeman: Centrality in Social Networks Conceptual Clarification, 1979.

== Use-cases

The Degree Centrality algorithm has been shown to be useful in many different applications. For example:

* Degree centrality is an important component of any attempt to determine the most important people in a social network. For example, in BrandWatch's most influential men and women on Twitter 2017 the top 5 people in each category have over 40m followers each, which is a lot higher than the average degree.

* Weighted degree centrality has been used to help separate fraudsters from legitimate users of an online auction. The weighted centrality for fraudsters is significantly higher because they tend to collude with each other to artificially increase the price of items. Read more in Two Step graph-based semi-supervised Learning for Online Auction Fraud Detection

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

.Degree Centrality syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ====== .Run Degree Centrality in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, score: Float ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored

in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | orientation | String | NATURAL | yes | The orientation used to compute node degrees. Supported orientations are `NATURAL`, `REVERSE` and `UNDIRECTED`. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | score | Float | Degree Centrality score.

======

[.include-with-stats] ====== .Run Degree Centrality in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.stats(graphName: String, configuration: Map) YIELD centralityDistribution: Map, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | orientation | String | NATURAL | yes | The orientation used to compute node degrees. Supported orientations are `NATURAL`, `REVERSE` and `UNDIRECTED`. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | centralityDistribution | Map | Map containing min, max, mean as well as p50,

p75, p90, p95, p99 and p999 percentile values of centrality values. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the statistics. | configuration | Map | Configuration used for running the algorithm.

```
======

[.include-with-mutate] ====== .Run Degree Centrality in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.mutate( graphName: String, configuration: Map ) YIELD centralityDistribution: Map, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['**'] | yes | **Filter the named graph using the given node labels. |
relationshipTypes | **List of String | ['**'] | yes | Filter the named graph using the given relationship types. |
concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | orientation | String | NATURAL | yes | The orientation used to compute node degrees. Supported orientations are `NATURAL`, `REVERSE` and `UNDIRECTED`. |
relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

```
.Results [opts="header",cols="1,1,6"]
```

| Name | Type | Description | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the statistics. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | nodePropertiesWritten | Integer | Number of properties added to the in-memory graph. | configuration | Map | Configuration used for running the algorithm.

```
======

[.include-with-write] ====== .Run Degree Centrality in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.write( graphName: String, configuration: Map ) YIELD centralityDistribution: Map, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored

in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | orientation | String | NATURAL | yes | The orientation used to compute node degrees. Supported orientations are `NATURAL`, `REVERSE` and `UNDIRECTED`. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the statistics. | writeMillis | Integer | Milliseconds for writing result data back. | nodePropertiesWritten | Integer | Number of properties written to Neo4j. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run Degree Centrality in write mode on an anonymous graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.write(configuration: Map) YIELD centralityDistribution: Map, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for

anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | orientation | String | NATURAL | yes | The orientation used to compute node degrees. Supported orientations are `NATURAL`, `REVERSE` and `UNDIRECTED`. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: degree-centrality-graph.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:User {name: 'Alice'}), (bridget:User {name: 'Bridget'}), (charles:User {name: 'Charles'}), (doug:User {name: 'Doug'}), (mark:User {name: 'Mark'}), (michael:User {name: 'Michael'}),

(alice)-[:FOLLOWS {score: 1}]→(doug), (alice)-[:FOLLOWS {score: -2}]→(bridget), (alice)-[:FOLLOWS {score: 5}]→(charles), (mark)-[:FOLLOWS {score: 1.5}]→(doug), (mark)-[:FOLLOWS {score: 4.5}]→(michael), (bridget)-[:FOLLOWS {score: 1.5}] →(doug), (charles)-[:FOLLOWS {score: 2}]→(doug), (michael)-[:FOLLOWS {score: 1.5}]→(doug) ----

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `User` nodes and the `FOLLOWS` relationships.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a reverse projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'User', { FOLLOWS: { orientation: 'REVERSE', properties: ['score'] } }) ----

The graph is projected in a `REVERSE` orientation in order to retrieve people with the most followers in the following examples. This will be demonstrated using the Degree Centrality algorithm on this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.write.estimate('myGraph', { writeProperty: 'degree' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 6 | 8 | 40 | 40 | "40 Bytes"

—

=== Stream

:stream-details: For example, we can order the results to find the nodes with the highest degree centrality. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm in `stream` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.stream('myGraph') YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score AS followers ORDER BY followers DESC, name DESC ----

.Results [opts="header",cols="1,1"]

| name | followers | "Doug" | 5.0 | "Michael" | 1.0 | "Charles" | 1.0 | "Bridget" | 1.0 | "Mark" | 0.0 | "Alice" | 0.0

— We can see that Doug is the most popular user in our imaginary social network graph, with 5 followers - all other users follow them, but they don't follow anybody back. In a real social network, celebrities have very high follower counts but tend to follow only very few people. We could therefore consider Doug quite the celebrity!

=== Stats

:stats-syntax: algorithms-degree-centrality-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm in `stats` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.stats('myGraph') YIELD centralityDistribution RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore ----

.Results [opts="header"]

| minimumScore | meanScore | 0.0 | 1.3333358764648438

— Comparing this to the results we saw in the stream example, we can find our minimum and mean values from the table.

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.mutate('myGraph', { mutateProperty: 'degree' }) YIELD centralityDistribution, nodePropertiesWritten RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore, nodePropertiesWritten ----

.Results [opts="header"]

| minimumScore | meanScore | nodePropertiesWritten | 0.0 | 1.3333358764648438 | 6

— The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `degree` which stores the degree centrality score for each node. To find out how to inspect the new schema of the in-memory graph, see Listing graphs in the catalog.

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.write('myGraph', { writeProperty: 'degree' }) YIELD centralityDistribution, nodePropertiesWritten RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore, nodePropertiesWritten ----

.Results [opts="header"]

| minimumScore | meanScore | nodePropertiesWritten | 0.0 | 1.3333358764648438 | 6

— The returned result is the same as in the `stats` example. Additionally, each of the seven nodes now has a new property `degree` in the Neo4j database, containing the degree centrality score for that node.

== Weighted Degree Centrality example

This example will explain the weighted Degree Centrality algorithm. This algorithm is a variant of the Degree Centrality algorithm, that measures the sum of positive weights of incoming and outgoing relationships.

[role=query-example] — .The following will run the algorithm in `stream` mode, showing which users have the highest weighted degree centrality: [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.stream('myGraph', { relationshipWeightProperty: 'score' }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score AS weightedFollowers ORDER BY weightedFollowers DESC, name DESC ----

.Results [opts="header",cols="1,1"]

| name | weightedFollowers | "Doug" | 7.5 | "Charles" | 5.0 | "Michael" | 4.5 | "Mark" | 0.0 | "Bridget" | 0.0 | "Alice" | 0.0

— Doug still remains our most popular user, but there isn't such a big gap to the next person. Charles and Michael both only have one follower, but those relationships have a high relationship weight. Note that Bridget also has a weighted score of 0.0, despite having a connection from Alice. That is because the `score` property value between Bridget and Alice is negative and will be ignored by the algorithm.

== Setting an orientation

By default, node centrality uses the `NATURAL` orientation to compute degrees. For some use-cases it makes sense to analyze a different orientation, for example, if we want to find out how many users follow another user. In order to change the orientation, we can use the `orientation` configuration key. Supported values are `NATURAL` (default), `REVERSE` and `UNDIRECTED`.

[role=query-example] — .The following will run the algorithm in `stream` mode, showing which users have the highest in-degree centrality using the reverse orientation of the relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.degree.stream('myGraph', { orientation: 'REVERSE' }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score AS followees ORDER BY followees DESC, name DESC ----

.Results [opts="header",cols="1,1"]

| name | followees | "Alice" | 3.0 | "Mark" | 2.0 | "Michael" | 1.0 | "Charles" | 1.0 | "Bridget" | 1.0 | "Doug" | 0.0

— The example shows that when looking at the reverse orientation, `Alice` is more central in the network than `Doug`.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Closeness Centrality algorithm in the Neo4j Graph Data Science library. [.alpha] = Closeness Centrality

[abstract] — This section describes the Closeness Centrality algorithm in the Neo4j Graph Data Science library. — Closeness centrality is a way of detecting nodes that are able to spread information very efficiently through a graph.

The closeness centrality of a node measures its average farness (inverse distance) to all other nodes. Nodes with a high closeness score have the shortest distances to all other nodes.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

For each node, the Closeness Centrality algorithm calculates the sum of its distances to all other nodes, based on calculating the shortest paths between all pairs of nodes. The resulting sum is then inverted to determine the closeness centrality score for that node.

The **raw closeness centrality** of a node is calculated using the following formula:

```
raw closeness centrality(node) = 1 / sum(distance from node to all other nodes)
```

It is more common to normalize this score so that it represents the average length of the shortest paths rather than their sum. This adjustment allow comparisons of the closeness centrality of nodes of graphs of different sizes

The formula for **normalized closeness centrality** is as follows:

```
normalized closeness centrality(node) = (number of nodes - 1) / sum(distance from node to all other nodes)
```

== Use-cases - when to use the Closeness Centrality algorithm

* Closeness centrality is used to research organizational networks, where individuals with high closeness centrality are in a favourable position to control and acquire vital information and resources within the organization. One such study is "Mapping Networks of Terrorist Cells" by Valdis E. Krebs.

* Closeness centrality can be interpreted as an estimated time of arrival of information flowing through telecommunications or package delivery networks where information flows through shortest paths to a predefined target. It can also be used in networks where information spreads through all shortest paths simultaneously, such as infection spreading through a social network. Find more details in "Centrality and network flow" by Stephen P. Borgatti.

* Closeness centrality has been used to estimate the importance of words in a document, based on a graph-based keyphrase extraction process. This process is described by Florian Boudin in "A Comparison of Centrality Measures for Graph-Based Keyphrase Extraction".

== Constraints - when not to use the Closeness Centrality algorithm

* Academically, closeness centrality works best on connected graphs. If we use the original formula on an unconnected graph, we can end up with an infinite distance between two nodes in separate connected components. This means that we'll end up with an infinite closeness centrality score when we sum up all the distances from that node. + In practice, a variation on the original formula is used so that we don't run into these issues.

== Syntax

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL

| Name | Type | Default | Optional | Description | concurrency | int | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | int | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | writeConcurrency | int | value of 'concurrency' | yes | The number of concurrent threads used for writing the result. | writeProperty | string | 'centrality' | yes | The property name written back to.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodes | int | The number of nodes considered. | createMillis | int | Milliseconds for loading data. | computeMillis | int | Milliseconds for running the algorithm. | writeMillis | int | Milliseconds for writing result data back. | writeProperty | string | The property name written back to. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.closeness.stream(configuration: Map) YIELD nodeId, centrality ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | concurrency | int | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | int | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph.

.Results [opts="header"]

| Name | Type | Description | node | long | Node ID | centrality | float | Closeness centrality score

== Closeness Centrality algorithm sample

image::closeness_centrality.png[]

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (a:Node{id:"A"}), (b:Node{id:"B"}), (c:Node{id:"C"}), (d:Node{id:"D"}), (e:Node{id:"E"}), (a)-[:LINK]→(b), (b)-[:LINK]→(a), (b)-[:LINK]→(c), (c)-[:LINK]→(b), (c)-[:LINK]→(d), (d)-[:LINK]→(c), (d)-[:LINK]→(e), (e)-[:LINK]→(d); ----

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.closeness.stream({ nodeProjection: 'Node', relationshipProjection: 'LINK' }) YIELD nodeId, centrality RETURN gds.util.asNode(nodeId).name AS user, centrality ORDER BY centrality DESC ----

.Results [opts="header",cols="1,1"]

| Name | Centrality weight | C | 0.6666666666666666 | B | 0.5714285714285714 | D | 0.5714285714285714 | A | 0.4 | E | 0.4

C is the best connected node in this graph, although B and D aren't far behind. A and E don't have close ties to many other nodes, so their scores are lower. Any node that has a direct connection to all other nodes would score 1.

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.closeness.write({ nodeProjection: 'Node', relationshipProjection: 'LINK', writeProperty: 'centrality' }) YIELD nodes, writeProperty ----

.Results [opts="header",cols="1,1"]

| nodes | writeProperty | 5 | "centrality"

== Cypher projection

If node labels and relationship types are not selective enough to project a graph, you can use Cypher queries instead. Cypher projections can also be used to run algorithms on a virtual graph. You can learn more in the Creating graphs using Cypher section of the manual.

[source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.closeness.write({ nodeQuery: 'MATCH (p:Node) RETURN id(p) AS id', relationshipQuery: 'MATCH (p1:Node)-[:LINK]→(p2:Node) RETURN id(p1) AS source, id(p2) AS target' }) YIELD nodes, writeProperty ----

.Results [opts="header",cols="1,1"]

| nodes | writeProperty | 5 | "centrality"

Calculation: - count farness in each msbfs-callback - divide by N-1 `N = 5` // number of nodes `k = N-1 = 4` // used for normalization .... A B C D E —	---------- ---------- --------- A	0 1 2 3 4 // farness between each pair of nodes B	1 0 1 2 3 C	2 1 0 1 2 D	3 2 1 0 1 E	4 3 2 1 0 —	---------- ---------- --------- S	10 7 6 7 10 // raw closeness centrality ==	======= ======= ======= ======= = k/S	0.4 0.57 0.67 0.57 0.4 // normalized closeness centrality .... :leveloffset: 2 :leveloffset: +3 :description: This section describes the Harmonic Centrality algorithm [.alpha] = Harmonic Centrality [abstract] — This section describes the Harmonic Centrality algorithm — Harmonic centrality (also known as valued centrality) is a variant of closeness centrality, that was invented to solve the problem the original formula had when dealing with

| Name | Type | Default | Optional | Description | concurrency | int | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | int | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | writeConcurrency | int | value of 'concurrency' | yes | The number of concurrent threads used for writing the result. | writeProperty | string | 'centrality' | yes | The property name written back to.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodes | int | The number of nodes considered. | createMillis | int | Milliseconds for loading data. | computeMillis | int | Milliseconds for running the algorithm. | writeMillis | int | Milliseconds for writing result data back. | writeProperty | string | The property name written back to. | centralityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.closeness.harmonic.stream(configuration: Map) YIELD nodeId, centrality ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | concurrency | int | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | int | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph.

.Results [opts="header"]

| Name | Type | Description | node | long | Node ID | centrality | float | Harmonic centrality score

== Harmonic Centrality algorithm sample

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (a:Node{id:"A"}), (b:Node{id:"B"}), (c:Node{id:"C"}), (d:Node{id:"D"}), (e:Node{id:"E"}), (a)-[:LINK]→(b), (b)-[:LINK]→(c), (d)-[:LINK]→(e) ----

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.closeness.harmonic.stream({ nodeProjection: 'Node', relationshipProjection: 'LINK' }) YIELD nodeId, centrality RETURN gds.util.asNode(nodeId).name AS user, centrality ORDER BY centrality DESC ----

.Results [opts="header",cols="1,1"]

| Name | Centrality weight | B | 0.5 | A | 0.375 | c | 0.375 | D | 0.25 | E | 0.25

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.closeness.harmonic.write({ nodeProjection: 'Node', relationshipProjection: 'LINK', writeProperty: 'centrality' }) YIELD nodes, writeProperty ----

.Results [opts="header",cols="1,1"]

| nodes | writeProperty | 5 | "centrality"

:leveloffset: 2

:leveloffset: +3

:description: This section describes the HITS algorithm in the Neo4j Graph Data Science library.

[.alpha] = HITS :entity: pregel :result: authority and hub scores :algorithm: HITS

[abstract] — This section describes the HITS algorithm in the Neo4j Graph Data Science library. —

== Introduction

The Hyperlink-Induced Topic Search (HITS) is a link analysis algorithm that rates nodes based on two scores, a `hub` score and an `authority` score. The `authority` score estimates the importance of the node within the network. The `hub` score estimates the value of its relationships to other nodes. The GDS implementation is based on the Authoritative Sources in a Hyperlinked Environment publication by Jon M. Kleinberg.

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

.HITS syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ======

.Run HITS in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.hits.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, values: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | hitsIterations | Integer | n/a | no | The number of hits iterations to run. The number of pregel iterations will be equal to `hitsIterations * 4 + 1` | authProperty | String | "auth" | yes | The name that is used for the auth property when using `STREAM`, `MUTATE` or `WRITE` modes. | hubProperty | String | "hub" | yes | The name that is used for the hub property when using `STREAM`, `MUTATE` or `WRITE` modes.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | values | Map | A map containing the `auth` and `hub` keys.

======

[.include-with-stats] ======

.Run HITS in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.hits.stats(graphName: String, configuration: Map) YIELD ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | hitsIterations | Integer | n/a | no | The number of hits iterations to run. The number of pregel iterations will be equal to `hitsIterations * 4 + 1` | authProperty | String | "auth" | yes | The name that is used for the auth property when using `STREAM`, `MUTATE` or `WRITE` modes. | hubProperty | String | "hub" | yes | The name that is used for the hub property when using `STREAM`, `MUTATE` or `WRITE` modes.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | Number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | configuration | Map | Configuration used for running the algorithm.

======

[.include-with-mutate] ======

.Run HITS in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.hits.mutate(graphName: String, configuration: Map) YIELD ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | hitsIterations | Integer | n/a | no | The number of hits iterations to run. The number of pregel iterations will be equal to `hitsIterations * 4 + 1` | authProperty | String | "auth" | yes | The name that is used for the auth property when using `STREAM`, `MUTATE` or `WRITE` modes. | hubProperty | String | "hub" | yes | The name that is used for the hub property when using `STREAM`, `MUTATE` or `WRITE` modes.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | nodePropertiesWritten | Integer | The number of properties that were written to Neo4j. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ======

.Run HITS in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.hits.write(graphName: String, configuration: Map) YIELD ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | hitsIterations | Integer | n/a | no | The number of hits iterations to run. The number of pregel iterations will be equal to `hitsIterations * 4 + 1` | authProperty | String | "auth" | yes | The name that is used for the auth property when using `STREAM`, `MUTATE` or `WRITE`

modes. | hubProperty | String | "hub" | yes | The name that is used for the hub property when using `STREAM`, `MUTATE` or `WRITE` modes.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back. | nodePropertiesWritten | Integer | The number of properties that were written to Neo4j. | configuration | Map | The configuration used for running the algorithm.

======

====

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: hits.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (a:Website {name: 'A'}), (b:Website {name: 'B'}), (c:Website {name: 'C'}), (d:Website {name: 'D'}), (e:Website {name: 'E'}), (f:Website {name: 'F'}), (g:Website {name: 'G'}), (h:Website {name: 'H'}), (i:Website {name: 'I'}),

(a)-[:LINK]→(b), (a)-[:LINK]→(c), (a)-[:LINK]→(d), (b)-[:LINK]→(c), (b)-[:LINK]→(d), (c)-[:LINK]→(d),

(e)-[:LINK]→(b), (e)-[:LINK]→(d), (e)-[:LINK]→(f), (e)-[:LINK]→(h),

(f)-[:LINK]→(g), (f)-[:LINK]→(i), (f)-[:LINK]→(h), (g)-[:LINK]→(h), (g)-[:LINK]→(i), (h)-[:LINK]→(i); ----

In the example, we will use the HITS algorithm to calculate the authority and hub scores.

.The following statement will create the graph and store it in the graph catalog. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Website', 'LINK'); ----

In the following examples we will demonstrate using the HITS algorithm on this graph.

=== Stream

In the `stream` execution mode, the algorithm returns the {result} for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm, and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.hits.stream('myGraph', {hitsIterations: 20}) YIELD nodeId, values RETURN gds.util.asNode(nodeId).name AS Name, values.auth AS auth, values.hub as hub ORDER BY Name ASC ----

.Results [opts="header"]

| Name | auth | hub | "A" | 0.0 | 0.5147630377521207 | "B" | 0.42644630743935796 | 0.3573686670593437 | "C" | 0.3218729455718005 | 0.23857061715828276 | "D" |

0.6463862608483191 | 0.0 | "E" | 0.0 | 0.640681017095129 | "F" | 0.23646490227616518 | 0.2763222153580397 | "G" | 0.10200264424057169 | 0.23867470447760597 | "H" | 0.426571816146601 | 0.0812340105698113 | "I" | 0.22009646020698218 | 0.0

— :leveloffset: 2

:leveloffset: +3

:description: This chapter provides explanations and examples for each of the influence maximization algorithms in the Neo4j Graph Data Science library. = Influence Maximization

[abstract] — This chapter provides explanations and examples for each of the influence maximization algorithms in the Neo4j Graph Data Science library. — The objective of influence maximization is to find a small subset of k nodes from a network in order to achieve maximization to the total number of nodes influenced by these k nodes. The Neo4j GDS library includes the following alpha influence maximization algorithms:

* Alpha Greedy CELF

:leveloffset: 2

:leveloffset: +4

:description: This section describes the Cost Effective Lazy Forward (CELF) influence maximization algorithm in the Neo4j Graph Data Science library. [.alpha] = CELF :entity: influenceMaximization :result: spread :algorithm: CELF

[abstract] — This section describes the Cost Effective Lazy Forward (CELF) influence maximization algorithm in the Neo4j Graph Data Science library. — [.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

This topic includes:

* Introduction * Syntax * Examples ** Stream

== Introduction The CELF algorithm for influence maximization aims to find k nodes that maximize the expected spread of influence in the network. It simulates the influence spread using the Independent Cascade model, which calculates the expected spread by taking the average spread over the mc Monte-Carlo simulations. In the propagation process, a node is influenced in case that a uniform random draw is less than the probability p.

Leskovec et al. 2007 introduced the CELF algorithm in their study Cost-effective Outbreak Detection in Networks to deal with the NP-hard problem of influence maximization. The CELF algorithm is based on a "lazy-forward" optimization. The CELF algorithm dramatically improves the efficiency of the Greedy algorithm and should be preferred for large networks.

== Syntax

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

.CELF syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ======

.Run CELF in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.influenceMaximization.celf.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, spread: Float ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | seedSetSize | Integer | n/a | no | The number of nodes that maximize the expected spread in the network. | monteCarloSimulations | Integer | 1000 | yes | The number of Monte-Carlo simulations. | propagationProbability | Float | 0.1 | yes | The probability of a node being activated by an active neighbour node.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | spread | Float | The spread gained by selecting the node.

====== [.include-with-stats] ======

.Run CELF in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.influenceMaximization.celf.stats(graphName: String, configuration: Map) YIELD nodes: Integer, computeMillis: Integer, ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | seedSetSize | Integer | n/a | no | The number of nodes that maximize the expected spread in the network. | monteCarloSimulations | Integer | 1000 | yes | The number of Monte-Carlo simulations. | propagationProbability | Float | 0.1 | yes | The probability of a node being activated by an active neighbour node.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodes | Integer | The number of nodes in the graph. | computeMillis | Integer |

Milliseconds for running the algorithm.

====== ====

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: influence-maximization.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=setup-query, indent=0] ---- CREATE (a:Person {name: 'Jimmy'}), (b:Person {name: 'Jack'}), (c:Person {name: 'Alice'}), (d:Person {name: 'Ceri'}), (e:Person {name: 'Mohammed'}), (f:Person {name: 'Michael'}), (g:Person {name: 'Ethan'}), (h:Person {name: 'Lara'}), (i:Person {name: 'Amir'}), (j:Person {name: 'Willie'}),

(b)-[:FRIEND_OF]→(c), (c)-[:FRIEND_OF]→(a), (c)-[:FRIEND_OF]→(g), (c)-[:FRIEND_OF]→(h), (c)-[:FRIEND_OF]→(i), (c)-[:FRIEND_OF]→(j), (d)-[:FRIEND_OF]→(g), (f)-[:FRIEND_OF]→(e), (f)-[:FRIEND_OF]→(g), (g)-[:FRIEND_OF]→(a), (g)-[:FRIEND_OF]→(b), (g)-[:FRIEND_OF]→(h), (g)-[:FRIEND_OF]→(e), (h)-[:FRIEND_OF]→(i); ----

In the example, we will use the CELF algorithm to find `k` nodes subset.

.The following statement will create the graph and store it in the graph catalog. [source, cypher, role=graph-create-query no-play, indent=0] ---- CALL gds.graph.create('myGraph', 'Person', 'FRIEND_OF'); ----

In the following examples we will demonstrate using the CELF algorithm on this graph.

=== Stream

In the `stream` execution mode, the algorithm returns the {result} for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm, and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.influenceMaximization.celf.stream('myGraph', {seedSetSize: 3, concurrency: 4}) YIELD nodeId, spread RETURN gds.util.asNode(nodeId).name AS Name, spread ORDER BY spread ASC ----

.Results [opts="header"]

| Name | spread | "Alice" | 1.519 | "Ethan" | 2.701 | "Michael" | 3.8

— :leveloffset: 2

:leveloffset: +4

:description: This section describes the Greedy influence maximization algorithm in the Neo4j Graph Data Science library. [.alpha] = Greedy :entity: influenceMaximization :result: spread :algorithm: Greedy

[abstract] — This section describes the Greedy influence maximization algorithm in the Neo4j Graph Data Science library. — [.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

This topic includes:

* Introduction * Syntax * Examples * Stream

== Introduction The Greedy algorithm for influence maximization aims to find `k` nodes that maximize the expected spread of influence in a network. It simulates the influence spread using the Independent Cascade model, which calculates the expected spread by taking the average spread over the `mc` Monte-Carlo simulations. In the propagation process, a node is influenced in case that a uniform random draw is less than the probability `p`.

Kempe et al. 2003 introduced the Greedy algorithm in their study Maximizing the Spread of Influence through a Social Network to deal with the *NP-hard problem of influence maximization. The Greedy algorithm successively selecting the node within the maximum marginal gain approximation in polynomial time. For large networks CELF algorithm should be used.

== Syntax

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

.Greedy syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ======

.Run Greedy in stream mode on a named graph. [source, cypher, role=no-play, indent=0] ---- CALL gds.alpha.influenceMaximization.greedy.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, spread : Float ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | seedSetSize | Integer | n/a | no | The number of nodes that maximize the expected spread in the network. | monteCarloSimulations | Integer | 1000 | yes | The number of Monte-Carlo simulations. | propagationProbability | Float | 0.1 | yes | The probability of a node being activated by an active neighbour node.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | spread | Float | The spread gained by selecting the node.

====== [.include-with-stats] ====== .Run Greedy in stats mode on a named graph. [source, cypher, role=no-play, indent=0] ---- CALL gds.alpha.influenceMaximization.greedy.stats(graphName: String, configuration: Map) YIELD nodes: Integer, computeMillis: Integer, ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['**] | yes | **Filter the named graph using the given node labels. | relationshipTypes | List of String | ['**] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | seedSetSize | Integer | n/a | no | The number of nodes that maximize the expected spread in the network. | monteCarloSimulations | Integer | 1000 | yes | The number of Monte-Carlo simulations. | propagationProbability | Float | 0.1 | yes | The probability of a node being activated by an active neighbour node.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodes | Integer | The number of nodes in the graph. | computeMillis | Integer | Milliseconds for running the algorithm.

====== ====

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: influence-maximization.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=setup-query no-play, indent=0] ---- CREATE (a:Person {name: 'Jimmy'}), (b:Person {name: 'Jack'}), (c:Person {name: 'Alice'}), (d:Person {name: 'Ceri'}), (e:Person {name: 'Mohammed'}), (f:Person {name: 'Michael'}), (g:Person {name: 'Ethan'}), (h:Person {name: 'Lara'}), (i:Person {name: 'Amir'}), (j:Person {name: 'Willie'}),

(b)-[:FRIEND_OF]→(c), (c)-[:FRIEND_OF]→(a), (c)-[:FRIEND_OF]→(g), (c)-[:FRIEND_OF]→(h), (c)-[:FRIEND_OF]→(i), (c)-[:FRIEND_OF]→(j), (d)-[:FRIEND_OF]→(g), (f)-[:FRIEND_OF]→(e), (f)-[:FRIEND_OF]→(g), (g)-[:FRIEND_OF]→(a), (g)-[:FRIEND_OF]→(b), (g)-[:FRIEND_OF]→(h), (g)-[:FRIEND_OF]→(e), (h)-[:FRIEND_OF]→(i); ----

In the example, we will use the Greedy algorithm to find `k` nodes subset.

.The following statement will create the graph and store it in the graph catalog. [source, cypher, role=graph-create-query no-play, indent=0] ---- CALL gds.graph.create('myGraph', 'Person', 'FRIEND_OF'); ----

In the following examples we will demonstrate using the Greedy algorithm on this graph.

=== Stream

In the `stream` execution mode, the algorithm returns the {result} for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm, and stream results: [source, cypher, role=no-play, indent=0] ---- CALL gds.alpha.influenceMaximization.greedy.stream('myGraph', {seedSetSize: 3, concurrency: 4}) YIELD nodeId, spread RETURN gds.util.asNode(nodeId).name AS Name, spread ORDER BY spread ASC ----

.Results [opts="header"]

| Name | spread | "Alice" | 1.519 | "Ethan" | 2.701 | "Michael" | 3.8

— :leveloffset: 2

:leveloffset: +2

:description: This chapter provides explanations and examples for each of the community detection algorithms in the Neo4j Graph Data Science library. = Community detection

[abstract] — This chapter provides explanations and examples for each of the community detection algorithms in the Neo4j Graph Data Science library. — Community detection algorithms are used to evaluate how groups of nodes are clustered or partitioned, as well as their tendency to strengthen or break apart. The Neo4j GDS library includes the following community detection algorithms, grouped by quality tier:

* Production-quality Louvain Label Propagation Weakly Connected Components Triangle Count Local Clustering Coefficient * Beta K-1 Coloring Modularity Optimization * Alpha Strongly Connected Components ** Speaker-Listener Label Propagation

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Louvain algorithm in the Neo4j Graph Data Science library. = Louvain :entity: node :result: community ID :algorithm: Louvain

[abstract] — This section describes the Louvain algorithm in the Neo4j Graph Data Science library. — :directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction The Louvain method is an algorithm to detect communities in large networks. It maximizes a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities. This means evaluating how much more densely connected the nodes within a community are, compared to how connected they would be in a random network.

The Louvain algorithm is a hierarchical clustering algorithm, that recursively merges communities into a single node and executes the modularity clustering on the condensed graphs.

For more information on this algorithm, see:

* Lu, Hao, Mahantesh Halappanavar, and Ananth Kalyanaraman "Parallel heuristics for scalable community detection." * https://en.wikipedia.org/wiki/Louvain_modularity

[NOTE] ==== Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read Memory Estimation. ====

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

.Louvain syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ====== .Run Louvain in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, communityId: Integer,

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial community for a node. The property value needs to be a number. | maxLevels | Integer | 10 | yes | The maximum number of levels in which the graph is clustered and then condensed. | maxIterations | Integer | 10 | yes | The maximum number of iterations that the modularity optimization will run for each level. | tolerance | Float | 0.0001 | yes | Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns. | includeIntermediateCommunities | Boolean | false | yes | Indicates whether to write intermediate communities. If set to false, only the final community is persisted. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the `includeIntermediateCommunities` flag.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeId | Integer | Node ID. | communityId | Integer | The community ID of the final level. | intermediateCommunityIds | List of Integer | Community IDs for each level. `Null` if `includeIntermediateCommunities` is set to false.

======

[.include-with-stats] ====== .Run Louvain in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.stats(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, communityCount: Integer, ranLevels: Integer, modularity: Float, modularities: List of Integer, communityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. |

concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial community for a node. The property value needs to be a number. | maxLevels | Integer | 10 | yes | The maximum number of levels in which the graph is clustered and then condensed. | maxIterations | Integer | 10 | yes | The maximum number of iterations that the modularity optimization will run for each level. | tolerance | Float | 0.0001 | yes | Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns. | includeIntermediateCommunities | Boolean | false | yes | Indicates whether to write intermediate communities. If set to false, only the final community is persisted. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the `includeIntermediateCommunities` flag.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing percentiles and community count. | communityCount | Integer | The number of communities found. | ranLevels | Integer | The number of supersteps the algorithm actually ran. | modularity | Float | The final modularity score. | modularities | List of Integer | The modularity scores for each level. | communityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size for the last level. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-mutate] ====== .Run Louvain in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, postProcessingMillis: Integer, communityCount: Integer, ranLevels: Integer, modularity: Float, modularities: List of Integer, nodePropertiesWritten: Integer, communityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial community for a node. The property value needs to be a number. | maxLevels | Integer | 10 | yes | The maximum number of levels in which the graph is clustered and then condensed. | maxIterations | Integer | 10 | yes | The maximum number of iterations that the modularity optimization will run for each level. | tolerance | Float | 0.0001 | yes | Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns. | includeIntermediateCommunities | Boolean | false | yes | Indicates whether to write intermediate communities. If set to false, only the final community is persisted. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the `includeIntermediateCommunities` flag.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | postProcessingMillis | Integer | Milliseconds for computing percentiles and community count. | communityCount | Integer | The number of communities found. | ranLevels | Integer | The number of supersteps the algorithm actually ran. | modularity | Float | The final modularity score. | modularities | List of Integer | The modularity scores for each level. | nodePropertiesWritten | Integer | Number of properties added to the in-memory graph. | communityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size for the last level. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ====== .Run Louvain in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.write(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, postProcessingMillis: Integer, nodePropertiesWritten: Integer, communityCount: Integer, ranLevels: Integer, modularity: Float, modularities: List of Integer, communityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial community for a node. The property value needs to be a number. | maxLevels | Integer | 10 | yes | The maximum number of levels in which the graph is clustered and then condensed. | maxIterations | Integer | 10 | yes | The maximum number of iterations that the modularity optimization will run for each level. | tolerance | Float | 0.0001 | yes | Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns. | includeIntermediateCommunities | Boolean | false | yes | Indicates whether to write intermediate communities. If set to false, only the final community is persisted. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the `includeIntermediateCommunities` flag. | minCommunitySize | Integer | 0 | yes | Only community ids of communities with a size greater than or equal to the given value are written to Neo4j.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back. | postProcessingMillis | Integer | Milliseconds for computing percentiles and community count. | nodePropertiesWritten | Integer | The number of node properties written. | communityCount | Integer | The number of communities found. | ranLevels | Integer | The number of supersteps the algorithm actually ran. | modularity | Float | The final modularity score. | modularities | List of Integer | The modularity scores for each level. | communityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size for the last level. | configuration | Map | The configuration used for running the algorithm.

```
======

====

=== Anonymous graphs

.Run Louvain in write mode on an anonymous graph. [source, cypher, role=noplay, indent=0] ---- CALL
gds.louvain.write(configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer,
postProcessingMillis: Integer, nodePropertiesWritten: Integer, communityCount: Integer, ranLevels: Integer, modularity:
Float, modularities: List of Integer, communityDistribution: Map, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of

concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial community for a node. The property value needs to be a number. | maxLevels | Integer | 10 | yes | The maximum number of levels in which the graph is clustered and then condensed. | maxIterations | Integer | 10 | yes | The maximum number of iterations that the modularity optimization will run for each level. | tolerance | Float | 0.0001 | yes | Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns. | includeIntermediateCommunities | Boolean | false | yes | Indicates whether to write intermediate communities. If set to false, only the final community is persisted. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the `includeIntermediateCommunities` flag.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: Louvain community detection :graph-description: social network :image-file: louvain.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (nAlice:User {name: 'Alice', seed: 42}), (nBridget:User {name: 'Bridget', seed: 42}), (nCharles:User {name: 'Charles', seed: 42}), (nDoug:User {name: 'Doug'}), (nMark:User {name: 'Mark'}), (nMichael:User {name: 'Michael'}),

(nAlice)-[:LINK {weight: 1}]→(nBridget), (nAlice)-[:LINK {weight: 1}]→(nCharles), (nCharles)-[:LINK {weight: 1}]→(nBridget),

(nAlice)-[:LINK {weight: 5}]→(nDoug),

(nMark)-[:LINK {weight: 1}]→(nDoug), (nMark)-[:LINK {weight: 1}]→(nMichael), (nMichael)-[:LINK {weight: 1}]→(nMark); ----

This graph has two clusters of *Users*, that are closely connected. Between those clusters there is one single edge. The relationships that connect the nodes in each component have a property `weight` which determines the strength of the relationship.

We can now create the graph and store it in the graph catalog. We load the `LINK` relationships with orientation set to `UNDIRECTED` as this works best with the Louvain algorithm.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create the graph and store it in the graph catalog. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'User', { LINK: { orientation: 'UNDIRECTED' } }, { nodeProperties: 'seed', relationshipProperties: 'weight' }) ----

In the following examples we will demonstrate using the Louvain algorithm on this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.write.estimate('myGraph', { writeProperty: 'community' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header", cols="1,1,1,1,1"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 6 | 14 | 5321 | 563904 | "[5321 Bytes … 550 KiB]"

—

=== Stream

In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.stream('myGraph') YIELD nodeId, communityId, intermediateCommunityIds RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds ORDER BY name ASC ----

.Results [opts="header"]

| name | communityId | intermediateCommunityIds | "Alice" | 2 | null | "Bridget" | 2 | null | "Charles" | 2 | null | "Doug" | 5 | null | "Mark" | 5 | null | "Michael" | 5 | null

— We use default values for the procedure configuration parameter. Levels and `innerIterations` are set to 10 and the tolerance value is 0.0001. Because we did not set the value of `includeIntermediateCommunities` to `true`, the column communities is always `null`.

=== Stats

:stats-syntax: algorithms-louvain-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm and returns the result in form of statistical and measurement values [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.stats('myGraph') YIELD communityCount ----

.Results [opts="header",cols="1"]

| communityCount | 2

—

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm and store the results in `myGraph`: [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.mutate('myGraph', { mutateProperty: 'communityId' }) YIELD communityCount, modularity, modularities ----

.Results [opts="header"]

| communityCount | modularity | modularities | 2 | 0.3571428571428571 | [0.3571428571428571] |1 row

— In `mutate` mode, only a single row is returned by the procedure. The result contains meta information, like the number of identified communities and the modularity values. In contrast to the `write` mode the result is written to the GDS in-memory graph instead of the Neo4j database.

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following run the algorithm, and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.write('myGraph', { writeProperty: 'community' }) YIELD communityCount, modularity, modularities ----

.Results [opts="header"]

| communityCount | modularity | modularities | 2 | 0.3571428571428571 | [0.3571428571428571] |1 row

— When writing back the results, only a single row is returned by the procedure. The result contains meta information, like the number of identified communities and the modularity values.

=== Weighted

The Louvain algorithm can also run on weighted graphs, taking the given relationship weights into concern when calculating the modularity.

[role=query-example] — .The following will run the algorithm on a weighted graph and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.stream('myGraph', { relationshipWeightProperty: 'weight' }) YIELD nodeId, communityId, intermediateCommunityIds RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds ORDER BY name ASC ----

.Results [opts="header"]

| name | communityId | intermediateCommunityIds | "Alice" | 3 | null | "Bridget" | 2 | null | "Charles" | 2 | null | "Doug" | 3 | null | "Mark" | 5 | null | "Michael" | 5 | null

— Using the weighted relationships, we see that `Alice` and `Doug` have formed their own community, as their link is much stronger than all the others.

=== Seeded

The Louvain algorithm can be run incrementally, by providing a seed property. With the seed property an initial community mapping can be supplied for a subset of the loaded nodes. The algorithm will try to keep the seeded community IDs.

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.stream('myGraph', { seedProperty: 'seed' }) YIELD nodeId, communityId, intermediateCommunityIds RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds ORDER BY name ASC ----

.Results [opts="header"]

| name | communityId | intermediateCommunityIds | "Alice" | 42 | null | "Bridget" | 42 | null | "Charles" | 42 | null | "Doug" | 47 | null | "Mark" | 47 | null | "Michael" | 47 | null

— Using the seeded graph, we see that the community around `Alice` keeps its initial community ID of `42`. The other community is assigned a new community ID, which is guaranteed to be larger than the largest seeded community ID. Note that the `consecutiveIds` configuration option cannot be used in combination with seeding in order to retain the seeding values.

=== Stream intermediate communities

As described before, Louvain is a hierarchical clustering algorithm. That means that after every clustering step all nodes that belong to the same cluster are reduced to a single node. Relationships between nodes of the same cluster become self-relationships, relationships to nodes of other clusters connect to the clusters representative. This condensed graph is then used to run the next level of clustering. The process is repeated until the clusters are stable.

In order to demonstrate this iterative behavior, we need to construct a more complex graph.

image::louvain-multilevel-graph.svg[align="center"]

[source, cypher, role=noplay setup-query, indent=0] ---- CREATE (a:Node {name: 'a'}) CREATE (b:Node {name: 'b'}) CREATE (c:Node {name: 'c'}) CREATE (d:Node {name: 'd'}) CREATE (e:Node {name: 'e'}) CREATE (f:Node {name: 'f'}) CREATE (g:Node {name: 'g'}) CREATE (h:Node {name: 'h'}) CREATE (i:Node {name: 'i'}) CREATE (j:Node {name: 'j'}) CREATE (k:Node {name: 'k'}) CREATE (l:Node {name: 'l'}) CREATE (m:Node {name: 'm'}) CREATE (n:Node {name: 'n'}) CREATE (x:Node {name: 'x'})

CREATE (a)-[:TYPE]→(b) CREATE (a)-[:TYPE]→(d) CREATE (a)-[:TYPE]→(f) CREATE (b)-[:TYPE]→(d) CREATE (b)-[:TYPE]→(x) CREATE (b)-[:TYPE]→(g) CREATE (b)-[:TYPE]→(e) CREATE (c)-[:TYPE]→(x) CREATE (c)-[:TYPE]→(f) CREATE (d)-[:TYPE]→(k) CREATE (e)-[:TYPE]→(x) CREATE (e)-[:TYPE]→(f) CREATE (e)-[:TYPE]→(h) CREATE (f)-[:TYPE]→(g) CREATE (g)-[:TYPE]→(h) CREATE (h)-[:TYPE]→(i) CREATE (h)-[:TYPE]→(j) CREATE (i)-[:TYPE]→(k) CREATE (j)-[:TYPE]→(k) CREATE (j)-[:TYPE]→(m) CREATE (j)-[:TYPE]→(n) CREATE (k)-[:TYPE]→(m) CREATE (k)-[:TYPE]→(l) CREATE (l)-[:TYPE]→(n) CREATE (m)-[:TYPE]→(n); ----

[role=query-example] — .The following will load the example graph, run the algorithm and stream results including the intermediate communities: [source, cypher, role=noplay, indent=0] ---- CALL gds.louvain.stream({ nodeProjection: 'Node', relationshipProjection: { TYPE: { type: 'TYPE', orientation: 'undirected', aggregation: 'NONE' } }, includeIntermediateCommunities: true }) YIELD nodeId, communityId, intermediateCommunityIds RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds ORDER BY name ASC ----

.Results [opts="header"]

| name | communityId | intermediateCommunityIds | "a" | 14 | [3, 14] | "b" | 14 | [3, 14] | "c" | 14 | [14, 14] | "d" | 14 | [3, 14] | "e" | 14 | [14, 14] | "f" | 14 | [14, 14] | "g" | 7 | [7, 7] | "h" | 7 | [7, 7] | "i" | 7 | [7, 7] | "j" | 12 | [12, 12] | "k" | 12 | [12, 12] | "l" | 12 | [12, 12] | "m" | 12 | [12, 12] | "n" | 12 | [12, 12] | "x" | 14 | [14, 14]

— In this example graph, after the first iteration we see 4 clusters, which in the second iteration are reduced to three.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Label Propagation algorithm in the Neo4j Graph Data Science library. = Label Propagation :entity: node :result: community ID :algorithm: Label Propagation

[abstract] — This section describes the Label Propagation algorithm in the Neo4j Graph Data Science library. — :directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

The Label Propagation algorithm (LPA) is a fast algorithm for finding communities in a graph. It detects these communities using network structure alone as its guide, and doesn't require a pre-defined objective function or prior information about the communities.

LPA works by propagating labels throughout the network and forming communities based on this process of label propagation.

The intuition behind the algorithm is that a single label can quickly become dominant in a densely connected group of nodes, but will have trouble crossing a sparsely connected region. Labels will get trapped inside a densely connected group of nodes, and those nodes that end up with the same label when the algorithms finish can be considered part of the same community.

The algorithm works as follows:

* Every node is initialized with a unique community label (an identifier). * These labels propagate through the network. * At every iteration of propagation, each node updates its label to the one that the maximum numbers of its neighbours belongs to. Ties are broken arbitrarily but deterministically. * LPA reaches convergence when each node has the majority label of its neighbours. * LPA stops if either convergence, or the user-defined maximum number of iterations is achieved.

As labels propagate, densely connected groups of nodes quickly reach a consensus on a unique label. At the end of the propagation only a few labels will remain - most will have disappeared. Nodes that have the same community label at convergence are said to belong to the same community.

One interesting feature of LPA is that nodes can be assigned preliminary labels to narrow down the range of solutions generated. This means that it can be used as semi-supervised way of finding communities where we hand-pick some initial communities.

For more information on this algorithm, see:

* "Near linear time algorithm to detect community structures in large-scale networks" * Use cases: Twitter polarity classification with label propagation over lexical links and the follower graph Label Propagation Prediction of Drug-Drug Interactions Based on Clinical Side Effects ** "Feature Inference Based on Label Propagation on Wikidata Graph for DST"

[NOTE] ==== Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read Memory Estimation. ====

== Syntax

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | 10 | yes | The maximum number of iterations to run. | nodeWeightProperty | String | null | yes | The name of a node property that contains node weights. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | The name of a node property that defines an initial numeric label. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | communityId | Integer | Community ID.

======

[.include-with-stats] ======

.Run Label Propagation in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.stats(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, communityCount: Integer, ranIterations: Integer, didConverge: Boolean, communityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | 10 | yes | The maximum number of iterations to run. | nodeWeightProperty | String | null | yes | The name of a node property that contains

node weights. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | The name of a node property that defines an initial numeric label. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing percentiles and community count. | communityCount | Integer | The number of communities found. | ranIterations | Integer | The number of iterations that were executed. | didConverge | Boolean | True if the algorithm did converge to a stable labelling within the provided number of maximum iterations. | communityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-mutate] ======

.Run Label Propagation in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, postProcessingMillis: Integer, nodePropertiesWritten: Integer, communityCount: Integer, ranIterations: Integer, didConverge: Boolean, communityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | 10 | yes | The maximum number of iterations to run. | nodeWeightProperty | String | null | yes | The name of a node property that contains node weights. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | The name of a node property that defines an initial numeric label. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer |

Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | postProcessingMillis | Integer | Milliseconds for computing percentiles and community count. | nodePropertiesWritten | Integer | The number of node properties written. | communityCount | Integer | The number of communities found. | ranIterations | Integer | The number of iterations that were executed. | didConverge | Boolean | True if the algorithm did converge to a stable labelling within the provided number of maximum iterations. | communityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size. | configuration | Map | The configuration used for running the algorithm.

```
======

[.include-with-write] ======

.Run Label Propagation in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.write( graphName: String, configuration: Map ) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, postProcessingMillis: Integer, nodePropertiesWritten: Integer, communityCount: Integer, ranIterations: Integer, didConverge: Boolean, communityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

```
.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]
```

| Name | Type | Default | Optional | Description | maxIterations | Integer | 10 | yes | The maximum number of iterations to run. | nodeWeightProperty | String | null | yes | The name of a node property that contains node weights. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | The name of a node property that defines an initial numeric label. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). | minCommunitySize | Integer | 0 | yes | Only community ids of communities with a size greater than or equal to the given value are written to Neo4j.

```
.Results [opts="header",cols="1,1,6"]
```

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back. | postProcessingMillis | Integer | Milliseconds for computing percentiles and community count. | nodePropertiesWritten | Integer | The number of node properties written. | communityCount | Integer | The

number of communities found. | ranIterations | Integer | The number of iterations that were executed. | didConverge | Boolean | True if the algorithm did converge to a stable labelling within the provided number of maximum iterations. | communityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run Label Propagation in write mode on an anonymous graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.write(configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, postProcessingMillis: Integer, nodePropertiesWritten: Integer, communityCount: Integer, ranIterations: Integer, didConverge: Boolean, communityDistribution: Map, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | 10 | yes | The maximum number of iterations to run. | nodeWeightProperty | String | null | yes | The name of a node property that contains node weights. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | The name of a node property that defines an initial numeric label. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: label-propagation-graph.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:User {name: 'Alice', seed_label: 52}), (bridget:User {name: 'Bridget', seed_label: 21}), (charles:User {name: 'Charles', seed_label: 43}), (doug:User {name: 'Doug', seed_label: 21}), (mark:User {name: 'Mark', seed_label: 19}), (michael:User {name: 'Michael', seed_label: 52}),

(alice)-[:FOLLOW {weight: 1}]→(bridget), (alice)-[:FOLLOW {weight: 10}]→(charles), (mark)-[:FOLLOW {weight: 1}]→(doug), (bridget)-[:FOLLOW {weight: 1}]→(michael), (doug)-[:FOLLOW {weight: 1}]→(mark), (michael)-[:FOLLOW {weight: 1}]→(alice), (alice)-[:FOLLOW {weight: 1}]→(michael), (bridget)-[:FOLLOW {weight: 1}]→(alice), (michael)-[:FOLLOW {weight: 1}] →(bridget), (charles)-[:FOLLOW {weight: 1}]→(doug) ----

This graph represents six users, some of whom follow each other. Besides a `name` property, each user also has a `seed_label` property. The `seed_label` property represents a value in the graph used to seed the node with a label. For example, this can be a result from a previous run of the Label Propagation algorithm. In addition, each relationship has a weight property.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'User', 'FOLLOW', { nodeProperties: 'seed_label', relationshipProperties: 'weight' }) ----

In the following examples we will demonstrate using the Label Propagation algorithm on this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.write.estimate('myGraph', { writeProperty: 'community' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 6 | 10 | 1608 | 1608 | "1608 Bytes"

—

=== Stream

:stream-details: For example, we can order the results to see the nodes that belong to the same communities displayed next to each other. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.stream('myGraph') YIELD nodeId, communityId AS Community RETURN gds.util.asNode(nodeId).name AS Name, Community ORDER BY Community, Name ----

.Results [opts="header",cols="1,1"]

| Name | Community | "Alice" | 1 | "Bridget" | 1 | "Michael" | 1 | "Charles" | 4 | "Doug" | 4 | "Mark" | 4

— In the above example we can see that our graph has two communities each containing three nodes. The default behaviour of the algorithm is to run `unweighted`, e.g. without using `node` or `relationship` weights. The `weighted` option will be demonstrated in Weighted

=== Stats

:stats-syntax: algorithms-label-propagation-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

.The following will run the algorithm in `stats` mode: [role=query-example] — [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.stats('myGraph') YIELD communityCount, ranIterations, didConverge ----

.Results [opts="header"]

| communityCount | ranIterations | didConverge | 2 | 3 | true

— As we can see from the example above the algorithm finds two communities and converges in three iterations. Note that we ran the algorithm `unweighted`.

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.mutate('myGraph', { mutateProperty: 'community' }) YIELD communityCount, ranIterations, didConverge ----

.Results [opts="header"]

| communityCount | ranIterations | didConverge | 2 | 3 | true

— The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `community` which stores the community ID for each node. To find out how to inspect the new schema of the in-memory graph, see Listing graphs.

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.write('myGraph', { writeProperty: 'community' }) YIELD communityCount, ranIterations, didConverge ----

.Results [opts="header"]

| communityCount | ranIterations | didConverge | 2 | 3 | true

— The returned result is the same as in the `stats` example. Additionally, each of the six nodes now has a new property `community` in the Neo4j database, containing the community ID for that node.

=== Weighted

The Label Propagation algorithm can also be configured to use node and/or relationship weights into account. By specifying a node weight via the `nodeWeightProperty` key, we can control the influence of a nodes community onto its neighbors. During the computation of the weight of a specific community, the node property will be multiplied by the weight of that nodes relationships.

When we created `myGraph`, we projected the relationship property `weight`. In order to tell the algorithm to consider this property as a relationship weight, we have to set the `relationshipWeightProperty` configuration parameter to `weight`.

[role=query-example] — .The following will run the algorithm on a graph with weighted relationships and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.stream('myGraph', { relationshipWeightProperty: 'weight' }) YIELD nodeId, communityId AS Community RETURN gds.util.asNode(nodeId).name AS Name, Community ORDER BY Community, Name ----

.Results [opts="header",cols="1,1"]

| Name | Community | "Bridget" | 2 | "Michael" | 2 | "Alice" | 4 | "Charles" | 4 | "Doug" | 4 | "Mark" | 4

— Compared to the unweighted run of the algorithm we still have two communities, but they contain two and four nodes respectively. Using the weighted relationships, the nodes `Alice` and `Charles` are now in the same community as there is a strong link between them.

NOTE: We have used the `stream` mode to demonstrate running the algorithm using weights, the configuration parameters are available for all the modes of the algorithm.

=== Seeded communities

At the beginning of the algorithm computation, every node is initialized with a unique label, and the labels propagate through the network.

An initial set of labels can be provided by setting the `seedProperty` configuration parameter. When we created `myGraph`, we projected the node property `seed_label`. We can use this node property as `seedProperty`.

The algorithm first checks if there is a seed label assigned to the node. If no seed label is present, the algorithm assigns new unique label to the node. Using this preliminary set of labels, it then sequentially updates each node's label to a new one, which is the most frequent label among its neighbors at every iteration of label propagation.

WARNING: The `consecutiveIds` configuration option cannot be used in combination with `seedProperty` in order to retain the seeding values.

[role=query-example] — .The following will run the algorithm with pre-defined labels: [source, cypher, role=noplay, indent=0] ---- CALL gds.labelPropagation.stream('myGraph', { seedProperty: 'seed_label' }) YIELD nodeId, communityId AS Community RETURN gds.util.asNode(nodeId).name AS Name, Community ORDER BY Community, Name ----

.Results [opts="header",cols="1,1"]

| Name | Community | "Charles" | 19 | "Doug" | 19 | "Mark" | 19 | "Alice" | 21 | "Bridget" | 21 | "Michael" | 21

— As we can see, the communities are based on the `seed_label` property, concretely `19` is from the node `Mark` and `21` from `Doug`.

NOTE: We have used the `stream` mode to demonstrate running the algorithm using `seedProperty`, this configuration parameter is available for all the modes of the algorithm.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Weakly Connected Components (WCC) algorithm in the Neo4j Graph Data Science library. = Weakly Connected Components :entity: node :result: component ID :algorithm: Weakly Connected Components

[abstract] — This section describes the Weakly Connected Components (WCC) algorithm in the Neo4j Graph Data Science library. — :directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

The WCC algorithm finds sets of connected nodes in an undirected graph, where all nodes in the same set form a connected component. WCC is often used early in an analysis to understand the structure of a graph. Using WCC to understand the graph structure enables running other algorithms independently on an identified cluster. As a preprocessing step for directed graphs, it helps quickly identify disconnected groups.

For more information on this algorithm, see:

* "An efficient domain-independent algorithm for detecting approximately duplicate database records". * One study uses WCC to work out how well connected the network is, and then to see whether the connectivity remains if 'hub' or 'authority' nodes are moved from the graph: "Characterizing and Mining Citation Graph of Computer Science Literature"

[NOTE] ==== Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read Memory Estimation. ====

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

.WCC syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ====== .Run WCC in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, componentId: Integer ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial component for a node. The property value needs to be a number. | threshold | Float | null | yes | The value of the weight above which the relationship is considered in the computation. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeId | Integer | Node ID. | componentId | Integer | Component ID.

======

[.include-with-stats] ====== .Run WCC in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.stats(graphName: String, configuration: Map) YIELD componentCount: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, componentDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial component for a node. The property value needs to be a number. | threshold | Float | null | yes | The value of the weight above which the relationship is considered in the computation. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | componentCount | Integer | The number of computed components. |

createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing component count and distribution statistics. | componentDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of component sizes. | configuration | Map | The configuration used for running the algorithm.

```
======

[.include-with-mutate] ====== .Run WCC in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ----
CALL gds.wcc.mutate( graphName: String, configuration: Map ) YIELD componentCount: Integer, nodePropertiesWritten:
Integer, createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, postProcessingMillis: Integer,
componentDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial component for a node. The property value needs to be a number. | threshold | Float | null | yes | The value of the weight above which the relationship is considered in the computation. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

```
.Results [opts="header",cols="1,1,6"]
```

| Name | Type | Description | componentCount | Integer | The number of computed components. | nodePropertiesWritten | Integer | The number of node properties written. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | postProcessingMillis | Integer | Milliseconds for computing component count and distribution statistics. | componentDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of component sizes. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ====== .Run WCC in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.write(graphName: String, configuration: Map) YIELD componentCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, postProcessingMillis: Integer, componentDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial component for a node. The property value needs to be a number. | threshold | Float | null | yes | The value of the weight above which the relationship is considered in the computation. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). | minComponentSize | Integer | 0 | yes | Only component ids of components with a size greater than or equal to the given value are written to Neo4j.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | componentCount | Integer | The number of computed components. | nodePropertiesWritten | Integer | The number of node properties written. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result back to Neo4j. | postProcessingMillis | Integer | Milliseconds for computing component count and distribution statistics. | componentDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of component sizes. | configuration | Map | The configuration used for running the algorithm.

====== ====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run WCC in write mode on an anonymous graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.write(configuration: Map) YIELD componentCount: Integer, nodePropertiesWritten: Integer, relationshipPropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, postProcessingMillis: Integer, componentDistribution: Map, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | seedProperty | String | n/a | yes | Used to set the initial component for a node. The property value needs to be a number. | threshold | Float | null | yes | The value of the weight above which the relationship is considered in the computation. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: user network :image-file: wcc-graph.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (nAlice:User {name: 'Alice'}), (nBridget:User {name: 'Bridget'}), (nCharles:User {name: 'Charles'}), (nDoug:User {name: 'Doug'}), (nMark:User {name: 'Mark'}), (nMichael:User {name: 'Michael'}),

(nAlice)-[:LINK {weight: 0.5}]→(nBridget), (nAlice)-[:LINK {weight: 4}]→(nCharles), (nMark)-[:LINK {weight: 1.1}]→(nDoug), (nMark)-[:LINK {weight: 2}]→(nMichael); ----

This graph has two connected components, each with three nodes. The relationships that connect the nodes in each component have a property `weight` which determines the strength of the relationship.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'User', 'LINK', { relationshipProperties: 'weight' }) ----

In the following examples we will demonstrate using the Weakly Connected Components algorithm on this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.write.estimate('myGraph', { writeProperty: 'component' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 6 | 4 | 176 | 176 | "176 Bytes"

—

=== Stream

:stream-details: For example, we can order the results to see the nodes that belong to the same component displayed next to each other. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.stream('myGraph') YIELD nodeId, componentId RETURN gds.util.asNode(nodeId).name AS name, componentId ORDER BY componentId, name ----

.Results [opts="header"]

| name | componentId | "Alice" | 0 | "Bridget" | 0 | "Charles" | 0 | "Doug" | 3 | "Mark" | 3 | "Michael" | 3

— The result shows that the algorithm identifies two components. This can be verified in the example graph.

The default behaviour of the algorithm is to run `unweighted`, e.g. without using `relationship` weights. The `weighted` option will be demonstrated in Weighted

=== Stats

:stats-syntax: algorithms-wcc-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm in `stats` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.stats('myGraph') YIELD componentCount ----

.Results [opts="header"]

| componentCount | 2

— The result shows that `myGraph` has two components and this can be verified by looking at the example graph.

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.mutate('myGraph', { mutateProperty: 'componentId' }) YIELD nodePropertiesWritten, componentCount; ----

.Results [opts="header"]

| nodePropertiesWritten | componentCount | 6 | 2

— === Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.write('myGraph', { writeProperty: 'componentId' }) YIELD nodePropertiesWritten, componentCount; ----

.Results [opts="header"]

| nodePropertiesWritten | componentCount | 6 | 2

— As we can see from the results, the nodes connected to one another are calculated by the algorithm as belonging to the same connected component.

=== Weighted

By configuring the algorithm to use a weight we can increase granularity in the way the algorithm calculates component assignment. We do this by specifying the property key with the `relationshipWeightProperty` configuration parameter. Additionally, we can specify a threshold for the weight value. Then, only weights greater than the threshold value will be considered by the algorithm. We do this by specifying the threshold value with the `threshold` configuration parameter.

If a relationship does not have the specified weight property, the algorithm falls back to using a default value of zero.

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.stream('myGraph', { relationshipWeightProperty: 'weight', threshold: 1.0 }) YIELD nodeId, componentId RETURN gds.util.asNode(nodeId).name AS Name, componentId AS ComponentId ORDER BY ComponentId, Name ----

.Results [opts="header",cols="1m,1m"]

| Name | ComponentId | "Alice" | 0 | "Charles" | 0 | "Bridget" | 1 | "Doug" | 3 | "Mark" | 3 | "Michael" | 3

— As we can see from the results, the node named 'Bridget' is now in its own component, due to its relationship weight being less than the configured threshold and thus ignored.

NOTE: We are using stream mode to illustrate running the algorithm as weighted or unweighted, all the other algorithm modes also support this configuration parameter.

=== Seeded components

It is possible to define preliminary component IDs for nodes using the `seedProperty` configuration parameter. This is helpful if we want to retain components from a previous run and it is known that no components have been split by removing relationships. The property value needs to be a number.

The algorithm first checks if there is a seeded component ID assigned to the node. If there is one, that component ID is used. Otherwise, a new unique component ID is assigned to the node.

Once every node belongs to a component, the algorithm merges components of connected nodes. When components are merged, the resulting component is always the one with the lower component ID. Note that the `consecutiveIds` configuration option cannot be used in combination with seeding in order to retain the seeding values.

[NOTE] ==== The algorithm assumes that nodes with the same seed value do in fact belong to the same component. If any two nodes in different components have the same seed, behavior is undefined. It is then recommended running WCC without seeds. ====

To demonstrate this in practice, we will go through a few steps:

. We will run the algorithm and write the results to Neo4j. . Then we will add another node to our graph, this node will not have the property computed in Step 1. . We will create a new in-memory graph that has the result from Step 1 as `nodeProperty` . And then we will run the algorithm again, this time in `stream` mode, and we will use the `seedProperty` configuration parameter.

We will use the weighted variant of WCC.

Step 1 [role=query-example, group=seeding] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.write('myGraph', { writeProperty: 'componentId', relationshipWeightProperty: 'weight', threshold: 1.0 }) YIELD nodePropertiesWritten, componentCount; ----

.Results [opts="header"]

| nodePropertiesWritten | componentCount | 6 | 3

— Step 2

After the algorithm has finished writing to Neo4j we want to create a new node in the database.

[role=query-example, no-result=true, group=seeding] — .The following will create a new node in the Neo4j graph, with no component ID: [source, cypher, role=noplay, indent=0] ---- MATCH (b:User {name: 'Bridget'}) CREATE (b)-[:LINK {weight: 2.0}]→(new:User {name: 'Mats'}) ---- — *Step 3*

Note, that we cannot use our already created graph as it does not contain the component id. We will therefore create a second in-memory graph that contains the previously computed component id.

[role=query-example, no-result=true, group=seeding] — .The following will create a new graph containing the previously computed component id: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('myGraph-seeded', 'User', 'LINK', { nodeProperties: 'componentId', relationshipProperties: 'weight' }) ---- — *Step 4*

[role=query-example, group=seeding] — .The following will run the algorithm in `stream` mode using `seedProperty`: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.stream('myGraph-seeded', { seedProperty: 'componentId', relationshipWeightProperty: 'weight', threshold: 1.0 }) YIELD nodeId, componentId RETURN gds.util.asNode(nodeId).name AS name, componentId ORDER BY componentId, name ----

.Results [opts="header"]

| name | componentId | "Alice" | 0 | "Charles" | 0 | "Bridget" | 1 | "Mats" | 1 | "Doug" | 3 | "Mark" | 3 | "Michael" | 3

— The result shows that despite not having the `seedProperty` when it was created, the node 'Mats' has been assigned to the same component as the node 'Bridget'. This is correct because these two nodes are connected.

=== Writing Seeded components

In the previous section we demonstrated the `seedProperty` usage in `stream` mode. It is also available in the other modes of the algorithm. Below is an example on how to use `seedProperty` in `write` mode. Note that the example below relies on Steps 1 - 3 from the previous section.

[role=query-example, group=seeding] — .The following will run the algorithm in `write` mode using `seedProperty`: [source, cypher, role=noplay, indent=0] ---- CALL gds.wcc.write('myGraph-seeded', { seedProperty: 'componentId', writeProperty: 'componentId', relationshipWeightProperty: 'weight', threshold: 1.0 }) YIELD nodePropertiesWritten, componentCount; ----

.Results [opts="header"]

| nodePropertiesWritten | componentCount | 1 | 3

— [NOTE] ==== If the `seedProperty` configuration parameter has the same value as `writeProperty`, the algorithm only writes properties for nodes where the component ID has changed. If they differ, the algorithm writes properties for all nodes. ====

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Triangle Count algorithm in the Neo4j Graph Data Science library. = Triangle Count :entity: node :result: triangle count :algorithm: Triangle Count

[abstract] — This section describes the Triangle Count algorithm in the Neo4j Graph Data Science library. — :undirected: :homogeneous: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

The Triangle Count algorithm counts the number of triangles for each node in the graph. A triangle is a set of three nodes where each node has a relationship to the other two. In graph theory terminology, this is sometimes referred to as a 3-clique. The Triangle Count algorithm in the GDS library only finds triangles in undirected graphs.

Triangle counting has gained popularity in social network analysis, where it is used to detect communities and measure the cohesiveness of those communities. It can also be used to determine the stability of a graph, and is often used as part of the computation of network indices, such as clustering coefficients. The Triangle Count algorithm is also used to compute the Local Clustering Coefficient.

For more information on this algorithm, see:

* Triangle count and clustering coefficient have been shown to be useful as features for classifying a given website as spam, or non-spam, content. This is described in "Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs".

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

WARNING: The named graphs must be projected in the `UNDIRECTED` orientation for the Triangle Count algorithm.

.Triangle Count syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ====== .Run Triangle Count in stream mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, triangleCount: Integer ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | **List of String** | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | maxDegree | Integer | 2^{63} - 1 | Yes | If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be `-1`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeId | Integer | Node ID. | triangleCount | Integer | Number of triangles the node is part of. Is `-1` if the node has been excluded from computation using the `maxDegree` configuration parameter.

======

[.include-with-stats] ====== .Run Triangle Count in stats mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.stats(graphName: String, configuration: Map) YIELD globalTriangleCount: Integer, nodeCount: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | **List of String** | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | maxDegree | Integer | 2^{63} - 1 | Yes | If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be `-1`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | globalTriangleCount | Integer | Total number of triangles in the graph. | nodeCount | Integer | Number of nodes in the graph. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | configuration | Map | The configuration used for running the algorithm.

```
======

[.include-with-mutate] ====== .Run Triangle Count in mutate mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.mutate( graphName: String, configuration: Map ) YIELD globalTriangleCount: Integer, nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | **relationshipTypes** | **List of String** | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | maxDegree | Integer | 2^{63} - 1 | Yes | If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be -1.

```
.Results [opts="header",cols="1,1,6"]
```

| Name | Type | Description | globalTriangleCount | Integer | Total number of triangles in the graph. | nodeCount | Integer | Number of nodes in the graph. | nodePropertiesWritten | Integer | Number of properties added to the in-memory graph. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | configuration | Map | The configuration used for running the algorithm.

```
======

[.include-with-write] ====== .Run Triangle Count in write mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.write( graphName: String, configuration: Map ) YIELD globalTriangleCount: Integer, nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** |

relationshipTypes | List of String | [''] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | maxDegree | Integer | 2^{63} - 1 | Yes | If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be `-1`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | globalTriangleCount | Integer | Total number of triangles in the graph. | nodeCount | Integer | Number of nodes in the graph. | nodePropertiesWritten | Integer | Number of properties written to Neo4j. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | writeMillis | Integer | Milliseconds for writing results back to Neo4j. | configuration | Map | The configuration used for running the algorithm.

====== ====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run Triangle Count in write mode on an anonymous graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.write(configuration: Map) YIELD globalTriangleCount: Integer, nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | maxDegree | Integer | 2^{63} - 1 | Yes | If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be -1.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

=== Triangles listing

In addition to the standard execution modes there is an `alpha` procedure `gds.alpha.triangles` that can be used to list all triangles in the graph.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

.The following will return a stream of node IDs for each triangle: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.triangles(graphName: String, configuration: Map) YIELD nodeA, nodeB, nodeC ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeA | Integer | The ID of the first node in the given triangle. | nodeB | Integer | The ID of the second node in the given triangle. | nodeC | Integer | The ID of the third node in the given triangle.

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: triangle_count.png In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:Person {name: 'Alice'}), (michael:Person {name: 'Michael'}), (karin:Person {name: 'Karin'}), (chris:Person {name: 'Chris'}), (will:Person {name: 'Will'}), (mark:Person {name: 'Mark'}),

(michael)-[:KNOWS]→(karin), (michael)-[:KNOWS]→(chris), (will)-[:KNOWS]→(michael), (mark)-[:KNOWS]→(michael), (mark)-[:KNOWS]→(will), (alice)-[:KNOWS]→(michael), (will)-[:KNOWS]→(chris), (chris)-[:KNOWS]→(karin) ----

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. For the relationships we must use the `UNDIRECTED` orientation. This is because the Triangle Count algorithm is defined only for undirected graphs.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Person', { KNOWS: { orientation: 'UNDIRECTED' } }) ----

WARNING: The Triangle Count algorithm requires the graph to be created using the `UNDIRECTED` orientation for relationships.

In the following examples we will demonstrate using the Triangle Count algorithm on this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.write.estimate('myGraph', { writeProperty: 'triangleCount' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 6 | 16 | 152 | 152 | "152 Bytes"

— Note that the relationship count is 16 although we only created 8 relationships in the original Cypher statement. This is because we used the `UNDIRECTED` orientation, which will project each relationship in each direction, effectively doubling the number of relationships.

=== Stream

:stream-details: For example, we can order the results to find the nodes with the highest triangle count. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

.The following will run the algorithm in `stream` mode: [role=query-example] — [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.stream('myGraph') YIELD nodeId, triangleCount RETURN gds.util.asNode(nodeId).name AS name, triangleCount ORDER BY triangleCount DESC ----

.Results [opts="header",cols="1,1"]

| name | triangleCount | "Michael" | 3 | "Chris" | 2 | "Will" | 2 | "Karin" | 1 | "Mark" | 1 | "Alice" | 0

— Here we find that the 'Michael' node has the most triangles. This can be verified in the example graph. Since the 'Alice' node only `KNOWS` one other node, it can not be part of any triangle, and indeed the algorithm reports a count of zero.

=== Stats

:stats-details: The summary result contains the global triangle count, which is the total number of triangles in the entire graph. :stats-syntax: algorithms-triangle-count-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

.The following will run the algorithm in `stats` mode: [role=query-example] — [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.stats('myGraph') YIELD globalTriangleCount, nodeCount ----

.Results [opts="header"]

| globalTriangleCount | nodeCount | 3 | 6

— Here we can see that the graph has six nodes with a total number of three triangles. Comparing this to the stream example we can see that the 'Michael' node has a triangle count equal to the global triangle count. In other words, that node is part of all of the triangles in the graph and thus has a very central position in the graph.

=== Mutate

:mutate-details: For example, using the triangle count to compute the local clustering coefficient. The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.mutate('myGraph', { mutateProperty: 'triangles' }) YIELD globalTriangleCount, nodeCount ----

.Results [opts="header"]

| globalTriangleCount | nodeCount | 3 | 6

— The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `triangles` which stores the triangle count for each node. To find out how to inspect the new schema of the in-memory graph, see Listing graphs.

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.write('myGraph', { writeProperty: 'triangles' }) YIELD globalTriangleCount, nodeCount ----

.Results [opts="header"]

| globalTriangleCount | nodeCount | 3 | 6

— The returned result is the same as in the `stats` example. Additionally, each of the six nodes now has a new property `triangles` in the Neo4j database, containing the triangle count for that node.

=== Maximum Degree

The Triangle Count algorithm supports a `maxDegree` configuration parameter that can be used to exclude nodes from processing if their degree is greater than the configured value. This can be useful to speed up the computation when there are nodes with a very high degree (so-called super nodes) in the graph. Super nodes have a great impact on the performance of the Triangle Count algorithm. To learn about the degree distribution of your graph, see Listing graphs.

The nodes excluded from the computation get assigned a triangle count of `-1`.

.The following will run the algorithm in `stream` mode with the `maxDegree` parameter: [role=query-example] — [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.stream('myGraph', { maxDegree: 4 }) YIELD nodeId, triangleCount RETURN gds.util.asNode(nodeId).name AS name, triangleCount ORDER BY name ASC ----

.Results [opts="header"]

| name | triangleCount | "Alice" | 0 | "Chris" | 0 | "Karin" | 0 | "Mark" | 0 | "Michael" | -1 | "Will" | 0

— Running the algorithm on the example graph with `maxDegree: 4` excludes the 'Michael' node from the computation, as it has a degree of 5.

As this node is part of all the triangles in the example graph excluding it results in no triangles.

== Triangles listing

It is also possible to list all the triangles in the graph. To do this we make use of the `alpha` procedure `gds.alpha.triangles`.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

[role=query-example] — .The following will compute a stream of node IDs for each triangle and return the name property of the nodes: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.triangles('myGraph') YIELD nodeA, nodeB, nodeC RETURN gds.util.asNode(nodeA).name AS nodeA, gds.util.asNode(nodeB).name AS nodeB, gds.util.asNode(nodeC).name AS nodeC ----

.Results [opts="header"]

| nodeA | nodeB | nodeC | "Michael" | "Karin" | "Chris" | "Michael" | "Chris" | "Will" | "Michael" | "Will" | "Mark"

— We can see that there are three triangles in the graph: "Will, Michael, and Chris", "Will, Mark, and Michael", and "Michael, Karin, and Chris". The node "Alice" is not part of any triangle and thus does not appear in the triangles listing.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Local Clustering Coefficient algorithm in the Neo4j Graph Data Science library. = Local Clustering Coefficient :entity: node :result: local clustering coefficient :algorithm: Local Clustering Coefficient

[abstract] — This section describes the Local Clustering Coefficient algorithm in the Neo4j Graph Data Science library. — :undirected: :homogeneous: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

The Local Clustering Coefficient algorithm computes the local clustering coefficient for each node in the graph. The local clustering coefficient C_n of a node *n* describes the likelihood that the neighbours of *n* are also connected. To compute C_n we use the number of triangles a node is a part of T_n, and the degree of the node d_n. The formula to compute the local clustering coefficient is as follows:

image::lcc_formula.svg[align="center"]

As we can see the triangle count is required to compute the local clustering coefficient. To do this the Triangle Count algorithm is utilised.

Additionally, the algorithm can compute the *average clustering coefficient* for the whole graph. This is the normalised sum over all the local clustering coefficients.

For more information, see Clustering Coefficient.

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

.Local Clustering Coefficient syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ====== .Run Local Clustering Coefficient in stream mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, localClusteringCoefficient: Double ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | triangleCountProperty | String | n/a | Yes | Node property that contains pre-computed triangle count.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | localClusteringCoefficient | Double | Local clustering coefficient.

======

[.include-with-stats] ====== .Run Local Clustering Coefficient in stats mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.stats(graphName: String, configuration: Map) YIELD averageClusteringCoefficient: Double, nodeCount: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | triangleCountProperty | String | n/a | Yes | Node property that contains pre-computed triangle count.

.Results [opts="header"]

| Name | Type | Description | averageClusteringCoefficient | Double | The average clustering coefficient. | nodeCount | Integer | Number of nodes in the graph. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | configuration | Map | The configuration used for running the algorithm.

```
======

[.include-with-mutate] ====== .Run Local Clustering Coefficient in mutate mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.mutate( graphName: String, configuration: Map ) YIELD averageClusteringCoefficient: Double, nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | triangleCountProperty | String | n/a | Yes | Node property that contains pre-computed triangle count.

```
.Results [opts="header"]
```

| Name | Type | Description | averageClusteringCoefficient | Double | The average clustering coefficient. | nodeCount | Integer | Number of nodes in the graph. | nodePropertiesWritten | Integer | Number of properties added to the in-memory graph. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | configuration | Map | The configuration used for running the algorithm.

```
======

[.include-with-write] ====== .Run Local Clustering Coefficient in write mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.write( graphName: String, configuration: Map ) YIELD averageClusteringCoefficient: Double, nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. |

relationshipTypes | List of String | [''] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | triangleCountProperty | String | n/a | Yes | Node property that contains pre-computed triangle count.

.Results [opts="header"]

| Name | Type | Description | averageClusteringCoefficient | Double | The average clustering coefficient. | nodeCount | Integer | Number of nodes in the graph. | nodePropertiesWritten | Integer | Number of properties written to Neo4j. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | writeMillis | Integer | Milliseconds for writing results back to Neo4j. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run Local Clustering Coefficient in write mode on an anonymous graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.write(configuration: Map) YIELD averageClusteringCoefficient: Double, nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes |

The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | triangleCountProperty | String | n/a | Yes | Node property that contains pre-computed triangle count.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: triangle_count.png In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:Person {name: 'Alice'}), (michael:Person {name: 'Michael'}), (karin:Person {name: 'Karin'}), (chris:Person {name: 'Chris'}), (will:Person {name: 'Will'}), (mark:Person {name: 'Mark'}),

(michael)-[:KNOWS]→(karin), (michael)-[:KNOWS]→(chris), (will)-[:KNOWS]→(michael), (mark)-[:KNOWS]→(michael), (mark)-[:KNOWS]→(will), (alice)-[:KNOWS]→(michael), (will)-[:KNOWS]→(chris), (chris)-[:KNOWS]→(karin) ----

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. For the relationships we must use the `UNDIRECTED` orientation. This is because the Local Clustering Coefficient algorithm is defined only for undirected graphs.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Person', { KNOWS: { orientation: 'UNDIRECTED' } }) ----

WARNING: The Local Clustering Coefficient algorithm requires the graph to be created using the `UNDIRECTED` orientation for relationships.

In the following examples we will demonstrate using the Local Clustering Coefficient algorithm on 'myGraph'.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.write.estimate('myGraph', { writeProperty: 'localClusteringCoefficient' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 6 | 16 | 296 | 296 | "296 Bytes"

— Note that the relationship count is 16 although we only created 8 relationships in the original Cypher statement. This is because we used the `UNDIRECTED` orientation, which will project each relationship in each direction, effectively doubling the number of relationships.

=== Stream

:stream-details: For example, we can order the results to find the nodes with the highest local clustering coefficient. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm in `stream` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.stream('myGraph') YIELD nodeId, localClusteringCoefficient RETURN gds.util.asNode(nodeId).name AS name, localClusteringCoefficient ORDER BY localClusteringCoefficient DESC ----

.Results [opts="header"]

| name | localClusteringCoefficient | "Karin" | 1.0 | "Mark" | 1.0 | "Chris" | 0.6666666666666666 | "Will" | 0.6666666666666666 | "Michael" | 0.3 | "Alice" | 0.0

— From the results we can see that the nodes 'Karin' and 'Mark' have the highest local clustering coefficients. This shows that they are the best at introducing their friends - all the people who know them, know each other! This can be verified in the example graph.

=== Stats

:stats-details: The summary result contains the avearage clustering coefficient of the graph, which is the normalised sum over all local clustering coefficients. :stats-syntax: algorithms-local-clustering-coefficient-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

.The following will run the algorithm in `stats` mode: [role=query-example] — [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.stats('myGraph') YIELD averageClusteringCoefficient, nodeCount ----

.Results [opts="header"]

| averageClusteringCoefficient | nodeCount | 0.6055555555555555 | 6

— The result shows that on average each node of our example graph has approximately 60% of its neighbours connected.

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.mutate('myGraph', { mutateProperty: 'localClusteringCoefficient' }) YIELD averageClusteringCoefficient, nodeCount ----

.Results [opts="header"]

| averageClusteringCoefficient | nodeCount | 0.6055555555555555 | 6

— The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `localClusteringCoefficient` which stores the local clustering coefficient for each node. To find out how to inspect the new schema of the in-memory graph, see Listing graphs.

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.write('myGraph', { writeProperty: 'localClusteringCoefficient' }) YIELD averageClusteringCoefficient, nodeCount ----

.Results [opts="header",cols="1,1"]

| averageClusteringCoefficient | nodeCount | 0.6055555555555555 | 6

— The returned result is the same as in the `stats` example. Additionally, each of the six nodes now has a new property `localClusteringCoefficient` in the Neo4j database, containing the local clustering coefficient for that node.

=== Pre-computed Counts

By default, the Local Clustering Coefficient algorithm executes Triangle Count as part of its computation. It is also possible to avoid the triangle count computation by configuring the Local Clustering Coefficient algorithm to read the triangle count from a node property. In order to do that we specify the `triangleCountProperty` configuration parameter. Please note that the Local Clustering Coefficient algorithm depends on the property holding actual triangle counts and not another number for the results to be actual local clustering coefficients.

To illustrate this we make use of the Triangle Count algorithm in `mutate` mode. The Triangle Count algorithm is going to store its result back into 'myGraph'. It is also possible to obtain the property value from the Neo4j database using a graph projection with a node property when creating the in-memory graph.

[role=query-example, no-result=true, group=triangleCountProperty] — .The following computes the triangle counts and stores the result into the in-memory graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.triangleCount.mutate('myGraph', { mutateProperty: 'triangles' }) ---- — [role=query-example, group=triangleCountProperty] — .The following will run the algorithm in `stream` mode using pre-computed triangle counts: [source, cypher, role=noplay, indent=0] ---- CALL gds.localClusteringCoefficient.stream('myGraph', { triangleCountProperty: 'triangles' }) YIELD nodeId, localClusteringCoefficient RETURN gds.util.asNode(nodeId).name AS name, localClusteringCoefficient ORDER BY localClusteringCoefficient DESC ----

.Results [opts="header"]

| name | localClusteringCoefficient | "Karin" | 1.0 | "Mark" | 1.0 | "Chris" | 0.6666666666666666 | "Will" | 0.6666666666666666 | "Michael" | 0.3 | "Alice" | 0.0

— As we can see the results are the same as in the `stream` example where we did not specify a `triangleCountProperty`.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the K-1 Coloring algorithm in the Neo4j Graph Data Science library. [.beta] = K-1 Coloring

[abstract] — This section describes the K-1 Coloring algorithm in the Neo4j Graph Data Science library. — [.beta-symbol] [.tier-note] This algorithm is in the beta tier. For more information on algorithm tiers, see Algorithms.

== Introduction

The K-1 Coloring algorithm assigns a color to every node in the graph, trying to optimize for two objectives:

1. To make sure that every neighbor of a given node has a different color than the node itself. 2. To use as few colors as possible.

Note that the graph coloring problem is proven to be NP-complete, which makes it intractable on anything but trivial graph sizes. For that reason the implemented algorithm is a greedy algorithm. Thus it is neither guaranteed that the result is an optimal solution, using as few colors as theoretically possible, nor does it always produce a correct result where no two neighboring nodes have different colors. However the precision of the latter can be controlled by the number of iterations this algorithm runs.

For more information on this algorithm, see:

* Çatalyürek, Ümit V., et al. "Graph coloring algorithms for multi-core and massively multithreaded architectures." * https://en.wikipedia.org/wiki/Graph_coloring#Vertex_coloring

[NOTE] ==== Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read Memory Estimation. ====

== Syntax

.K-1 Coloring syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ====== .The following describes the API for running the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.k1coloring.stream(graphName: String, configuration: Map) YIELD nodeId, color ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | null | yes | The name of an existing graph on which to run the algorithm. If no graph name is provided, the configuration map must contain configuration for creating a graph. | configuration | Map | {} | yes | Additional configuration, see below.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String | null | yes | The projection of nodes to use when creating the implicit graph. | relationshipProjection | String | null | yes | The projection of relationships to use when creating the implicit graph. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. This is dependent on the Neo4j edition; for more information, see CPU. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | maxIterations | Integer | 10 | yes | The maximum number of iterations of K1 Coloring to

run.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeId | Integer | The ID of the Node | color | Integer | The color of the Node

====== [.include-with-stats] ====== .The following describes the API for running the algorithm and returning the computation statistics: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.k1coloring.stats(graphName: String, configuration: Map) YIELD nodeCount, colorCount, ranIterations, didConverge, configuration, createMillis, computeMillis ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String or Map | n/a | no | Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String | null | yes | The projection of nodes to use when creating the implicit graph. | relationshipProjection | String | null | yes | The projection of relationships to use when creating the implicit graph. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. This is dependent on the Neo4j edition; for more information, see CPU. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | maxIterations | Integer | 10 | yes | The maximum number of iterations of K1 Coloring to run.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes considered. | ranIterations | Integer | The actual number of iterations the algorithm ran. | didConverge | Boolean | An indicator of whether the algorithm found a correct coloring. | colorCount | Integer | The number of colors used. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-mutate] ====== .The following describes the API for running the algorithm and mutating the in-memory graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.k1coloring.mutate(graphName: String, configuration: Map) YIELD nodeCount, colorCount, ranIterations, didConverge, configuration, createMillis, computeMillis, mutateMillis ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String or Map | n/a | no | Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

The configuration for the `mutate` mode is similar to the `write` mode. Instead of specifying a `writeProperty`, we need to specify a `mutateProperty`. Also, specifying `writeConcurrency` is not possible in `mutate` mode.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes considered. | ranIterations | Integer | The actual number of iterations the algorithm ran. | didConverge | Boolean | An indicator of whether the algorithm found a correct coloring. | colorCount | Integer | The number of colors used. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ====== .The following describes the API for running the algorithm and writing results back to Neo4j: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.k1coloring.write(graphName: String, configuration: Map) YIELD nodeCount, colorCount, ranIterations, didConverge, configuration, createMillis, computeMillis, writeMillis ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String or Map | n/a | no | Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String | null | yes | The projection of nodes to use when creating the implicit graph. | relationshipProjection | String | null | yes | The projection of relationships to use when creating the implicit graph. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. This is dependent on the Neo4j edition; for more information, see CPU. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result. | maxIterations | Integer | 10 | yes | The maximum number of iterations of K1 Coloring to run. | writeProperty | String | n/a | no | The node property this procedure writes the color to.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes considered. | ranIterations | Integer | The actual number of iterations the algorithm ran. | didConverge | Boolean | An indicator of whether the algorithm found a correct coloring. | colorCount | Integer | The number of colors used. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | configuration | Map | The configuration used for running the algorithm.

======

====

== Examples

Consider the graph created by the following Cypher statement:

[source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:User {name: 'Alice'}), (bridget:User {name: 'Bridget'}), (charles:User {name: 'Charles'}), (doug:User {name: 'Doug'}),

(alice)-[:LINK]→(bridget), (alice)-[:LINK]→(charles), (alice)-[:LINK]→(doug), (bridget)-[:LINK]→(charles) ----

This graph has a super node with name "Alice" that connects to all other nodes. It should therefore not be possible for any other node to be assigned the same color as the Alice node.

[source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'User', { LINK : { orientation: 'UNDIRECTED' } }) ---- We can now go ahead and create an in-memory graph with all the `User` nodes and the `LINK` relationships with `UNDIRECTED` orientation.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create the graph and store it in the graph catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.create('myGraph', 'Person', 'LIKES') ----

In the following examples we will demonstrate using the K-1 Coloring algorithm on this graph.

[role=query-example] — .Running the K-1 Coloring algorithm in stream mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.k1coloring.stream('myGraph') YIELD nodeId, color RETURN gds.util.asNode(nodeId).name AS name, color ORDER BY name ----

.Results [opts="header",cols="1m,1m"]

| name | color | "Alice" | 0 | "Bridget" | 1 | "Charles" | 2 | "Doug" | 1

— It is also possible to write the assigned colors back to the database using the `write` mode.

[role=query-example] — .Running the K-1 Coloring algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.k1coloring.write('myGraph', {writeProperty: 'color'}) YIELD nodeCount, colorCount, ranIterations, didConverge ----

.Results [opts="header",cols="1m,1m,1m,1m"]

| nodeCount | colorCount | ranIterations | didConverge | 4 | 3 | 1 | true

— When using `write` mode the procedure will return information about the algorithm execution. In this example we return the number of processed nodes, the number of colors used to color the graph, the number of iterations and information whether the algorithm converged.

To instead mutate the in-memory graph with the assigned colors, the `mutate` mode can be used as follows.

[role=query-example] — .Running the K-1 Coloring algorithm in mutate mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.k1coloring.mutate('myGraph', {mutateProperty: 'color'}) YIELD nodeCount, colorCount, ranIterations, didConverge ----

.Results [opts="header",cols="1m,1m,1m,1m"]

| nodeCount | colorCount | ranIterations | didConverge | 4 | 3 | 1 | true

— Similar to the `write` mode, `stats` mode can run the algorithm and return only the execution statistics without persisting the results.

[role=query-example] — .Running the K-1 Coloring algorithm in stats mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.k1coloring.stats('myGraph') YIELD nodeCount, colorCount, ranIterations, didConverge ----

.Results [opts="header",cols="1m,1m,1m,1m"]

| nodeCount | colorCount | ranIterations | didConverge | 4 | 3 | 1 | true

— :leveloffset: 2

:leveloffset: +3

:description: This section describes the Modularity Optimization algorithm in the Neo4j Graph Data Science library. [.beta] = Modularity Optimization

[abstract] — This section describes the Modularity Optimization algorithm in the Neo4j Graph Data Science library. — [.beta-symbol] [.tier-note] This algorithm is in the beta tier. For more information on algorithm tiers, see Algorithms.

== Introduction

The Modularity Optimization algorithm tries to detect communities in the graph based on their *modularity*. *Modularity* is a measure of the structure of a graph, measuring the density of connections within a module or community. Graphs with a high modularity score will have many connections within a community but only few pointing outwards to other communities. The algorithm will explore for every node if its modularity score might increase if it changes its community to one of its neighboring nodes.

For more information on this algorithm, see:

* MEJ Newman, M Girvan "Finding and evaluating community structure in networks" * https://en.wikipedia.org/wiki/Modularity_(networks)

[NOTE] ==== Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read Memory Estimation. ====

== Syntax

.Modularity Optimization syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ====== .Run Modularity Optimization in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.modularityOptimization.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, communityId: Integer ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String or Map | n/a | no | Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

.General configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | concurrency | Integer | 4 | yes | The number of concurrent

threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result (applicable in WRITE mode). | nodeProjection | Map or List | null | yes | The node projection used for implicit graph loading or filtering nodes of an explicitly loaded graph. | relationshipProjection | Map or List | null | yes | The relationship projection used for implicit graph loading or filtering relationship of an explicitly loaded graph. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for implicit graph loading via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for implicit graph loading via a Cypher projection. | nodeProperties | Map or List | null | yes | The node properties to load during implicit graph loading. | relationshipProperties | Map or List | null | yes | The relationship properties to load during implicit graph loading.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | 10 | yes | The maximum number of iterations to run. | tolerance | Float | 0.0001 | yes | Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns. | seedProperty | String | n/a | yes | Used to define initial set of labels (must be a number). | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID | communityId | Integer | Community ID

| ======

[.include-with-mutate] ====== .Run Modularity Optimization in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.modularityOptimization.mutate(graphName: String | Map, configuration: Map}) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, communityCount: Integer, communityDistribution: Map, modularity: Float, ranIterations: Integer, didConverge: Boolean, nodes: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String or Map | n/a | no | Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

The configuration for the `mutate` mode is similar to the `write` mode. Instead of specifying a `writeProperty`, we need to specify a `mutateProperty`. Also, specifying `writeConcurrency` is not possible in `mutate` mode.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | postProcessingMillis | Integer | Milliseconds for computing percentiles and community count. | nodes | Integer | The number of nodes considered. | didConverge | Boolean | True if the algorithm

did converge to a stable modularity score within the provided number of maximum iterations. | ranIterations | Integer | The number of iterations run. | modularity | Float | The final modularity score. | communityCount | Integer | The number of communities found. | communityDistribution | Map | The containing min, max, mean as well as 50, 75, 90, 95, 99 and 999 percentile of community size. | configuration | Map | The configuration used for running the algorithm.

====== [.include-with-write] ====== .Run Modularity Optimization in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.modularityOptimization.write(graphName: String	Map, configuration: Map}) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, communityCount: Integer, communityDistribution: Map, modularity: Float, ranIterations: Integer, didConverge: Boolean, nodes: Integer, configuration: Map ---- .Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String or Map | n/a | no | Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

.General configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result (applicable in WRITE mode). | nodeProjection | Map or List | null | yes | The node projection used for implicit graph loading or filtering nodes of an explicitly loaded graph. | relationshipProjection | Map or List | null | yes | The relationship projection used for implicit graph loading or filtering relationship of an explicitly loaded graph. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for implicit graph loading via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for implicit graph loading via a Cypher projection. | nodeProperties | Map or List | null | yes | The node properties to load during implicit graph loading. | relationshipProperties | Map or List | null | yes | The relationship properties to load during implicit graph loading.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | seedProperty | String | n/a | yes | Used to set the initial community for a node. The property value needs to be a number. | writeProperty | String | n/a | yes | The property name written back the ID of the partition particular node belongs to. | maxIterations | Integer | 10 | yes | The maximum number of iterations that the modularity optimization will run for each level. | tolerance | Float | 0.0001 | yes | Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns. | consecutiveIds | Boolean | false | yes | Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back. | postProcessingMillis | Integer | Milliseconds for computing percentiles and community count. | nodes | Integer | The number of nodes considered. | didConverge | Boolean | True if the algorithm did converge to a stable modularity score within the provided number of maximum iterations. | ranIterations | Integer | The number of iterations run. | modularity | Float | The final modularity score. | communityCount | Integer | The number of communities found. | communityDistribution | Map | The containing min, max, mean as well as 50, 75, 90, 95, 99 and 999 percentile of community size. | configuration | Map | The configuration used for running the algorithm.

====== ====

== Examples

Consider the graph created by the following Cypher statement:

[source, cypher, role=noplay setup-query, indent=0] ---- CREATE (a:Person {name:'Alice'}) , (b:Person {name:'Bridget'}) , (c:Person {name:'Charles'}) , (d:Person {name:'Doug'}) , (e:Person {name:'Elton'}) , (f:Person {name:'Frank'}) , (a)-[:KNOWS {weight: 0.01}]→(b) , (a)-[:KNOWS {weight: 5.0}]→(e) , (a)-[:KNOWS {weight: 5.0}]→(f) , (b)-[:KNOWS {weight: 5.0}]→(c) , (b)-[:KNOWS {weight: 5.0}]→(d) , (c)-[:KNOWS {weight: 0.01}]→(e) , (f)-[:KNOWS {weight: 0.01}]→(d) ----

This graph consists of two center nodes "Alice" and "Bridget" each of which have two more neighbors. Additionally, each neighbor of "Alice" is connected to one of the neighbors of "Bridget". Looking at the weights of the relationships, it can be seen that the connections from the two center nodes to their neighbors are very strong, while connections between those groups are weak. Therefore the Modularity Optimization algorithm should detect two communities: "Alice" and "Bob" together with their neighbors respectively.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create the graph and store it in the graph catalog. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Person', { KNOWS: { type: 'KNOWS', orientation: 'UNDIRECTED', properties: ['weight'] } }) ----

The following example demonstrates using the Modularity Algorithm on this weighted graph.

[role=query-example] — .Running the Modularity Optimization algorithm in stream mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.modularityOptimization.stream('myGraph', { relationshipWeightProperty: 'weight' }) YIELD nodeId, communityId RETURN gds.util.asNode(nodeId).name AS name, communityId ORDER BY name ----

.Results [opts="header"]

| name | communityId | "Alice" | 4 | "Bridget" | 1 | "Charles" | 1 | "Doug" | 1 | "Elton" | 4 | "Frank" | 4

— It is also possible to write the assigned community ids back to the database using the `write` mode.

[role=query-example] — .Running the Modularity Optimization algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.modularityOptimization.write('myGraph', { relationshipWeightProperty: 'weight', writeProperty: 'community' }) YIELD nodes, communityCount, ranIterations, didConverge ----

.Results [opts="header"]

| nodes | communityCount | ranIterations | didConverge | 6 | 2 | 3 | true

— When using `write` mode the procedure will return information about the algorithm execution. In this example we return the number of processed nodes, the number of communities assigned to the nodes in the graph, the number of iterations and information whether the algorithm converged.

Running the algorithm without specifying the `relationshipWeightProperty` will default all relationship weights to 1.0.

To instead mutate the in-memory graph with the assigned community ids, the `mutate` mode is used.

[role=query-example] — .Running the Modularity Optimization algorithm in mutate mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.modularityOptimization.mutate('myGraph', { relationshipWeightProperty: 'weight', mutateProperty: 'community' }) YIELD nodes, communityCount, ranIterations, didConverge ----

.Results [opts="header"]

| nodes | communityCount | ranIterations | didConverge | 6 | 2 | 3 | true

— When using `mutate` mode the procedure will return information about the algorithm execution as in `write` mode.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Strongly Connected Components algorithm in the Neo4j Graph Data Science library. [.alpha] = Strongly Connected Components

[abstract] — This section describes the Strongly Connected Components algorithm in the Neo4j Graph Data Science library. — The Strongly Connected Components (SCC) algorithm finds maximal sets of connected nodes in a directed graph. A set is considered a strongly connected component if there is a directed path between each pair of nodes within the set. It is often used early in a graph analysis process to help us get an idea of how our graph is structured.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

SCC is one of the earliest graph algorithms, and the first linear-time algorithm was described by Tarjan in 1972. Decomposing a directed graph into its strongly connected components is a classic application of the depth-first search algorithm.

== Use-cases - when to use the Strongly Connected Components algorithm

* In the analysis of powerful transnational corporations, SCC can be used to find the set of firms in which every member owns directly and/or indirectly owns shares in every other member. Although it has benefits, such as reducing transaction costs and increasing trust, this type of structure can weaken market competition. Read more in "The Network of Global Corporate Control". * SCC can be used to compute the connectivity of different network configurations when measuring routing performance in multihop wireless networks. Read more in "Routing performance in the presence of unidirectional links in multihop wireless networks" * Strongly Connected Components algorithms can be used as a first step in many graph algorithms that work only on strongly connected graph. In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages, or play common games. The SCC algorithms can be used to find such groups, and suggest the commonly liked pages or games to the people in the group who have not yet liked those pages or games.

== Syntax

Map, configuration: Map) YIELD createMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | writeProperty | String | 'componentId' | yes | The property name written back to. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back. | postProcessingMillis | Integer | Milliseconds for computing percentiles and community count. | nodes | Integer | The number of nodes considered. | communityCount | Integer | The number of communities found. | p1 | Float | The 1 percentile of community size. | p5 | Float | The 5 percentile of community size. | p10 | Float | The 10 percentile of community size. | p25 | Float | The 25 percentile of community size. | p50 | Float | The 50 percentile of community size. | p75 | Float | The 75 percentile of community size. | p90 | Float | The 90 percentile of community size. | p95 | Float | The 95 percentile of community size. | p99 | Float | The 99 percentile of community size. | p100 | Float | The 100 percentile of community size. | writeProperty | String | The property name written back to.

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.scc.stream(graphName: String, configuration: Map) YIELD nodeId, componentId ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | componentId | Integer | Component ID.

== Strongly Connected Components algorithm example

image::strongly_connected_components.png[]

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (nAlice:User {name:'Alice'}) CREATE (nBridget:User {name:'Bridget'}) CREATE (nCharles:User {name:'Charles'}) CREATE (nDoug:User {name:'Doug'}) CREATE (nMark:User {name:'Mark'}) CREATE (nMichael:User {name:'Michael'})

CREATE (nAlice)-[:FOLLOW]→(nBridget) CREATE (nAlice)-[:FOLLOW]→(nCharles) CREATE (nMark)-[:FOLLOW]→(nDoug) CREATE (nMark)-[:FOLLOW]→(nMichael) CREATE (nBridget)-[:FOLLOW]→(nMichael) CREATE (nDoug)-[:FOLLOW]→(nMark) CREATE (nMichael)-[:FOLLOW]→(nAlice) CREATE (nAlice)-[:FOLLOW]→(nMichael) CREATE (nBridget)-[:FOLLOW]→(nAlice) CREATE (nMichael)-[:FOLLOW]→(nBridget); ----

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.scc.write({ nodeProjection: 'User', relationshipProjection: 'FOLLOW', writeProperty: 'componentId' }) YIELD setCount, maxSetSize, minSetSize; ----

.Results [opts="header",cols="1m,1m,1m"]

| setCount | maxSetSize | minSetSize | 3 | 3 | 1

.The following will run the algorithm and stream back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.scc.stream({ nodeProjection: 'User', relationshipProjection: 'FOLLOW' }) YIELD nodeId, componentId RETURN gds.util.asNode(nodeId).name AS Name, componentId AS Component ORDER BY Component DESC ----

.Results [opts="header",cols="1m,1m"]

| Name | Component | "Doug" | 3 | "Mark" | 3 | "Charles" | 2 | "Alice" | 0 | "Bridget" | 0 | "Michael" | 0

We have 3 strongly connected components in our sample graph.

The first, and biggest, component has members Alice, Bridget, and Michael, while the second component has Doug and Mark. Charles ends up in his own component because there isn't an outgoing relationship from that node to any of the others.

.The following will find the largest partition: [source, cypher, role=noplay, indent=0] ---- MATCH (u:User) RETURN u.componentId AS Component, count(*) AS ComponentSize ORDER BY ComponentSize DESC LIMIT 1 ----

.Results [opts="header",cols="1m,1m"]

| Component | ComponentSize | 0 | 3

== Cypher projection

If node labels and relationship types are not selective enough to project a graph, you can use Cypher queries instead. Cypher projections can also be used to run algorithms on a virtual graph. You can learn more in the Creating graphs using Cypher section of the manual.

.Use `nodeQuery` and `relationshipQuery` in the config:

[source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.scc.stream({ nodeQuery: 'MATCH (u:User) RETURN id(u) AS id', relationshipQuery: 'MATCH (u1:User)-[:FOLLOW]→(u2:User) RETURN id(u1) AS source, id(u2) AS target' }) YIELD nodeId, componentId RETURN gds.util.asNode(nodeId).name AS Name, componentId AS Component ORDER BY Component DESC ----

.Results [opts="header",cols="1m,1m"]

| Name | Component | "Doug" | 3 | "Mark" | 3 | "Charles" | 2 | "Alice" | 0 | "Bridget" | 0 | "Michael" | 0

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Speaker-Listener Label Propagation algorithm in the Neo4j Graph Data Science library. [.alpha] = Speaker-Listener Label Propagation :entity: pregel :result: community IDs :algorithm: SLLPA

[abstract] — This section describes the Speaker-Listener Label Propagation algorithm in the Neo4j Graph Data Science library. —

== Introduction

The Speaker-Listener Label Propagation Algorithm (SLLPA) is a variation of the Label Propagation algorithm that is able to detect multiple communities per node. The GDS implementation is based on the SLPA: Uncovering Overlapping Communities in Social Networks via A Speaker-listener Interaction Dynamic Process publication by Xie et al.

The algorithm is randomized in nature and will not produce deterministic results. To accommodate this, we recommend using a higher number of iterations.

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

.SLLPA syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ======

.Run SLLPA in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.sllpa.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, values: Map { communtiyIds: List of Integer } ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | n/a | no | Maximum number of iterations to run. | minAssociationStrength | String | 0.2 | yes | Minimum influence required for a community to retain a node.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | values | Map | A map that contains the key `communityIds`.

```
======

[.include-with-stats] ======

.Run SLLPA in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.sllpa.stats( graphName: String, configuration: Map ) YIELD ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

```
.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]
```

| Name | Type | Default | Optional | Description | maxIterations | Integer | n/a | no | Maximum number of iterations to run. | minAssociationStrength | String | 0.2 | yes | Minimum influence required for a community to retain a node.

```
.Results [opts="header",cols="1,1,6"]
```

| Name | Type | Description | ranIterations | Integer | Number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | configuration | Map | Configuration used for running the algorithm.

```
======

[.include-with-mutate] ======

.Run SLLPA in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.sllpa.mutate( graphName: String, configuration: Map ) YIELD ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | n/a | no | Maximum number of iterations to run. | minAssociationStrength | String | 0.2 | yes | Minimum influence required for a community to retain a node.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | nodePropertiesWritten | Integer | The number of properties that were written to Neo4j. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ======

.Run SLLPA in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.sllpa.write(graphName: String, configuration: Map) YIELD ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | maxIterations | Integer | n/a | no | Maximum number of iterations to run. | minAssociationStrength | String | 0.2 | yes | Minimum influence required for a community to retain a node.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | ranIterations | Integer | The number of iterations run. | didConverge | Boolean | Indicates if the algorithm converged. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back. | nodePropertiesWritten | Integer | The number of properties that were written to Neo4j. | configuration | Map | The configuration used for running the algorithm.

```
======

====

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: sllpa.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'}), (c:Person {name: 'Carol'}), (d:Person {name: 'Dave'}), (e:Person {name: 'Eve'}), (f:Person {name: 'Fredrick'}), (g:Person {name: 'Gary'}), (h:Person {name: 'Hilda'}), (i:Person {name: 'Ichabod'}), (j:Person {name: 'James'}), (k:Person {name: 'Khalid'}),

(a)-[:KNOWS]→(b), (a)-[:KNOWS]→(c), (a)-[:KNOWS]→(d), (b)-[:KNOWS]→(c), (b)-[:KNOWS]→(d), (c)-[:KNOWS]→(d),

(b)-[:KNOWS]→(e), (e)-[:KNOWS]→(f), (f)-[:KNOWS]→(g), (g)-[:KNOWS]→(h),

(h)-[:KNOWS]→(i), (h)-[:KNOWS]→(j), (h)-[:KNOWS]→(k), (i)-[:KNOWS]→(j), (i)-[:KNOWS]→(k), (j)-[:KNOWS]→(k); ----

In the example, we will use the SLLPA algorithm to find the communities in the graph.

.The following statement will create the graph and store it in the graph catalog. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create( 'myGraph', 'Person', { KNOWS: { orientation: 'UNDIRECTED' } } ); ----

In the following examples we will demonstrate using the SLLPA algorithm on this graph.

=== Stream

In the stream execution mode, the algorithm returns the {result} for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the stream mode in general, see Stream.

[role=query-example, no-result=true] — .The following will run the algorithm, and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.sllpa.stream('myGraph', {maxIterations: 100, minAssociationStrength: 0.1}) YIELD nodeId, values RETURN gds.util.asNode(nodeId).name AS Name, values.communityIds AS communityIds ORDER BY Name ASC ----

.Results [opts="header"]
```

| Name | communityIds | "Alice" | [0] | "Bob" | [0] | "Carol" | [0] | "Dave" | [0] | "Eve" | [0, 1] | "Fredrick" | [0, 1] | "Gary" | [0, 1] | "Hilda" | [1] | "Ichabod" | [1] | "James" | [1] | "Khalid" | [1]

— Due to the randomness of the algorithm, the results will tend to vary between runs.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Approximate Maximum k-cut algorithm in the Neo4j Graph Data Science library. [.alpha] = Approximate Maximum k-cut :entity: node :result: approximate maximum k-cut :algorithm: Approximate Maximum k-cut

[abstract] — This section describes the Approximate Maximum k-cut algorithm in the Neo4j Graph Data Science library. —

== Introduction

A k-cut of a graph is an assignment of its nodes into k disjoint communities. So for example a 2-cut of a graph with nodes `a,b,c,d` could be the communities `{a,b,c}` and `{d}`.

A Maximum k-cut is a k-cut such that the total weight of relationships between nodes from different communities in the k-cut is maximized. That is, a k-cut that maximizes the sum of weights of relationships whose source and target nodes are assigned to different communities in the k-cut. Suppose in the simple `a,b,c,d` node set example above we only had one relationship `b → c`, and it was of weight `1.0`. The 2-cut we outlined above would then not be a maximum 2-cut (with a cut cost of `0.0`), whereas for example the 2-cut with communities `{a,b}` and `{c,d}` would be one (with a cut cost of `1.0`).

[NOTE] ==== Maximum k-cut is the same as Maximum Cut when `k = 2`. ====

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

=== Applications

Finding the maximum k-cut for a graph has several known applications, for example it is used to:

* analyze protein interaction * design circuit (VLSI) layouts * solve wireless communication problems * analyze cryptocurrency transaction patterns * design computer networks

=== Approximation

In practice, finding the best cut is not feasible for larger graphs and only an approximation can be computed in reasonable time.

The approximate heuristic algorithm implemented in GDS is a parallelized GRASP style algorithm optionally enhanced (via config) with variable neighborhood search (VNS).

For detailed information about a serial version of the algorithm, with a slightly different construction phase, when `k = 2` see `GRASP+VNR` in the paper:

* Festa et al. Randomized Heuristics for the Max-Cut Problem, 2002.

To see how the algorithm above performs in terms of solution quality compared to other algorithms when `k = 2` see `FES02GV` in the paper:

* Dunning et al. What Works Best When? A Systematic Evaluation of Heuristics for Max-Cut and QUBO, 2018.

[NOTE] ==== By the stochastic nature of the algorithm, the results it yields will not be deterministic unless running single-threaded (`concurrency = 1`) and using the same random seed (`randomSeed = SOME_FIXED_VALUE`). ====

== Tuning the algorithm parameters

There are two important algorithm specific parameters which lets you trade solution quality for shorter runtime.

=== Iterations GRASP style algorithms are iterative by nature. Every iteration they run the same well-defined steps to derive

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="3,1,1m,1,4"]

| Name | Type | Default | Optional | Description | k | Integer | 2 | yes | The number of disjoint communities the nodes will be divided into. | iterations | Integer | 8 | yes | The number of iterations the algorithm will run before returning the best solution among all the iterations. | vnsMaxNeighborhoodOrder | Integer | 0 (VNS off) | yes | The maximum number of nodes VNS will swap when perturbing solutions. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in the computation. Requires `concurrency = 1`. | relationshipWeightProperty | String | null | yes | If set, the values stored at the given property are used as relationship weights during the computation. If not set, the graph is considered unweighted.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | communityId | Integer | Community ID.

======

[.include-with-mutate] ====== .Run Approximate Maximum k-cut in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.maxkcut.mutate(graphName: String, configuration: Map) YIELD cutCost: Float, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="3,1,1m,1,4"]

| Name | Type | Default | Optional | Description | k | Integer | 2 | yes | The number of disjoint communities the nodes will be divided into. | iterations | Integer | 8 | yes | The number of iterations the algorithm will run

before returning the best solution among all the iterations. | vnsMaxNeighborhoodOrder | Integer | 0 (VNS off) | yes | The maximum number of nodes VNS will swap when perturbing solutions. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in the computation. Requires `concurrency = 1`. | relationshipWeightProperty | String | null | yes | If set, the values stored at the given property are used as relationship weights during the computation. If not set, the graph is considered unweighted.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | cutCost | Float | Sum of weights of all relationships connecting nodes from different communities. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the statistics. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | nodePropertiesWritten | Integer | Number of properties added to the in-memory graph. | configuration | Map | Configuration used for running the algorithm.

====== ====

== Examples

:algorithm-name: {algorithm} :graph-description: Bitcoin transactions :image-file: approx-max-k-cut-example.png In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:Person {name: 'Alice'}), (bridget:Person {name: 'Bridget'}), (charles:Person {name: 'Charles'}), (doug:Person {name: 'Doug'}), (eric:Person {name: 'Eric'}), (fiona:Person {name: 'Fiona'}), (george:Person {name: 'George'}), (alice)-[:TRANSACTION {value: 81.0}]→(bridget), (alice)-[:TRANSACTION {value: 7.0}]→(doug), (bridget)-[:TRANSACTION {value: 1.0}]→(doug), (bridget)-[:TRANSACTION {value: 1.0}]→(eric), (bridget)-[:TRANSACTION {value: 1.0}]→(fiona), (bridget)-[:TRANSACTION {value: 1.0}]→(george), (charles)-[:TRANSACTION {value: 45.0}]→(bridget), (charles)-[:TRANSACTION {value: 3.0}]→(eric), (doug)-[:TRANSACTION {value: 3.0}]→(charles), (doug)-[:TRANSACTION {value: 1.0}]→(bridget), (eric)-[:TRANSACTION {value: 1.0}]→(bridget), (fiona)-[:TRANSACTION {value: 3.0}]→(alice), (fiona)-[:TRANSACTION {value: 1.0}]→(bridget), (george)-[:TRANSACTION {value: 1.0}]→(bridget), (george)-[:TRANSACTION {value: 4.0}]→(charles) ----

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `TRANSACTION` relationships.

.The following statement will create a graph store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Person', { TRANSACTION: { properties: ['value'] } }) ----

=== Memory Estimation

:mode: mutate First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.maxkcut.mutate.estimate('myGraph', {mutateProperty: 'community'}) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 7 | 15 | 488 | 488 | "488 Bytes"

—

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.maxkcut.mutate('myGraph', {mutateProperty: 'community'}) YIELD cutCost, nodePropertiesWritten ----

.Results [opts="header"]

| cutCost | nodePropertiesWritten | 13.0 | 7

— We can see that when relationship weight is not taken into account we derive a cut into two (since we didn't override the default `k = 2`) communities of cost `13.0`. The total cost is represented by the `cutCost` column here. This is the value we want to be as high as possible. Additionally, the graph 'myGraph' now has a node property `community` which stores the community to which each node belongs.

To inspect which community each node belongs to we can stream node properties.

[role=query-example] — .Stream node properties: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamNodeProperty('myGraph', 'community') YIELD nodeId, propertyValue RETURN gds.util.asNode(nodeId).name as name, propertyValue AS community ----

.Results [opts="header"]

| name | community | "Alice" | 0 | "Bridget" | 0 | "Charles" | 0 | "Doug" | 1 | "Eric" | 1 | "Fiona" | 1 | "George" | 1

— Looking at our graph topology we can see that there are no relationships between the nodes of community `1`, and two relationships between the nodes of community `0`, namely `Alice → Bridget` and `Charles → Bridget`. However, since there are a total of eight relationships between `Bridget` and nodes of community `1`, and our graph is unweighted assigning `Bridget` to community `1` would not yield a cut of a higher total weight. Thus, since the number of relationships connecting nodes of different communities greatly outnumber the number of relationships connecting nodes of the same community it seems like a good solution. In fact, this is the maximum 2-cut for this graph.

[NOTE] ==== Because of the inherent randomness in the Approximate Maximum k-Cut algorithm (unless having `concurrency = 1` and fixed `randomSeed`), running it another time might yield a different solution. For our case here it would be equally plausible to get the inverse solution, i.e. when our community `0` nodes are mapped to community `1` instead, and vice versa. Note however, that for that solution the cut cost would remain the same. ====

=== Mutate with relationship weights

In this example we will have a look at how adding relationship weight can affect our solution.

[role=query-example] — .The following will run the algorithm in `mutate` mode, diving our nodes into two communities once again: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.maxkcut.mutate('myGraph', { relationshipWeightProperty: 'value', mutateProperty: 'weightedCommunity' }) YIELD cutCost, nodePropertiesWritten ----

.Results [opts="header"]

| cutCost | nodePropertiesWritten | 146.0 | 7

— Since the `value` properties on our `TRANSACTION` relationships were all at least `1.0` and several of a larger value it's not surprising that we obtain a cut with a larger cost in the weighted case.

Let us now stream node properties to once again inspect the node community distribution.

[role=query-example] — .Stream node properties: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamNodeProperties('myGraph', 'weightedCommunity') YIELD nodeId, propertyValue RETURN gds.util.asNode(nodeId).name as name, propertyValue AS weightedCommunity ----

.Results [opts="header"]

| name | weightedCommunity | "Alice" | 0 | "Bridget" | 1 | "Charles" | 0 | "Doug" | 1 | "Eric" | 1 | "Fiona" | 1 | "George" | 1

— Comparing this result with that of unweighted case we can see that `Bridget` has moved to another community but the output is otherwise the same. Indeed, this makes sense by looking at our graph. `Bridget` is connected to nodes of community `1` by eight relationships, but these relationships all have weight `1.0`. And although `Bridget` is only connected to two community `0` nodes, these relationships are of weight `81.0` and `45.0`. Moving `Bridget` back to community `0` would lower the total cut cost of `81.0 + 45.0 - 8 * 1.0 = 118.0`. Hence, it does make sense that `Bridget` is now in community `1`. In fact, this is the maximum 2-cut in the weighted case.

[NOTE] ==== Because of the inherent randomness in the Approximate Maximum k-Cut algorithm (unless having `concurrency = 1` and fixed `randomSeed`), running it another time might yield a different solution. For our case here it would be equally plausible to get the inverse solution, i.e. when our community `0` nodes are mapped to community `1` instead, and vice versa. Note however, that for that solution the cut cost would remain the same. ====

=== Stream

In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm in `stream` mode using default configuration parameters: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.maxkcut.stream('myGraph') YIELD nodeId, communityId RETURN gds.util.asNode(nodeId).name AS name, communityId ----

.Results [opts="header"]

| name | communityId | "Alice" | 0 | "Bridget" | 0 | "Charles" | 0 | "Doug" | 1 | "Eric" | 1 | "Fiona" | 1 | "George" | 1

— We can see that the result is what we expect, namely the same as in the mutate unweighted example.

[NOTE] ==== Because of the inherent randomness in the Approximate Maximum k-Cut algorithm (unless having `concurrency = 1` and fixed `randomSeed`), running it another time might yield a different solution. For our case here it would be equally plausible to get the inverse solution, i.e. when our community `0` nodes are mapped to community `1` instead, and vice versa. Note however, that for that solution the cut cost would remain the same. ====

:leveloffset: 2

:leveloffset: +2

:description: This chapter provides explanations and examples for each of the similarity algorithms in the Neo4j Graph Data Science library. = Similarity

[abstract] — This chapter provides explanations and examples for each of the similarity algorithms in the Neo4j Graph Data Science library. — Similarity algorithms compute the similarity of pairs of nodes using different vector-based metrics. The Neo4j GDS library includes the following similarity algorithms, grouped by quality tier:

* Production-quality Node Similarity * Beta K-Nearest Neighbors * Alpha Approximate Nearest Neighbors Cosine Similarity Euclidean Similarity Jaccard Similarity Overlap Similarity Pearson Similarity

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Node Similarity algorithm in the Neo4j Graph Data Science library. = Node Similarity :entity: relationship :result: similarity score :algorithm: Node Similarity

[abstract] — This section describes the Node Similarity algorithm in the Neo4j Graph Data Science library. The algorithm is based on the Jaccard Similarity score. — :directed: :undirected: :homogeneous: :heterogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

The Node Similarity algorithm compares a set of nodes based on the nodes they are connected to. Two nodes are considered similar if they share many of the same neighbors. Node Similarity computes pair-wise similarities based on the Jaccard metric, also known as the Jaccard Similarity Score.

Jaccard Similarity is computed using the following formula:

image::jaccard.png[role="middle"]

The input of this algorithm is a bipartite, connected graph containing two disjoint node sets. Each relationship starts from a node in the first node set and ends at a node in the second node set.

The Node Similarity algorithm compares each node that has outgoing relationships with each other such node. For every node `n`, we collect the outgoing neighborhood `N(n)` of that node, that is, all nodes `m` such that there is a relationship from `n` to `m`. For each pair `n`, `m`, the algorithm computes a similarity for that pair which is the Jaccard similarity of `N(n)` and `N(m)`.

The complexity of this comparison grows quadratically with the number of nodes to compare. The algorithm reduces the complexity by ignoring disconnected nodes.

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | similarityCutoff | Float | 1E-42 | yes | Lower limit for the similarity score to be present in the result. Values must be between 0 and 1. | degreeCutoff | Integer | 1 | yes | Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1. | topK | Integer | 10 | yes | Limit on the number of scores per node. The K largest results are returned. This value cannot be lower than 1. | bottomK | Integer | 10 | yes | Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1. | topN | Integer | 0 | yes | Global limit on the number of scores computed. The N largest total results are returned. This value cannot be negative, a value of 0 means no global limit. | bottomN | Integer | 0 | yes | Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header",cols="1m,1,6"]

| Name | Type | Description | node1 | Integer | Node ID of the first node. | node2 | Integer | Node ID of the second node. | similarity | Float | Similarity score for the two nodes.

======

[.include-with-stats] ====== .Run Node Similarity in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.stats(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, nodesCompared: Integer, similarityPairs: Integer, similarityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | similarityCutoff | Float | 1E-42 | yes | Lower limit for the similarity score to be present in the result. Values must be between 0 and 1. | degreeCutoff | Integer | 1 | yes | Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1. | topK | Integer | 10 | yes | Limit on the number of scores per node. The K largest results are returned. This value cannot be lower than 1. | bottomK | Integer | 10 | yes | Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1. | topN | Integer | 0 | yes | Global limit on the number of scores computed. The N largest total results are returned. This value cannot be negative, a value of 0 means no global limit. | bottomN | Integer | 0 | yes | Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | nodesCompared | Integer | The number of nodes compared. | postProcessingMillis | Integer | Milliseconds for computing component count and distribution statistics. | similarityPairs | Integer | The number of pairs of similar nodes computed. | similarityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of the computed similarity results. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-mutate] ====== .Run Node Similarity in mutate mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, postProcessingMillis: Integer, relationshipsWritten: Integer, nodesCompared: Integer, similarityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | similarityCutoff | Float | 1E-42 | yes | Lower limit for the similarity score to be present in the result. Values must be between 0 and 1. | degreeCutoff | Integer | 1 | yes | Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1. | topK | Integer | 10 | yes | Limit on the number of scores per node. The K largest results are

returned. This value cannot be lower than 1. | bottomK | Integer | 10 | yes | Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1. | topN | Integer | 0 | yes | Global limit on the number of scores computed. The N largest total results are returned. This value cannot be negative, a value of 0 means no global limit. | bottomN | Integer | 0 | yes | Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodesCompared | Integer | The number of nodes compared. | relationshipsWritten | Integer | The number of relationships created. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | postProcessingMillis | Integer | Milliseconds for computing percentiles. | similarityDistribution | Map | Map containing min, max, mean, stdDev and p1, p5, p10, p25, p75, p90, p95, p99, p100 percentile values of the computed similarity results. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ====== .Run Node Similarity in write mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.write(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, postProcessingMillis: Integer, nodesCompared: Integer, relationshipsWritten: Integer, similarityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | similarityCutoff | Float | 1E-42 | yes | Lower limit for the similarity score to be present in the result. Values must be between 0 and 1. | degreeCutoff | Integer | 1 | yes | Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1. | topK | Integer | 10 | yes | Limit on the number of scores per node. The K largest results are returned. This value cannot be lower than 1. | bottomK | Integer | 10 | yes | Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1. | topN | Integer | 0 | yes | Global limit on the number of scores computed. The N largest total results are returned. This value cannot

be negative, a value of 0 means no global limit. | bottomN | Integer | 0 | yes | Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodesCompared | Integer | The number of nodes compared. | relationshipsWritten | Integer | The number of relationships created. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | postProcessingMillis | Integer | Milliseconds for computing percentiles. | similarityDistribution | Map | Map containing min, max, mean, stdDev and p1, p5, p10, p25, p75, p90, p95, p99, p100 percentile values of the computed similarity results. | configuration | Map | The configuration used for running the algorithm.

====== ====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run Node Similarity in write mode on an anonymous graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.write(configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, postProcessingMillis: Integer, nodesCompared: Integer, relationshipsWritten: Integer, similarityDistribution: Map, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | similarityCutoff | Float | 1E-42 | yes | Lower limit for the

similarity score to be present in the result. Values must be between 0 and 1. | degreeCutoff | Integer | 1 | yes | Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1. | topK | Integer | 10 | yes | Limit on the number of scores per node. The K largest results are returned. This value cannot be lower than 1. | bottomK | Integer | 10 | yes | Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1. | topN | Integer | 0 | yes | Global limit on the number of scores computed. The N largest total results are returned. This value cannot be negative, a value of 0 means no global limit. | bottomN | Integer | 0 | yes | Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: knowledge :image-file: node-similarity.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:Person {name: 'Alice'}), (bob:Person {name: 'Bob'}), (carol:Person {name: 'Carol'}), (dave:Person {name: 'Dave'}), (eve:Person {name: 'Eve'}), (guitar:Instrument {name: 'Guitar'}), (synth:Instrument {name: 'Synthesizer'}), (bongos:Instrument {name: 'Bongos'}), (trumpet:Instrument {name: 'Trumpet'}),

(alice)-[:LIKES]→(guitar), (alice)-[:LIKES]→(synth), (alice)-[:LIKES {strength: 0.5}]→(bongos), (bob)-[:LIKES]→(guitar), (bob)-[:LIKES]→(synth), (carol)-[:LIKES]→(bongos), (dave)-[:LIKES]→(guitar), (dave)-[:LIKES]→(synth), (dave)-[:LIKES]→(bongos); ----

This bipartite graph has two node sets, Person nodes and Instrument nodes. The two node sets are connected via LIKES relationships. Each relationship starts at a Person node and ends at an Instrument node.

In the example, we want to use the Node Similarity algorithm to compare people based on the instruments they like.

The Node Similarity algorithm will only compute similarity for nodes that have a degree of at least 1. In the example graph, the Eve node will not be compared to other Person nodes.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create the graph and store it in the graph catalog. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', ['Person', 'Instrument'], { LIKES: { type: 'LIKES', properties: { strength: { property: 'strength', defaultValue: 1.0 } } } }); ----

In the following examples we will demonstrate using the Node Similarity algorithm on this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.write.estimate('myGraph', { writeRelationshipType: 'SIMILAR', writeProperty: 'score' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header", cols="1,1,1,1,1"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 9 | 9 | 2592 | 2808 | "[2592 Bytes … 2808 Bytes]"

—

=== Stream

In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm, and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.stream('myGraph') YIELD node1, node2, similarity RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity ORDER BY similarity DESCENDING, Person1, Person2 ----

.Results [opts="header"]

| Person1 | Person2 | similarity | "Alice" | "Dave" | 1.0 | "Dave" | "Alice" | 1.0 | "Alice" | "Bob" | 0.6666666666666666 | "Bob" | "Alice" | 0.6666666666666666 | "Bob" | "Dave" | 0.6666666666666666 | "Dave" | "Bob" | 0.6666666666666666 | "Alice" | "Carol" | 0.3333333333333333 | "Carol" | "Alice" | 0.3333333333333333 | "Carol" | "Dave" | 0.3333333333333333 | "Dave" | "Carol" | 0.3333333333333333

— We use default values for the procedure configuration parameter. TopK is set to 10, topN is set to 0. Because of that the result set contains the top 10 similarity scores for each node.

[NOTE] ==== If we would like to instead compare the Instruments to each other, we would then project the `LIKES` relationship type using `REVERSE` orientation. This would return similarities for pairs of Instruments and not compute any similarities between Persons. ====

=== Stats

:stats-syntax: algorithms-node-similarity-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm and returns the result in form of statistical and measurement values [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.stats('myGraph') YIELD nodesCompared, similarityPairs ----

.Results [opts="header",cols="2"]

| nodesCompared | similarityPairs | 4 | 10

—

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm, and write back results to the in-memory graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.mutate('myGraph', { mutateRelationshipType: 'SIMILAR', mutateProperty: 'score' }) YIELD nodesCompared, relationshipsWritten ----

.Results [opts="header"]

| nodesCompared | relationshipsWritten | 4 | 10

— As we can see from the results, the number of created relationships is equal to the number of rows in the streaming example.

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: for each pair of nodes we create a relationship with the Jaccard similarity score as a property to the Neo4j database. The type of the new relationship is specified using the mandatory configuration parameter `writeRelationshipType`. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm, and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.write('myGraph', { writeRelationshipType: 'SIMILAR', writeProperty: 'score' }) YIELD nodesCompared, relationshipsWritten ----

.Results [opts="header"]

| nodesCompared | relationshipsWritten | 4 | 10

— As we can see from the results, the number of created relationships is equal to the number of rows in the streaming example.

=== Limit results

There are four limits that can be applied to the similarity results. Top limits the result to the highest similarity scores. Bottom limits the result to the lowest similarity scores. Both top and bottom limits can apply to the result as a whole ("N"), or to the result per node ("K").

[NOTE] ==== There must always be a "K" limit, either bottomK or topK, which is a positive number. The default value for topK and bottomK is 10. ====

.Result limits [opts="header", cols="1h,1,1"]

| | total results | results per node | highest score | topN | topK | lowest score | bottomN | bottomK

==== topK and bottomK

TopK and bottomK are limits on the number of scores computed per node. For topK, the K largest similarity scores per node are returned. For bottomK, the K smallest similarity scores per node are returned. TopK and bottomK cannot be 0, used in conjunction, and the default value is 10. If neither is specified, topK is used.

[role=query-example] — .The following will run the algorithm, and stream the top 1 result per node: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.stream('myGraph', { topK: 1 }) YIELD node1, node2, similarity RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity ORDER BY Person1 ----

.Results [opts="header"]

| Person1 | Person2 | similarity | "Alice" | "Dave" | 1.0 | "Bob" | "Alice" | 0.6666666666666666 | "Carol" | "Alice" | 0.3333333333333333 | "Dave" | "Alice" | 1.0

—

[role=query-example] — .The following will run the algorithm, and stream the bottom 1 result per node: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.stream('myGraph', { bottomK: 1 }) YIELD node1, node2, similarity RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity ORDER BY Person1 ----

.Results [opts="header"]

| Person1 | Person2 | similarity | "Alice" | "Carol" | 0.3333333333333333 | "Bob" | "Alice" | 0.6666666666666666 | "Carol" | "Alice" | 0.3333333333333333 | "Dave" | "Carol" | 0.3333333333333333

—

==== topN and bottomN

TopN and bottomN limit the number of similarity scores across all nodes. This is a limit on the total result set, in addition to the topK or bottomK limit on the results per node. For topN, the N largest similarity scores are returned. For bottomN, the N smallest similarity scores are returned. A value of 0 means no global limit is imposed and all results from topK or bottomK are returned.

[role=query-example] — .The following will run the algorithm, and stream the 3 highest out of the top 1 results per node: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.stream('myGraph', { topK: 1, topN: 3 }) YIELD node1, node2, similarity RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity ORDER BY similarity DESC, Person1, Person2 ----

.Results [opts="header"]

| Person1 | Person2 | similarity | "Alice" | "Dave" | 1.0 | "Dave" | "Alice" | 1.0 | "Bob" | "Alice" | 0.6666666666666666

—

=== Degree cutoff and similarity cutoff

Degree cutoff is a lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1.

[role=query-example] — .The following will ignore nodes with less than 3 LIKES relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.stream('myGraph', { degreeCutoff: 3 }) YIELD node1, node2, similarity RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity ORDER BY Person1 ----

.Results [opts="header"]

| Person1 | Person2 | similarity | "Alice" | "Dave" | 1.0 | "Dave" | "Alice" | 1.0

— Similarity cutoff is a lower limit for the similarity score to be present in the result. The default value is very small (`1E-42`) to exclude results with a similarity score of 0.

[NOTE] ==== Setting similarity cutoff to 0 may yield a very large result set, increased runtime and memory consumption. ====

[role=query-example] — .The following will ignore node pairs with a similarity score less than 0.5: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.stream('myGraph', { similarityCutoff: 0.5 }) YIELD node1, node2, similarity RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity ORDER BY Person1 ----

.Results [opts="header"]

| Person1 | Person2 | similarity | "Alice" | "Dave" | 1.0 | "Alice" | "Bob" | 0.6666666666666666 | "Bob" | "Dave" | 0.6666666666666666 | "Bob" | "Alice" | 0.6666666666666666 | "Dave" | "Alice" | 1.0 | "Dave" | "Bob" | 0.6666666666666666

—

=== Weighted Jaccard Similarity

Relationship properties can be used to modify the similarity induced by certain relationships. For example a relationship value of 2 is equal to counting that relationship twice while computing the jaccard similarity.

[NOTE] ==== Weighted jaccard similarity is only defined for values greater or equal to 0. ====

[role=query-example] — .The following query will respect relationship properties in the similarity computation: [source, cypher, role=noplay, indent=0] ---- CALL gds.nodeSimilarity.stream('myGraph', { relationshipWeightProperty: 'strength', similarityCutoff: 0.5 }) YIELD node1, node2, similarity RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity ORDER BY Person1 ----

.Results [opts="header"]

| Person1 | Person2 | similarity | "Alice" | "Dave" | 0.8333333333333334 | "Alice" | "Bob" | 0.8 | "Bob" | "Alice" | 0.8 | "Bob" | "Dave" | 0.6666666666666666 | "Dave" | "Alice" | 0.8333333333333334 | "Dave" | "Bob" | 0.6666666666666666

— It can be seen that the similarity between Alice and Dave decreased compared to the non-weighted version of this algorithm. This is the case as the strength of the relationship between Alice and Bongos is reduced and both persons now only share 2.5 out of 3 possible instruments. Analogous the similarity between Alice and Bob increased as the missing liked instrument has a lower impact on the similarity score.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the K-Nearest Neighbors (KNN) algorithm in the Neo4j Graph Data Science library.
[.beta] = K-Nearest Neighbors

:entity: relationship :result: similarity score :algorithm: K-Nearest Neighbors

[abstract] — This section describes the K-Nearest Neighbors (KNN) algorithm in the Neo4j Graph Data Science library. —

== Introduction

The K-Nearest Neighbors algorithm computes a distance value for all node pairs in the graph and creates new relationships between each node and its k nearest neighbors. The distance is calculated based on node properties.

The input of this algorithm is a monopartite graph. The graph does not need to be connected, in fact, existing relationships between nodes will be ignored. New relationships are created between each node and its k nearest neighbors.

The K-Nearest Neighbors algorithm compares a given property of each node. The `k` nodes where this property is most similar are the k-nearest neighbors.

The initial set of neighbors is picked at random and verified and refined in multiple iterations. The number of iterations is limited by the configuration parameter `maxIterations`. The algorithm may stop earlier if the neighbor lists only change by a small amount, which can be controlled by the configuration parameter `deltaThreshold`.

The particular implementation is based on Efficient k-nearest neighbor graph construction for generic similarity measures by Wei Dong et al. Instead of comparing every node with every other node, the algorithm selects possible neighbors based on the assumption, that the neighbors-of-neighbors of a node are most likely already the nearest one. The algorithm scales quasi-linear with respect to the node count, instead of being quadratic.

Furthermore, the algorithm only compares a sample of all possible neighbors on each iteration, assuming that eventually all possible neighbors will be seen. This can be controlled with the configuration parameter `sampleRate`:

* A valid sample rate must be in between 0 (exclusive) and 1 (inclusive). * The default value is `0.5`. * The parameter is used to control the trade-off between accuracy and runtime-performance. * A higher sample rate will increase the accuracy of the result. **The algorithm will also require more memory and will take longer to compute. * A lower sample rate will increase the runtime-performance.** Some potential nodes may be missed in the comparison and may not be included in the result.

The output of the algorithm are new relationships between nodes and their k-nearest neighbors. Similarity scores are expressed via relationship properties.

For more information on this algorithm, see:

* Efficient k-nearest neighbor graph construction for generic similarity measures * Nearest neighbor graph (Wikipedia)

[NOTE] ==== Running this algorithm requires sufficient available memory. Before running this algorithm, we recommend that you read Memory Estimation. ====

=== Similarity measures

The similarity measure used in the KNN algorithm depends on the type of the configured node property. KNN supports both scalar numeric values as well as lists of numbers.

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeWeightProperty | String | n/a | no | The name of a node property that contains node weights which will be used for similarity computation. | topK | Integer | 10 | yes | The number of neighbors to find for each node. The K-nearest neighbors are returned. This value cannot be lower than 1. | sampleRate | Float | 0.5 | yes | Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive). | deltaThreshold | Float | 0.001 | yes | Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive). | maxIterations | Integer | 100 | yes | Hard limit to stop the algorithm after that many iterations. | randomJoins | Integer | 10 | yes | Between every iteration, how many attempts are being made to connect new node neighbors based on random selection. | randomSeed | Integer | -1 | yes | The seed value to control the randomness of the algorithm. The value -1 means that a new seed is generated for every execution, all other values (including negative ones) are used as the seed value.

.Results [opts="header",cols="1m,1,6"]

| Name | Type | Description | node1 | Integer | Node ID of the first node. | node2 | Integer | Node ID of the second node. | similarity | Float | Similarity score for the two nodes.

======

[.include-with-stats] ====== .Run K-Nearest Neighbors in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.knn.stats(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, nodesCompared: Integer, similarityPairs: Integer, similarityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeWeightProperty | String | n/a | no | The name of a node property that contains node weights which will be used for similarity computation. | topK | Integer | 10 | yes | The number of neighbors to find for each node. The K-nearest neighbors are returned. This value cannot be lower than 1. | sampleRate | Float | 0.5 | yes | Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive). | deltaThreshold | Float | 0.001 | yes | Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive). | maxIterations | Integer | 100 | yes | Hard limit to stop the algorithm after that many iterations. | randomJoins | Integer | 10 | yes | Between every iteration, how many attempts are being made to connect new node neighbors based on random selection. | randomSeed | Integer | -1 | yes | The seed value to control the randomness of the algorithm. The value -1 means that a new seed is generated for every execution, all other values (including negative ones) are used as the seed value.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing similarity value distribution statistics. | nodesCompared | Integer | The number of nodes compared. | similarityPairs | Integer | The number of pairs of similar nodes computed. | similarityDistribution | Map | Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of the computed similarity results. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-mutate] ====== .Run K-Nearest Neighbors in mutate mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.knn.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, postProcessingMillis: Integer, relationshipsWritten: Integer, nodesCompared: Integer, similarityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeWeightProperty | String | n/a | no | The name of a node property that contains node weights which will be used for similarity computation. | topK | Integer | 10 | yes | The number of neighbors to find for each node. The K-nearest neighbors are returned. This value

cannot be lower than 1. | sampleRate | Float | 0.5 | yes | Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive). | deltaThreshold | Float | 0.001 | yes | Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive). | maxIterations | Integer | 100 | yes | Hard limit to stop the algorithm after that many iterations. | randomJoins | Integer | 10 | yes | Between every iteration, how many attempts are being made to connect new node neighbors based on random selection. | randomSeed | Integer | -1 | yes | The seed value to control the randomness of the algorithm. The value -1 means that a new seed is generated for every execution, all other values (including negative ones) are used as the seed value.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | postProcessingMillis | Integer | Milliseconds for computing similarity value distribution statistics. | nodesCompared | Integer | The number of nodes compared. | relationshipsWritten | Integer | The number of relationships created. | similarityDistribution | Map | Map containing min, max, mean, stdDev and p1, p5, p10, p25, p75, p90, p95, p99, p100 percentile values of the computed similarity results. | configuration | Map | The configuration used for running the algorithm.

```
======

[.include-with-write] ====== .Run K-Nearest Neighbors in write mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.knn.write( graphName: String, configuration: Map ) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, postProcessingMillis: Integer, nodesCompared: Integer, relationshipsWritten: Integer, similarityDistribution: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeWeightProperty | String | n/a | no | The name of a node property that contains node weights which will be used for similarity computation. | topK | Integer | 10 | yes | The number of neighbors to find for each node. The K-nearest neighbors are returned. This value cannot be lower than 1. | sampleRate | Float | 0.5 | yes | Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive). | deltaThreshold | Float | 0.001 | yes |

Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive). | maxIterations | Integer | 100 | yes | Hard limit to stop the algorithm after that many iterations. | randomJoins | Integer | 10 | yes | Between every iteration, how many attempts are being made to connect new node neighbors based on random selection. | randomSeed | Integer | -1 | yes | The seed value to control the randomness of the algorithm. The value -1 means that a new seed is generated for every execution, all other values (including negative ones) are used as the seed value.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | postProcessingMillis | Integer | Milliseconds for computing similarity value distribution statistics. | nodesCompared | Integer | The number of nodes compared. | relationshipsWritten | Integer | The number of relationships created. | similarityDistribution | Map | Map containing min, max, mean, stdDev and p1, p5, p10, p25, p75, p90, p95, p99, p100 percentile values of the computed similarity results. | configuration | Map | The configuration used for running the algorithm.

====== ====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run K-Nearest Neighbors in write mode on an anonymous graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.knn.write(configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, postProcessingMillis: Integer, nodesCompared: Integer, relationshipsWritten: Integer, similarityDistribution: Map, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

[NOTE] ==== The KNN algorithm does not read any relationships, but the values for `relationshipProjection` or `relationshipQuery` are still being used and respected for the graph loading. ====

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeWeightProperty | String | n/a | no | The name of a node property that contains node weights which will be used for similarity computation. | topK | Integer | 10 | yes | The number of neighbors to find for each node. The K-nearest neighbors are returned. This value cannot be lower than 1. | sampleRate | Float | 0.5 | yes | Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive). | deltaThreshold | Float | 0.001 | yes | Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive). | maxIterations | Integer | 100 | yes | Hard limit to stop the algorithm after that many iterations. | randomJoins | Integer | 10 | yes | Between every iteration, how many attempts are being made to connect new node neighbors based on random selection. | randomSeed | Integer | -1 | yes | The seed value to control the randomness of the algorithm. The value -1 means that a new seed is generated for every execution, all other values (including negative ones) are used as the seed value.

The results are the same as running write mode on a named graph, see write mode syntax above.

== Examples

Consider the graph created by the following Cypher statement:

[source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:Person {name: 'Alice', age: 24}) CREATE (bob:Person {name: 'Bob', age: 73}) CREATE (carol:Person {name: 'Carol', age: 24}) CREATE (dave:Person {name: 'Dave', age: 48}) CREATE (eve:Person {name: 'Eve', age: 67}); ----

In the example, we want to use the K-Nearest Neighbors algorithm to compare people based on their age.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create the graph and store it in the graph catalog. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', { Person: { label: 'Person', properties: 'age' } }, '*'); ----

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.knn.write.estimate('myGraph', { nodeWeightProperty: 'age', writeRelationshipType: 'SIMILAR', writeProperty: 'score', topK: 1 }) YIELD nodeCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header", cols="1,1,1,1"]

| nodeCount | bytesMin | bytesMax | requiredMemory | 5 | 1224 | 2184 | "[1224 Bytes ... 2184 Bytes]"

—

=== Stream

In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm, and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.knn.stream('myGraph', { topK: 1, nodeWeightProperty: 'age', // The following parameters are set to produce a deterministic result randomSeed: 42, concurrency: 1, sampleRate: 1.0, deltaThreshold: 0.0 }) YIELD node1, node2, similarity RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity ORDER BY similarity DESCENDING, Person1, Person2 ----

.Results [opts="header"]

| Person1 | Person2 | similarity | "Alice" | "Carol" | 1.0 | "Carol" | "Alice" | 1.0 | "Bob" | "Eve" | 0.14285714285714285 | "Eve" | "Bob" | 0.14285714285714285 | "Dave" | "Eve" | 0.05

— We use default values for the procedure configuration parameter for most parameters. The `randomSeed` is set to produce the same result on every invocation. The `topK` parameter is set to 1 to only return the single nearest neighbor for every node.

=== Stats

:stats-syntax: algorithms-knn-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm and return the result in form of statistical and measurement values: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.knn.stats('myGraph', {topK: 1, randomSeed: 42, nodeWeightProperty: 'age'}) YIELD nodesCompared, similarityPairs ----

.Results [opts="header",cols="2"]

| nodesCompared | similarityPairs | 5 | 5

—

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm, and write back results to the in-memory graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.knn.mutate('myGraph', { mutateRelationshipType: 'SIMILAR', mutateProperty: 'score', topK: 1, randomSeed: 42, nodeWeightProperty: 'age' }) YIELD nodesCompared, relationshipsWritten ----

.Results [opts="header"]

| nodesCompared | relationshipsWritten | 5 | 5

— As we can see from the results, the number of created relationships is equal to the number of rows in the streaming example.

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: for each pair of nodes we create a relationship with the similarity score as a property to the Neo4j database. The type of the new relationship is specified using the mandatory configuration parameter `writeRelationshipType`. Each new relationship stores the similarity score between the two nodes it represents. The relationship property key is set using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm, and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.knn.write('myGraph', { writeRelationshipType: 'SIMILAR', writeProperty: 'score', topK: 1, randomSeed: 42, nodeWeightProperty: 'age' }) YIELD nodesCompared, relationshipsWritten ----

.Results [opts="header"]

| nodesCompared | relationshipsWritten | 5 | 5

— As we can see from the results, the number of created relationships is equal to the number of rows in the streaming example.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Jaccard Similarity algorithm in the Neo4j Graph Data Science library. [.alpha] = Jaccard Similarity

[abstract] — This section describes the Jaccard Similarity algorithm in the Neo4j Graph Data Science library. — Jaccard Similarity (coefficient), a term coined by Paul Jaccard, measures similarities between sets. It is defined as the size of the intersection divided by the size of the union of two sets. This notion has been generalized for multisets, where duplicate elements are counted as weights.

The GDS Jaccard Similarity function is defined for lists, which are interpreted as multisets.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

A related procedure for computing Jaccard similarity is described in Node Similarity.

== History and explanation

Jaccard Similarity is computed using the following formula:

image::jaccard.png[role="middle"]

The library contains functions to calculate similarity between sets of data. The Jaccard Similarity function is best used when calculating the similarity between small numbers of sets.

== Use-cases - when to use the Jaccard Similarity algorithm

We can use the Jaccard Similarity algorithm to work out the similarity between two things. We might then use the computed similarity as part of a recommendation query. For example, you can use the Jaccard Similarity algorithm to show the products that were purchased by similar customers, in terms of previous products purchased.

== Jaccard Similarity algorithm function sample

The Jaccard Similarity function computes the similarity of two lists of numbers.

We can use it to compute the similarity of two hardcoded lists.

.The following will return the Jaccard Similarity of two lists of numbers: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.similarity.jaccard([1,2,3], [1,2,4,5]) AS similarity ----

.Results [opts="header",cols="1"]

| `similarity` | 0.4

These two lists of numbers have a Jaccard Similarity of 0.4. We can see how this result is derived by breaking down the formula:

---- J(A,B) = ▯A ∩ B▯ / ▯A▯ + ▯B▯ - ▯A ∩ B

J(A,B) = 2 / 3 + 4 - 2 = 2 / 5 = 0.4 ----

We can also use it to compute the similarity of nodes based on lists computed by a Cypher query.

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (french:Cuisine {name:'French'}), (italian:Cuisine {name:'Italian'}), (indian:Cuisine {name:'Indian'}), (lebanese:Cuisine {name:'Lebanese'}), (portuguese:Cuisine {name:'Portuguese'}),

(zhen:Person {name: 'Zhen'}), (praveena:Person {name: 'Praveena'}), (michael:Person {name: 'Michael'}), (arya:Person {name: 'Arya'}), (karin:Person {name: 'Karin'}),

(praveena)-[:LIKES]→(indian), (praveena)-[:LIKES]→(portuguese),

(zhen)-[:LIKES]→(french), (zhen)-[:LIKES]→(indian),

(michael)-[:LIKES]→(french), (michael)-[:LIKES]→(italian), (michael)-[:LIKES]→(indian),

(arya)-[:LIKES]→(lebanese), (arya)-[:LIKES]→(italian), (arya)-[:LIKES]→(portuguese),

(karin)-[:LIKES]→(lebanese), (karin)-[:LIKES]→(italian) ----

.The following will return the Jaccard Similarity of Karin and Arya: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Karin'})-[:LIKES]→(cuisine1) WITH p1, collect(id(cuisine1)) AS p1Cuisine MATCH (p2:Person {name: "Arya"})-[:LIKES]→(cuisine2) WITH p1, p1Cuisine, p2, collect(id(cuisine2)) AS p2Cuisine RETURN p1.name AS from, p2.name AS to, gds.alpha.similarity.jaccard(p1Cuisine, p2Cuisine) AS similarity ----

.Results [opts="header"]

| from | to | similarity | "Karin" | "Arya" | 0.6666666666666666

.The following will return the Jaccard Similarity of Karin and the other people that have a cuisine in common: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Karin'})-[:LIKES]→(cuisine1) WITH p1, collect(id(cuisine1)) AS p1Cuisine MATCH (p2:Person)-[:LIKES]→(cuisine2) WHERE p1 <> p2 WITH p1, p1Cuisine, p2, collect(id(cuisine2)) AS p2Cuisine RETURN p1.name AS from, p2.name AS to, gds.alpha.similarity.jaccard(p1Cuisine, p2Cuisine) AS similarity ORDER BY to, similarity DESC ----

.Results [opts="header"]

| from | to | similarity | "Karin" | "Arya" | 0.6666666666666666 | "Karin" | "Michael" | 0.25 | "Karin" | "Praveena" | 0.0 | "Karin" | "Zhen" | 0.0

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Cosine Similarity algorithm in the Neo4j Graph Data Science library. :procedure-name: Cosine Similarity [.alpha] = Cosine Similarity

[abstract] — This section describes the Cosine Similarity algorithm in the Neo4j Graph Data Science library. — Cosine similarity is the cosine of the angle between two n-dimensional vectors in an n-dimensional space. It is the dot product of the two vectors divided by the product of the two vectors' lengths (or magnitudes).

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

Cosine similarity is computed using the following formula:

image::cosine-similarity.png[role="middle"]

Values range between -1 and 1, where -1 is perfectly dissimilar and 1 is perfectly similar.

The library contains both procedures and functions to calculate similarity between sets of data. The function is best used when calculating the similarity between small numbers of sets. The procedures parallelize the computation and are therefore more appropriate for computing similarities on bigger datasets.

== Use-cases - when to use the Cosine Similarity algorithm

We can use the Cosine Similarity algorithm to work out the similarity between two things. We might then use the computed similarity as part of a recommendation query. For example, to get movie recommendations based on the preferences of users who have given similar ratings to other movies that you've seen.

== Syntax

.The following will create an anonymous graph to run the algorithm on and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.similarity.cosine.write(configuration: Map) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100 ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | configuration | Map | n/a | no | Algorithm-specific configuration.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | data | List of String | null | no | A list of maps of the following structure: `{item: nodeId, weights: [double, double, double]}` or a Cypher query. | top | Integer | 0 | yes | The number of similar pairs to return. If `0`, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. If `0`, it will return as many as it finds. | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. | degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the `targets` list. If the list contains less than this amount, that node will be excluded from the calculation. | skipValue | Float | gds.util.NaN() | yes | Value to skip when executing similarity computation. A value of `null` means that skipping is disabled. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result. | graph | String | dense |

yes | The graph type ('dense' or 'cypher'). | writeBatchSize | Integer | 10000 | yes | The batch size to use when storing results. | writeRelationshipType | String | SIMILAR | yes | The relationship type to use when storing results. | writeProperty | String | score | yes | The property to use when storing results. | sourceIds | List of Integer | null | yes | The ids of items from which we need to compute similarities. Defaults to all the items provided in the `data` parameter. | targetIds | List of Integer | null | yes | The ids of items to which we need to compute similarities. Defaults to all the items provided in the `data` parameter.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodes | Integer | The number of nodes passed in. | similarityPairs | Integer | The number of pairs of similar nodes computed. | writeRelationshipType | String | The relationship type used when storing results. | writeProperty | String | The property used when storing results. | min | Float | The minimum similarity score computed. | max | Float | The maximum similarity score computed. | mean | Float | The mean of similarities scores computed. | stdDev | Float | The standard deviation of similarities scores computed. | p25 | Float | The 25 percentile of similarities scores computed. | p50 | Float | The 50 percentile of similarities scores computed. | p75 | Float | The 75 percentile of similarities scores computed. | p90 | Float | The 90 percentile of similarities scores computed. | p95 | Float | The 95 percentile of similarities scores computed. | p99 | Float | The 99 percentile of similarities scores computed. | p999 | Float | The 99.9 percentile of similarities scores computed. | p100 | Float | The 100 percentile of similarities scores computed.

.The following will create an anonymous graph to run the algorithm on and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.similarity.cosine.stream(configuration: Map) YIELD item1, item2, count1, count2, intersection, similarity ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | configuration | Map | n/a | no | Algorithm-specific configuration.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | data | List of String | null | no | A list of maps of the following structure: `{item: nodeId, weights: [double, double, double]}` or a Cypher query. | top | Integer | 0 | yes | The number of similar pairs to return. If `0`, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. If `0`, it will return as many as it finds. | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. | degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the `targets` list. If the list contains less than this amount, that node will be excluded from the calculation. | skipValue | Float | null | yes | Value to skip when executing similarity computation. A value of `null` means that skipping is disabled. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. | graph | String | dense | yes | The graph type ('dense' or 'cypher'). | sourceIds | List of Integer | null | yes | The ids of items from which we need to compute similarities. Defaults to all the items provided in the `data` parameter. | targetIds | List of Integer | null | yes | The ids of items to which we need to compute similarities. Defaults to all the items provided in the `data` parameter.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | item1 | Integer | The ID of one node in the similarity pair. | item2 | Integer | The ID of other node in the similarity pair. | count1 | Integer | The size of the `targets` list of one node. | count2 | Integer | The size of the `targets` list of other node. | intersection | Integer | The number of intersecting values in the two nodes `targets` lists. | similarity | Integer | The cosine similarity of the two nodes.

== Cosine Similarity algorithm function sample

The Cosine Similarity function computes the similarity of two lists of numbers.

[NOTE] ==== {procedure-name} is only calculated over non-NULL dimensions. When calling the function, we should provide lists that contain the overlapping items. ====

We can use it to compute the similarity of two hardcoded lists.

.The following will return the cosine similarity of two lists of numbers: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.similarity.cosine([3,8,7,5,2,9], [10,8,6,6,4,5]) AS similarity ----

.Results [opts="header",cols="1"]

| `similarity` | 0.8638935626791597

These two lists of numbers have a Cosine similarity of 0.863. We can see how this result is derived by breaking down the formula:

image::cosine-similarity2.png[role="middle"]

We can also use it to compute the similarity of nodes based on lists computed by a Cypher query.

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (french:Cuisine {name:'French'}) CREATE (italian:Cuisine {name:'Italian'}) CREATE (indian:Cuisine {name:'Indian'}) CREATE (lebanese:Cuisine {name:'Lebanese'}) CREATE (portuguese:Cuisine {name:'Portuguese'}) CREATE (british:Cuisine {name:'British'}) CREATE (mauritian:Cuisine {name:'Mauritian'})

CREATE (zhen:Person {name: "Zhen"}) CREATE (praveena:Person {name: "Praveena"}) CREATE (michael:Person {name: "Michael"}) CREATE (arya:Person {name: "Arya"}) CREATE (karin:Person {name: "Karin"})

CREATE (praveena)-[:LIKES {score: 9}]→(indian) CREATE (praveena)-[:LIKES {score: 7}]→(portuguese) CREATE (praveena)-[:LIKES {score: 8}]→(british) CREATE (praveena)-[:LIKES {score: 1}]→(mauritian)

CREATE (zhen)-[:LIKES {score: 10}]→(french) CREATE (zhen)-[:LIKES {score: 6}]→(indian) CREATE (zhen)-[:LIKES {score: 2}]→(british)

CREATE (michael)-[:LIKES {score: 8}]→(french) CREATE (michael)-[:LIKES {score: 7}]→(italian) CREATE (michael)-[:LIKES {score: 9}]→(indian) CREATE (michael)-[:LIKES {score: 3}]→(portuguese)

CREATE (arya)-[:LIKES {score: 10}]→(lebanese) CREATE (arya)-[:LIKES {score: 10}]→(italian) CREATE (arya)-[:LIKES {score: 7}]→(portuguese) CREATE (arya)-[:LIKES {score: 9}]→(mauritian)

CREATE (karin)-[:LIKES {score: 9}]→(lebanese) CREATE (karin)-[:LIKES {score: 7}]→(italian) CREATE (karin)-[:LIKES {score: 10}]→(portuguese) ----

.The following will return the Cosine similarity of Michael and Arya: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'})-[likes1:LIKES]→(cuisine) MATCH (p2:Person {name: "Arya"})-[likes2:LIKES]→(cuisine) RETURN p1.name AS from, p2.name AS to, gds.alpha.similarity.cosine(collect(likes1.score), collect(likes2.score)) AS similarity ----

.Results [opts="header"]

| from | to | similarity | "Michael" | "Arya" | 0.9788908326303921

.The following will return the Cosine similarity of Michael and the other people that have a cuisine in common: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'})-[likes1:LIKES]→(cuisine) MATCH (p2:Person)-[likes2:LIKES]→(cuisine) WHERE p2 <> p1 RETURN p1.name AS from, p2.name AS to, gds.alpha.similarity.cosine(collect(likes1.score), collect(likes2.score)) AS similarity ORDER BY similarity DESC ----

.Results [opts="header"]

| from | to | similarity | "Michael" | "Arya" | 0.9788908326303921 | "Michael" | "Zhen" | 0.9542262139256075 | "Michael" | "Praveena" | 0.9429903335828894 | "Michael" | "Karin" | 0.8498063272285821

== Cosine Similarity algorithm procedures examples

The {procedure-name} procedure computes similarity between all pairs of items. It is a symmetrical algorithm, which means that the result from computing the similarity of Item A to Item B is the same as computing the similarity of Item B to Item A. We can therefore compute the score for each pair of nodes once. We don't compute the similarity of items to themselves.

The number of computations is # items)^2 / 2) - # items, which can be very computationally expensive if we have a lot of items. [NOTE] ==== {procedure-name} is only calculated over non-NULL dimensions. The procedures expect to receive the same length lists for all items. Otherwise, longer lists will be trimmed to the length of the shortest list. ==== .The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (french:Cuisine {name:'French'}) CREATE (italian:Cuisine {name:'Italian'}) CREATE (indian:Cuisine {name:'Indian'}) CREATE (lebanese:Cuisine {name:'Lebanese'}) CREATE (portuguese:Cuisine {name:'Portuguese'}) CREATE (british:Cuisine {name:'British'}) CREATE (mauritian:Cuisine {name:'Mauritian'}) CREATE (zhen:Person {name: "Zhen"}) CREATE (praveena:Person {name: "Praveena"}) CREATE (michael:Person {name: "Michael"}) CREATE (arya:Person {name: "Arya"}) CREATE (karin:Person {name: "Karin"}) CREATE (praveena)-[:LIKES {score: 9}]→(indian) CREATE (praveena)-[:LIKES {score: 7}]→(portuguese) CREATE (praveena)-[:LIKES {score: 8}]→(british) CREATE (praveena)-[:LIKES {score: 1}]→(mauritian) CREATE (zhen)-[:LIKES {score: 10}]→(french) CREATE (zhen)-[:LIKES {score: 6}]→(indian) CREATE (zhen)-[:LIKES {score: 2}]→(british) CREATE (michael)-[:LIKES {score: 8}]→(french) CREATE (michael)-[:LIKES {score: 7}]→(italian) CREATE (michael)-[:LIKES {score: 9}]→(indian) CREATE (michael)-[:LIKES {score: 3}]→(portuguese) CREATE (arya)-[:LIKES {score: 10}]→(lebanese) CREATE (arya)-[:LIKES {score: 10}]→(italian) CREATE (arya)-[:LIKES {score: 7}]→(portuguese) CREATE (arya)-[:LIKES {score: 9}]→(mauritian) CREATE (karin)-[:LIKES {score: 9}]→(lebanese) CREATE (karin)-[:LIKES {score: 7}]→(italian) CREATE (karin)-[:LIKES {score: 10}]→(portuguese) ---- === Stream .The following will return a stream of node pairs along with their Cosine similarities: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.cosine.stream({data: data}) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header"]

| from | to | similarity | "Praveena" | "Karin" | 1.0 | "Michael" | "Arya" | 0.9788908326303921 | "Arya" | "Karin" | 0.9610904115204073 | "Zhen" | "Michael" | 0.9542262139256075 | "Praveena" | "Michael" | 0.9429903335828895 | "Zhen" | "Praveena" | 0.9191450300180579 | "Michael" | "Karin" | 0.8498063272285821 | "Praveena" | "Arya" | 0.7194014606174091 | "Zhen" | "Arya" | 0.0 | "Zhen" | "Karin" | 0.0

Praveena and Karin have the most similar food tastes, with a score of 1.0, and there are also several other pairs of users with similar tastes. The scores here are unusually high because our users haven't liked many of the same cuisines. We also have 2 pairs of users who are not similar at all. We'd probably want to filter those out, which we can do by passing in the `similarityCutoff` parameter.

.The following will return a stream of node pairs that have a similarity of at least 0.1, along with their cosine similarities: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.cosine.stream({ data: data, similarityCutoff: 0.0 }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header"]

| `from` | `to` | `similarity` | "Praveena" | "Karin" | 1.0 | "Michael" | "Arya" | 0.9788908326303921 | "Arya" | "Karin" | 0.9610904115204073 | "Zhen" | "Michael" | 0.9542262139256075 | "Praveena" | "Michael" | 0.9429903335828895 | "Zhen" | "Praveena" | 0.9191450300180579 | "Michael" | "Karin" | 0.8498063272285821 | "Praveena" | "Arya" | 0.7194014606174091

We can see that those users with no similarity have been filtered out. If we're implementing a k-Nearest Neighbors type query we might instead want to find the most similar `k` users for a given user. We can do that by passing in the `topK` parameter.

.The following will return a stream of users along with the most similar user to them (i.e. `k=1`): [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.cosine.stream({ data: data, similarityCutoff: 0.0, topK: 1 }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY from ----

.Results [opts="header",cols="1,1,1"]

| `from` | `to` | `similarity` | "Arya" | "Michael" | 0.9788908326303921 | "Karin" | "Praveena" | 1.0 | "Michael" | "Arya" | 0.9788908326303921 | "Praveena" | "Karin" | 1.0 | "Zhen" | "Michael" | 0.9542262139256075

These results will not be symmetrical. For example, the person most similar to Zhen is Michael, but the person most similar to Michael is Arya.

=== Write

.The following will find the most similar user for each user, and store a relationship between those users: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.cosine.write({ data: data, topK: 1, similarityCutoff: 0.1 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

.Results [opts="header"]

| `nodes` | `similarityPairs` | `writeRelationshipType` | `writeProperty` | `min` | `max` | `mean` | `p95` | 5 | 5 | "SIMILAR" | "score" | 0.9542236328125 | 1.0000038146972656 | 0.9824020385742187 | 1.0000038146972656

We then could write a query to find out what types of cuisine that other people similar to us might like.

.The following will find the most similar user to Praveena, and return their favourite cuisines that Praveena doesn't (yet!) like: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person {name: "Praveena"})-[:SIMILAR]→(other), (other)-[:LIKES]→(cuisine) WHERE notp)-[:LIKES]→(cuisine RETURN cuisine.name AS cuisine ----

.Results [opts="header",cols="1"]

| `cuisine` | Italian | Lebanese

=== Stats

.The following will run the algorithm and returns the result in form of statistical and measurement values [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.cosine.stats({ data: data, topK: 1, similarityCutoff: 0.1 }) YIELD nodes, similarityPairs, min, max, mean, p95 RETURN nodes, similarityPairs, min, max, mean, p95 ----

== Specifying source and target ids

Sometimes, we don't want to compute all pairs similarity, but would rather specify subsets of items to compare to each other. We do this using the `sourceIds` and `targetIds` keys in the config.

We could use this technique to compute the similarity of a subset of items to all other items.

.The following will find the most similar person (i.e. `k=1`) to Arya and Praveena: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), name: p.name, weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS personCuisines WITH personCuisines, [value in personCuisines WHERE value.name IN ["Praveena", "Arya"] value.item] AS sourceIds CALL gds.alpha.similarity.cosine.stream({ data: personCuisines, sourceIds: sourceIds, topK: 1 }) YIELD item1, item2, similarity WITH gds.util.asNode(item1) AS from, gds.util.asNode(item2) AS to, similarity RETURN from.name AS from, to.name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header",cols="1,1,1"]

| `from` | `to` | `similarity` | Praveena | Karin | 1.0 | Arya | Michael | 0.9788908326303921

== Skipping values

The algorithm checks every value in the input vectors against the `skipValue` to determine whether that value should be considered as part of the similarity computation. Vectors of different length are padded with `NaN` values which are skipped by default. Setting a `skipValue` allows skipping an additional value. A common value to skip is `0.0`.

.The following will create a sample graph storing an embedding vector for each node: [source, cypher, role=noplay, indent=0] ---- CREATE (french:Cuisine {name:'French'}) SET french.embedding = [0.0, 0.33, 0.81, 0.52, 0.41] CREATE (italian:Cuisine {name:'Italian'}) SET italian.embedding = [0.31, 0.72, 0.58, 0.67, 0.31] CREATE (indian:Cuisine {name:'Indian'}) SET indian.embedding = [0.43, 0.0, 0.98, 0.51, 0.76] CREATE (lebanese:Cuisine {name:'Lebanese'}) SET lebanese.embedding = [0.12, 0.23, 0.35, 0.31, 0.39] CREATE (portuguese:Cuisine {name:'Portuguese'}) SET portuguese.embedding = [0.47, 0.98, 0.0, 0.72, 0.89] CREATE (british:Cuisine {name:'British'}) SET british.embedding = [0.94, 0.12, 0.23, 0.4, 0.71] CREATE (mauritian:Cuisine {name:'Mauritian'}) SET mauritian.embedding = [0.31, 0.56, 0.98, 0.0, 0.62] ----

.The following will find the top 3 similarities between cuisines based on the `embedding` property: [source, cypher, role=noplay, indent=0] ---- MATCH (c:Cuisine) WITH {item:id(c), weights: c.embedding} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.cosine.stream({ data: data, skipValue: 0.0 }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity DESC, from ASC LIMIT 3 ----

.Results with skipping `0.0` values: [opts="header"]

| `from` | `to` | `similarity` | "Mauritian" | "Portuguese" | 0.9955829148132149 | "Portuguese" | "Mauritian" | 0.9955829148132149 | "Indian" | "Portuguese" | 0.9954426605601884

Without skipping `0.0` values the result would look different:

.Results without skipping `0.0` values: [opts="header"]

| `from` | `to` | `similarity` | "Lebanese" | "French" | 0.9372771447068958 | "French" | "Lebanese" | 0.9372771447068958 | "Indian" | "Lebanese" | 0.9110882139221992

== Cypher projection

If the similarity lists are very large they can take up a lot of memory. For cases where those lists contain lots of values that should be skipped, you can use the less memory-intensive approach of using Cypher statements to project the graph instead.

The Cypher projection expects to receive 3 fields:

* `item` - should contain node ids, which we can return using the `id` function. * `category` - should contain node ids, which we can return using the `id` function. * `weight` - should contain a double value.

.Set `graph:'cypher'` in the config:

[source, cypher, role=noplay, indent=0] ---- WITH 'MATCH (person:Person)-[likes:LIKES]→(c) RETURN id(person) AS item, id(c) AS category, likes.score AS weight' AS query CALL gds.alpha.similarity.cosine.write({ data: query, graph: 'cypher', topK: 1, similarityCutoff: 0.1 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p95 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Pearson Similarity algorithm in the Neo4j Graph Data Science library. :procedure-name: Pearson Similarity [.alpha] = Pearson Similarity

[abstract] — This section describes the Pearson Similarity algorithm in the Neo4j Graph Data Science library. — Pearson similarity is the covariance of the two *n*-dimensional vectors divided by the product of their standard deviations.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

Pearson similarity is computed using the following formula:

image::pearson-similarity.png[role="middle"]

Values range between -1 and 1, where -1 is perfectly dissimilar and 1 is perfectly similar.

The library contains both procedures and functions to calculate similarity between sets of data. The function is best used when calculating the similarity between small numbers of sets. The procedures parallelize the computation and are therefore more appropriate for computing similarities on bigger datasets.

== Use-cases - when to use the Pearson Similarity algorithm

We can use the Pearson Similarity algorithm to work out the similarity between two things. We might then use the computed similarity as part of a recommendation query. For example, to get movie recommendations based on the preferences of users who have given similar ratings to other movies that you've seen.

== Pearson Similarity algorithm function sample

The Pearson Similarity function computes the similarity of two lists of numbers.

[NOTE] ==== {procedure-name} is only calculated over non-NULL dimensions. When calling the function, we should provide lists that contain the overlapping items. ====

We can use it to compute the similarity of two hardcoded lists.

.The following will return the Pearson similarity of two lists of numbers: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.similarity.pearson([5,8,7,5,4,9], [7,8,6,6,4,5]) AS similarity ----

| `similarity` | 0.28767798089123053

We can also use it to compute the similarity of nodes based on lists computed by a Cypher query.

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- MERGE (home_alone:Movie {name:'Home Alone'}) MERGE (matrix:Movie {name:'The Matrix'}) MERGE (good_men:Movie {name:'A Few Good Men'}) MERGE (top_gun:Movie {name:'Top Gun'}) MERGE (jerry:Movie {name:'Jerry Maguire'}) MERGE (gruffalo:Movie {name:'The Gruffalo'})

MERGE (zhen:Person {name: 'Zhen'}) MERGE (praveena:Person {name: 'Praveena'}) MERGE (michael:Person {name: 'Michael'}) MERGE (arya:Person {name: 'Arya'}) MERGE (karin:Person {name: 'Karin'})

MERGE (zhen)-[:RATED {score: 2}]→(home_alone) MERGE (zhen)-[:RATED {score: 2}]→(good_men) MERGE (zhen)-[:RATED {score: 3}]→(matrix) MERGE (zhen)-[:RATED {score: 6}]→(jerry)

MERGE (praveena)-[:RATED {score: 6}]→(home_alone) MERGE (praveena)-[:RATED {score: 7}]→(good_men) MERGE (praveena)-[:RATED {score: 8}]→(matrix) MERGE (praveena)-[:RATED {score: 9}]→(jerry)

MERGE (michael)-[:RATED {score: 7}]→(home_alone) MERGE (michael)-[:RATED {score: 9}]→(good_men) MERGE (michael)-[:RATED {score: 3}]→(jerry) MERGE (michael)-[:RATED {score: 4}]→(top_gun)

MERGE (arya)-[:RATED {score: 8}]→(top_gun) MERGE (arya)-[:RATED {score: 1}]→(matrix) MERGE (arya)-[:RATED {score: 10}]→(jerry) MERGE (arya)-[:RATED {score: 10}]→(gruffalo)

MERGE (karin)-[:RATED {score: 9}]→(top_gun) MERGE (karin)-[:RATED {score: 7}]→(matrix) MERGE (karin)-[:RATED {score: 7}]→(home_alone) MERGE (karin)-[:RATED {score: 9}]→(gruffalo) ----

.The following will return the Pearson similarity of Arya and Karin: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Arya'})-[rated:RATED]→(movie) WITH p1, gds.alpha.similarity.asVector(movie, rated.score) AS p1Vector MATCH (p2:Person {name: 'Karin'})-[rated:RATED]→(movie) WITH p1, p2, p1Vector, gds.alpha.similarity.asVector(movie, rated.score) AS p2Vector RETURN p1.name AS from, p2.name AS to, gds.alpha.similarity.pearson(p1Vector, p2Vector, {vectorType: "maps"}) AS similarity ----

.Results [opts="header"]

| `from`| `to` | `similarity` |"Arya" | "Karin" | 0.8194651785206903

In this example, we pass in `vectorType: "maps"` as an extra parameter, as well as using the `gds.alpha.similarity.asVector` function to construct a vector of maps containing each movie and the corresponding rating. We do this because the Pearson Similarity algorithm needs to compute the average of **all** the movies that a user has reviewed, not just the ones that they have in common with the user we're comparing them to. We can't therefore just pass in collections of the ratings of movies that have been reviewed by both people.

.The following will return the Pearson similarity of Arya and other people that have rated at least one movie: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Arya'})-[rated:RATED]→(movie) WITH p1, gds.alpha.similarity.asVector(movie, rated.score) AS p1Vector MATCH (p2:Person)-[rated:RATED]→(movie) WHERE p2 <> p1 WITH p1, p2, p1Vector, gds.alpha.similarity.asVector(movie, rated.score) AS p2Vector RETURN p1.name AS from, p2.name AS to, gds.alpha.similarity.pearson(p1Vector, p2Vector, {vectorType: "maps"}) AS similarity ORDER BY similarity DESC ----

.Results [opts="header"]

| `from` | `to` | `similarity` | "Arya" | "Karin" | 0.8194651785206903 | "Arya" | "Zhen" | 0.4839533792540704 | "Arya" | "Praveena" | 0.09262336892949784 | "Arya" | "Michael" | -0.9551953674747637

== Pearson Similarity algorithm procedures sample

The {procedure-name} procedure computes similarity between all pairs of items. It is a symmetrical algorithm, which means that the result from computing the similarity of Item A to Item B is the same as computing the similarity of Item B to Item A. We can therefore compute the score for each pair of nodes once. We don't compute the similarity of items to themselves.

The number of computations is `# items)^2 / 2) - # items`, which can be very computationally expensive if we have a lot of items. [NOTE] ==== {procedure-name} is only calculated over non-NULL dimensions. The procedures expect to receive the same length lists for all items. Otherwise, longer lists will be trimmed to the length of the shortest list. ==== .The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- MERGE (home_alone:Movie {name:'Home Alone'}) MERGE (matrix:Movie {name:'The Matrix'}) MERGE (good_men:Movie {name:'A Few Good Men'}) MERGE (top_gun:Movie {name:'Top Gun'}) MERGE (jerry:Movie {name:'Jerry Maguire'}) MERGE (gruffalo:Movie {name:'The Gruffalo'}) MERGE (zhen:Person {name: 'Zhen'}) MERGE (praveena:Person {name: 'Praveena'}) MERGE (michael:Person {name: 'Michael'}) MERGE (arya:Person {name: 'Arya'}) MERGE (karin:Person {name: 'Karin'}) MERGE (zhen)-[:RATED {score: 2}]→(home_alone) MERGE (zhen)-[:RATED {score: 2}]→(good_men) MERGE (zhen)-[:RATED {score: 3}]→(matrix) MERGE (zhen)-[:RATED {score: 6}]→(jerry) MERGE (praveena)-[:RATED {score: 6}]→(home_alone) MERGE (praveena)-[:RATED {score: 7}]→(good_men) MERGE (praveena)-[:RATED {score: 8}]→(matrix) MERGE (praveena)-[:RATED {score: 9}]→(jerry) MERGE (michael)-[:RATED {score: 7}]→(home_alone) MERGE (michael)-[:RATED {score: 9}]→(good_men) MERGE (michael)-[:RATED {score: 3}]→(jerry) MERGE (michael)-[:RATED {score: 4}]→(top_gun) MERGE (arya)-[:RATED {score: 8}]→(top_gun) MERGE (arya)-[:RATED {score: 1}]→(matrix) MERGE (arya)-[:RATED {score: 10}]→(jerry) MERGE (arya)-[:RATED {score: 10}]→(gruffalo) MERGE (karin)-[:RATED {score: 9}]→(top_gun) MERGE (karin)-[:RATED {score: 7}]→(matrix) MERGE (karin)-[:RATED {score: 7}]→(home_alone) MERGE (karin)-[:RATED {score: 9}]→(gruffalo) ---- === Stream .The following will return a stream of node pairs along with their Pearson similarities: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (m:Movie) OPTIONAL MATCH (p)-[rated:RATED]→(m) WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.pearson.stream({ data: data, topK: 0 }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header"]

| `from` | `to` | `similarity` | "Zhen" | "Praveena" | 0.8865926413116155 | "Zhen" | "Karin" | 0.8320502943378437 | "Arya" | "Karin" | 0.8194651785206903 | "Zhen" | "Arya" | 0.4839533792540704 | "Praveena" | "Karin" | 0.4472135954999579 | "Praveena" | "Arya" | 0.09262336892949784 | "Praveena" | "Michael" | -0.788492846568306 | "Zhen" | "Michael" | -0.9091365607973364 | "Michael" | "Arya" | -0.9551953674747637 | "Michael" | "Karin" | -0.9863939238321437

Zhen and Praveena are the most similar with a score of 0.88. The maximum score is 1.0 We also have 4 pairs of users who are not similar at all. We'd probably want to filter those out, which we can do by passing in the `similarityCutoff` parameter.

.The following will return a stream of node pairs that have a similarity of at least 0.1, along with their Pearson similarities: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (m:Movie) OPTIONAL MATCH (p)-[rated:RATED]→(m) WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.pearson.stream({ data: data, similarityCutoff: 0.1, topK: 0 }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header"]

| `from` | `to` | `similarity` | "Zhen" | "Praveena" | 0.8865926413116155 | "Zhen" | "Karin" | 0.8320502943378437 | "Arya" | "Karin" | 0.8194651785206903 | "Zhen" | "Arya" | 0.4839533792540704 | "Praveena" | "Karin" | 0.4472135954999579

We can see that those users with no similarity have been filtered out. If we're implementing a k-Nearest Neighbors type query we might instead want to find the most similar k users for a given user. We can do that by passing in the topK parameter.

.The following will return a stream of users along with the most similar user to them (i.e. k=1): [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (m:Movie) OPTIONAL MATCH (p)-[rated:RATED]→(m) WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.pearson.stream({ data: data, topK:1, similarityCutoff: 0.0 }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header",cols="1,1,1"]

| from | to | similarity | "Zhen" | "Praveena" | 0.8865926413116155 | "Praveena" | "Zhen" | 0.8865926413116155 | "Karin" | "Zhen" | 0.8320502943378437 | "Arya" | "Karin" | 0.8194651785206903

These results will not necessarily be symmetrical. For example, the person most similar to Arya is Karin, but the person most similar to Karin is Zhen.

=== Write

.The following will find the most similar user for each user, and store a relationship between those users: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (m:Movie) OPTIONAL MATCH (p)-[rated:RATED]→(m) WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.pearson.write({ data: data, topK: 1, similarityCutoff: 0.1 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

.Results [opts="header"]

| nodes | similarityPairs | writeRelationshipType | writeProperty | min | max | mean | p95 | 5 | 4 | "SIMILAR" | "score" | 0.8194618225097656 | 0.8865890502929688 | 0.8561716079711914 | 0.8865890502929688

We then could write a query to find out which are the movies that other people similar to us liked.

.The following will find the most similar user to Karin, and return their movies that Karin didn't (yet!) rate: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person {name: 'Karin'})-[:SIMILAR]→(other), (other)-[r:RATED]→(movie) WHERE notp)-[:RATED]→(movie and r.score >= 5 RETURN movie.name AS movie ----

.Results [opts="header",cols="1"]

| movie | Jerry Maguire

=== Stats

.The following will run the algorithm and returns the result in form of statistical and measurement values [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (m:Movie) OPTIONAL MATCH (p)-[rated:RATED]→(m) WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.pearson.stats({ data: data, topK: 1, similarityCutoff: 0.1 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

== Specifying source and target ids

Sometimes, we don't want to compute all pairs similarity, but would rather specify subsets of items to compare to each other. We do this using the `sourceIds` and `targetIds` keys in the config.

We could use this technique to compute the similarity of a subset of items to all other items.

.The following will find the most similar person (i.e. `k=1`) to Arya and Praveena: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (m:Movie) OPTIONAL MATCH (p)-[rated:RATED]→(m) WITH {item:id(p), name: p.name, weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData WITH collect(userData) AS personCuisines WITH personCuisines, [value in personCuisines WHERE value.name IN ["Praveena", "Arya"] value.item] AS sourceIds CALL gds.alpha.similarity.pearson.stream({ data: personCuisines, sourceIds: sourceIds, topK: 1 }) YIELD item1, item2, similarity WITH gds.util.asNode(item1) AS from, gds.util.asNode(item2) AS to, similarity RETURN from.name AS from, to.name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header",cols="1,1,1"]

| `from` | `to` | `similarity` | Praveena | Zhen | 0.8865926413116155 | Arya | Karin | 0.8194651785206903

== Skipping values

By default the `skipValue` parameter is `gds.util.NaN()`. The algorithm checks every value against the `skipValue` to determine whether that value should be considered as part of the similarity result. For cases where no values should be skipped, skipping can be disabled by setting `skipValue` to `null`.

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- MERGE (home_alone:Movie {name:'Home Alone'}) SET home_alone.embedding = [0.71, 0.33, 0.81, 0.52, 0.41] MERGE (matrix:Movie {name:'The Matrix'}) SET matrix.embedding = [0.31, 0.72, 0.58, 0.67, 0.31] MERGE (good_men:Movie {name:'A Few Good Men'}) SET good_men.embedding = [0.43, 0.26, 0.98, 0.51, 0.76] MERGE (top_gun:Movie {name:'Top Gun'}) SET top_gun.embedding = [0.12, 0.23, 0.35, 0.31, 0.3] MERGE (jerry:Movie {name:'Jerry Maguire'}) SET jerry.embedding = [0.47, 0.98, 0.81, 0.72, 0] ----

.The following will find the similarity between movies based on the `embedding` property: [source, cypher, role=noplay, indent=0] ---- MATCH (m:Movie) WITH {item:id(m), weights: m.embedding} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.pearson.stream({ data: data, skipValue: null }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header",cols="1,1,1"]

| from | to | similarity | The Matrix | Jerry Maguire | 0.8689113641953199 | A Few Good Men | Top Gun | 0.6846566091701214 | Home Alone | A Few Good Men | 0.556559508845268 | The Matrix | Top Gun | 0.39320549183813097 | Home Alone | Jerry Maguire | 0.10026787755714502 | Top Gun | Jerry Maguire | 0.056232940630734043 | Home Alone | Top Gun | 0.006048691083898151 | Home Alone | The Matrix | -0.23435051666541426 | The Matrix | A Few Good Men | -0.2545273235448378 | A Few Good Men | Jerry Maquire | -0.31099199179883635

== Cypher projection

If the similarity lists are very large they can take up a lot of memory. For cases where those lists contain lots of values that should be skipped, you can use the less memory-intensive approach of using Cypher statements to project the graph instead.

The Cypher projection expects to receive 3 fields:

* item - should contain node ids, which we can return using the id function. * category - should contain node ids, which we can return using the id function. * weight - should contain a double value.

.Set graph:'cypher' in the config:

[source, cypher, role=noplay, indent=0] ---- WITH "MATCH (person:Person)-[rated:RATED]→(c) RETURN id(person) AS item, id(c) AS category, rated.score AS weight" AS query CALL gds.alpha.similarity.pearson({ data: query, graph: 'cypher', topK: 1, similarityCutoff: 0.1 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p95 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

.Results [opts="header"]

| nodes | similarityPairs | writeRelationshipType | writeProperty | min | max | mean | p95 | 5 | 4 | "SIMILAR" | "score" | 0.8194618225097656 | 0.8865890502929688 | 0.8561716079711914 | 0.8865890502929688

== Syntax

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.similarity.pearson.write(configuration: Map) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100 ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | configuration | Map | n/a | no | Algorithm-specific configuration.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | data | List or String | null | no | A list of maps of the following structure: {item: nodeId, weights: [double, double, double]} or a Cypher query. | top | Integer | 0 | yes | The number of similar pairs to return. If 0, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. If 0, it will return as many as it finds. | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. | degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the targets list. If the list contains less than this amount, that node will be excluded from the calculation. | skipValue | Float | gds.util.NaN() | yes | Value to skip when executing similarity computation. A value of null means that skipping is disabled. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running

the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result. | graph | String | dense | yes | The graph name ('dense' or 'cypher'). | writeBatchSize | Integer | 10000 | yes | The batch size to use when storing results. | writeRelationshipType | String | SIMILAR | yes | The relationship type to use when storing results. | writeProperty | String | score | yes | The property to use when storing results. | sourceIds | List of String | null | yes | The ids of items from which we need to compute similarities. Defaults to all the items provided in the `data` parameter. | targetIds | List of String | null | yes | The ids of items to which we need to compute similarities. Defaults to all the items provided in the `data` parameter.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodes | Integer | The number of nodes passed in. | similarityPairs | Integer | The number of pairs of similar nodes computed. | writeRelationshipType | String | The relationship type used when storing results. | writeProperty | String | The property used when storing results. | min | Float | The minimum similarity score computed. | max | Float | The maximum similarity score computed. | mean | Float | The mean of similarities scores computed. | stdDev | Float | The standard deviation of similarities scores computed. | p25 | Float | The 25 percentile of similarities scores computed. | p50 | Float | The 50 percentile of similarities scores computed. | p75 | Float | The 75 percentile of similarities scores computed. | p90 | Float | The 90 percentile of similarities scores computed. | p95 | Float | The 95 percentile of similarities scores computed. | p99 | Float | The 99 percentile of similarities scores computed. | p999 | Float | The 99.9 percentile of similarities scores computed. | p100 | Float | The 100 percentile of similarities scores computed.

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.similarity.pearson.stream(configuration: Map) YIELD item1, item2, count1, count2, intersection, similarity ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | configuration | Map | n/a | no | Algorithm-specific configuration.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | data | List or String | null | no | A list of maps of the following structure: `{item: nodeId, weights: [double, double, double]}` or a Cypher query. | top | Integer | 0 | yes | The number of similar pairs to return. If `0`, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. If `0`, it will return as many as it finds. | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. | degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the `targets` list. If the list contains less than this amount, that node will be excluded from the calculation. | skipValue | Float | gds.util.NaN() | yes | Value to skip when executing similarity computation. A value of `null` means that skipping is disabled. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. | graph | String | dense | yes | The graph name ('dense' or 'cypher'). | sourceIds | List of Integer | null | yes | The ids of items from which we need to compute similarities. Defaults to all the items provided in the `data` parameter. | targetIds | List of Integer | null | yes | The ids of items to which we need to compute similarities. Defaults to all the items provided in the `data` parameter.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | item1 | Integer | The ID of one node in the similarity pair. | item2 | Integer | The ID of other node in the similarity pair. | count1 | Integer | The size of the `targets` list of one node. | count2 | Integer | The size of the `targets` list of other node. | intersection | Integer | The number of intersecting values in the two nodes `targets` lists. | similarity | Integer | The pearson similarity of the two nodes.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Euclidean Distance algorithm in the Neo4j Graph Data Science library. :procedure-name: Euclidean Distance

[.alpha] = Euclidean Distance

[abstract] — This section describes the Euclidean Distance algorithm in the Neo4j Graph Data Science library. — Euclidean distance measures the straight line distance between two points in n-dimensional space.

[.alpha-note] [.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

Euclidean distance is computed using the following formula:

image::euclidean.png[role="middle"]

The library contains both procedures and functions to calculate similarity between sets of data. The function is best used when calculating the similarity between small numbers of sets. The procedures parallelize the computation and are therefore more appropriate for computing similarities on bigger datasets.

== Use-cases - when to use the Euclidean Distance algorithm

We can use the Euclidean Distance algorithm to work out the similarity between two things. We might then use the computed similarity as part of a recommendation query. For example, to get movie recommendations based on the preferences of users who have given similar ratings to other movies that you've seen.

== Euclidean Distance algorithm function sample

The Euclidean Distance function computes the similarity of two lists of numbers.

[NOTE] ==== {procedure-name} is only calculated over non-NULL dimensions. When calling the function, we should provide lists that contain the overlapping items. ====

We can use it to compute the similarity of two hardcoded lists.

.The following will return the euclidean similarity of two lists of numbers: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.similarity.euclideanDistance([3,8,7,5,2,9], [10,8,6,6,4,5]) AS similarity ----

.Results [opts="header",cols="1"]

| `similarity` | 8.426149773176359

These two lists of numbers have a euclidean distance of 8.42.

We can also use it to compute the similarity of nodes based on lists computed by a Cypher query.

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- MERGE (french:Cuisine {name:'French'}) MERGE (italian:Cuisine {name:'Italian'}) MERGE (indian:Cuisine {name:'Indian'}) MERGE (lebanese:Cuisine {name:'Lebanese'}) MERGE (portuguese:Cuisine {name:'Portuguese'}) MERGE (british:Cuisine {name:'British'}) MERGE (mauritian:Cuisine {name:'Mauritian'})

MERGE (zhen:Person {name: "Zhen"}) MERGE (praveena:Person {name: "Praveena"}) MERGE (michael:Person {name: "Michael"}) MERGE (arya:Person {name: "Arya"}) MERGE (karin:Person {name: "Karin"})

MERGE (praveena)-[:LIKES {score: 9}]→(indian) MERGE (praveena)-[:LIKES {score: 7}]→(portuguese) MERGE (praveena)-[:LIKES {score: 8}]→(british) MERGE (praveena)-[:LIKES {score: 1}]→(mauritian)

MERGE (zhen)-[:LIKES {score: 10}]→(french) MERGE (zhen)-[:LIKES {score: 6}]→(indian) MERGE (zhen)-[:LIKES {score: 2}]→(british)

MERGE (michael)-[:LIKES {score: 8}]→(french) MERGE (michael)-[:LIKES {score: 7}]→(italian) MERGE (michael)-[:LIKES {score: 9}]→(indian) MERGE (michael)-[:LIKES {score: 3}]→(portuguese)

MERGE (arya)-[:LIKES {score: 10}]→(lebanese) MERGE (arya)-[:LIKES {score: 10}]→(italian) MERGE (arya)-[:LIKES {score: 7}]→(portuguese) MERGE (arya)-[:LIKES {score: 9}]→(mauritian)

MERGE (karin)-[:LIKES {score: 9}]→(lebanese) MERGE (karin)-[:LIKES {score: 7}]→(italian) MERGE (karin)-[:LIKES {score: 10}]→(portuguese) ----

.The following will return the Euclidean distance of Zhen and Praveena: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Zhen'})-[likes1:LIKES]→(cuisine) MATCH (p2:Person {name: 'Praveena'})-[likes2:LIKES]→(cuisine) RETURN p1.name AS from, p2.name AS to, gds.alpha.similarity.euclideanDistance(collect(likes1.score), collect(likes2.score)) AS similarity ----

.Results [opts="header"]

| `from` | `to` | `similarity` | "Zhen" | "Praveena" | 6.708203932499369

.The following will return the Euclidean distance of Zhen and the other people that have a cuisine in common: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Zhen'})-[likes1:LIKES]→(cuisine) MATCH (p2:Person)-[likes2:LIKES]→(cuisine) WHERE p2 <> p1 RETURN p1.name AS from, p2.name AS to, gds.alpha.similarity.euclideanDistance(collect(likes1.score), collect(likes2.score)) AS similarity ORDER BY similarity DESC ----

.Results [opts="header"]

| `from` | `to` | `similarity` | "Zhen" | "Praveena" | 6.708203932499369 | "Zhen" | "Michael" | 3.605551275463989

== Euclidean Distance algorithm procedures sample

The {procedure-name} procedure computes similarity between all pairs of items. It is a symmetrical algorithm, which means that the result from computing the similarity of Item A to Item B is the same as computing the similarity of Item B to Item A. We can therefore compute the score for each pair of nodes once. We don't compute the similarity of items to themselves.

The number of computations is `# items)^2 / 2) - # items`, which can be very computationally expensive if we have a lot of items. [NOTE] ==== {procedure-name} is only calculated over non-NULL dimensions. The procedures expect to receive the same length lists for all items. Otherwise, longer lists will be trimmed to the length of the shortest list. ==== .The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- MERGE (french:Cuisine {name:'French'}) MERGE (italian:Cuisine {name:'Italian'}) MERGE (indian:Cuisine {name:'Indian'}) MERGE (lebanese:Cuisine {name:'Lebanese'}) MERGE (portuguese:Cuisine {name:'Portuguese'}) MERGE (karin:Person {name: "Karin"}) MERGE (praveena)-[:LIKES {score: 9}]→(indian) MERGE (praveena)-[:LIKES {score: 7}]→(portuguese) MERGE (praveena)-[:LIKES {score: 8}]→(british) MERGE (praveena)-[:LIKES {score: 1}]→(mauritian) MERGE (zhen)-[:LIKES {score: 10}]→(french) MERGE (zhen)-[:LIKES {score: 6}]→(indian) MERGE (zhen)-[:LIKES {score: 2}]→(british) MERGE (british:Cuisine {name:'British'}) MERGE (mauritian:Cuisine {name:'Mauritian'}) MERGE (zhen:Person {name: "Zhen"}) MERGE (praveena:Person {name: "Praveena"}) MERGE (michael:Person {name: "Michael"}) MERGE (arya:Person {name: "Arya"}) MERGE (michael)-[:LIKES {score: 8}]→(french) MERGE (michael)-[:LIKES {score: 7}]→(italian) MERGE (michael)-[:LIKES {score: 9}]→(indian) MERGE (michael)-[:LIKES {score: 3}]→(portuguese) MERGE (arya)-[:LIKES {score: 10}]→(lebanese) MERGE (arya)-[:LIKES {score: 10}]→(italian) MERGE (arya)-[:LIKES {score: 7}]→(portuguese) MERGE (arya)-[:LIKES {score: 9}]→(mauritian) MERGE (karin)-[:LIKES {score: 9}]→(lebanese) MERGE (karin)-[:LIKES {score: 7}]→(italian) MERGE (karin)-[:LIKES {score: 10}]→(portuguese) ---- === Stream .The following will return a stream of node pairs, along with their intersection and euclidean similarities: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.euclidean.stream({ data: data, topK: 0 }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity ----

.Results [opts="header"]

| `from` | `to` | `similarity` | "Praveena" | "Karin" | 3.0 | "Zhen" | "Michael" | 3.605551275463989 | "Praveena" | "Michael" | 4.0 | "Arya" | "Karin" | 4.358898943540674 | "Michael" | "Arya" | 5.0 | "Zhen" | "Praveena" | 6.708203932499369 | "Michael" | "Karin" | 7.0 | "Praveena" | "Arya" | 8.0 | "Zhen" | "Arya" | NaN | "Zhen" | "Karin" | NaN

Praveena and Karin have the most similar food preferences, with a euclidean distance of 3.0. Lower scores are better here; a score of 0 would indicate that users have exactly the same preferences.

We can also see at the bottom of the list that Zhen and Arya and Zhen and Karin have a similarity of `NaN`. We get this result because there is no overlap in their food preferences.

We can filter those results out using the `gds.util.isFinite` function.

.The following will return a stream of node pairs, along with their intersection and finite euclidean similarities: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.euclidean.stream({ data: data, topK: 0 }) YIELD item1, item2, count1, count2, similarity WHERE gds.util.isFinite(similarity) RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity ----

.Results [opts="header"]

| `from` | `to` | `similarity` | "Praveena" | "Karin" | 3.0 | "Zhen" | "Michael" | 3.605551275463989 | "Praveena" | "Michael" | 4.0 | "Arya" | "Karin" | 4.358898943540674 | "Michael" | "Arya" | 5.0 | "Zhen" | "Praveena" | 6.708203932499369 | "Michael" | "Karin" | 7.0 | "Praveena" | "Arya" | 8.0

We can see in these results that Zhen and Arya and Zhen and Karin have been removed.

We might decide that we don't want to see users with a similarity above 4 returned in our results. If so, we can filter those out by passing in the `similarityCutoff` parameter.

.The following will return a stream of node pairs that have a similarity of at most 4, along with their euclidean distance: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.euclidean.stream({ data: data, similarityCutoff: 4.0, topK: 0 }) YIELD item1, item2, count1, count2, similarity WHERE gds.util.isFinite(similarity) RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity ----

.Results [opts="header"]

| `from` | `to` | `similarity` | "Praveena" | "Karin" | 3.0 | "Zhen" | "Michael" | 3.605551275463989 | "Praveena" | "Michael" | 4.0

We can see that those users with a high score have been filtered out. If we're implementing a k-Nearest Neighbors type query we might instead want to find the most similar `k` users for a given user. We can do that by passing in the `topK` parameter.

.The following will return a stream of users along with the most similar user to them (i.e. `k=1`): [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.euclidean.stream({ data: data, topK: 1 }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY from ----

.Results [opts="header",cols="1,1,1"]

| `from` | `to` | `similarity` | "Arya" | "Karin" | 4.358898943540674 | "Karin" | "Praveena" | 3.0 | "Michael" | "Zhen" | 3.605551275463989 | "Praveena" | "Karin" | 3.0 | "Zhen" | "Michael" | 3.605551275463989

These results will not necessarily be symmetrical. For example, the person most similar to Arya is Karin, but the person most similar to Karin is Praveena.

=== Write

.The following will find the most similar user for each user, and store a relationship between those users: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.euclidean.write({ data: data, topK: 1 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

.Results [opts="header"]

| `nodes` | `similarityPairs` | `writeRelationshipType` | `writeProperty` | `min` | `max` | `mean` | `p95` | 5 | 5 | "SIMILAR" | "score" | 3.0 | 4.3589019775390625 | 3.5139984130859374 | 4.3589019775390625

We then could write a query to find out what types of cuisine that other people similar to us might like.

.The following will find the most similar user to Praveena, and return their favorite cuisines that Praveena doesn't (yet!) like: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person {name: "Praveena"})-[:SIMILAR]→(other), (other)-[:LIKES]→(cuisine) WHERE notp)-[:LIKES]→(cuisine RETURN cuisine.name AS cuisine ----

.Results [opts="header",cols="1"]

| `cuisine` | Italian | Lebanese

=== Stats

.The following will run the algorithm and returns the result in form of statistical and measurement values [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.euclidean.stats({ data: data, topK: 1 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

== Specifying source and target ids

Sometimes, we don't want to compute all pairs similarity, but would rather specify subsets of items to compare to each other. We do this using the `sourceIds` and `targetIds` keys in the config.

We could use this technique to compute the similarity of a subset of items to all other items.

.The following will find the most similar person (i.e. `k=1`) to Arya and Praveena: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person), (c:Cuisine) OPTIONAL MATCH (p)-[likes:LIKES]→(c) WITH {item:id(p), name: p.name, weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData WITH collect(userData) AS personCuisines WITH personCuisines, [value in personCuisines WHERE value.name IN ["Praveena", "Arya"] value.item] AS sourceIds CALL gds.alpha.similarity.euclidean.stream({ data: personCuisines, sourceIds: sourceIds, topK: 1 }) YIELD item1, item2, similarity WITH gds.util.asNode(item1) AS from, gds.util.asNode(item2) AS to, similarity RETURN from.name AS from, to.name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header",cols="1,1,1"]

| `from` | `to` | `similarity` | "Arya" | "Karin" | 4.358898943540674 | "Praveena" | "Karin" | 3.0

== Skipping values

By default the `skipValue` parameter is `gds.util.NaN()`. The algorithm checks every value against the `skipValue` to determine whether that value should be considered as part of the similarity result. For cases where no values should be skipped, skipping can be disabled by setting `skipValue` to `null`.

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- MERGE (french:Cuisine {name:'French'}) SET french.embedding = [0.71, 0.33, 0.81, 0.52, 0.41] MERGE (italian:Cuisine {name:'Italian'}) SET italian.embedding = [0.31, 0.72, 0.58, 0.67, 0.31] MERGE (indian:Cuisine {name:'Indian'}) SET indian.embedding = [0.43, 0.26, 0.98, 0.51, 0.76] MERGE (lebanese:Cuisine {name:'Lebanese'}) SET lebanese.embedding = [0.12, 0.23, 0.35, 0.31, 0.39] MERGE (portuguese:Cuisine {name:'Portuguese'}) SET portuguese.embedding = [0.47, 0.98, 0.81, 0.72, 0.89] MERGE (british:Cuisine {name:'British'}) SET british.embedding = [0.94, 0.12, 0.23, 0.4, 0.71] MERGE (mauritian:Cuisine {name:'Mauritian'}) SET mauritian.embedding = [0.31, 0.56, 0.98, 0.21, 0.62] ----

.The following will find the similarity between cuisines based on the `embedding` property: [source, cypher, role=noplay, indent=0] ---- MATCH (c:Cuisine) WITH {item:id(c), weights: c.embedding} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.euclidean.stream({ data: data, skipValue: null }) YIELD item1, item2, count1, count2, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity DESC ----

== Cypher projection

If the similarity lists are very large they can take up a lot of memory. For cases where those lists contain lots of values that should be skipped, you can use the less memory-intensive approach of using Cypher statements to project the graph instead.

The Cypher projection expects to receive 3 fields:

* `item` - should contain node ids, which we can return using the `id` function. * `category` - should contain node ids, which we can return using the `id` function. * `weight` - should contain a double value.

.Set `graph:'cypher'` in the config:

[source, cypher, role=noplay, indent=0] ---- WITH "MATCH (person:Person)-[likes:LIKES]→(c) RETURN id(person) AS item, id(c) AS category, likes.score AS weight" AS query CALL gds.alpha.similarity.euclidean.write({ data: query, graph: 'cypher', topK: 1, similarityCutoff: 4.0 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p95 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

== Syntax

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.similarity.euclidean.write(configuration: Map) YIELD nodes, similarityPair, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100 ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | configuration | Map | n/a | no | Algorithm-specific configuration.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | data | List or String | null | no | A list of maps of the following structure: `{item: nodeId, weights: [double, double, double]}` or a Cypher query. | top | Integer | 0 | yes | The number of similar pairs to return. If `0`, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. If `0`, it will return as many as it finds. | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. |

degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the `targets` list. If the list contains less than this amount, that node will be excluded from the calculation. | skipValue | Float | gds.util.NaN() | yes | Value to skip when executing similarity computation. A value of `null` means that skipping is disabled. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result. | graph | String | dense | yes | The graph name ('dense' or 'cypher'). | writeBatchSize | Integer | 10000 | yes | The batch size to use when storing results. | writeRelationshipType | String | SIMILAR | yes | The relationship type to use when storing results. | writeProperty | String | score | yes | The property to use when storing results. | sourceIds | List of String | null | yes | The ids of items from which we need to compute similarities. Defaults to all the items provided in the `data` parameter. | targetIds | List of String | null | yes | The ids of items to which we need to compute similarities. Defaults to all the items provided in the `data` parameter.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodes | Integer | The number of nodes passed in. | similarityPairs | Integer | The number of pairs of similar nodes computed. | writeRelationshipType | String | The relationship type used when storing results. | writeProperty | String | The property used when storing results. | min | Float | The minimum similarity score computed. | max | Float | The maximum similarity score computed. | mean | Float | The mean of similarities scores computed. | stdDev | Float | The standard deviation of similarities scores computed. | p25 | Float | The 25 percentile of similarities scores computed. | p50 | Float | The 50 percentile of similarities scores computed. | p75 | Float | The 75 percentile of similarities scores computed. | p90 | Float | The 90 percentile of similarities scores computed. | p95 | Float | The 95 percentile of similarities scores computed. | p99 | Float | The 99 percentile of similarities scores computed. | p999 | Float | The 99.9 percentile of similarities scores computed. | p100 | Float | The 100 percentile of similarities scores computed.

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.similarity.euclidean.stream(configuration: Map) YIELD item1, item2, count1, count2, intersection, similarity ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | configuration | Map | n/a | no | Algorithm-specific configuration.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | data | List or String | null | no | A list of maps of the following structure: `{item: nodeId, weights: [double, double, double]}` or a Cypher query. | top | Integer | 0 | yes | The number of similar pairs to return. If `0`, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. If `0`, it will return as many as it finds. | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. | degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the `targets` list. If the list contains less than this amount, that node will be excluded from the calculation. | skipValue | Float | gds.util.NaN() | yes | Value to skip when executing similarity computation. A value of `null` means that skipping is disabled. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. | graph | String | dense | yes | The graph name ('dense' or 'cypher'). | sourceIds | List of Integer | null | yes | The ids of items from which we need to compute similarities. Defaults to all the items

provided in the `data` parameter. | targetIds | List of Integer | null | yes | The ids of items to which we need to compute similarities. Defaults to all the items provided in the `data` parameter.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | item1 | Integer | The ID of one node in the similarity pair. | item2 | Integer | The ID of other node in the similarity pair. | count1 | Integer | The size of the `targets` list of one node. | count2 | Integer | The size of the `targets` list of other node. | intersection | Integer | The number of intersecting values in the two nodes `targets` lists. | similarity | Integer | The euclidean similarity of the two nodes.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Overlap Similarity algorithm in the Neo4j Graph Data Science library. [.alpha] = Overlap Similarity

[abstract] — This section describes the Overlap Similarity algorithm in the Neo4j Graph Data Science library. — Overlap similarity measures overlap between two sets. It is defined as the size of the intersection of two sets, divided by the size of the smaller of the two sets.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

Overlap similarity is computed using the following formula:

image::overlap.png[role="middle"]

The library contains both procedures and functions to calculate similarity between sets of data. The function is best used when calculating the similarity between small numbers of sets. The procedures parallelize the computation, and are therefore more appropriate for computing similarities on bigger datasets.

== Use-cases - when to use the Overlap Similarity algorithm

We can use the Overlap Similarity algorithm to work out which things are subsets of others. We might then use these computed subsets to learn a taxonomy from tagged data, as described by Jesús Barrasa.

== Overlap Similarity algorithm function sample

.The following will return the Overlap similarity of two lists of numbers: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.similarity.overlap([1,2,3], [1,2,4,5]) AS similarity ----

.Results [opts="header",cols="1"]

| `similarity` | 0.6666666666666666

These two lists of numbers have an overlap similarity of 0.66. We can see how this result is derived by breaking down the formula: ---- O(A,B) = (▯A ∩ B▯) / (min(▯A	,	B	)) O(A,B) = 2 / min(3,4) = 2 / 3 = 0.66 ---- == Overlap Similarity algorithm procedures sample .The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (fahrenheit451:Book {title:'Fahrenheit 451'}), (dune:Book {title:'Dune'}), (hungerGames:Book {title:'The Hunger Games'}), (nineteen84:Book {title:'1984'}), (gatsby:Book {title:'The Great Gatsby'}), (scienceFiction:Genre {name: "Science Fiction"}), (fantasy:Genre {name: "Fantasy"}), (dystopia:Genre {name: "Dystopia"}), (classics:Genre {name: "Classics"}), (fahrenheit451)-[:HAS_GENRE]→(dystopia), (fahrenheit451)-[:HAS_GENRE]→(scienceFiction), (fahrenheit451)-[:HAS_GENRE]→(fantasy), (fahrenheit451)-[:HAS_GENRE]→(classics), (hungerGames)-[:HAS_GENRE]→(scienceFiction), (hungerGames)-[:HAS_GENRE]→(fantasy), (nineteen84)-[:HAS_GENRE]→(scienceFiction), (nineteen84)-[:HAS_GENRE]→(dystopia), (nineteen84)-[:HAS_GENRE]→(classics), (dune)-[:HAS_GENRE]→(scienceFiction), (dune)-[:HAS_GENRE]→(fantasy), (dune)-[:HAS_GENRE]→(classics), (gatsby)-[:HAS_GENRE]→(classics) ---- === Stream

| from | to | count1 | count2 | intersection | similarity | Fantasy | Science Fiction | 3 | 4 | 3 | 1.0 | Dystopia | Science Fiction | 2 | 4 | 2 | 1.0 | Dystopia | Classics | 2 | 4 | 2 | 1.0 | Science Fiction | Classics | 4 | 4 | 3 | 0.75 | Fantasy | Classics | 3 | 4 | 2 | 0.66 | Dystopia | Fantasy | 2 | 3 | 1 | 0.5

Fantasy and Dystopia are both clear subgenres of Science Fiction - 100% of the books that list those as genres also list Science Fiction as a genre. Dystopia is also a subgenre of Classics. The others are less obvious; Dystopia probably isn't a subgenre of Fantasy, but the other two pairs could be subgenres.

.The following will return a stream of node pairs that have a similarity of at least 0.75, along with their intersection and overlap similarities: [source, cypher, role=noplay, indent=0] ---- MATCH (book:Book)-[:HAS_GENRE]→(genre) WITH {item:id(genre), categories: collect(id(book))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.overlap.stream({ data: data, similarityCutoff: 0.75 }) YIELD item1, item2, count1, count2, intersection, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, count1, count2, intersection, similarity ORDER BY similarity DESC ----

.Results [opts="header"]

| from | to | count1 | count2 | intersection | similarity | Fantasy | Science Fiction | 3 | 4 | 3 | 1.0 | Dystopia | Classics | 2 | 4 | 2 | 1.0 | Dystopia | Science Fiction | 2 | 4 | 2 | 1.0 | Science Fiction | Classics | 4 | 4 | 3 | 0.75

We can see that those genres with lower similarity have been filtered out. If we're implementing a k-Nearest Neighbors type query we might instead want to find the most similar k super genres for a given genre. We can do that by passing in the topK parameter.

.The following will return a stream of genres, along with the two most similar super genres to them (i.e. k=2): [source, cypher, role=noplay, indent=0] ---- MATCH (book:Book)-[:HAS_GENRE]→(genre) WITH {item:id(genre), categories: collect(id(book))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.overlap.stream({ data: data, topK: 2 }) YIELD item1, item2, count1, count2, intersection, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, count1, count2, intersection, similarity ORDER BY from ----

.Results [opts="header"]

| from | to | count1 | count2 | intersection | similarity | Dystopia | Classics | 2 | 4 | 2 | 1.0 | Dystopia | Science Fiction | 2 | 4 | 2 | 1.0 | Fantasy | Science Fiction | 3 | 4 | 3 | 1.0 | Fantasy | Classics | 3 | 4 | 2 | 0.6666666666666666 | Science Fiction | Classics | 4 | 4 | 3 | 0.75

=== Write

.The following will find the most similar genre for each genre, and store a relationship between those genres: [source, cypher, role=noplay, indent=0] ---- MATCH (book:Book)-[:HAS_GENRE]→(genre) WITH {item:id(genre), categories: collect(id(book))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.overlap.write({ data: data, topK: 2, similarityCutoff: 0.5 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

.Results [opts="header"]

| nodes | similarityPairs | writeRelationshipType | writeProperty | min | max | mean | p95 | 4 | 5 | NARROWER_THAN | score | 0.6666641235351562 | 1.0000038146972656 | 0.8833351135253906 | 1.0000038146972656

We then could write a query to find out the genre hierarchy for a specific genre.

.The following will find the genre hierarchy for the Fantasy genre [source, cypher, role=noplay, indent=0] ---- MATCH path = (fantasy:Genre {name: "Fantasy"})-[:NARROWER_THAN*]→(genre) RETURN [node in nodes(path) node.name] AS hierarchy ORDER BY length(path) ----

.Results [opts="header",cols="1"]

| `hierarchy` | ["Fantasy", "Science Fiction"] | ["Fantasy", "Classics"] | ["Fantasy", "Science Fiction", "Classics"]

=== Stats

.The following will run the algorithm and returns the result in form of statistical and measurement values [source, cypher, role=noplay, indent=0] ---- MATCH (book:Book)-[:HAS_GENRE]→(genre) WITH {item:id(genre), categories: collect(id(book))} AS userData WITH collect(userData) AS data CALL gds.alpha.similarity.overlap.stats({ data: data, topK: 2, similarityCutoff: 0.5 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

== Specifying source and target ids

Sometimes, we don't want to compute all pairs similarity, but would rather specify subsets of items to compare to each other. We do this using the `sourceIds` and `targetIds` keys in the config.

We could use this technique to compute the similarity of a subset of items to all other items.

.The following will return the super genres for the `Fantasy` and `Classics` genres: [source, cypher, role=noplay, indent=0] ---- MATCH (book:Book)-[:HAS_GENRE]→(genre) WITH {item:id(genre), name: genre.name, categories: collect(id(book))} AS userData WITH collect(userData) AS data WITH data, [value in data WHERE value.name IN ["Fantasy", "Classics"] value.item] AS sourceIds CALL gds.alpha.similarity.overlap.stream({ data: data, sourceIds: sourceIds }) YIELD item1, item2, count1, count2, intersection, similarity RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY similarity DESC ----

.Results [opts="header",cols="1,1,1"]

| `from` | `to` | `similarity` | Fantasy | Science Fiction | 1.0 | Classics | Science Fiction | 0.75 | Fantasy | Classics | 0.6666666666666666

== Syntax

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.similarity.overlap.write(configuration: Map) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100 ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | configuration | Map | n/a | no | Algorithm-specific

configuration.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | data | List or String | null | no | A list of maps of the following structure: `{item: nodeId, weights: [double, double, double]}` or a Cypher query. | top | Integer | 0 | yes | The number of similar pairs to return. If `0`, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. If `0`, it will return as many as it finds. | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. | degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the `targets` list. If the list contains less than this amount, that node will be excluded from the calculation. | skipValue | Float | gds.util.NaN() | yes | Value to skip when executing similarity computation. A value of `null` means that skipping is disabled. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result. | graph | String | dense | yes | The graph name ('dense' or 'cypher'). | writeBatchSize | Integer | 10000 | yes | The batch size to use when storing results. | writeRelationshipType | String | SIMILAR | yes | The relationship type to use when storing results. | writeProperty | String | score | yes | The property to use when storing results. | sourceIds | List of String | null | yes | The ids of items from which we need to compute similarities. Defaults to all the items provided in the `data` parameter. | targetIds | List of String | null | yes | The ids of items to which we need to compute similarities. Defaults to all the items provided in the `data` parameter.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodes | Integer | The number of nodes passed in. | similarityPairs | Integer | The number of pairs of similar nodes computed. | writeRelationshipType | String | The relationship type used when storing results. | writeProperty | String | The property used when storing results. | min | Float | The minimum similarity score computed. | max | Float | The maximum similarity score computed. | mean | Float | The mean of similarities scores computed. | stdDev | Float | The standard deviation of similarities scores computed. | p25 | Float | The 25 percentile of similarities scores computed. | p50 | Float | The 50 percentile of similarities scores computed. | p75 | Float | The 75 percentile of similarities scores computed. | p90 | Float | The 90 percentile of similarities scores computed. | p95 | Float | The 95 percentile of similarities scores computed. | p99 | Float | The 99 percentile of similarities scores computed. | p999 | Float | The 99.9 percentile of similarities scores computed. | p100 | Float | The 100 percentile of similarities scores computed.

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.similarity.overlap.stream(configuration: Map) YIELD item1, item2, count1, count2, similarity ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | configuration | Map | n/a | no | Algorithm-specific configuration.

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | data | List or String | null | no | A list of maps of the following structure: `{item: nodeId, weights: [double, double, double]}` or a Cypher query. | top |

Integer | 0 | yes | The number of similar pairs to return. If `0`, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. If `0`, it will return as many as it finds. | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. | degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the `targets` list. If the list contains less than this amount, that node will be excluded from the calculation. | skipValue | Float | gds.util.NaN() | yes | Value to skip when executing similarity computation. A value of `null` means that skipping is disabled. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. | graph | String | dense | yes | The graph name ('dense' or 'cypher'). | sourceIds | List of Integer | null | yes | The ids of items from which we need to compute similarities. Defaults to all the items provided in the `data` parameter. | targetIds | List of Integer | null | yes | The ids of items to which we need to compute similarities. Defaults to all the items provided in the `data` parameter.

```
.Results [opts="header",cols="1,1,6"]
```

| Name | Type | Description | item1 | Integer | The ID of one node in the similarity pair. | item2 | Integer | The ID of other node in the similarity pair. | count1 | Integer | The size of the `targets` list of one node. | count2 | Integer | The size of the `targets` list of other node. | intersection | Integer | The number of intersecting values in the two nodes `targets` lists. | similarity | Integer | The overlap similarity of the two nodes.

```
:leveloffset: 2

:leveloffset: +3

:description: This section describes the Approximate Nearest Neighbors algorithm in the Neo4j Graph Data Science library.
[.alpha] = Approximate Nearest Neighbors (ANN)

[abstract] — This section describes the Approximate Nearest Neighbors algorithm in the Neo4j Graph Data Science library. —  [.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

The Approximate Nearest Neighbors algorithm constructs a k-Nearest Neighbors Graph for a set of objects based on a provided similarity algorithm. The similarity of items is computed based on Jaccard Similarity, Cosine Similarity, Euclidean Distance, or Pearson Similarity.

The implementation in the library is based on Dong, Charikar, and Li's paper Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures.

== Syntax

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.ann.write(configuration: Map) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100 ----

.Configuration [opts="header",cols="1,1,1,1,4"]
```

| Name | Type | Default | Optional | Description | algorithm | String | null | no | The similarity algorithm to use. Valid values: jaccard', 'cosine', 'pearson', 'euclidean'. | data | List | null | no | If algorithm is `jaccard`, a list of maps of the following structure: `{item: nodeId, categories: [nodeId, nodeId, nodeId]}`. Otherwise a list of maps of the following structure: `{item: nodeId, weights: [double, double, double]}` or a Cypher query. | top | Integer | 0 | yes | The number of similar pairs to return. If `0`, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. | randomSeed | Integer | n/a | yes | The random-seed used for neighbor-sampling. | sampling | Boolean | true | yes |

Whether the potential neighbors should be sampled. | p | Float | 0.5 | yes | Influences the sample size: `min(1.0, p) * \|topK\|`. | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. | degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the `targets` list. If the list contains less than this amount, that node will be excluded from the calculation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result. | writeBatchSize | Integer | 10000 | yes | The batch size to use when storing results. | writeRelationshipType | String | SIMILAR | yes | The relationship type to use when storing results. | writeProperty | String | score | yes | The property to use when storing results.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodes | Integer | The number of nodes passed in. | similarityPairs | Integer | The number of pairs of similar nodes computed. | writeRelationshipType | String | The relationship type used when storing results. | writeProperty | String | The property used when storing results. | min | Float | The minimum similarity score computed. | max | Float | The maximum similarity score computed. | mean | Float | The mean of similarities scores computed. | stdDev | Float | The standard deviation of similarities scores computed. | p25 | Float | The 25 percentile of similarities scores computed. | p50 | Float | The 50 percentile of similarities scores computed. | p75 | Float | The 75 percentile of similarities scores computed. | p90 | Float | The 90 percentile of similarities scores computed. | p95 | Float | The 95 percentile of similarities scores computed. | p99 | Float | The 99 percentile of similarities scores computed. | p999 | Float | The 99.9 percentile of similarities scores computed. | p100 | Float | The 25 percentile of similarities scores computed.

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.ann.stream(configuration: Map) YIELD item1, item2, count1, count2, intersection, similarity

----

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | algorithm | String | null | no | The similarity algorithm to use. Valid values: jaccard', 'cosine', 'pearson', 'euclidean' | data | List | null | no | If algorithm is 'jaccard', a list of maps of the following structure: `{item: nodeId, categories: [nodeId, nodeId, nodeId]}`. Otherwise a list of maps of the following structure: `{item: nodeId, weights: [double, double, double]}` or a Cypher query. | top | Integer | 0 | yes | The number of similar pairs to return. If `0`, it will return as many as it finds. | topK | Integer | 3 | yes | The number of similar values to return per node. | randomSeed | Integer | 1 | yes | The random-seed used for neighbor-sampling. | sampling | Boolean | true | yes | Whether the potential neighbors should be sampled. | p | Float | 0.5 | yes | Influences the sample size: `min(1.0, p) * \|topK\|` | similarityCutoff | Integer | -1 | yes | The threshold for similarity. Values below this will not be returned. | degreeCutoff | Integer | 0 | yes | The threshold for the number of items in the `targets` list. If the list contains less than this amount, that node will be excluded from the calculation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | item1 | Integer | The ID of one node in the similarity pair. | item2 | Integer | The ID of other node in the similarity pair. | count1 | Integer | The size of the `targets` list of one node. | count2 |

Integer | The size of the `targets` list of other node. | intersection | Integer | The number of intersecting values in the two nodes `targets` lists. | similarity | Integer | The similarity of the two nodes.

== Use-cases - when to use the Approximate Nearest Neighbors algorithm

We can use the Approximate Nearest Neighbors algorithm to work out the approximate k most similar items to each other. The corresponding k-Nearest Neighbors Graph can then be used as part of recommendation queries.

== Approximate Nearest Neighbors algorithm sample

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (french:Cuisine {name:'French'}), (italian:Cuisine {name:'Italian'}), (indian:Cuisine {name:'Indian'}), (lebanese:Cuisine {name:'Lebanese'}), (portuguese:Cuisine {name:'Portuguese'}),

(zhen:Person {name: 'Zhen'}), (praveena:Person {name: 'Praveena'}), (michael:Person {name: 'Michael'}), (arya:Person {name: 'Arya'}), (karin:Person {name: 'Karin'}),

(praveena)-[:LIKES]→(indian), (praveena)-[:LIKES]→(portuguese),

(zhen)-[:LIKES]→(french), (zhen)-[:LIKES]→(indian),

(michael)-[:LIKES]→(french), (michael)-[:LIKES]→(italian), (michael)-[:LIKES]→(indian),

(arya)-[:LIKES]→(lebanese), (arya)-[:LIKES]→(italian), (arya)-[:LIKES]→(portuguese),

(karin)-[:LIKES]→(lebanese), (karin)-[:LIKES]→(italian) ----

.The following will return a stream of nodes, along with up to the 3 most similar nodes to them based on Jaccard Similarity: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person)-[:LIKES]→(cuisine) WITH {item:id(p), categories: collect(id(cuisine))} AS userData WITH collect(userData) AS data CALL gds.alpha.ml.ann.stream({ data: data, algorithm: 'jaccard', similarityCutoff: 0.1, randomSeed: 1, concurrency: 1 }) YIELD item1, item2, similarity return gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity ORDER BY from ----

.Results [opts="header",cols="1,1,1"]

| from | to | similarity | Arya | Karin | 0.6666666666666666 | Arya | Praveena | 0.25 | Arya | Michael | 0.2 | Karin | Arya | 0.6666666666666666 | Karin | Michael | 0.25 | Michael | Karin | 0.25 | Michael | Praveena | 0.25 | Michael | Arya | 0.2 | Praveena | Arya | 0.25 | Praveena | Michael | 0.25 | Zhen | Michael | 0.6666666666666666

Arya and Karin, and Zhen and Michael have the most similar food preferences, with two overlapping cuisines for a similarity of 0.66. We also have 3 pairs of users who are not similar at all. We'd probably want to filter those out, which we can do by passing in the `similarityCutoff` parameter.

.The following will find up to 3 similar users for each user, and store a relationship between those users: [source, cypher, role=noplay, indent=0] ---- MATCH (p:Person)-[:LIKES]→(cuisine) WITH {item:id(p), categories: collect(id(cuisine))} AS userData WITH collect(userData) AS data CALL gds.alpha.ml.ann.write({ algorithm: 'jaccard', data: data, similarityCutoff: 0.1, showComputations: true, randomSeed: 1, concurrency: 1 }) YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95 ----

.Results [opts="header"]

| nodes | similarityPairs | writeRelationshipType | writeProperty | min | max | mean | p95 | 5 | 13 | "SIMILAR" | "score" | 0.19999980926513672 | 0.6666669845581055 | 0.3512822664701022 | 0.6666669845581055

We then could write a query to find out what types of cuisine that other people similar to us might like.

.The following will find the most similar user to Praveena, and return their favorite cuisines that Praveena doesn't (yet!) like:
[source, cypher, role=noplay, indent=0] ---- MATCH (p:Person {name: 'Praveena'})-[:SIMILAR]→(other), (other)-[:LIKES]→(cuisine) WHERE notp)-[:LIKES]→(cuisine RETURN cuisine.name AS cuisine, count(*) AS count ORDER BY cuisine DESC ----

.Results [opts="header"]

| cuisine | count | "French" | 1 | "Italian" | 2 | "Lebanese" | 1

=== Usage

When executing ApproximateNearestNeighbors in parallel, it is possible that results are flaky because of the asynchronous execution fashion of the algorithm.

:leveloffset: 2

:leveloffset: 2

:description: This chapter provides explanations and examples for each of the path finding algorithms in the Neo4j Graph Data Science library. [[algorithms-path-finding]] = Path finding

[abstract] -- This chapter provides explanations and examples for each of the path finding algorithms in the Neo4j Graph Data Science library. --

Path finding algorithms find the shortest path between two or more nodes or evaluate the availability and quality of paths. The Neo4j GDS library includes the following path finding algorithms, grouped by quality tier:

* Production-quality ** <<algorithms-dijkstra-source-target, Dijkstra Source-Target>> ** <<algorithms-dijkstra-single-source, Dijkstra Single-Source>> ** <<algorithms-astar, A*>> ** <<algorithms-yens, Yen's algorithm>> * Alpha ** <<alpha-algorithms-minimum-weight-spanning-tree, Minimum Weight Spanning Tree>> ** <<alpha-algorithms-single-source-shortest-path, Single Source Shortest Path>> ** <<alpha-algorithm-all-pairs-shortest-path, All Pairs Shortest Path>> ** <<alpha-algorithms-random-walk, Random Walk>> ** <<algorithms-bfs, Breadth First Search>> ** <<algorithms-dfs, Depth First Search>>

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Dijkstra Shortest Path algorithm in the Neo4j Graph Data Science library. [[algorithms-dijkstra-source-target]] = Dijkstra Source-Target :entity: source-target-pair :result: shortest path :algorithm: Dijkstra :source-target: true :procedure-name: gds.shortestPath.dijkstra

[abstract] -- This section describes the Dijkstra Shortest Path algorithm in the Neo4j Graph Data Science library. --

:directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] -- [.not-supported] <<introduction-algorithms-directed, Directed>>

[.not-supported] <<introduction-algorithms-undirected, Undirected>>

[.not-supported] <<introduction-algorithms-homogeneous, Homogeneous>>

[.not-supported] <<introduction-algorithms-heterogeneous, Heterogeneous>>

[.not-supported] <<introduction-algorithms-weighted, Weighted>> --

[[algorithms-dijkstra-source-target-intro]] == Introduction

The Dijkstra Shortest Path algorithm computes the shortest path between nodes. The algorithm supports weighted graphs with positive relationship weights. The Dijkstra Source-Target algorithm computes the shortest path between a source and a target node. To compute all paths from a source node to all reachable nodes, <<algorithms-dijkstra-single-source, Dijkstra Single-Source>> can be used.

The GDS implementation is based on the http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf[original description] and uses a binary heap as priority queue. The implementation is also used for the <<algorithms-astar, A*>> and <<algorithms-yens, Yen's>> algorithms. The algorithm implementation is executed using a single thread. Altering the concurrency configuration has no effect.

[[algorithms-dijkstra-source-target-syntax]] == Syntax

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header"]

| Name | Type | Description | index | Integer | 0-based index of the found path. | sourceNode | Integer | Source node of the path. | targetNode | Integer | Target node of the path. | totalCost | Float | Total cost from source to target. | nodeIds | List of Integer | Node ids on the path in traversal order. | costs | List of Float | Accumulated costs for each node on the path. | path | Path | The path represented as Cypher entity.

======

[.include-with-mutate] ======

The mutate mode creates new relationships in the in-memory graph. Each relationship represents a path from the source node to the target node. The total cost of a path is stored via the `totalCost` relationship property.

.Run {algorithm} in mutate mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.mutate(graphName: String, configuration: Map) YIELD relationshipsWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Unused. | mutateMillis | Integer | Milliseconds for adding relationships to the in-memory graph. | relationshipsWritten | Integer | The number of relationships that were added. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ======

The write mode creates new relationships in the Neo4j database. Each relationship represents a path from the source node to the target node. Additional path information is stored using relationship properties. By default, the write mode stores a `totalCost` property. Optionally, one can also store `nodeIds` and `costs` of intermediate nodes on the path.

.Run {algorithm} in write mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.write(graphName: String, configuration: Map) YIELD relationshipsWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id. | writeNodeIds | Boolean | false | yes | If true, the written relationship has a nodeIds list property. | writeCosts | Boolean | false | yes | If true, the written relationship has a costs list property.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Unused. | writeMillis | Integer | Milliseconds for writing relationships to Neo4j. | relationshipsWritten | Integer | The number of relationships that were written. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run {algorithm} in write mode on an anonymous graph: [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.write(configuration: Map) YIELD relationshipsWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j node id of the source node. | targetNode | Integer | n/a | no | The Neo4j node id of the target node. | writeNodeIds | Boolean | false | yes | Iff true, the written relationship has a nodeIds list property. | writeCosts | Boolean | false | yes | Iff true, the written relationship has a costs list property.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: transport network :image-file: dijkstra.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (a:Location {name: 'A'}), (b:Location {name: 'B'}), (c:Location {name: 'C'}), (d:Location {name: 'D'}), (e:Location {name: 'E'}), (f:Location {name: 'F'}), (a)-[:ROAD {cost: 50}]→(b), (a)-[:ROAD {cost: 50}]→(c), (a)-[:ROAD {cost: 100}]→(d), (b)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 80}]→(e), (d)-[:ROAD {cost: 30}]→(e), (d)-[:ROAD {cost: 80}]→(f), (e)-[:ROAD {cost: 40}]→(f); ----

This graph builds a transportation network with roads between locations. Like in the real world, the roads in the graph have different lengths. These lengths are represented by the `cost` relationship property.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Location', 'ROAD', { relationshipProperties: 'cost' }) ----

In the following example we will demonstrate the use of the {algorithm} Shortest Path algorithm using this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'}) CALL gds.shortestPath.dijkstra.write.estimate('myGraph', { sourceNode: source, targetNode: target, relationshipWeightProperty: 'cost', writeRelationshipType: 'PATH' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 6 | 9 | 696 | 696 | "696 Bytes"

—

=== Stream

:!stream-details: In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'}) CALL gds.shortestPath.dijkstra.stream('myGraph', { sourceNode: source, targetNode: target, relationshipWeightProperty: 'cost' }) YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path RETURN index, gds.util.asNode(sourceNode).name AS sourceNodeName, gds.util.asNode(targetNode).name AS targetNodeName, totalCost, [nodeId IN nodeIds gds.util.asNode(nodeId).name] AS nodeNames, costs, nodes(path) as path ORDER BY index ----

.Results [opts="header"]

| index | sourceNodeName | targetNodeName | totalCost | nodeNames | costs | path | 0 | "A" | "F" | 160.0 | [A, B, D, E, F] | [0.0, 50.0, 90.0, 120.0, 160.0] | [Node[0], Node[1], Node[3], Node[4], Node[5]]

— The result shows the total cost of the shortest path between node `A` and node `F`. It also shows an ordered list of node ids that were traversed to find the shortest path as well as the accumulated costs of the visited nodes. This can be verified in the example graph. Cypher Path objects can be returned by the `path` return field. The Path objects contain the node objects and virtual relationships which have a `cost` property.

=== Mutate

The `mutate` execution mode updates the named graph with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `mutateRelationshipType` option. The total path cost is stored using the `totalCost` property.

The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'}) CALL gds.shortestPath.dijkstra.mutate('myGraph', { sourceNode: source, targetNode: target, relationshipWeightProperty: 'cost', mutateRelationshipType: 'PATH' }) YIELD relationshipsWritten RETURN relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 1

— After executing the above query, the in-memory graph will be updated with a new relationship of type `PATH`. The new relationship will store a single property `totalCost`.

=== Write

The `write` execution mode updates the Neo4j database with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `writeRelationshipType` option. The total path cost is stored using the `totalCost` property. The intermediate node ids are stored using the `nodeIds` property. The accumulated costs to reach an intermediate node are stored using the `costs` property.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'}) CALL gds.shortestPath.dijkstra.write('myGraph', { sourceNode: source, targetNode: target, relationshipWeightProperty: 'cost', writeRelationshipType: 'PATH', writeNodeIds: true, writeCosts: true }) YIELD relationshipsWritten RETURN relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 1

— The above query will write a single relationship of type `PATH` back to Neo4j. The relationship stores three properties describing the path: `totalCost`, `nodeIds` and `costs`.

:leveloffset: 2

:leveloffset: 3

:description: This section describes the Dijkstra Shortest Path algorithm in the Neo4j Graph Data Science library. [[algorithms-dijkstra-single-source]] = Dijkstra Single-Source :entity: source-target-pair :result: shortest path :algorithm: Dijkstra :source-target: false :procedure-name: gds.allShortestPaths.dijkstra

[abstract] -- This section describes the Dijkstra Shortest Path algorithm in the Neo4j Graph Data Science library. --

:directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] -- [.not-supported] <<introduction-algorithms-directed, Directed>>

[.not-supported] <<introduction-algorithms-undirected, Undirected>>

[.not-supported] <<introduction-algorithms-homogeneous, Homogeneous>>

[.not-supported] <<introduction-algorithms-heterogeneous, Heterogeneous>>

[.not-supported] <<introduction-algorithms-weighted, Weighted>> --

[[algorithms-dijkstra-single-source-intro]] == Introduction

The Dijkstra Shortest Path algorithm computes the shortest path between nodes. The algorithm supports weighted graphs with positive relationship weights. The Dijkstra Single-Source algorithm computes the shortest paths between a source node and all nodes reachable from that node. To compute the shortest path between a source and a target node, <<algorithms-dijkstra-source-target, Dijkstra Source-Target>> can be used.

The GDS implementation is based on the http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf[original description] and uses a binary heap as priority queue. The implementation is also used for the <<algorithms-astar, A*>> and <<algorithms-yens, Yen's>> algorithms. The algorithm implementation is executed using a single thread. Altering the concurrency configuration has no effect.

[[algorithms-dijkstra-single-source-syntax]] == Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see <<algorithms-syntax, Syntax overview>>.

.{algorithm} syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ======

.Run {algorithm} in stream mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.stream(graphName: String, configuration: Map) YIELD index: Integer, sourceNode: Integer, targetNode: Integer, totalCost: Float, nodeIds: List of Integer, costs: List of Float, path: Path ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header"]

| Name | Type | Description | index | Integer | 0-based index of the found path. | sourceNode | Integer | Source node of the path. | targetNode | Integer | Target node of the path. | totalCost | Float | Total cost from source to target. | nodeIds | List of Integer | Node ids on the path in traversal order. | costs | List of Float | Accumulated costs for each node on the path. | path | Path | The path represented as Cypher entity.

======

[.include-with-mutate] ======

The mutate mode creates new relationships in the in-memory graph. Each relationship represents a path from the source node to the target node. The total cost of a path is stored via the `totalCost` relationship property.

.Run {algorithm} in mutate mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.mutate(graphName: String, configuration: Map) YIELD relationshipsWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis |

Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Unused. | mutateMillis | Integer | Milliseconds for adding relationships to the in-memory graph. | relationshipsWritten | Integer | The number of relationships that were added. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ======

The write mode creates new relationships in the Neo4j database. Each relationship represents a path from the source node to the target node. Additional path information is stored using relationship properties. By default, the write mode stores a `totalCost` property. Optionally, one can also store `nodeIds` and `costs` of intermediate nodes on the path.

.Run {algorithm} in write mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.write(graphName: String, configuration: Map) YIELD relationshipsWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id. | writeNodeIds | Boolean | false | yes | If true, the written relationship has a nodeIds list property. | writeCosts | Boolean | false | yes | If true, the written relationship has a costs list property.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Unused. | writeMillis | Integer | Milliseconds for writing relationships to Neo4j. | relationshipsWritten | Integer | The number of relationships that were written. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run {algorithm} in write mode on an anonymous graph: [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.write(configuration: Map) YIELD relationshipsWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j node id of the source node. | targetNode | Integer | n/a | no | The Neo4j node id of the target node. | writeNodeIds | Boolean | false | yes | Iff true, the written relationship has a nodeIds list property. | writeCosts | Boolean | false | yes | Iff true, the written relationship has a costs list property.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: transport network :image-file: dijkstra.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (a:Location {name: 'A'}), (b:Location {name: 'B'}), (c:Location {name: 'C'}), (d:Location {name: 'D'}), (e:Location {name: 'E'}), (f:Location {name: 'F'}), (a)-[:ROAD {cost: 50}]→(b), (a)-[:ROAD {cost: 50}]→(c), (a)-[:ROAD {cost: 100}]→(d), (b)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 80}]→(e), (d)-[:ROAD {cost: 30}]→(e), (d)-[:ROAD {cost: 80}]→(f), (e)-[:ROAD {cost: 40}]→(f); ----

This graph builds a transportation network with roads between locations. Like in the real world, the roads in the graph have different lengths. These lengths are represented by the `cost` relationship property.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Location', 'ROAD', { relationshipProperties: 'cost' }) ----

In the following example we will demonstrate the use of the {algorithm} Shortest Path algorithm using this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}) CALL gds.allShortestPaths.dijkstra.write.estimate('myGraph', { sourceNode: source, relationshipWeightProperty: 'cost', writeRelationshipType: 'PATH' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 6 | 9 | 696 | 696 | "696 Bytes"

—

=== Stream

:!stream-details: In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}) CALL gds.allShortestPaths.dijkstra.stream('myGraph', { sourceNode: source, relationshipWeightProperty: 'cost' }) YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path RETURN index, gds.util.asNode(sourceNode).name AS sourceNodeName, gds.util.asNode(targetNode).name AS targetNodeName, totalCost, [nodeId IN nodeIds gds.util.asNode(nodeId).name] AS nodeNames, costs, nodes(path) as path ORDER BY index ----

.Results [opts="header"]

| index | sourceNodeName | targetNodeName | totalCost | nodeNames | costs | path | 0 | "A" | "A" | 0.0 | [A] | [0.0] | [Node[0]] | 1 | "A" | "B" | 50.0 | [A, B] | [0.0, 50.0] | [Node[0], Node[1]] | 2 | "A" | "C" | 50.0 | [A, C] | [0.0, 50.0] | [Node[0], Node[2]] | 3 | "A" | "D" | 90.0 | [A, B, D] | [0.0, 50.0, 90.0] | [Node[0], Node[1], Node[3]] | 4 | "A" | "E" | 120.0 | [A, B, D, E] | [0.0, 50.0, 90.0, 120.0] | [Node[0], Node[1], Node[3], Node[4]] | 5 | "A" | "F" | 160.0 | [A, B, D, E, F] | [0.0, 50.0, 90.0, 120.0, 160.0] | [Node[0], Node[1], Node[3], Node[4], Node[5]]

— The result shows the total cost of the shortest path between node `A` and all other reachable nodes in the graph. It also shows ordered lists of node ids that were traversed to find the shortest paths as well as the accumulated costs of the visited nodes. This can be verified in the example graph. Cypher Path objects can be returned by the `path` return field. The Path objects contain the node objects and virtual relationships which have a `cost` property.

=== Mutate

The `mutate` execution mode updates the named graph with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `mutateRelationshipType` option. The total path cost is stored using the `totalCost` property.

The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}) CALL gds.allShortestPaths.dijkstra.mutate('myGraph', { sourceNode: source, relationshipWeightProperty: 'cost', mutateRelationshipType: 'PATH' }) YIELD relationshipsWritten RETURN relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 6

— After executing the above query, the in-memory graph will be updated with new relationships of type `PATH`. The new relationships will store a single property `totalCost`.

=== Write

The `write` execution mode updates the Neo4j database with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `writeRelationshipType` option. The total path cost is stored using the `totalCost` property. The intermediate node ids are stored using the `nodeIds` property. The accumulated costs to reach an intermediate node are stored using the `costs` property.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}) CALL gds.allShortestPaths.dijkstra.write('myGraph', { sourceNode: source, relationshipWeightProperty: 'cost', writeRelationshipType: 'PATH', writeNodeIds: true, writeCosts: true }) YIELD relationshipsWritten RETURN relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 6

— The above query will write 6 relationships of type `PATH` back to Neo4j. The relationships store three properties describing the path: `totalCost`, `nodeIds` and `costs`.

:leveloffset: 2

:leveloffset: 3

:description: This section describes the A* Shortest Path algorithm in the Neo4j Graph Data Science library. [[algorithms-astar]] = A* :entity: source-target-pair :result: shortest path :algorithm: A* :source-target: true :procedure-name: gds.shortestPath.astar

[abstract] -- This section describes the A* Shortest Path algorithm in the Neo4j Graph Data Science library. --

:directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] -- [.not-supported] <<introduction-algorithms-directed, Directed>>

[.not-supported] <<introduction-algorithms-undirected, Undirected>>

[.not-supported] <<introduction-algorithms-homogeneous, Homogeneous>>

[.not-supported] <<introduction-algorithms-heterogeneous, Heterogeneous>>

[.not-supported] <<introduction-algorithms-weighted, Weighted>> --

[[algorithms-astar-intro]] == Introduction

The A* (pronounced "A-Star") Shortest Path algorithm computes the shortest path between two nodes. A* is an informed search algorithm as it uses a heuristic function to guide the graph traversal. The algorithm supports weighted graphs with positive relationship weights.

Unlike <<algorithms-dijkstra-source-target, Dijkstra's shortest path algorithm>>, the next node to search from is not solely picked on the already computed distance. Instead, the algorithm combines the already computed distance with the result of a heuristic function. That function takes a node as input and returns a value that corresponds to the cost to reach the target node from that node. In each iteration, the graph traversal is continued from the node with the lowest combined cost.

In GDS, the A* algorithm is based on the <<algorithms-dijkstra-source-target, Dijkstra's shortest path algorithm>>. The heuristic function is the haversine distance, which defines the distance between two points on a sphere. Here, the sphere is the earth and the points are geo-coordinates stored on the nodes in the graph.

The algorithm implementation is executed using a single thread. Altering the concurrency configuration has no effect.

[[algorithms-astar-requirements]] == Requirements

In GDS, the heuristic function used to guide the search is the https://en.wikipedia.org/wiki/Haversine_formula[haversine formula]. The formula computes the distance between two points on a sphere given their longitudes and latitudes. The distance is computed in nautical miles.

In order to guarantee finding the optimal solution, i.e., the shortest path between two points, the heuristic must be admissible. To be admissible, the function must not overestimate the distance to the target, i.e., the lowest possible cost of a path must always be greater or equal to the heuristic.

This leads to a requirement on the relationship weights of the input graph. Relationship weights must represent the distance between two nodes and ideally scaled to nautical miles. Kilometers or miles also work, but the heuristic works best for nautical miles.

[[algorithms-astar-syntax]] == Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see <<algorithms-syntax, Syntax

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header"]

| Name | Type | Description | index | Integer | 0-based index of the found path. | sourceNode | Integer | Source node of the path. | targetNode | Integer | Target node of the path. | totalCost | Float | Total cost from source to target. | nodeIds | List of Integer | Node ids on the path in traversal order. | costs | List of Float | Accumulated costs for each node on the path. | path | Path | The path represented as Cypher entity.

======

[.include-with-mutate] ======

The mutate mode creates new relationships in the in-memory graph. Each relationship represents a path from the source node to the target node. The total cost of a path is stored via the `totalCost` relationship property.

.Run {algorithm} in mutate mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.mutate(graphName: String, configuration: Map) YIELD relationshipsWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Unused. | mutateMillis | Integer | Milliseconds for adding relationships to the in-memory graph. | relationshipsWritten | Integer | The number of relationships that were added. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ======

The write mode creates new relationships in the Neo4j database. Each relationship represents a path from the source node to the target node. Additional path information is stored using relationship properties. By default, the write mode stores a `totalCost` property. Optionally, one can also store `nodeIds` and `costs` of intermediate nodes on the path.

.Run {algorithm} in write mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.write(graphName: String, configuration: Map) YIELD relationshipsWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id. | writeNodeIds | Boolean | false | yes | If true, the written relationship has a nodeIds list property. | writeCosts | Boolean | false | yes | If true, the written relationship has a costs list property.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Unused. | writeMillis | Integer | Milliseconds for writing relationships to Neo4j. | relationshipsWritten | Integer | The number of relationships that were written. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run {algorithm} in write mode on an anonymous graph: [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.write(configuration: Map) YIELD relationshipsWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j node id of the source node. | targetNode | Integer | n/a | no | The Neo4j node id of the target node. | writeNodeIds | Boolean | false | yes | Iff true, the written relationship has a nodeIds list property. | writeCosts | Boolean | false | yes | Iff true, the written relationship has a costs list property.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: transport network :image-file: astar.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (a:Station {name: 'Kings Cross', latitude: 51.5308, longitude: -0.1238}), (b:Station {name: 'Euston', latitude: 51.5282, longitude: -0.1337}), (c:Station {name: 'Camden Town', latitude: 51.5392, longitude: -0.1426}), (d:Station {name: 'Mornington Crescent', latitude: 51.5342, longitude: -0.1387}), (e:Station {name: 'Kentish Town', latitude: 51.5507, longitude: -0.1402}), (a)-[:CONNECTION {distance: 0.7}]→(b), (b)-[:CONNECTION {distance: 1.3}]→(c), (b)-[:CONNECTION {distance: 0.7}]→(d), (d)-[:CONNECTION {distance: 0.6}]→(c), (c)-[:CONNECTION {distance: 1.3}]→(e) ----

The graph represents a transport network of stations. Each station has a geo-coordinate, expressed by `latitude` and `longitude` properties. Stations are connected via connections. We use the `distance` property as relationship weight which represents the distance between stations in kilometers. The algorithm will pick the next node in the search based on the already traveled distance and the distance to the target station.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Station', 'CONNECTION', { nodeProperties: ['latitude', 'longitude'], relationshipProperties: 'distance' }) ----

In the following example we will demonstrate the use of the {algorithm} Shortest Path algorithm using this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'}) CALL gds.shortestPath.astar.write.estimate('myGraph', { sourceNode: source, targetNode: target, latitudeProperty: 'latitude', longitudeProperty: 'longitude', writeRelationshipType: 'PATH' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 5 | 5 | 984 | 984 | "984 Bytes"

— === Stream :!stream-details: In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects. For more details on the `stream` mode in general, see Stream. [role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'}) CALL gds.shortestPath.astar.stream('myGraph', { sourceNode: source, targetNode: target, latitudeProperty: 'latitude', longitudeProperty: 'longitude', relationshipWeightProperty: 'distance' }) YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path RETURN index, gds.util.asNode(sourceNode).name AS sourceNodeName, gds.util.asNode(targetNode).name AS targetNodeName, totalCost, [nodeId IN nodeIds	gds.util.asNode(nodeId).name] AS nodeNames, costs, nodes(path) as path ORDER BY index ---- .Results [opts="header"]	

| index | sourceNodeName | targetNodeName | totalCost | nodeNames | costs | path | 0 | "Kings Cross" | "Kentish Town" | 3.3 | [Kings Cross, Euston, Camden Town, Kentish Town] | [0.0, 0.7, 2.0, 3.3] | [Node[0], Node[1], Node[2], Node[4]]

— The result shows the total cost of the shortest path between node `King's Cross` and `Kentish Town` in the graph. It also shows ordered lists of node ids that were traversed to find the shortest paths as well as the accumulated costs of the visited nodes. This can be verified in the example graph. Cypher Path objects can be returned by the `path` return field. The Path objects contain the node objects and virtual relationships which have a `cost` property. === Mutate The `mutate` execution mode updates the named graph with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `mutateRelationshipType` option. The total path cost is stored using the `totalCost` property. The `mutate` mode is especially useful when multiple algorithms are used in conjunction. For more details on the `mutate` mode in general, see Mutate. [role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'}) CALL gds.shortestPath.astar.mutate('myGraph', { sourceNode: source, targetNode: target, latitudeProperty: 'latitude', longitudeProperty: 'longitude', relationshipWeightProperty: 'distance', mutateRelationshipType: 'PATH' }) YIELD relationshipsWritten RETURN relationshipsWritten ---- .Results [opts="header"]

| relationshipsWritten | 1

— After executing the above query, the in-memory graph will be updated with new relationships of type `PATH`. The new relationships will store a single property `totalCost`.

=== Write

The `write` execution mode updates the Neo4j database with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `writeRelationshipType` option. The total path cost is stored using the `totalCost` property. The intermediate node ids are stored using the `nodeIds` property. The accumulated costs to reach an intermediate node are stored using the `costs` property.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'}) CALL gds.shortestPath.astar.write('myGraph', { sourceNode: source, targetNode: target, latitudeProperty: 'latitude', longitudeProperty: 'longitude', relationshipWeightProperty: 'distance', writeRelationshipType: 'PATH', writeNodeIds: true, writeCosts: true }) YIELD relationshipsWritten RETURN relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 1

— The above query will write one relationship of type `PATH` back to Neo4j. The relationship stores three properties describing the path: `totalCost`, `nodeIds` and `costs`.

:leveloffset: 2

:leveloffset: 3

:description: This section describes the Yen's Shortest Path algorithm in the Neo4j Graph Data Science library. [[algorithms-yens]] = Yen's algorithm :entity: source-target-pair :result: shortest path :algorithm: Yen's :source-target: true :procedure-name: gds.shortestPath.yens

[abstract] -- This section describes the Yen's Shortest Path algorithm in the Neo4j Graph Data Science library. --

:directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] -- [.not-supported] <<introduction-algorithms-directed, Directed>>

[.not-supported] <<introduction-algorithms-undirected, Undirected>>

[.not-supported] <<introduction-algorithms-homogeneous, Homogeneous>>

[.not-supported] <<introduction-algorithms-heterogeneous, Heterogeneous>>

[.not-supported] <<introduction-algorithms-weighted, Weighted>> --

[[algorithms-yens-intro]] == Introduction

Yen's Shortest Path algorithm computes a number of shortest paths between two nodes. The algorithm is often referred to as Yen's k-Shortest Path algorithm, where k is the number of shortest paths to compute. The algorithm supports weighted graphs with positive relationship weights. It also respects parallel relationships between the same two nodes when computing multiple shortest paths.

For `k = 1`, the algorithm behaves exactly like <<algorithms-dijkstra-source-target, Dijkstra's shortest path algorithm>> and returns the shortest path. For `k = 2`, the algorithm returns the shortest path and the second shortest path between the same source and target node. Generally, for `k = n`, the algorithm computes at most `n` paths which are discovered in the order of their total cost.

The GDS implementation is based on the https://www.ams.org/journals/qam/1970-27-04/S0033-569X-1970-0253822-7/[original description]. For the actual path computation, Yen's algorithm uses <<algorithms-dijkstra-source-target, Dijkstra's shortest path algorithm>>. The algorithm makes sure that an already discovered shortest path will not be traversed again.

The algorithm implementation is executed using a single thread. Altering the concurrency configuration has no effect.

[[algorithms-yens-syntax]] == Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see <<algorithms-syntax, Syntax overview>>.

.{algorithm} syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ======

.Run {algorithm} in stream mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.stream(graphName: String, configuration: Map) YIELD index: Integer, sourceNode: Integer, targetNode: Integer, totalCost: Float, nodeIds: List of Integer, costs: List of Float, path: Path ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored

in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header"]

| Name | Type | Description | index | Integer | 0-based index of the found path. | sourceNode | Integer | Source node of the path. | targetNode | Integer | Target node of the path. | totalCost | Float | Total cost from source to target. | nodeIds | List of Integer | Node ids on the path in traversal order. | costs | List of Float | Accumulated costs for each node on the path. | path | Path | The path represented as Cypher entity.

======

[.include-with-mutate] ======

The mutate mode creates new relationships in the in-memory graph. Each relationship represents a path from the source node to the target node. The total cost of a path is stored via the `totalCost` relationship property.

.Run {algorithm} in mutate mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.mutate(graphName: String, configuration: Map) YIELD relationshipsWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or

node id.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Unused. | mutateMillis | Integer | Milliseconds for adding relationships to the in-memory graph. | relationshipsWritten | Integer | The number of relationships that were added. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ======

The write mode creates new relationships in the Neo4j database. Each relationship represents a path from the source node to the target node. Additional path information is stored using relationship properties. By default, the write mode stores a `totalCost` property. Optionally, one can also store `nodeIds` and `costs` of intermediate nodes on the path.

.Run {algorithm} in write mode on a named graph. [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.write(graphName: String, configuration: Map) YIELD relationshipsWritten: Integer, createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j source node or node id. | writeNodeIds | Boolean | false | yes | If true, the written relationship has a nodeIds list property. | writeCosts | Boolean | false | yes | If true, the written relationship has a costs list property.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Unused. | writeMillis | Integer | Milliseconds for writing relationships to Neo4j. | relationshipsWritten | Integer | The number of relationships that were written. | configuration | Map | The configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run {algorithm} in write mode on an anonymous graph: [source, cypher, role=noplay, subs="quotes,attributes+", indent=0, indent=0, indent=0, indent=0] ---- CALL {procedure-name}.write(configuration: Map) YIELD relationshipsWritten: Integer, ranIterations: Integer, didConverge: Boolean, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | sourceNode | Integer | n/a | no | The Neo4j node id of the source node. | targetNode | Integer | n/a | no | The Neo4j node id of the target node. | writeNodeIds | Boolean | false | yes | Iff true, the written relationship has a nodeIds list property. | writeCosts | Boolean | false | yes | Iff true, the written relationship has a costs list property.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: transport network :image-file: dijkstra.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (a:Location {name: 'A'}), (b:Location {name: 'B'}), (c:Location {name: 'C'}), (d:Location {name: 'D'}), (e:Location {name: 'E'}), (f:Location {name: 'F'}), (a)-[:ROAD {cost: 50}]→(b), (a)-[:ROAD {cost: 50}]→(c), (a)-[:ROAD {cost: 100}]→(d), (b)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 80}]→(e), (d)-[:ROAD {cost: 30}]→(e), (d)-[:ROAD {cost: 80}]→(f), (e)-[:ROAD {cost: 40}]→(f); ----

This graph builds a transportation network with roads between locations. Like in the real world, the roads in the graph have different lengths. These lengths are represented by the `cost` relationship property.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Location', 'ROAD', { relationshipProperties: 'cost' }) ----

In the following example we will demonstrate the use of the {algorithm} Shortest Path algorithm using this graph.

=== Memory Estimation

:mode: write First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'}) CALL gds.shortestPath.yens.write.estimate('myGraph', { sourceNode: source, targetNode: target, k: 3, relationshipWeightProperty: 'cost', writeRelationshipType: 'PATH' }) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 6 | 9 | 1008 | 1008 | "1008 Bytes"

—

=== Stream

:!stream-details: In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'}) CALL gds.shortestPath.yens.stream('myGraph', { sourceNode: source, targetNode: target, k: 3, relationshipWeightProperty: 'cost' }) YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path RETURN index, gds.util.asNode(sourceNode).name AS sourceNodeName, gds.util.asNode(targetNode).name AS targetNodeName, totalCost, [nodeId IN nodeIds gds.util.asNode(nodeId).name] AS nodeNames, costs, nodes(path) as path ORDER BY index ----

.Results [opts="header"]

| index | sourceNodeName | targetNodeName | totalCost | nodeNames | costs | path | 0 | "A" | "F" | 160.0 | [A, B, D, E, F] | [0.0, 50.0, 90.0, 120.0, 160.0] | [Node[0], Node[1], Node[3], Node[4], Node[5]] | 1 | "A" | "F" | 160.0 | [A, C, D, E, F] | [0.0, 50.0, 90.0, 120.0, 160.0] | [Node[0], Node[2], Node[3], Node[4], Node[5]] | 2 | "A" | "F" | 170.0 | [A, B, D, F] | [0.0, 50.0, 90.0, 170.0] | [Node[0], Node[1], Node[3], Node[5]]

— The result shows the three shortest paths between node `A` and node `F`. The first two paths have the same total cost, however the first one traversed from `A` to `D` via the `B` node, while the second traversed via the `C` node. The third path has a higher total cost as it goes directly from `D` to `F` using the relationship with a cost of `80`, whereas the detour via `E` for the first two paths costs `70`. This can be verified in the example graph. Cypher Path objects can be returned by the `path` return field. The Path objects contain the node objects and virtual relationships which have a `cost` property.

=== Mutate

The `mutate` execution mode updates the named graph with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `mutateRelationshipType` option. The total path cost is stored using the `totalCost` property.

The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'}) CALL gds.shortestPath.yens.mutate('myGraph', { sourceNode: source, targetNode: target, k: 3, relationshipWeightProperty: 'cost', mutateRelationshipType: 'PATH' }) YIELD relationshipsWritten RETURN relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 3

— After executing the above query, the in-memory graph will be updated with a new relationship of type `PATH`. The new relationship will store a single property `totalCost`.

=== Write

The `write` execution mode updates the Neo4j database with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `writeRelationshipType` option. The total path cost is stored using the `totalCost` property. The intermediate node ids are stored using the `nodeIds` property. The accumulated costs to reach an intermediate node are stored using the `costs` property.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'}) CALL gds.shortestPath.yens.write('myGraph', { sourceNode: source, targetNode: target, k: 3, relationshipWeightProperty: 'cost', writeRelationshipType: 'PATH', writeNodeIds: true, writeCosts: true }) YIELD relationshipsWritten RETURN relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 3

— The above query will write a single relationship of type `PATH` back to Neo4j. The relationship stores three properties describing the path: `totalCost`, `nodeIds` and `costs`.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Minimum Weight Spanning Tree algorithm in the Neo4j Graph Data Science library.
[.alpha] = Minimum Weight Spanning Tree

[abstract] — This section describes the Minimum Weight Spanning Tree algorithm in the Neo4j Graph Data Science library. — The Minimum Weight Spanning Tree (MST) starts from a given node, and finds all its reachable nodes and the set of relationships that connect the nodes together with the minimum possible weight. Prim's algorithm is one of the simplest and best-known minimum spanning tree algorithms. The K-Means variant of this algorithm can be used to detect clusters in the graph.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

The first known algorithm for finding a minimum spanning tree was developed by the Czech scientist Otakar Borůvka in 1926, while trying to find an efficient electricity network for Moravia. Prim's algorithm was invented by Jarnik in 1930 and rediscovered by Prim in 1957. It is similar to Dijkstra's shortest path algorithm but, rather than minimizing the total length of a path ending at each relationship, it minimizes the length of each relationship individually. Unlike Dijkstra's, Prim's can tolerate negative-weight relationships.

The algorithm operates as follows:

* Start with a tree containing only one node (and no relationships). * Select the minimal-weight relationship coming from that node, and add it to our tree. * Repeatedly choose a minimal-weight relationship that joins any node in the tree to one that is not in the tree, adding the new relationship and node to our tree. * When there are no more nodes to add, the tree we have built is a minimum spanning tree.

== Use-cases - when to use the Minimum Weight Spanning Tree algorithm

* Minimum spanning tree was applied to analyze airline and sea connections of Papua New Guinea, and minimize the travel cost of exploring the country. It could be used to help design low-cost tours that visit many destinations across the country. The research mentioned can be found in "An Application of Minimum Spanning Trees to Travel Planning". * Minimum spanning tree has been used to analyze and visualize correlations in a network of currencies, based on the correlation between currency returns. This is described in "Minimum Spanning Tree Application in the Currency Market". * Minimum spanning tree has been shown to be a useful tool to trace the history of transmission of infection, in an outbreak supported by exhaustive clinical research. For more information, see Use of the Minimum Spanning Tree Model for Molecular Epidemiological Investigation of a Nosocomial Outbreak of Hepatitis C Virus Infection.

== Constraints - when not to use the Minimum Weight Spanning Tree algorithm

The MST algorithm only gives meaningful results when run on a graph, where the relationships have different weights. If the graph has no weights, or all relationships have the same weight, then any spanning tree is a minimum spanning tree.

== Syntax

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.spanningTree.write(configuration: Map) YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount ----

.The following will compute the minimum weight spanning tree and write the results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.spanningTree.minimum.write(configuration: Map) YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount ----

.The following will compute the maximum weight spanning tree and write the results: [source, cypher, role=noplay,

| Name | Type | Default | Optional | Description | startNodeId | Integer | null | no | The start node ID | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | writeProperty | String | 'mst' | yes | The relationship type written back as result | weightWriteProperty | String | n/a | no | The weight property of the `writeProperty` relationship type written back

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | effectiveNodeCount | Integer | The number of visited nodes | createMillis | Integer | Milliseconds for loading data | computeMillis | Integer | Milliseconds for running the algorithm | writeMillis | Integer | Milliseconds for writing result data back

.The following will run the k-spanning tree algorithms and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.spanningTree.kmin.write(configuration: Map) YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount ----

[source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.spanningTree.kmax.write(configuration: Map) YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount ----

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | k | Integer | null | no | The result is a tree with `k` nodes and `k - 1` relationships | startNodeId | Integer | null | no | The start node ID | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | writeProperty | String | 'MST' | yes | The relationship type written back as result | weightWriteProperty | String | n/a | no | The weight property of the `writeProperty` relationship type written back

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | effectiveNodeCount | Integer | The number of visited nodes | createMillis | Integer | Milliseconds for loading data | computeMillis | Integer | Milliseconds for running the algorithm | writeMillis | Integer | Milliseconds for writing result data back

== Minimum Weight Spanning Tree algorithm sample

image::mst.png[]

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (a:Place {id: 'A'}), (b:Place {id: 'B'}), (c:Place {id: 'C'}), (d:Place {id: 'D'}), (e:Place {id: 'E'}), (f:Place {id: 'F'}), (g:Place {id: 'G'}), (d)-[:LINK {cost:4}]→(b), (d)-[:LINK {cost:6}]→(e), (b)-[:LINK {cost:1}]→(a), (b)-[:LINK {cost:3}]→(c), (a)-[:LINK {cost:2}]→(c), (c)-[:LINK {cost:5}]→(e), (f)-[:LINK {cost:1}]→(g); ----

Minimum weight spanning tree visits all nodes that are in the same connected component as the starting node, and returns a spanning tree of all nodes in the component where the total weight of the relationships is minimized.

.The following will run the Minimum Weight Spanning Tree algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Place {id: 'D'}) CALL gds.alpha.spanningTree.minimum.write({ nodeProjection: 'Place', relationshipProjection: { LINK: { type: 'LINK', properties: 'cost', orientation: 'UNDIRECTED' } }, startNodeId: id(n), relationshipWeightProperty: 'cost', writeProperty: 'MINST', weightWriteProperty: 'writeCost' }) YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount RETURN createMillis, computeMillis, writeMillis, effectiveNodeCount; ----

.To find all pairs of nodes included in our minimum spanning tree, run the following query: [source, cypher, role=noplay, indent=0] ---- MATCH path = (n:Place {id: 'D'})-[:MINST*]-() WITH relationships(path) AS rels UNWIND rels AS rel WITH DISTINCT rel AS rel RETURN startNode(rel).id AS source, endNode(rel).id AS destination, rel.writeCost AS cost ----

.Results image::minst_result.png[]

.Results [opts="header",cols="1,1,1"]

| Source | Destination | Cost | D | B | 4 | B | A | 1 | A | C | 2 | C | E | 5

The minimum spanning tree excludes the relationship with cost 6 from D to E, and the one with cost 3 from B to C. Nodes F and G aren't included because they're unreachable from D.

Maximum weighted tree spanning algorithm is similar to the minimum one, except that it returns a spanning tree of all nodes in the component where the total weight of the relationships is maximized.

.The following will run the maximum weight spanning tree algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Place{id: 'D'}) CALL gds.alpha.spanningTree.maximum.write({ nodeProjection: 'Place', relationshipProjection: { LINK: { type: 'LINK', properties: 'cost' } }, startNodeId: id(n), relationshipWeightProperty: 'cost', writeProperty: 'MAXST', weightWriteProperty: 'writeCost' }) YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount RETURN createMillis,computeMillis, writeMillis, effectiveNodeCount; ----

.Results image::maxst_result.png[]

=== K-Spanning tree

Sometimes we want to limit the size of our spanning tree result, as we are only interested in finding a smaller tree within our graph that does not span across all nodes. K-Spanning tree algorithm returns a tree with `k` nodes and `k - 1` relationships.

In our sample graph we have 5 nodes. When we ran MST above, we got a 5-minimum spanning tree returned, that covered all five nodes. By setting the `k=3`, we define that we want to get returned a 3-minimum spanning tree that covers 3 nodes and has 2 relationships.

.The following will run the k-minimum spanning tree algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Place{id: 'D'}) CALL gds.alpha.spanningTree.kmin.write({ nodeProjection: 'Place', relationshipProjection: { LINK: { type: 'LINK', properties: 'cost' } }, k: 3, startNodeId: id(n), relationshipWeightProperty: 'cost', writeProperty:'kminst' }) YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount RETURN createMillis,computeMillis,writeMillis, effectiveNodeCount; ----

.Find nodes that belong to our k-spanning tree result: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Place) WITH n.id AS Place, n.kminst AS Partition, count(*) AS count WHERE count = 3 RETURN Place, Partition ----

.Results [opts="header",cols="1,1"]

| Place | Partition | A | 1 | B | 1 | C | 1 | D | 3 | E | 4

Nodes A, B, and C are the result 3-minimum spanning tree of our graph.

.The following will run the k-maximum spanning tree algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Place{id: 'D'}) CALL gds.alpha.spanningTree.kmax.write({ nodeProjection: 'Place', relationshipProjection: { LINK: { type: 'LINK', properties: 'cost' } }, k: 3, startNodeId: id(n), relationshipWeightProperty: 'cost', writeProperty:'kmaxst' }) YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount RETURN createMillis,computeMillis,writeMillis, effectiveNodeCount; ----

.Find nodes that belong to our k-spanning tree result: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Place) WITH n.id AS Place, n.kmaxst AS Partition, count(*) AS count WHERE count = 3 RETURN Place, Partition ----

.Results [opts="header",cols="1,1"]

| Place | Partition | A | 0 | B | 1 | C | 3 | D | 3 | E | 3

Nodes C, D, and E are the result 3-maximum spanning tree of our graph.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Single Source Shortest Path algorithm in the Neo4j Graph Data Science library.
[.alpha] = Single Source Shortest Path

[abstract] — This section describes the Single Source Shortest Path algorithm in the Neo4j Graph Data Science library. — The Single Source Shortest Path (SSSP) algorithm calculates the shortest (weighted) path from a node to all other nodes in the graph.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

SSSP came into prominence at the same time as the shortest path algorithm and Dijkstra's algorithm can act as an implementation for both problems.

We implement a delta-stepping algorithm that has been shown to outperform Dijkstra's.

== Use-cases - when to use the Single Source Shortest Path algorithm

* Open Shortest Path First is a routing protocol for IP networks. It uses Dijkstra's algorithm to help detect changes in topology, such as link failures, and come up with a new routing structure in seconds.

== Constraints - when not to use the Single Source Shortest Path algorithm

Delta stepping does not support negative weights. The algorithm assumes that adding a relationship to a path can never make a path shorter - an invariant that would be violated with negative weights.

== Syntax

.The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.shortestPath.deltaStepping.write(configuration: Map) YIELD nodeCount, loadDuration, evalDuration, writeDuration ----

.Configuration [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | startNode | Node | null | no | The start node | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | delta | Float | null | yes | The grade of concurrency to use. | writeProperty | String | 'sssp' | yes | The property name written back to the node sequence of the node in the path. The property contains the cost it takes to get from the start node to the specific node.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes considered | loadDuration | Integer | Milliseconds for loading data | evalDuration | Integer | Milliseconds for running the algorithm | writeDuration | Integer | Milliseconds for writing result data back

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.shortestPath.deltaStepping.stream(configuration: Map) YIELD nodeId, distance ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | startNode | Node | null | no | The start node | delta | Float | null | no | The grade of concurrency to use. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID | distance | Integer | The cost it takes to get from the start node to the specific node.

== Single Source Shortest Path algorithm sample

image::sssp.png[]

.The following will create a sample graph: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (a:Loc {name: "A"}), (b:Loc {name: "B"}), (c:Loc {name: "C"}), (d:Loc {name: "D"}), (e:Loc {name: "E"}), (f:Loc {name: "F"}), (a)-[:ROAD {cost: 50}]→(b), (a)-[:ROAD {cost: 50}]→(c), (a)-[:ROAD {cost: 100}]→(d), (b)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 80}]→(e), (d)-[:ROAD {cost: 30}]→(e), (d)-[:ROAD {cost: 80}]→(f), (e)-[:ROAD {cost: 40}]→(f); ----

=== Delta stepping algorithm

[role=query-example] — .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Loc {name: 'A'}) CALL gds.alpha.shortestPath.deltaStepping.stream({ nodeProjection: 'Loc', relationshipProjection: { ROAD: { type: 'ROAD', properties: 'cost' } }, startNode: n, relationshipWeightProperty: 'cost', delta: 3.0 }) YIELD nodeId, distance RETURN gds.util.asNode(nodeId).name AS Name, distance AS Cost ----

.Results [opts="header",cols="1,1"]

| Name | Cost | "A" | 0.0 | "B" | 50.0 | "C" | 50.0 | "D" | 90.0 | "E" | 120.0 | "F" | 160.0

— The above table shows the cost of going from A to each of the other nodes, including itself at a cost of 0.

[role=query-example] — .The following will run the algorithm and write back results: [source, cypher, role=noplay, indent=0] ---- MATCH (n:Loc {name: 'A'}) CALL gds.alpha.shortestPath.deltaStepping.write({ nodeProjection: 'Loc', relationshipProjection: { ROAD: { type: 'ROAD', properties: 'cost' } }, startNode: n, relationshipWeightProperty: 'cost', delta: 3.0, writeProperty: 'sssp' }) YIELD nodeCount RETURN nodeCount ----

.Results [opts="header",cols="1"]

| nodeCount | 6

— ==== Cypher projection

If node labels and relationship types are not selective enough to project a graph, you can use Cypher queries instead. Cypher projections can also be used to run algorithms on a virtual graph. You can learn more in the Creating graphs using Cypher section of the manual.

[role=query-example] — [source, cypher, role=noplay, indent=0] ---- MATCH (start:Loc {name: 'A'}) CALL gds.alpha.shortestPath.deltaStepping.write({ nodeQuery:'MATCH(n:Loc) WHERE not n.name = "C" RETURN id(n) AS id', relationshipQuery:'MATCH(n:Loc)-[r:ROAD]→(m:Loc) WHERE not (n.name = "C" OR m.name = "C") RETURN id(n) AS source, id(m) AS target, r.cost AS weight', startNode: start, relationshipWeightProperty: 'weight', delta: 3.0, writeProperty: 'sssp' }) YIELD nodeCount RETURN nodeCount ----

.Results [opts="header",cols="1"]

| nodeCount | 5

— :leveloffset: 2

:leveloffset: +3

:description: This section describes the All Pairs Shortest Path algorithm in the Neo4j Graph Data Science library. [.alpha] = All Pairs Shortest Path

[abstract] — This section describes the All Pairs Shortest Path algorithm in the Neo4j Graph Data Science library. — The All Pairs Shortest Path (APSP) calculates the shortest (weighted) path between all pairs of nodes. This algorithm has optimizations that make it quicker than calling the Single Source Shortest Path algorithm for every pair of nodes in the graph.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

Some pairs of nodes might not be reachable between each other, so no shortest path exists between these pairs. In this scenario, the algorithm will return `Infinity` value as a result between these pairs of nodes.

Plain cypher does not support filtering `Infinity` values, so `gds.util.isFinite` function was added to help filter `Infinity` values from results.

== Use-cases - when to use the All Pairs Shortest Path algorithm

* The All Pairs Shortest Path algorithm is used in urban service system problems, such as the location of urban facilities or the distribution or delivery of goods. One example of this is determining the traffic load expected on different segments of a transportation grid. For more information, see Urban Operations Research. * All pairs shortest path is used as part of the REWIRE data center design algorithm that finds a network with maximum bandwidth and minimal latency. There are more details about this approach in "REWIRE: An Optimization-based Framework for Data Center Network Design"

== Syntax .The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.allShortestPaths.stream(configuration: Map) YIELD startNodeId, targetNodeId, distance ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. This is dependent on the Neo4j edition; for more information, see CPU. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph.

== All Pairs Shortest Path algorithm sample

image::sssp.png[]

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (a:Loc {name: 'A'}), (b:Loc {name: 'B'}), (c:Loc {name: 'C'}), (d:Loc {name: 'D'}), (e:Loc {name: 'E'}), (f:Loc {name: 'F'}), (a)-[:ROAD {cost: 50}]→(b), (a)-[:ROAD {cost: 50}]→(c), (a)-[:ROAD {cost: 100}]→(d), (b)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 40}]→(d), (c)-[:ROAD {cost: 80}]→(e), (d)-[:ROAD {cost: 30}]→(e), (d)-[:ROAD {cost: 80}]→(f), (e)-[:ROAD {cost: 40}]→(f);

.The following will run the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.allShortestPaths.stream({ nodeProjection: 'Loc', relationshipProjection: { ROAD: { type: 'ROAD', properties: 'cost' } }, relationshipWeightProperty: 'cost' }) YIELD sourceNodeId, targetNodeId, distance WITH sourceNodeId, targetNodeId, distance WHERE gds.util.isFinite(distance) = true

MATCH (source:Loc) WHERE id(source) = sourceNodeId MATCH (target:Loc) WHERE id(target) = targetNodeId WITH source, target, distance WHERE source <> target

RETURN source.name AS source, target.name AS target, distance ORDER BY distance DESC, source ASC, target ASC LIMIT 10 ----

.Results [opts="header",cols="1,1,1"]

| Source | Target | Cost | A | F | 160 | A | E | 120 | B | F | 110 | C | F | 110 | A | D | 90 | B | E | 70 | C | E | 70 | D | F | 70 | A | B | 50 | A | C | 50

This query returned the top 10 pairs of nodes that are the furthest away from each other. F and E appear to be quite distant from the others.

For now, only single-source shortest path support loading the relationship as undirected, but we can use Cypher loading to help us solve this. Undirected graph can be represented as Bidirected graph, which is a directed graph in which the reverse of every relationship is also a relationship.

We do not have to save this reversed relationship, we can project it using **Cypher loading**. Note that relationship query does not specify direction of the relationship. This is applicable to all other algorithms that use Cypher loading.

.The following will run the algorithm, treating the graph as undirected: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.allShortestPaths.stream({ nodeQuery: 'MATCH (n:Loc) RETURN id(n) AS id', relationshipQuery: 'MATCH (n:Loc)-[r:ROAD]-(p:Loc) RETURN id(n) AS source, id(p) AS target, r.cost AS cost', relationshipWeightProperty: 'cost' }) YIELD sourceNodeId, targetNodeId, distance WITH sourceNodeId, targetNodeId, distance WHERE gds.util.isFinite(distance) = true

MATCH (source:Loc) WHERE id(source) = sourceNodeId MATCH (target:Loc) WHERE id(target) = targetNodeId WITH source, target, distance WHERE source <> target

RETURN source.name AS source, target.name AS target, distance ORDER BY distance DESC, source ASC, target ASC LIMIT 10 ----

.Results [opts="header",cols="1,1,1"]

| Source | Target | Cost | A | F | 160 | F | A | 160 | A | E | 120 | E | A | 120 | B | F | 110 | C | F | 110 | F | B | 110 | F | C | 110 | A | D | 90 | D | A | 90

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Random Walk algorithm in the Neo4j Graph Data Science library. [.alpha] = Random Walk

[abstract] — This section describes the Random Walk algorithm in the Neo4j Graph Data Science library. — Random Walk is an algorithm that provides random paths in a graph.

A random walk means that we start at one node, choose a neighbor to navigate to at random or based on a provided probability distribution, and then do the same from that node, keeping the resulting path in a list. It's similar to how a drunk person traverses a city.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

The term "random walk" was first mentioned by Karl Pearson in 1905 in a letter to Nature magazine titled The Problem of the Random Walk. Study of random walks date back even further to the Gambler's ruin problem, where it could be used to show that a gambler would eventually go bankrupt against an opponent with infinite wealth.

It's only in the last couple of decades, however, that researchers have studied them with respect to networks.

== Use-cases - when to use the Random Walk algorithm

* It has be shown to relate to Brownian motion and also to the movement and dispersal of animals in the study of Random walk models in biology.

* It has been used to analyse ALSI index of the JSE stock exchange and show that the index followed the random walk hypothesis between years 2000 and 2011. This means the movement of stock prices was random and the ability of investors to perform relied more on luck than anything else. Find this study in The Random Walk Theory And Stock Prices: Evidence From Johannesburg Stock Exchange

Random Walk is often used as part of other algorithms:

* It can be used as part of the **node2vec** and **graph2vec** algorithms, that create node embeddings.

* It can be used as part of the **Walktrap** and **Infomap community detection** algorithms. If a random walk returns a small set of nodes repeatedly, then it indicates that those set of nodes may have a community structure.

* It can be used as part of the training process of machine learning model, as described in David Mack's article Review prediction with Neo4j and TensorFlow.

You can read about more use cases in Random walks and diffusion on networks.

== Constraints - when not to use the Random Walk algorithm

The constraints of Page Rank also apply to Random Walks:

* Dead-ends occur when pages have no out-links. In this case, the random walk will abort and a path containing only the first first node will be returned. This problem can be avoided by running on an undirected graph, so that the random walk will traverse relationships in both directions.

* If there are no links from within a group of pages to outside of the group, then the group is considered a spider trap. Random walks starting from any of the nodes in that group will only traverse to the others in the group - our implementation of the algorithm doesn't allow a random walk to jump to non-neighbouring nodes.

* Sinks can occur when a network of links form an infinite cycle.

| Name | Type | Default | Optional | Description | start | Object | null | yes | Starting points: null - whole graph, "Label" - nodes with that label, node-id - that node, list of node-ids - these nodes. | steps | Integer | 10 | yes | Length of paths returned, in case of error only path of lenght 1 is returned. | walks | Integer | 1 | yes | Number of paths returned. | mode | String | random | yes | Strategy for choosing the next relationship, modes: random and node2vec. | inOut | Float | 1.0 | yes | Parameter for node2vec. | return | Float | 1.0 | yes | Parameter for node2vec. | path | Boolean | false | yes | If the more expensive operation of creating a path from node-ids should be performed and returned in results. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph.

.Results [opts="header"]

| Name | Type | Description | startNodeId | Integer | Node ID starting the path. | nodeIds | List of Integer | List of Node ID forming a path. | path | Path | Optional Path (with virtual relationships).

== Random Walk algorithm sample

This sample will explain the Random Walk algorithm, using a simple graph:

image::pagerank.png[]

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (home:Page {name: 'Home'}), (about:Page {name: 'About'}), (product:Page {name: 'Product'}), (links:Page {name: 'Links'}), (a:Page {name: 'Site A'}), (b:Page {name: 'Site B'}), (c:Page {name: 'Site C'}), (d:Page {name: 'Site D'}),

(home)-[:LINKS]→(about), (about)-[:LINKS]→(home), (product)-[:LINKS]→(home), (home)-[:LINKS]→(product), (links)-[:LINKS]→(home), (home)-[:LINKS]→(links), (links)-[:LINKS]→(a), (a)-[:LINKS]→(home), (links)-[:LINKS]→(b), (b)-[:LINKS]→(home), (links)-[:LINKS]→(c), (c)-[:LINKS]→(home), (links)-[:LINKS]→(d), (d)-[:LINKS]→(home) ----

.The following will run the algorithm starting from the Home page and returning a 1 random walk, of path length 3: [source, cypher, role=noplay, indent=0] ---- MATCH (home:Page {name: 'Home'}) CALL gds.alpha.randomWalk.stream({ nodeProjection: '*', relationshipProjection: { LINKS: { type: 'LINKS', orientation: 'UNDIRECTED' } }, start: id(home), steps: 3, walks: 1 }) YIELD nodeIds UNWIND nodeIds AS nodeId RETURN gds.util.asNode(nodeId).name AS page ----

.Results [opts="header", cols="1"]

| page |"Home" |"Site C" |"Links" |"Site A"

== Cypher projection

If node labels and relationship types are not selective enough to project a graph, you can use Cypher queries instead. Cypher projections can also be used to run algorithms on a virtual graph. You can learn more in the Creating graphs using Cypher section of the manual.

[source, cypher, role=noplay, indent=0] ---- MATCH (home:Page {name: 'Home'}) CALL gds.alpha.randomWalk.stream({ nodeQuery: 'MATCH (p:Page) RETURN id(p) AS id', relationshipQuery: 'MATCH (p1:Page)-[:LINKS]→(p2:Page) RETURN id(p1) AS source, id(p2) AS target', start: id(home), steps: 5, walks: 1 }) YIELD nodeIds UNWIND nodeIds AS nodeId RETURN gds.util.asNode(nodeId).name AS page ----

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Breadth First Search traversal algorithm in the Neo4j Graph Data Science library. [.alpha] = Breadth First Search

[abstract] — This section describes the Breadth First Search traversal algorithm in the Neo4j Graph Data Science library. — [.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== Introduction

The Breadth First Search algorithm is a graph traversal algorithm that given a start node visits nodes in order of increasing distance, see https://en.wikipedia.org/wiki/Breadth-first_search. A related algorithm is the Depth First Search algorithm, Depth First Search. This algorithm is useful for searching when the likelihood of finding the node searched for decreases with distance. There are multiple termination conditions supported for the traversal, based on either reaching one of several target nodes, reaching a maximum depth, exhausting a given budget of traversed relationship cost, or just traversing the whole graph. The output of the procedure contains information about which nodes were visited and in what order.

== Syntax

.The following describes the API for running the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.bfs.stream(graphName: string, configuration: map) YIELD // general stream return columns startNodeId: int, nodeIds: int, path: Path ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String or Map | n/a | no | Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

.General configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result (applicable in WRITE mode). | nodeProjection | Map or List | null | yes | The node projection used for implicit graph loading or filtering nodes of an explicitly loaded graph. | relationshipProjection | Map or List | null | yes | The relationship projection used for implicit graph loading or filtering relationship of an explicitly loaded graph. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for implicit graph loading via a Cypher projection. | relationshipQuery |

String | null | yes | The Cypher query used to select the relationships for implicit graph loading via a Cypher projection. | nodeProperties | Map or List | null | yes | The node properties to load during implicit graph loading. | relationshipProperties | Map or List | null | yes | The relationship properties to load during implicit graph loading.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | startNodeId | Integer | n/a | no | The node id of the node where to start the traversal. | targetNodes | List of Integer | empty list | yes | Ids for target nodes. Traversal terminates when any target node is visited. | maxDepth | Integer | -1 | yes | The maximum distance from the start node at which nodes are visited.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | startNodeId | Integer | The node id of the node where to start the traversal. | nodeIds | List of Integer | The ids of all nodes that were visited during the traversal. | path | Path | A path containing all the nodes that were visited during the traversal.

== Examples Consider the graph created by the following Cypher statement: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (nA:Node {tag: 'a'}), (nB:Node {tag: 'b'}), (nC:Node {tag: 'c'}), (nD:Node {tag: 'd'}), (nE:Node {tag: 'e'}), (nA)-[:REL {cost: 8.0}]→(nB), (nA)-[:REL {cost: 9.0}]→(nC), (nB)-[:REL {cost: 1.0}]→(nE), (nC)-[:REL {cost: 5.0}]→(nD) ---- .The following statement will create the graph and store it in the graph catalog. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Node', 'REL', { relationshipProperties: 'cost' }) ---- In the following examples we will demonstrate using the Breadth First Search algorithm on this graph. [role=query-example] — .Running the Breadth First Search algorithm: [source, cypher, role=noplay, indent=0] ---- MATCH (a:Node{tag:'a'}) WITH id(a) AS startNode CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode}) YIELD path UNWIND [n in nodes(path)	n.tag] AS tags RETURN tags ORDER BY tags ---- .Results [opts="header"]

| tags | "a" | "b" | "c" | "d" | "e"

— Since none of the options for early termination are specified, the whole graph is visited during the traversal.

[role=query-example] — .Running the Breadth First Search algorithm with target nodes: [source, cypher, role=noplay, indent=0] ---- MATCH (a:Node{tag:'a'}), (d:Node{tag:'d'}), (e:Node{tag:'e'}) WITH id(a) AS startNode, [id(d), id(e)] AS targetNodes CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode, targetNodes: targetNodes}) YIELD path UNWIND [n in nodes(path)

n.tag] AS tags RETURN tags ORDER BY tags ----

.Results [opts="header"]

| tags | "a" | "b" | "c" | "e"

— [role=query-example] — .Running the Breadth First Search algorithm with maxDepth: [source, cypher, role=noplay, indent=0] ---- MATCH (a:Node{tag:'a'}) WITH id(a) AS startNode CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode, maxDepth: 1}) YIELD path UNWIND [n in nodes(path)

n.tag] AS tags RETURN tags ORDER BY tags ----

.Results [opts="header"]

| tags | "a" | "b" | "c"

— In the above example, nodes d and e were not visited since they are at distance 2 from a.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Depth First Search traversal algorithm in the Neo4j Graph Data Science library. [.alpha] = Depth First Search

[abstract] — This section describes the Depth First Search traversal algorithm in the Neo4j Graph Data Science library. — [.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== Introduction

The Depth First Search algorithm is a graph traversal that starts at a given node and explores as far as possible along each branch before backtracking, see https://en.wikipedia.org/wiki/Depth-first_search. A related algorithm is the Breath First Search algorithm, Breath First Search. This algorithm can be preferred over Breath First Search for example if one wants to find a target node at a large distance and exploring a random path has decent probability of success. There are multiple termination conditions supported for the traversal, based on either reaching one of several target nodes, reaching a maximum depth, exhausting a given budget of traversed relationship cost, or just traversing the whole graph. The output of the procedure contains information about which nodes were visited and in what order.

== Syntax

.The following describes the API for running the algorithm and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.dfs.stream(graphName: String, configuration: Map) YIELD // general stream return columns startNodeId: Integer, nodeIds: Integer, path: Path ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String or Map | n/a | no | Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering. Must be empty if

graphNameOrConfig is a Map.

.General configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for reading the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result (applicable in WRITE mode). | nodeProjection | Map or List | null | yes | The node projection used for implicit graph loading or filtering nodes of an explicitly loaded graph. | relationshipProjection | Map or List | null | yes | The relationship projection used for implicit graph loading or filtering relationship of an explicitly loaded graph. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for implicit graph loading via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for implicit graph loading via a Cypher projection. | nodeProperties | Map or List | null | yes | The node properties to load during implicit graph loading. | relationshipProperties | Map or List | null | yes | The relationship properties to load during implicit graph loading.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | startNodeId | Integer | n/a | no | The node id of the node where to start the traversal. | targetNodes | List of Integer | empty list | yes | Ids for target nodes. Traversal terminates when any target node is visited. | maxDepth | Integer | -1 | yes | The maximum distance from the start node at which nodes are visited.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | startNodeId | Integer | The node id of the node where to start the traversal. | nodeIds | List of Integer | The ids of all nodes that were visited during the traversal. | path | Path | A path containing all the nodes that were visited during the traversal.

== Examples Consider the graph created by the following Cypher statement: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (nA:Node {tag: 'a'}), (nB:Node {tag: 'b'}), (nC:Node {tag: 'c'}), (nD:Node {tag: 'd'}), (nE:Node {tag: 'e'}), (nA)-[:REL {cost: 8.0}]→(nB), (nA)-[:REL {cost: 9.0}]→(nC), (nB)-[:REL {cost: 1.0}]→(nE), (nC)-[:REL {cost: 5.0}]→(nD) ---- .The following statement will create the graph and store it in the graph catalog. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Node', 'REL', { relationshipProperties: 'cost' }) ---- In the following examples we will demonstrate using the Depth First Search algorithm on this graph. If we do not specify any of the options for early termination, the whole graph is visited: [role=query-example] — .Running the Depth First Search algorithm: [source, cypher, role=noplay, indent=0] ---- MATCH (a:Node{tag:'a'}) WITH id(a) AS startNode CALL gds.alpha.dfs.stream('myGraph', {startNode: startNode}) YIELD path UNWIND [n in nodes(path)	n.tag] AS tags RETURN tags ORDER BY tags ---- .Results [opts="header"]

| tags | "a" | "b" | "c" | "d" | "e"

— If specifying d and e as target nodes, not all nodes at distance 1 will be visited due to the depth first traversal order, in which node d is reached before b: [role=query-example] — .Running the Depth First Search algorithm with target nodes: [source, cypher, role=noplay, indent=0] ---- MATCH (a:Node{tag:'a'}), (d:Node{tag:'d'}), (e:Node{tag:'e'}) WITH id(a) AS startNode, [id(d), id(e)] AS targetNodes CALL gds.alpha.dfs.stream('myGraph', {startNode: startNode, targetNodes: targetNodes}) YIELD path UNWIND [n in nodes(path)	n.tag] AS tags RETURN tags ORDER BY tags ---- .Results [opts="header"]

| tags | "a" | "c" | "d"

— [role=query-example] — .Running the Depth First Search algorithm with maxDepth: [source, cypher, role=noplay, indent=0] ---- MATCH (a:Node{tag:'a'}) WITH id(a) AS startNode CALL gds.alpha.dfs.stream('myGraph', {startNode: startNode, maxDepth: 1}) YIELD path UNWIND [n in nodes(path)	n.tag] AS tags RETURN tags ORDER BY tags ---- .Results [opts="header"]

| tags | "a" | "b" | "c"

— In the above case, nodes d and e were not visited since they are at distance 2 from a.

:leveloffset: 2

:leveloffset: +2

:description: This chapter provides explanations and examples for each of the link prediction algorithms in the Neo4j Graph Data Science library. = Topological link prediction

[abstract] — This chapter provides explanations and examples for each of the link prediction algorithms in the Neo4j Graph Data Science library. — Link prediction algorithms help determine the closeness of a pair of nodes using the topology of the graph. The computed scores can then be used to predict new relationships between them.

[NOTE] ==== The following algorithms use only the topology of the graph to make predictions about relationships between nodes. To make predictions also utilizing node properties one can use the machine learning based methods Link prediction and Link prediction pipelines. ====

The Neo4j GDS library includes the following link prediction algorithms, grouped by quality tier:

* Alpha Adamic Adar Common Neighbors Preferential Attachment Resource Allocation Same Community Total Neighbors

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Adamic Adar algorithm in the Neo4j Graph Data Science library. [.alpha] = Adamic Adar

[abstract] — This section describes the Adamic Adar algorithm in the Neo4j Graph Data Science library. — Adamic Adar is a measure used to compute the closeness of nodes based on their shared neighbors.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

The Adamic Adar algorithm was introduced in 2003 by Lada Adamic and Eytan Adar to predict links in a social network. It is computed using the following formula:

image::adamic-adar.svg[role="middle"]

where `N(u)` is the set of nodes adjacent to `u`.

A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

The library contains a function to calculate closeness between two nodes.

== Syntax

.The following will run the algorithm and return the result: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.linkprediction.adamicAdar(node1:Node, node2:Node, { relationshipQuery:String, direction:String }) ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | `node1` | Node | null | no | A node | `node2` | Node | null | no | Another node | `relationshipQuery` | String | null | yes | The relationship type used to compute similarity between `node1` and `node2` | `direction` | String | BOTH | yes | The relationship direction used to compute similarity between `node1` and `node2`. Possible values are `OUTGOING`, `INCOMING` and `BOTH`.

== Adamic Adar algorithm sample

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (zhen:Person {name: 'Zhen'}), (praveena:Person {name: 'Praveena'}), (michael:Person {name: 'Michael'}), (arya:Person {name: 'Arya'}), (karin:Person {name: 'Karin'}),

(zhen)-[:FRIENDS]→(arya), (zhen)-[:FRIENDS]→(praveena), (praveena)-[:WORKS_WITH]→(karin), (praveena)-[:FRIENDS]→(michael), (michael)-[:WORKS_WITH]→(karin), (arya)-[:FRIENDS]→(karin) ----

.The following will return the Adamic Adar score for Michael and Karin: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.adamicAdar(p1, p2) AS score ----

.Results [opts="header",cols="1"]

| `score` | 0.9102392266268373

We can also compute the score of a pair of nodes based on a specific relationship type.

.The following will return the Adamic Adar score for Michael and Karin based only on the `FRIENDS` relationships: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.adamicAdar(p1, p2, {relationshipQuery: 'FRIENDS'}) AS score ----

.Results [opts="header",cols="1"]

| `score` | 0.0

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Common Neighbors algorithm in the Neo4j Graph Data Science library. [.alpha] = Common Neighbors

[abstract] — This section describes the Common Neighbors algorithm in the Neo4j Graph Data Science library. — Common neighbors captures the idea that two strangers who have a friend in common are more likely to be introduced than those who don't have any friends in common.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

It is computed using the following formula:

image::common-neighbors.svg[role="middle"]

where `N(x)` is the set of nodes adjacent to node `x`, and `N(y)` is the set of nodes adjacent to node `y`.

A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

The library contains a function to calculate closeness between two nodes.

== Syntax

.The following will run the algorithm and return the result: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.linkprediction.commonNeighbors(node1:Node, node2:Node, { relationshipQuery:String, direction:String }) ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | `node1` | Node | null | no | A node | `node2` | Node | null | no | Another node | `relationshipQuery` | String | null | yes | The relationship type used to compute similarity between `node1` and `node2`. | `direction` | String | BOTH | yes | The relationship direction used to compute similarity between `node1` and `node2`. Possible values are `OUTGOING`, `INCOMING` and `BOTH`.

== Common Neighbors algorithm sample

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (zhen:Person {name: 'Zhen'}), (praveena:Person {name: 'Praveena'}), (michael:Person {name: 'Michael'}), (arya:Person {name: 'Arya'}), (karin:Person {name: 'Karin'}),

(zhen)-[:FRIENDS]→(arya), (zhen)-[:FRIENDS]→(praveena), (praveena)-[:WORKS_WITH]→(karin), (praveena)-[:FRIENDS]→(michael), (michael)-[:WORKS_WITH]→(karin), (arya)-[:FRIENDS]→(karin) ----

.The following will return the number of common neighbors for Michael and Karin: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.commonNeighbors(p1, p2) AS score ----

.Results [opts="header",cols="1"]

| `score` | 1.0

We can also compute the score of a pair of nodes based on a specific relationship type.

.The following will return the number of common neighbors for Michael and Karin based only on the `FRIENDS` relationships: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.commonNeighbors(p1, p2, {relationshipQuery: "FRIENDS"}) AS score ----

.Results [opts="header",cols="1"]

| `score` | 0.0

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Preferential Attachment algorithm in the Neo4j Graph Data Science library. [.alpha] = Preferential Attachment

[abstract] — This section describes the Preferential Attachment algorithm in the Neo4j Graph Data Science library. — Preferential Attachment is a measure used to compute the closeness of nodes, based on their shared neighbors.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

Preferential attachment means that the more connected a node is, the more likely it is to receive new links. This algorithm was popularised by Albert-László Barabási and Réka Albert through their work on scale-free networks. It is computed using the following formula:

image::preferential-attachment.svg[role="middle"]

where `N(u)` is the set of nodes adjacent to `u`.

A value of 0 indicates that two nodes are not close, while higher values indicate that nodes are closer.

The library contains a function to calculate closeness between two nodes.

== Syntax

.The following will run the algorithm and return the result: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.linkprediction.preferentialAttachment(node1:Node, node2:Node, { relationshipQuery:String, direction:String }) ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | `node1` | Node | null | no | A node | `node2` | Node | null | no | Another node | `relationshipQuery` | String | null | yes | The relationship type used to compute similarity between `node1` and `node2` | `direction` | String | BOTH | yes | The relationship direction used to compute similarity between `node1` and `node2`. Possible values are `OUTGOING`, `INCOMING` and `BOTH`.

== Preferential Attachment algorithm sample

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (zhen:Person {name: 'Zhen'}), (praveena:Person {name: 'Praveena'}), (michael:Person {name: 'Michael'}), (arya:Person {name: 'Arya'}), (karin:Person {name: 'Karin'}),

(zhen)-[:FRIENDS]→(arya), (zhen)-[:FRIENDS]→(praveena), (praveena)-[:WORKS_WITH]→(karin), (praveena)-[:FRIENDS]→(michael), (michael)-[:WORKS_WITH]→(karin), (arya)-[:FRIENDS]→(karin) ----

.The following will return the Preferential Attachment score for Michael and Karin: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.preferentialAttachment(p1, p2) AS score ----

.Results [opts="header",cols="1"]

| `score` | 6.0

We can also compute the score of a pair of nodes based on a specific relationship type.

.The following will return the Preferential Attachment score for Michael and Karin based only on the `FRIENDS` relationship: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.preferentialAttachment(p1, p2, {relationshipQuery: "FRIENDS"}) AS score ----

.Results [opts="header",cols="1"]

| `score` | 1.0

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Resource Allocation algorithm in the Neo4j Graph Data Science library. [.alpha] = Resource Allocation

[abstract] — This section describes the Resource Allocation algorithm in the Neo4j Graph Data Science library. — Resource Allocation is a measure used to compute the closeness of nodes based on their shared neighbors.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

The Resource Allocation algorithm was introduced in 2009 by Tao Zhou, Linyuan Lü, and Yi-Cheng Zhang as part of a study to predict links in various networks. It is computed using the following formula:

image::resource-allocation.svg[role="middle"]

where `N(u)` is the set of nodes adjacent to `u`.

A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

The library contains a function to calculate closeness between two nodes.

== Syntax

.The following will run the algorithm and return the result: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.linkprediction.resourceAllocation(node1:Node, node2:Node, { relationshipQuery:String, direction:String }) ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | `node1` | Node | null | no | A node | `node2` | Node | null | no | Another node | `relationshipQuery` | String | null | yes | The relationship type to use to compute similarity between `node1` and `node2` | `direction` | String | BOTH | yes | The relationship direction used to compute similarity between `node1` and `node2`. Possible values are `OUTGOING`, `INCOMING` and `BOTH`.

== Resource Allocation algorithm sample

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (zhen:Person {name: 'Zhen'}), (praveena:Person {name: 'Praveena'}), (michael:Person {name: 'Michael'}), (arya:Person {name: 'Arya'}), (karin:Person {name: 'Karin'}),

(zhen)-[:FRIENDS]→(arya), (zhen)-[:FRIENDS]→(praveena), (praveena)-[:WORKS_WITH]→(karin), (praveena)-[:FRIENDS]→(michael), (michael)-[:WORKS_WITH]→(karin), (arya)-[:FRIENDS]→(karin) ----

.The following will return the Resource Allocation score for Michael and Karin: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.resourceAllocation(p1, p2) AS score ----

.Results [opts="header",cols="1"]

| `score` | 0.3333333333333333

We can also compute the score of a pair of nodes based on a specific relationship type.

.The following will return the Resource Allocation score for Michael and Karin based only on the `FRIENDS` relationships: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.resourceAllocation(p1, p2, {relationshipQuery: "FRIENDS"}) AS score ----

.Results [opts="header",cols="1"]

| `score` | 0.0

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Same Community algorithm in the Neo4j Graph Data Science library. [.alpha] = Same Community

[abstract] — This section describes the Same Community algorithm in the Neo4j Graph Data Science library. — Same Community is a way of determining whether two nodes belong to the same community. These communities could be computed by using one of the Community detection.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

If two nodes belong to the same community, there is a greater likelihood that there will be a relationship between them in future, if there isn't already.

A value of 0 indicates that two nodes are not in the same community. A value of 1 indicates that two nodes are in the same community.

The library contains a function to calculate closeness between two nodes.

== Syntax

.The following will run the algorithm and return the result: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.linkprediction.sameCommunity(node1:Node, node2:Node, communityProperty:String) ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | `node1` | Node | null | no | A node | `node2` | Node | null | no | Another node | `communityProperty` | String | 'community' | yes | The property that contains the community to which nodes belong

== Same Community algorithm sample

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (zhen:Person {name: 'Zhen', community: 1}), (praveena:Person {name: 'Praveena', community: 2}), (michael:Person {name: 'Michael', community: 1}), (arya:Person {name: 'Arya', partition: 5}), (karin:Person {name: 'Karin', partition: 5}), (jennifer:Person {name: 'Jennifer'}) ----

.The following will indicate that Michael and Zhen belong to the same community: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Zhen'}) RETURN gds.alpha.linkprediction.sameCommunity(p1, p2) AS score ----

.Results [opts="header",cols="1"]

| score | 1.0

.The following will indicate that Michael and Praveena do not belong to the same community: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Praveena'}) RETURN gds.alpha.linkprediction.sameCommunity(p1, p2) AS score ----

.Results [opts="header",cols="1"]

| score | 0.0

If one of the nodes doesn't have a community, this means it doesn't belong to the same community as any other node.

.The following will indicate that Michael and Jennifer do not belong to the same community: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Jennifer'}) RETURN gds.alpha.linkprediction.sameCommunity(p1, p2) AS score ----

.Results [opts="header",cols="1"]

| score | 0.0

By default, the community is read from the `community` property, but it is possible to explicitly state which property to read from.

.The following will indicate that Arya and Karin belong to the same community, based on the `partition` property: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Arya'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.sameCommunity(p1, p2, 'partition') AS score ----

.Results [opts="header",cols="1"]

| score | 1.0

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Total Neighbors algorithm in the Neo4j Graph Data Science library. [.alpha] = Total Neighbors

[abstract] — This section describes the Total Neighbors algorithm in the Neo4j Graph Data Science library. — Total Neighbors computes the closeness of nodes, based on the number of unique neighbors that they have. It is based on the idea that the more connected a node is, the more likely it is to receive new links.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== History and explanation

Total Neighbors is computed using the following formula:

image::total-neighbors.svg[role="middle"]

where `N(x)` is the set of nodes adjacent to `x`, and `N(y)` is the set of nodes adjacent to `y`.

A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

The library contains a function to calculate the closeness between two nodes.

== Syntax

.The following will run the algorithm and return the result: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.linkprediction.totalNeighbors(node1:Node, node2:Node, { relationshipQuery: null, direction: "BOTH" }) ----

.Parameters [opts="header",cols="1,1,1,1,4"]

| Name | Type | Default | Optional | Description | `node1` | Node | null | no | A node | `node2` | Node | null | no | Another node | `relationshipQuery` | String | null | yes | The relationship type used to compute similarity between `node1` and `node2` | `direction` | String | BOTH | yes | The relationship direction used to compute similarity between `node1` and `node2`. Possible values are `OUTGOING`, `INCOMING` and `BOTH`.

== Total Neighbors algorithm sample

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (zhen:Person {name: 'Zhen'}), (praveena:Person {name: 'Praveena'}), (michael:Person {name: 'Michael'}), (arya:Person {name: 'Arya'}), (karin:Person {name: 'Karin'}),

(zhen)-[:FRIENDS]→(arya), (zhen)-[:FRIENDS]→(praveena), (praveena)-[:WORKS_WITH]→(karin), (praveena)-[:FRIENDS]→(michael), (michael)-[:WORKS_WITH]→(karin), (arya)-[:FRIENDS]→(karin) ----

.The following will return the Total Neighbors score for Michael and Karin: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.totalNeighbors(p1, p2) AS score ----

.Results [opts="header",cols="1"]

| `score` | 4.0

We can also compute the score of a pair of nodes, based on a specific relationship type.

.The following will return the Total Neighbors score for Michael and Karin based only on the `FRIENDS` relationship: [source, cypher, role=noplay, indent=0] ---- MATCH (p1:Person {name: 'Michael'}) MATCH (p2:Person {name: 'Karin'}) RETURN gds.alpha.linkprediction.totalNeighbors(p1, p2, {relationshipQuery: "FRIENDS"}) AS score ----

.Results [opts="header",cols="1"]

| `score` | 2.0

:leveloffset: 2

:leveloffset: +2

:description: This chapter provides explanations and examples for the node embedding algorithms in the Neo4j Graph Data Science library. = Node embeddings

[abstract] — This chapter provides explanations and examples for the node embedding algorithms in the Neo4j Graph Data Science library. — Node embedding algorithms compute low-dimensional vector representations of nodes in a graph. These vectors, also called embeddings, can be used for machine learning. The Neo4j Graph Data Science library contains the following node embedding algorithms:

* Production-quality FastRP

* Beta GraphSAGE ** Node2Vec

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Fast Random Projection (FastRP) node embedding algorithm in the Neo4j Graph Data Science library. = Fast Random Projection :entity: node :result: embedding :algorithm: FastRP

[abstract] — This section describes the Fast Random Projection (FastRP) node embedding algorithm in the Neo4j Graph Data Science library. — :directed: :undirected: :homogeneous: :weighted: .Supported algorithm traits: [.graph-variants, caption=] — [.not-supported] Directed

[.not-supported] Undirected

[.not-supported] Homogeneous

[.not-supported] Heterogeneous

[.not-supported] Weighted —

== Introduction

Fast Random Projection, or FastRP for short, is a node embedding algorithm in the family of random projection algorithms. These algorithms are theoretically backed by the Johnsson-Lindenstrauss lemma according to which one can project *n* vectors of *arbitrary* dimension into *O(log(n))* dimensions and still approximately preserve pairwise distances among the points. In fact, a linear projection chosen in a random way satisfies this property.

Such techniques therefore allow for aggressive dimensionality reduction while preserving most of the distance information. The FastRP algorithm operates on graphs, in which case we care about preserving similarity between nodes and their neighbors. This means that two nodes that have similar neighborhoods should be assigned similar embedding vectors. Conversely, two nodes that are not similar should be not be assigned similar embedding vectors.

The FastRP algorithm initially assigns random vectors to all nodes using a technique called *very sparse random projection*, see (Achlioptas, 2003) below. The algorithm then iteratively constructs *intermediate* embeddings by averaging either neighboring intermediate embeddings from the previous iteration, or the generated random vectors during the first iteration. In each iteration, the intermediate embedding is normalised using a standard euclidean norm. That is, each element in the embedding is divided by the square root of the sum of squares of the emdedding elements.

In the end, the resulting embedding for each node is a weighted sum of the intermediate embeddings, where the weights are a configuration parameter called `iterationWeights`.

Therefore, each node's embedding depends on a neighborhood of radius equal to the number of iterations. This way FastRP exploits higher-order relationships in the graph while still being highly scalable.

The present implementation extends the original algorithm to support weighted graphs, which computes weighted

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | embeddingDimension | Integer | n/a | no | The dimension of the computed node embeddings. Minimum value is 1. | iterationWeights | List of Float | [0.0, 1.0, 1.0] | yes | Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding. | normalizationStrength | Float | 0 | yes | The initial random vector for each node is scaled by its degree to the power of `normalizationStrength`. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in computing the embeddings. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted. 5+| The number of iterations is equal to the length of `iterationWeights`, which must be at least one.

.Results [opts="header"]

| Name | Type | Description | nodeId | Integer | Node ID. | embedding | List of Float | FastRP node embedding.

======

[.include-with-stats] ======

.Run FastRP in stats mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.stats(graphName: String, configuration: Map) YIELD nodeCount: Integer, createMillis: Integer, computeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | embeddingDimension | Integer | n/a | no | The dimension of the computed node embeddings. Minimum value is 1. | iterationWeights | List of Float | [0.0, 1.0, 1.0] | yes | Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding. | normalizationStrength | Float | 0 | yes | The initial random vector for each node is scaled by its degree to the power of `normalizationStrength`. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in computing the embeddings. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted. 5+| The number of iterations is equal to the length of `iterationWeights`, which must be at least one.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | Number of nodes processed. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | configuration | Map | Configuration used for running the algorithm.

======

[.include-with-mutate] ======

.Run FastRP in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.mutate(graphName: String, configuration: Map) YIELD nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | embeddingDimension | Integer | n/a | no | The dimension of the computed node embeddings. Minimum value is 1. | iterationWeights | List of Float | [0.0, 1.0, 1.0] | yes | Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding. | normalizationStrength | Float | 0 | yes | The initial random vector for each node is scaled by its degree to the power of `normalizationStrength`. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in computing the embeddings. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted. 5+| The number of iterations is equal to the length of `iterationWeights`, which must be at least one.

.Results [opts="header"]

| Name | Type | Description | nodeCount | Integer | Number of nodes processed. | nodePropertiesWritten | Integer | Number of node properties written. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | configuration | Map | Configuration used for running the algorithm.

======

[.include-with-write] ======

.Run FastRP in write mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.write(graphName: String, configuration: Map) YIELD nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | embeddingDimension | Integer | n/a | no | The dimension of the computed node embeddings. Minimum value is 1. | iterationWeights | List of Float | [0.0, 1.0, 1.0] | yes | Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding. | normalizationStrength | Float | 0 | yes | The initial random vector for each node is scaled by its degree to the power of `normalizationStrength`. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in computing the embeddings. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted. 5+| The number of iterations is equal to the length of `iterationWeights`, which must be at least one.

.Results [opts="header"]

| Name | Type | Description | nodeCount | Integer | Number of nodes processed. | nodePropertiesWritten | Integer | Number of node properties written. | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | configuration | Map | Configuration used for running the algorithm.

======

====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run FastRP in write mode on an anonymous graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.write(configuration: Map) YIELD nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | WRITE mode only: The number of concurrent threads used for writing the result. | writeProperty | String | n/a | no | WRITE mode only: The {entity} property to which the {result} is written to.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | embeddingDimension | Integer | n/a | no | The dimension of the computed node embeddings. Minimum value is 1. | iterationWeights | List of Float | [0.0, 1.0, 1.0] | yes | Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding. | normalizationStrength | Float | 0 | yes | The initial random vector for each node is scaled by its degree to the power of `normalizationStrength`. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in computing the embeddings. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted. 5+| The number of iterations is equal to the length of `iterationWeights`, which must be at least one.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: FastRP node embedding :graph-description: social network :image-file: fastrp.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (dan:Person {name: 'Dan'}), (annie:Person {name: 'Annie'}), (matt:Person {name: 'Matt'}), (jeff:Person {name: 'Jeff'}), (brie:Person {name: 'Brie'}), (elsa:Person {name: 'Elsa'}), (john:Person {name: 'John'}),

(dan)-[:KNOWS {weight: 1.0}]→(annie), (dan)-[:KNOWS {weight: 1.0}]→(matt), (annie)-[:KNOWS {weight: 1.0}]→(matt), (annie)-[:KNOWS {weight: 1.0}]→(jeff), (annie)-[:KNOWS {weight: 1.0}]→(brie), (matt)-[:KNOWS {weight: 3.5}]→(brie), (brie)-[:KNOWS {weight: 1.0}]→(elsa), (brie)-[:KNOWS {weight: 2.0}]→(jeff), (john)-[:KNOWS {weight: 1.0}]→(jeff); ----

This graph represents seven people who know one another. A relationship property `weight` denotes the strength of the knowledge between two persons.

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. For the relationships we will use the `UNDIRECTED` orientation. This is because the FastRP algorithm has been measured to compute more predictive node embeddings in undirected graphs. We will also add the `weight` relationship property which we will make use of when running the weighted version of FastRP.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'persons'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('persons', 'Person', { KNOWS: { orientation: 'UNDIRECTED', properties: 'weight' } }) ----

=== Memory Estimation

:mode: stream First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm: [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.stream.estimate('persons', {embeddingDimension: 128}) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header", cols="1,1,1,1,1"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 7 | 18 | 11416 | 11416 | "11416 Bytes"

—

=== Stream

:stream-details: For example, we can collect the results and pass them into a similarity algorithm. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm, and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.stream('persons', { embeddingDimension: 4, randomSeed: 42 }) YIELD nodeId, embedding ----

[opts=header] .Results

| nodeId | embedding | 0 | [0.4774002134799957, -0.6602408289909363, -0.36686956882476807, -1.7089111804962158] | 1 | [0.7989360094070435, -0.4918718934059143, -0.41281944513320923, -1.6314401626586914] | 2 | [0.47275322675704956, -0.49587157368659973, -0.3340468406677246, -1.7141895294189453] | 3 | [0.8290714025497437, -0.3260476291179657, -0.3317275643348694, -1.4370529651641846] | 4 | [0.7749264240264893, -0.4773247539997101, 0.0675133764743805, -1.5248265266418457] | 5 | [0.8408374190330505, -0.37151476740837097, 0.12121132016181946, -1.530960202217102] | 6 | [1.0, -0.11054422706365585, -0.3697933852672577, -0.9225144982337952]

— The results of the algorithm are not very intuitively interpretable, as the node embedding format is a mathematical abstraction of the node within its neighborhood, designed for machine learning programs. What we can see is that the embeddings have four elements (as configured using `embeddingDimension`) and that the numbers are relatively small (they all fit in the range of `[-2, 2]`). The magnitude of the numbers is controlled by the `embeddingDimension`, the number of nodes in the graph, and by the fact that FastRP performs euclidean normalization on the intermediate embedding vectors.

[NOTE] ==== Due to the random nature of the algorithm the results will vary between the runs. However, this does not necessarily mean that the pairwise distances of two node embeddings vary as much. ====

=== Stats

:stats-syntax: algorithms-embeddings-fastrp-syntax In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in the syntax section.

For more details on the `stats` mode in general, see Stats.

[role=query-example] — .The following will run the algorithm and returns the result in form of statistical and measurement values [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.stats('persons', { embeddingDimension: 8 }) YIELD nodeCount ----

[opts=header] .Results

| nodeCount | 7

— The `stats` mode does not currently offer any statistical results for the embeddings themselves. We can however see that the algorithm has successfully processed all seven nodes in our example graph.

=== Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new {entity} property containing the {result} for that {entity}. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.mutate('persons', { embeddingDimension: 8, mutateProperty: 'fastrp-embedding' }) YIELD nodePropertiesWritten ----

[opts=header] .Results

| nodePropertiesWritten | 7

— The returned result is similar to the `stats` example. Additionally, the graph 'persons' now has a node property `fastrp-embedding` which stores the node embedding for each node. To find out how to inspect the new schema of the in-memory graph, see Listing graphs.

=== Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the {result} for each {entity} as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

[role=query-example] — .The following will run the algorithm in `write` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.write('persons', { embeddingDimension: 8, writeProperty: 'fastrp-embedding' }) YIELD nodePropertiesWritten ----

[opts=header] .Results

| nodePropertiesWritten | 7

— The returned result is similar to the `stats` example. Additionally, each of the seven nodes now has a new property `fastrp-embedding` in the Neo4j database, containing the node embedding for that node.

=== Weighted

By default, the algorithm is considering the relationships of the graph to be unweighted. To change this behaviour we can use configuration parameter called `relationshipWeightProperty`. Below is an example of running the weighted variant of algorithm.

[role=query-example] — .The following will run the algorithm, and stream results: [source, cypher, role=noplay, indent=0] ---- CALL gds.fastRP.stream('persons', { embeddingDimension: 4, randomSeed: 42, relationshipWeightProperty: 'weight' }) YIELD nodeId, embedding ----

[opts=header] .Results

| nodeId | embedding | 0 | [0.10945529490709305, -0.5032674074172974, 0.464673787355423, -1.7539862394332886] | 1 | [0.3639600872993469, -0.39210301637649536, 0.46271592378616333, -1.829423427581787] | 2 | [0.12314096093177795, -0.3213110864162445, 0.40100979804992676, -1.471055269241333] | 3 | [0.30704641342163086, -0.24944794178009033, 0.3947891891002655, -1.3463698625564575] | 4 | [0.23112300038337708, -0.30148714780807495, 0.584831714630127, -1.2822188138961792] | 5 | [0.14497177302837372, -0.2312137484550476, 0.5552002191543579, -1.2605633735656738] | 6 | [0.5139184594154358, -0.07954332232475281, 0.3690345287322998, -0.9176374077796936]

— Since the initial state of the algorithm is randomised, it isn't possible to intuitively analyse the effect of the relationship weights.

== Extended Algorithm: Using Node Properties

[.beta-symbol] [.tier-note] The following aspects of the algorithm are in the beta tier. For more information on algorithm tiers, see Algorithms.

Most real-world graphs contain node properties which store information about the nodes and what they represent. An embedding algorithm which can process the node properties as features and incorporate them in the embeddings can therefore represent the graph more accurately.

The extended FastRP algorithm (FastRPExtended) has the additional configuration parameters `featureProperties` and `propertyDimension`. Each node property specified in the former is associated with a randomly generated vector of dimension `propertyDimension`. Each node is then initialized with a vector of size `embeddingDimension` formed by concatenation of two parts: The first part is formed like in the FastRP algorithm, and the second one is a linear combination of the property vectors, using the property values of the node as weights. The algorithm then proceeds with the same logic as the FastRP algorithm. Therefore, the algorithm will output arrays of size `embeddingDimension`. The last `propertyDimension` coordinates in the embedding captures information about property values of nearby nodes, and the remaining coordinates (`embeddingDimension` - `propertyDimension` of them) captures information about nearby presence of nodes.

As other configuration parameters, `propertyDimension` needs to be tuned for optimal performance. We suggest keeping the previously selected value for `embeddingDimension` and setting `propertyDimension` to half that value as a starting point for using FastRPExtended. Other factors that may influence the choice of `propertyDimension` are vaguely the amount of valuable information contained in the node properties, i.e. number of properties and how independent they are, as well as how relevant the properties are to the problem at hand.

.FastRPExtended syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ====== .Run FastRPExtended in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.fastRPExtended.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, embedding: List of Float ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | propertyDimension | Integer | 0 | yes | The dimension of the projected node properties. Maximum value is `embeddingDimension`. | featureProperties | List of String | [] | yes | The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float. | embeddingDimension | Integer | n/a | no | The dimension of the computed node embeddings. Minimum value is 1. | iterationWeights | List of Float | [0.0, 1.0, 1.0] | yes | Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding. | normalizationStrength | Float | 0 | yes | The initial random vector for each node is scaled by its degree to the power of `normalizationStrength`. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in computing the embeddings. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted. 5+| The number of iterations is equal to the length of `iterationWeights`, which must be at least one.

.Results [opts="header",cols="1m,1,6"]

| Name | Type | Description | nodeId | Integer | The Neo4j node ID. | embedding | List of Float | The computed node embedding.

======

[.include-with-stats] ====== .Run FastRPExtended in stats mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.fastRPExtended.stats(graphName: String, configuration: Map) YIELD nodeCount: Integer, createMillis: Integer, computeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | propertyDimension | Integer | 0 | yes | The dimension of the projected node properties. Maximum value is `embeddingDimension`. | featureProperties | List of String | [] | yes | The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float. | embeddingDimension | Integer | n/a | no | The dimension of the computed node embeddings. Minimum value is 1. | iterationWeights | List of Float | [0.0, 1.0, 1.0] | yes | Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding. | normalizationStrength | Float | 0 | yes | The initial random vector for each node is scaled by its degree to the power of `normalizationStrength`. |

randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in computing the embeddings. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted. 5+| The number of iterations is equal to the length of `iterationWeights`, which must be at least one.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes processed. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-mutate] ====== .Run FastRPExtended in mutate mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.fastRPExtended.mutate(graphName: String, configuration: Map) YIELD nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | propertyDimension | Integer | 0 | yes | The dimension of the projected node properties. Maximum value is `embeddingDimension`. | featureProperties | List of String | [] | yes | The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float. | embeddingDimension | Integer | n/a | no | The dimension of the computed node embeddings. Minimum value is 1. | iterationWeights | List of Float | [0.0, 1.0, 1.0] | yes | Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding. | normalizationStrength | Float | 0 | yes | The initial random vector for each node is scaled by its degree to the power of `normalizationStrength`. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in computing the embeddings. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted. 5+| The number of iterations is equal to the length of `iterationWeights`, which must be at least one.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes processed. |

nodePropertiesWritten | Integer | The number of node properties written. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | configuration | Map | The configuration used for running the algorithm.

```
======

[.include-with-write] ====== .Run FastRPExtended in write mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.fastRPExtended.write( graphName: String, configuration: Map ) YIELD nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | propertyDimension | Integer | 0 | yes | The dimension of the projected node properties. Maximum value is `embeddingDimension`. | featureProperties | List of String | [] | yes | The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float. | embeddingDimension | Integer | n/a | no | The dimension of the computed node embeddings. Minimum value is 1. | iterationWeights | List of Float | [0.0, 1.0, 1.0] | yes | Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding. | normalizationStrength | Float | 0 | yes | The initial random vector for each node is scaled by its degree to the power of `normalizationStrength`. | randomSeed | Integer | n/a | yes | A random seed which is used for all randomness in computing the embeddings. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted. 5+| The number of iterations is equal to the length of `iterationWeights`, which must be at least one.

```
.Results [opts="header",cols="1,1,6"]
```

| Name | Type | Description | nodeCount | Integer | The number of nodes processed. | nodePropertiesWritten | Integer | The number of node properties written. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | configuration | Map | The configuration used for running the algorithm.

====== ====

Memory estimation for FastRPExtended works similarly to FastRP, which we can see in the following example for the stream mode:

[role=query-example] — .The following will estimate the memory requirements for FastRPExtended: [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.fastRPExtended.stream.estimate('persons', {embeddingDimension: 128, propertyDimension: 64, featureProperties: ['p1', 'p2']}) YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory ----

.Results [opts="header", cols="1,1,1,1,1"]

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory | 7 | 18 | 11928 | 11928 | "11928 Bytes"

— :leveloffset: 2

:leveloffset: +3

:description: This section describes the GraphSAGE node embedding algorithm in the Neo4j Graph Data Science library. [.beta] = GraphSAGE :entity: node :result: embedding :modelType: GraphSAGE :algorithm: GraphSAGE

[abstract] — This section describes the GraphSAGE node embedding algorithm in the Neo4j Graph Data Science library. — GraphSAGE is an *inductive* algorithm for computing node embeddings. GraphSAGE is using node feature information to generate node embeddings on unseen nodes or graphs. Instead of training individual embeddings for each node, the algorithm learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood.

NOTE: The algorithm is defined for UNDIRECTED graphs.

For more information on this algorithm see:

* William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs." 2018. * Amit Pande, Kai Ni and Venkataramani Kini. "SWAG: Item Recommendations using Convolutions on Weighted Graphs." 2019.

== Syntax

.GraphSAGE syntax per mode [.tabbed-example, caption =] ====

[.include-with-train] ====== .Run GraphSAGE in train mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graphSage.train(graphName: String, configuration: Map) YIELD graphName: String, graphCreateConfig: Map, modelInfo: Map, configuration: Map, trainMillis: Integer ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | **List of String** | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="2,1,1m,1,3"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of the model to train, must not exist in the Model Catalog. | featureProperties | List of String | n/a | no | The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float. | embeddingDimension | Integer | 64 | yes | The dimension of the generated node embeddings as well as their hidden layer representations. | aggregator | String | "mean" | yes | The aggregator to be used by the layers. Supported values are "mean" and "pool". | activationFunction | String | "sigmoid" | yes | The activation function to be used in the model architecture. Supported values are "sigmoid" and "relu". | sampleSizes | List of Integer | [25, 10] | yes | A list of Integer values, the size of the list determines the number of layers and the values determine how many nodes will be sampled by the layers. | projectedFeatureDimension | Integer | n/a | yes | The dimension of the projected `featureProperties`. This enables multi-label GraphSage, where each label can have a subset of the `featureProperties`. | batchSize | Integer | 100 | yes | The number of nodes per batch. | tolerance | Float | 1e-4 | yes | Tolerance used for the early convergence of an epoch. | learningRate | Float | 0.1 | yes | The learning rate determines the step size at each iteration while moving toward a minimum of a loss function. | epochs | Integer | 1 | yes | Number of times to traverse the graph. | maxIterations | Integer | 10 | yes | Maximum number of weight updates per batch. Batches can also converge early based on `tolerance`. | searchDepth | Integer | 5 | yes | Maximum depth of the RandomWalks to sample nearby nodes for the training. | negativeSampleWeight | Integer | 20 | yes | The weight of the negative samples. Higher values increase the impact of negative samples in the loss. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | randomSeed | Integer | random | yes | A random seed which is used to control the randomness in computing the embeddings.

.Results [opts="header",cols="2m,1,6"]

| Name | Type | Description | graphName | String | The name of the in-memory graph used during training. | graphCreateConfig | Map | Configuration used to create in-memory graph. Only has value if `anonymous graph` was used. | modelInfo | Map | Details of the trained model. | configuration | Map | The configuration used to run the procedure. | trainMillis | Integer | Milliseconds to train the model.

.Details on `modelInfo` [opts="header",cols="1m,1,6"]

| Name | Type | Description | name | String | The name of the trained model. | type | String | The type of the trained model. Always `graphSage`. | metrics | Map | Metrics related to running the training, details in the table below.

.Metrics collected during training [opts="header",cols="2m,1,6"]

| Name | Type | Description | ranEpochs | Integer | The number of ran epochs during training. | epochLosses | List | Ordered list of the losses after each epoch. | didConverge | Boolean | Indicates if the training has converged.

```
======

[.include-with-stream] ====== .Run GraphSAGE in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graphSage.stream( graphName: String, configuration: Map ) YIELD nodeId: Integer, embedding: List ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | batchSize | Integer | 100 | yes | The number of nodes per batch.

```
.Results [opts="header",cols="1m,1,6"]
```

| Name | Type | Description | nodeId | Integer | The Neo4j node ID. | embedding | List of Float | The computed node embedding.

```
======

[.include-with-mutate] ====== .Run GraphSAGE in mutate mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graphSage.mutate( graphName: String, configuration: Map ) YIELD nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | batchSize | Integer | 100 | yes | The number of nodes per

batch.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes processed. | nodePropertiesWritten | Integer | The number of node properties written. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for writing result data back to the in-memory graph. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ====== .Run GraphSAGE in write mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graphSage.write(graphName: String, configuration: Map) YIELD nodeCount: Integer, nodePropertiesWritten: Integer, createMillis: Integer, computeMillis: Integer, writeMillis: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | batchSize | Integer | 100 | yes | The number of nodes per batch.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes processed. | nodePropertiesWritten | Integer | The number of node properties written. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | configuration | Map | The configuration used for running the algorithm.

====== ====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run GraphSAGE in write mode on an anonymous graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graphSage.write(configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, nodeCount: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | WRITE mode only: The number of concurrent threads used for writing the result. | writeProperty | String | n/a | no | WRITE mode only: The {entity} property to which the {result} is written to.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | batchSize | Integer | 100 | yes | The number of nodes per batch.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

:algorithm-name: {algorithm} :graph-description: friends network :image-file: graph-sage-graph.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE // Persons (dan:Person {name: 'Dan', age: 20, heightAndWeight: [185, 75]}), (annie:Person {name: 'Annie', age: 12, heightAndWeight: [124, 42]}), (matt:Person {name: 'Matt', age: 67, heightAndWeight: [170, 80]}), (jeff:Person {name: 'Jeff', age: 45, heightAndWeight: [192, 85]}), (brie:Person {name: 'Brie', age: 27, heightAndWeight: [176, 57]}), (elsa:Person {name: 'Elsa', age: 32, heightAndWeight: [158, 55]}), (john:Person {name: 'John', age: 35, heightAndWeight: [172, 76]}),

(dan)-[:KNOWS {relWeight: 1.0}]→(annie), (dan)-[:KNOWS {relWeight: 1.6}]→(matt), (annie)-[:KNOWS {relWeight: 0.1}] →(matt), (annie)-[:KNOWS {relWeight: 3.0}]→(jeff), (annie)-[:KNOWS {relWeight: 1.2}]→(brie), (matt)-[:KNOWS {relWeight: 10.0}]→(brie), (brie)-[:KNOWS {relWeight: 1.0}]→(elsa), (brie)-[:KNOWS {relWeight: 2.2}]→(jeff), (john)-[:KNOWS {relWeight: 5.0}]→(jeff) ----

[source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('persons', { Person: { label: 'Person', properties: ['age', 'heightAndWeight'] } }, { KNOWS: { type: 'KNOWS', orientation: 'UNDIRECTED', properties: ['relWeight'] } }) ----

NOTE: The algorithm is defined for UNDIRECTED graphs.

=== Train

Before we are able to generate node embeddings we need to train a model and store it in the model catalog. Below is an example of how to do that.

NOTE: The names specified in the `featureProperties` configuration parameter must exist in the in-memory graph.

[source, cypher, role=noplay query-example, no-result=true, indent=0] ---- CALL gds.beta.graphSage.train('persons', { modelName: 'exampleTrainModel', featureProperties: ['age', 'heightAndWeight'], aggregator: 'mean', activationFunction: 'sigmoid', sampleSizes: [25, 10] }) YIELD modelInfo as info RETURN info.name as modelName, info.metrics.didConverge as didConverge, info.metrics.ranEpochs as ranEpochs, info.metrics.epochLosses as epochLosses ----

[opts="header",cols="2,1,1,4"] .Results

| modelName | didConverge | ranEpochs | epochLosses | exampleTrainModel | true | 1 | [186.0494816886275, 186.04946806237382]

NOTE: Due to the random initialisation of the weight variables the results may vary between different runs.

Looking at the results we can draw the following conclusions, the training converged after a single epoch, the losses are almost identical. Tuning the algorithm parameters, such as trying out different `sampleSizes`, `searchDepth`, `embeddingDimension` or `batchSize` can improve the losses. For different datasets, GraphSAGE may require different train parameters for producing good models.

The trained model is automatically registered in the model catalog.

=== Train with multiple node labels

In this section we describe how to train on a graph with multiple labels. The different labels may have different sets of properties. To run on such a graph, GraphSAGE is run in *multi-label mode*, in which the feature properties are projected into a common feature space. Therefore, all nodes have feature vectors of the same dimension after the projection.

The projection for a label is linear and given by a matrix of weights. The weights for each label are learned jointly with the other weights of the GraphSAGE model.

In the multi-label mode, the following is applied prior to the usual aggregation layers:

. A property representing the label is added to the feature properties for that label . The feature properties for each label are projected into a feature vector of a shared dimension

The projected feature dimension is configured with `projectedFeatureDimension`, and specifying it enables the multi-label mode.

The feature properties used for a label are those present in the `featureProperties` configuration parameter which exist in the graph for that label. In the multi-label mode, it is no longer required that all labels have all the specified properties.

==== Assumptions

- A requirement for multi-label mode is that each node belongs to exactly one label. - A GraphSAGE model trained in this mode must be applied on graphs with the same schema with regards to node labels and properties.

==== Examples

In order to demonstrate GraphSAGE with multiple labels, we add instruments and relationships of type `LIKE` between person and instrument to the example graph.

image::example-graphs/graph-sage-multi-label-graph.svg[Visualization of the multi-label example graph,align="center"]

.The following Cypher statement will extend the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- MATCH (dan:Person {name: "Dan"}), (annie:Person {name: "Annie"}), (matt:Person {name: "Matt"}), (brie:Person {name: "Brie"}), (john:Person {name: "John"}) CREATE (guitar:Instrument {name: 'Guitar', cost: 1337.0}), (synth:Instrument {name: 'Synthesizer', cost: 1337.0}), (bongos:Instrument {name: 'Bongos', cost: 42.0}), (trumpet:Instrument {name: 'Trumpet', cost: 1337.0}), (dan)-[:LIKES]→(guitar), (dan)-[:LIKES]→(synth), (dan)-[:LIKES]→(bongos), (annie)-[:LIKES]→(guitar), (annie)-[:LIKES]→(synth), (matt)-[:LIKES]→(bongos), (brie)-[:LIKES]→(guitar), (brie)-[:LIKES]→(synth), (brie)-[:LIKES]→(bongos), (john)-[:LIKES]→(trumpet) ----

[source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('persons_with_instruments', { Person: { label: 'Person', properties: ['age', 'heightAndWeight'] }, Instrument: { label: 'Instrument', properties: ['cost'] } }, { KNOWS: { type: 'KNOWS', orientation: 'UNDIRECTED' }, LIKES: { type: 'LIKES', orientation: 'UNDIRECTED' } }) ----

We can now run GraphSAGE in multi-label mode on that graph by specifying the `projectedFeatureDimension` parameter. Multi-label GraphSAGE removes the requirement, that each node in the in-memory graph must have all `featureProperties`. However, the projections are independent per label and even if two labels have the same `featureProperty` they are considered as different features before projection. The `projectedFeatureDimension` equals the maximum length of the feature-array, i.e., `age` and `cost` both are scalar features plus the list feature `heightAndWeight` which has a length of two. For each node its unique labels properties is projected using a label specific projection to vector space of dimension

| nodeId | embedding | 0 | [0.5285002502143177, 0.4682181762801141, 0.7081378570737874] | 1 | [0.5285002502147674, 0.46821817628034773, 0.7081378570732975] | 2 | [0.5285002502143014, 0.46821817628010554, 0.7081378570738053] | 3 | [0.5285002502129178, 0.46821817627938667, 0.7081378570753134] | 4 | [0.5285002502572376, 0.46821817630241636, 0.7081378570270093] | 5 | [0.5285002503196665, 0.46821817633485613, 0.7081378569589678] | 6 | [0.528500250213112, 0.46821817627948753, 0.7081378570751017]

— NOTE: Due to the random initialisation of the weight variables the results may vary slightly between the runs.

=== Mutate

The model trained as part of the stream example can be reused to write the results to the in-memory graph using the `mutate` mode of the procedure. Below is an example of how to achieve this. — [source, cypher, role=noplay query-example, no-result=true, group=graphSage, indent=0] ---- CALL gds.beta.graphSage.mutate('persons', { mutateProperty: 'inMemoryEmbedding', modelName: 'graphSage' }) YIELD nodeCount, nodePropertiesWritten ----

[opts=header] .Results

| nodeCount | nodePropertiesWritten | 7 | 7

— === Write

The model trained as part of the stream example can be reused to write the results to Neo4j. Below is an example of how to achieve this. — [source, cypher, role=noplay query-example, no-result=true, group=graphSage, indent=0] ---- CALL gds.beta.graphSage.write('persons', { writeProperty: 'embedding', modelName: 'graphSage' }) YIELD nodeCount, nodePropertiesWritten ----

[opts=header] .Results

| nodeCount | nodePropertiesWritten | 7 | 7

— == Caveats

If you are embedding a graph that has an isolated node, the aggregation step in GraphSAGE can only draw information from the node itself. When all the properties of that node are `0.0`, and the activation function is relu, this leads to an all-zero vector for that node. However, since GraphSAGE normalizes node embeddings using the L2-norm, and a zero vector cannot be normalized, we assign all-zero embeddings to such nodes under these special circumstances. In scenarios where you generate all-zero embeddings for orphan nodes, that may have impacts on downstream tasks such as nearest neighbor or other similarity algorithms. It may be more appropriate to filter out these disconnected nodes prior to running GraphSAGE.

When running `gds.beta.graphSage.train.estimate`, the feature dimension is computed as if each feature property is scalar.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Node2Vec node embedding algorithm in the Neo4j Graph Data Science library. [.beta] = Node2Vec :entity: node :result: embedding

[abstract] — This section describes the Node2Vec node embedding algorithm in the Neo4j Graph Data Science library. — Node2Vec is a node embedding algorithm that computes a vector representation of a node based on random walks in the graph. The neighborhood is sampled through random walks. Using a number of random neighborhood samples, the algorithm trains a single hidden layer neural network. The neural network is trained to predict the likelihood that a node will occur in a walk based on the occurrence of another node.

For more information on this algorithm, see:

* Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for networks." Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. 2016. * https://snap.stanford.edu/node2vec/

== Random Walks

A main concept of the Node2Vec algorithm are the second order random walks. A random walk simulates a traversal of the graph in which the traversed relationships are chosen at random. In a classic random walk, each relationship has the same, possibly weighted, probability of being picked. This probability is not influenced by the previously visited nodes. The concept of second order random walks, however, tries to model the transition probability based on the currently visited node `v`, the node `t` visited before the current one, and the node `x` which is the target of a candidate relationship. Node2Vec random walks are thus influenced by two parameters: the `returnFactor` and the `inOutFactor`:

* The `returnFactor` is used if `t` equals `x`, i.e., the random walk returns to the previously visited node. * The `inOutFactor` is used if the distance from `t` to `x` is equal to 2, i.e., the walk traverses further away from the node `t`

image::randomWalk.svg[Visuzalition of random walk parameters,align="center"]

The probabilities for traversing a relationship during a random walk can be further influenced by specifying a `relationshipWeightProperty`. A relationship property value greater than 1 will increase the likelihood of a relationship being traversed, a property value between 0 and 1 will decrease that probability.

For every node in the graph Node2Vec generates a series of random walks with the particular node as start node. The number of random walks per node can be influenced by the `walkPerNode` configuration parameters, the walk length is controlled by the `walkLength` parameter.

== Syntax

.Node2Vec syntax per mode [.tabbed-example, caption =] ====

[.include-with-stream] ====== .Run Node2Vec in stream mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.node2vec.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, embedding: List of Float ----

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | walkLength | Integer | 80 | yes | The number of steps in a single random walk. | walksPerNode | Integer | 10 | yes | The number of random walks generated for each node. | inOutFactor | Float | 1.0 | yes | Tendency of the random walk to stay close to the start node or fan out in the graph. Higher value means stay local. | returnFactor | Float | 1.0 | yes | Tendency of the random walk to return to the last visited node. A value below 1.0 means a higher tendency. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights to influence the probabilities of the random walks. The weights need to be >= 0. If unspecified, the algorithm runs unweighted. | windowSize | Integer | 10 | yes | Size of the context window when training the neural network. | negativeSamplingRate | Integer | 5 | yes | Number of negative samples to produce for each positive sample. | positiveSamplingFactor | Float | 0.001 | yes | Factor for influencing the distribution for positive samples. A higher value increases the probability that frequent nodes are down-sampled. | negativeSamplingExponent | Float | 0.75 | yes | Exponent applied to the node frequency to obtain the negative sampling distribution. A value of 1.0 samples proportionally to the frequency. A value of 0.0 samples each node equally. | embeddingDimension | Integer | 128 | yes | Size of the computed node embeddings. | iterations | Integer | 1 | yes | Number of training iterations. | initialLearningRate | Float | 0.01 | yes | Learning rate used initially for training the neural network. The learning rate decreases after each training iteration. | minLearningRate | Float | 0.0001 | yes | Lower bound for learning rate as it is decreased during training. | randomSeed | Integer | random | yes | Seed value for the random number generator used to generate the random walks. | walkBufferSize | Integer | 1000 | yes | The number of random walks to complete before starting training.

.Results [opts="header",cols="1m,1,6"]

| Name | Type | Description | nodeId | Integer | The Neo4j node ID. | embedding | List of Float | The computed node embedding.

======

[.include-with-mutate] ====== .Run Node2Vec in mutate mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.node2vec.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, nodeCount: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored

in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | walkLength | Integer | 80 | yes | The number of steps in a single random walk. | walksPerNode | Integer | 10 | yes | The number of random walks generated for each node. | inOutFactor | Float | 1.0 | yes | Tendency of the random walk to stay close to the start node or fan out in the graph. Higher value means stay local. | returnFactor | Float | 1.0 | yes | Tendency of the random walk to return to the last visited node. A value below 1.0 means a higher tendency. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights to influence the probabilities of the random walks. The weights need to be >= 0. If unspecified, the algorithm runs unweighted. | windowSize | Integer | 10 | yes | Size of the context window when training the neural network. | negativeSamplingRate | Integer | 5 | yes | Number of negative samples to produce for each positive sample. | positiveSamplingFactor | Float | 0.001 | yes | Factor for influencing the distribution for positive samples. A higher value increases the probability that frequent nodes are down-sampled. | negativeSamplingExponent | Float | 0.75 | yes | Exponent applied to the node frequency to obtain the negative sampling distribution. A value of 1.0 samples proportionally to the frequency. A value of 0.0 samples each node equally. | embeddingDimension | Integer | 128 | yes | Size of the computed node embeddings. | iterations | Integer | 1 | yes | Number of training iterations. | initialLearningRate | Float | 0.01 | yes | Learning rate used initially for training the neural network. The learning rate decreases after each training iteration. | minLearningRate | Float | 0.0001 | yes | Lower bound for learning rate as it is decreased during training. | randomSeed | Integer | random | yes | Seed value for the random number generator used to generate the random walks. | walkBufferSize | Integer | 1000 | yes | The number of random walks to complete before starting training.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes processed. | nodePropertiesWritten | Integer | The number of node properties written. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | postProcessingMillis | Integer | Milliseconds for post-processing of the results. | configuration | Map | The configuration used for running the algorithm.

======

[.include-with-write] ====== .Run Node2Vec in write mode on a graph stored in the catalog. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.node2vec.write(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, nodeCount: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | walkLength | Integer | 80 | yes | The number of steps in a single random walk. | walksPerNode | Integer | 10 | yes | The number of random walks generated for each node. | inOutFactor | Float | 1.0 | yes | Tendency of the random walk to stay close to the start node or fan out in the graph. Higher value means stay local. | returnFactor | Float | 1.0 | yes | Tendency of the random walk to return to the last visited node. A value below 1.0 means a higher tendency. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights to influence the probabilities of the random walks. The weights need to be >= 0. If unspecified, the algorithm runs unweighted. | windowSize | Integer | 10 | yes | Size of the context window when training the neural network. | negativeSamplingRate | Integer | 5 | yes | Number of negative samples to produce for each positive sample. | positiveSamplingFactor | Float | 0.001 | yes | Factor for influencing the distribution for positive samples. A higher value increases the probability that frequent nodes are down-sampled. | negativeSamplingExponent | Float | 0.75 | yes | Exponent applied to the node frequency to obtain the negative sampling distribution. A value of 1.0 samples proportionally to the frequency. A value of 0.0 samples each node equally. | embeddingDimension | Integer | 128 | yes | Size of the computed node embeddings. | iterations | Integer | 1 | yes | Number of training iterations. | initialLearningRate | Float | 0.01 | yes | Learning rate used initially for training the neural network. The learning rate decreases after each training iteration. | minLearningRate | Float | 0.0001 | yes | Lower bound for learning rate as it is decreased during training. | randomSeed | Integer | random | yes | Seed value for the random number generator used to generate the random walks. | walkBufferSize | Integer | 1000 | yes | The number of random walks to complete before starting training.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeCount | Integer | The number of nodes processed. | nodePropertiesWritten | Integer | The number of node properties written. | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | configuration | Map | The configuration used for running the algorithm.

====== ====

=== Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see Syntax overview.

.Run Node2Vec in write mode on an anonymous graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.node2vec.write(configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, nodeCount: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.General configuration for algorithm execution on an anonymous graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProjection | String, List of String or Map | null | yes | The node projection used for anonymous graph creation via a Native projection. | relationshipProjection | String, List of String or Map | null | yes | The relationship projection used for anonymous graph creation a Native projection. | nodeQuery | String | null | yes | The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection. | relationshipQuery | String | null | yes | The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection. | nodeProperties | String, List of String or Map | null | yes | The node properties to project during anonymous graph creation. | relationshipProperties | String, List of String or Map | null | yes | The relationship properties to project during anonymous graph creation. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. | readConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for creating the graph. | writeConcurrency | Integer | value of 'concurrency' | yes | WRITE mode only: The number of concurrent threads used for writing the result. | writeProperty | String | n/a | no | WRITE mode only: The {entity} property to which the {result} is written to.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | walkLength | Integer | 80 | yes | The number of steps in a single random walk. | walksPerNode | Integer | 10 | yes | The number of random walks generated for each node. | inOutFactor | Float | 1.0 | yes | Tendency of the random walk to stay close to the start node or fan out in the graph. Higher value means stay local. | returnFactor | Float | 1.0 | yes | Tendency of the random walk to return to the last visited node. A value below 1.0 means a higher tendency. | relationshipWeightProperty | String | null | yes | Name of the relationship property to use as weights to influence the probabilities of the random walks. The weights need to be >= 0. If unspecified, the algorithm runs unweighted. | windowSize | Integer | 10 | yes | Size of the context window when training the neural network. | negativeSamplingRate | Integer | 5 | yes | Number of negative samples to produce for each positive sample. | positiveSamplingFactor | Float | 0.001 | yes | Factor for influencing the distribution for positive samples. A higher value increases the probability that frequent nodes are down-sampled. | negativeSamplingExponent | Float | 0.75 | yes | Exponent applied to the node frequency to obtain the negative sampling distribution. A value of 1.0 samples proportionally to the frequency. A value of 0.0 samples each node equally. | embeddingDimension | Integer | 128 | yes | Size of the computed node embeddings. | iterations | Integer | 1 | yes | Number of training iterations. | initialLearningRate | Float | 0.01

| yes | Learning rate used initially for training the neural network. The learning rate decreases after each training iteration. | minLearningRate | Float | 0.0001 | yes | Lower bound for learning rate as it is decreased during training. | randomSeed | Integer | random | yes | Seed value for the random number generator used to generate the random walks. | walkBufferSize | Integer | 1000 | yes | The number of random walks to complete before starting training.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

== Examples

Consider the graph created by the following Cypher statement:

[source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:Person {name: 'Alice'}) CREATE (bob:Person {name: 'Bob'}) CREATE (carol:Person {name: 'Carol'}) CREATE (dave:Person {name: 'Dave'}) CREATE (eve:Person {name: 'Eve'}) CREATE (guitar:Instrument {name: 'Guitar'}) CREATE (synth:Instrument {name: 'Synthesizer'}) CREATE (bongos:Instrument {name: 'Bongos'}) CREATE (trumpet:Instrument {name: 'Trumpet'})

CREATE (alice)-[:LIKES]→(guitar) CREATE (alice)-[:LIKES]→(synth) CREATE (alice)-[:LIKES]→(bongos) CREATE (bob)-[:LIKES]→(guitar) CREATE (bob)-[:LIKES]→(synth) CREATE (carol)-[:LIKES]→(bongos) CREATE (dave)-[:LIKES]→(guitar) CREATE (dave)-[:LIKES]→(synth) CREATE (dave)-[:LIKES]→(bongos); ----

[source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', ['Person', 'Instrument'], 'LIKES'); ----

[role=query-example,no-result=true] — .Run the Node2Vec algorithm on `myGraph` [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.node2vec.stream('myGraph', {embeddingDimension: 2}) YIELD nodeId, embedding RETURN nodeId, embedding ----

.Results [opts="header"]

| nodeId | embedding | 0 | [-0.14295829832553864, 0.08884537220001221] | 1 | [0.016700705513358116, 0.2253911793231964] | 2 | [-0.06589698046445847, 0.042405471205711365] | 3 | [0.05862073227763176, 0.1193704605102539] | 4 | [0.10888434946537018, -0.18204474449157715] | 5 | [0.16728264093399048, 0.14098615944385529] | 6 | [-0.007779224775731564, 0.02114257402718067] | 7 | [-0.213893860578537, 0.06195802614092827] | 8 | [0.2479933649301529, -0.137322798371315]

— :leveloffset: 2

:leveloffset: +2

:description: This chapter provides explanations and examples for the supervised machine learning models in the Neo4j Graph Data Science library. = Machine learning models

[abstract] — This chapter provides explanations and examples for the supervised machine learning models in the Neo4j Graph Data Science library. — The machine learning procedures in Neo4j GDS allow you to train supervised machine learning models. Models can then be accessed via the Model catalog and used to make predictions about your graph.

To help with working with the ML models, there are additional guides for pre-processing and hyperparameter tuning available in:

* Pre-processing * Tuning parameters for training

The Neo4j GDS library includes the following machine learning models, grouped by quality tier:

* Alpha Node Classification Link Prediction Link Prediction Pipelines

:leveloffset: 2

:leveloffset: +3

= Pre-processing

In most machine learning scenarios, several pre-processing steps are applied to produce data that is amenable to machine learning algorithms. This is also true for graph data. The goal of pre-processing is to provide good features for the learning algorithm. In GDS some options include:

* Node embeddings * Centrality algorithms * Auxiliary algorithms Of special interest are Scale Properties and Split Relationships for Link Prediction.

:leveloffset: 2

:leveloffset: +3

= Tuning parameters

Both Node Classification and Link Prediction have training parameters that can be tuned automatically given a set of allowed values. The parameters `maxEpochs`, `tolerance` and `patience` control for how long the training will run until termination. These parameters give ways to limit a computational budget. In general, higher `maxEpochs` and `patience` and lower `tolerance` lead to longer training but higher quality models. It is however well-known that restricting the computational budget can serve the purpose of regularization and mitigate overfitting.

When faced with a heavy training task, a strategy to perform hyperparameter optimization faster, is to initially use lower values for the budget related parameters while exploring better ranges for other general or algorithm specific parameters.

More precisely, `maxEpochs` is the maximum number of epochs trained until termination. Whether the training exhausted the maximum number of epochs or converged prior is reported in the neo4j debug log.

As for `patience` and `tolerance`, the former is the maximum number of consecutive epochs that do not improve the training loss at least by a `tolerance` fraction of the current loss. After `patience` such unproductive epochs, the training is terminated. In our experience, reasonable values for `patience` are in the range `1` to `3`.

It is also possible, via `minEpochs`, to control a minimum number of epochs before the above termination criteria enter into play.

The training algorithm applied to the above algorithms is gradient descent. The gradient updates are computed batch-wise on batches of `batchSize` examples, and batches are computed concurrently on `concurrency` threads. Thus `batchSize`

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of the model to train, must not exist in the Model Catalog. | featureProperties | List of String | [] | yes | The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float. | nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | targetProperty | String | n/a | no | The class of the node. Must be of type Integer. | holdoutFraction | Float | n/a | no | Portion of the graph reserved for testing. Must be in the range (0, 1). | validationFolds | Integer | n/a | no | Number of divisions of the train graph used for model selection. | metrics | List of String | n/a | no | Metrics used to evaluate the models. | params | List of Map | n/a | no | List of model configurations to be trained. See next table for details. | randomSeed | Integer | n/a | yes | Seed for the random number generator used during training.

.Model configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | penalty | Float | n/a | no | Penalty used for the logistic regression. | batchSize | Integer | 100 | yes | Number of nodes per batch. | minEpochs | Integer | 1 | yes | Minimum number of training epochs. | maxEpochs | Integer | 100 | yes | Maximum number of training epochs. | patience | Integer | 1 | yes | Maximum number of iterations that do not improve the loss before stopping. | tolerance | Float | 0.001 | yes | Minimum acceptable loss before stopping. | concurrency | Integer | see description | yes | Concurrency for training the model candidate. By default the value of the top level `concurrency` parameter is used.

For hyperparameter tuning ideas, look here.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | trainMillis | Integer | Milliseconds used for training. | modelInfo | Map | Information about the training and the winning model. | configuration | Map | Configuration used for the train procedure.

The `modelInfo` can also be retrieved at a later time by using the Model List Procedure. The `modelInfo` return field has the following algorithm-specific subfields:

.Model info fields [opts="header",cols="1,1,6"]

| Name | Type | Description | classes | List of Integer | Sorted list of class ids which are the distinct values of `targetProperty` over the entire graph. | bestParameters | Map | The model parameters which performed best on average on validation folds according to the primary metric. | metrics | Map | Map from metric description to evaluated metrics for various models and subsets of the data, see below.

The structure of `modelInfo` is:

[listing] ---- { bestParameters: Map, // <1> classes: List of Integer, // <2> metrics: { // <3> <METRIC_NAME>: { // <4> test: Float, // <5> outerTrain: Float, // <6> train: [{ // <7> avg: Float, max: Float, min: Float, params: Map }, { avg: Float, max: Float, min: Float, params: Map }, ...], validation: [{ // <8> avg: Float, max: Float, min: Float, params: Map }, { avg: Float, max: Float, min: Float, params: Map }, ...] } } } ---- <1> The best scoring model candidate configuration. <2> Sorted list of class ids which are the distinct values of `targetProperty` over the entire graph. <3> The `metrics` map contains an entry for each metric description, and the corresponding results for that metric. <4> A metric name specified in the configuration of the procedure, e.g., `F1_MACRO` or `RECALL(class=4)`. <5> Numeric value for the evaluation of the best model on the test set. <6> Numeric value for the evaluation of the best model on the outer train set. <7> The `train` entry lists the scores over the `train` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`. <8> The `validation` entry lists the scores over the `validation` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.

======

[.include-with-stream] ====== .Run Node Classification in stream mode on a named graph: [source] ---- CALL gds.alpha.ml.nodeClassification.predict.stream(graphName: String, configuration: Map) YIELD nodeId: Integer, predictedClass: Integer, predictedProbabilities: List[Float] ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | includePredictedProbabilities | Boolean | false | yes | Whether to return the probability for each class. If `false` then `null` is returned in `predictedProbabilites`. | batchSize | Integer | 100 | yes | Number of nodes per batch.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeId | Integer | Node ID. | predictedClass | Integer | Predicted class for this node. | predictedProbabilities | List[Float] | Probabilities for all classes, for this node.

======

[.include-with-mutate] ====== .Run Node Classification in mutate mode on a named graph: [source] ---- CALL gds.alpha.ml.nodeClassification.predict.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | predictedProbabilityProperty | String | n/a | yes | The node property in which the class probability list is stored. If omitted, the probability list is discarded. | batchSize | Integer | 100 | yes | Number of nodes per batch.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | nodePropertiesWritten | Integer | Number of relationships created. | configuration | Map | Configuration used for running the algorithm.

======

[.include-with-write] ====== .Run Node Classification in write mode on a named graph: [source] ---- CALL gds.alpha.ml.nodeClassification.predict.write(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, writeMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'. | writeConcurrency | Integer | value of 'concurrency' | yes | The number of concurrent threads used for writing the result to Neo4j.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | predictedProbabilityProperty | String | n/a | yes | The node property in which the class probability list is stored. If omitted, the probability list is discarded. | batchSize |

Integer | 100 | yes | Number of nodes per batch.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | writeMillis | Integer | Milliseconds for writing result back to Neo4j. | nodePropertiesWritten | Integer | Number of relationships created. | configuration | Map | Configuration used for running the algorithm.

====== ====

== Examples

In this section we will show examples of training a Node Classification Model on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the model in a real setting. We will do this on a small graph of a handful of nodes representing houses. The example graph looks like this:

image::example-graphs/node_classification.svg[align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (:House {color: 'Gold', sizePerStory: [15.5, 23.6, 33.1], class: 0}), (:House {color: 'Red', sizePerStory: [15.5, 23.6, 100.0], class: 0}), (:House {color: 'Blue', sizePerStory: [11.3, 35.1, 22.0], class: 0}), (:House {color: 'Green', sizePerStory: [23.2, 55.1, 0.0], class: 1}), (:House {color: 'Gray', sizePerStory: [34.3, 24.0, 0.0], class: 1}), (:House {color: 'Black', sizePerStory: [71.66, 55.0, 0.0], class: 1}), (:House {color: 'White', sizePerStory: [11.1, 111.0, 0.0], class: 1}), (:House {color: 'Teal', sizePerStory: [80.8, 0.0, 0.0], class: 2}), (:House {color: 'Beige', sizePerStory: [106.2, 0.0, 0.0], class: 2}), (:House {color: 'Magenta', sizePerStory: [99.9, 0.0, 0.0], class: 2}), (:House {color: 'Purple', sizePerStory: [56.5, 0.0, 0.0], class: 2}), (:UnknownHouse {color: 'Pink', sizePerStory: [23.2, 55.1, 56.1]}), (:UnknownHouse {color: 'Tan', sizePerStory: [22.32, 102.0, 0.0]}), (:UnknownHouse {color: 'Yellow', sizePerStory: [39.0, 0.0, 0.0]}); ----

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `House` and `UnknownHouse` labels. We will also project the `sizeOfStory` property to use as a model feature, and the `class` property to use as a target feature.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', { House: { properties: ['sizePerStory', 'class'] }, UnknownHouse: { properties: 'sizePerStory' } }, '*') ----

In the following examples we will demonstrate using the Node Classification model on this graph.

=== Memory Estimation :mode: train

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `{mode}` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see Automatic estimation and execution blocking.

For more details on `estimate` in general, see Memory Estimation.

[role=query-example] — .The following will estimate the memory requirements for running the algorithm in write mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.nodeClassification.train.estimate('myGraph', { nodeLabels: ['House'], modelName: 'nc-model', featureProperties: ['sizePerStory'], targetProperty: 'class', randomSeed: 2, holdoutFraction: 0.2, validationFolds: 5, metrics: ['F1_WEIGHTED'], params: [{penalty: 0.0625}, {penalty: 0.5}, {penalty: 1.0}, {penalty: 4.0}] }) YIELD bytesMin, bytesMax, requiredMemory ----

.Results [opts="header"]

| bytesMin | bytesMax | requiredMemory | 66874368 | 66906328 | "[63 MiB ... 63 MiB]"

—

=== Train

In this example we will train a model to predict the class in which a house belongs, based on its `sizePerStory` property.

.Train a Node Classification model: [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.alpha.ml.nodeClassification.train('myGraph', { nodeLabels: ['House'], modelName: 'nc-model', featureProperties: ['sizePerStory'], targetProperty: 'class', randomSeed: 2, holdoutFraction: 0.2, validationFolds: 5, metrics: ['F1_WEIGHTED'], params: [{penalty: 0.0625}, {penalty: 0.5}, {penalty: 1.0}, {penalty: 4.0}] }) YIELD modelInfo RETURN {penalty: modelInfo.bestParameters.penalty} AS winningModel, modelInfo.metrics.F1_WEIGHTED.outerTrain AS trainGraphScore, modelInfo.metrics.F1_WEIGHTED.test AS testGraphScore ----

.Results [opts="header"]

| winningModel | trainGraphScore | testGraphScore | {penalty=0.0625} | 0.999999990909091 | 0.6363636286363638

Here we can observe that the model candidate with penalty `0.0625` performed the best in the training phase, with a score of almost 100% over the train graph. On the test graph, the model scores a bit lower at about 64%. This indicates that the model reacted very well to the train graph, and was able to generalize fairly well to unseen data. In order to achieve a higher test score, we may need to use better features, a larger graph, or different model configuration.

=== Stream

In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

In this example we will show how to use a trained model to predict the class of a node in your in-memory graph. In addition to the predicted class, we will also produce the probability for each class in another node property. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the train example which we gave the name `'nc-model'`.

[role=query-example, no-result=true, group=stream] — [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.nodeClassification.predict.stream('myGraph', { nodeLabels: ['House', 'UnknownHouse'], modelName: 'nc-model', includePredictedProbabilities: true }) YIELD nodeId, predictedClass, predictedProbabilities WITH gds.util.asNode(nodeId) AS houseNode, predictedClass, predictedProbabilities WHERE houseNode:UnknownHouse RETURN houseNode.color AS classifiedHouse, predictedClass, floor(predictedProbabilities[predictedClass] * 100) AS confidence ORDER BY classifiedHouse ----

.Results [opts="header",cols="m,m,m"]

| classifiedHouse | predictedClass | confidence | "Pink" | 0 | 98.0 | "Tan" | 1 | 98.0 | "Yellow" | 2 | 79.0

— As we can see, the model was able to predict the pink house into class 0, tan house into class 1, and yellow house into class 2. This makes sense, as all houses in class 0 had three stories, class 1 two stories and class 2 one story, and the same is true of the pink, tan and yellow houses, respectively. Additionally, we see that the model is confident in these predictions, as the highest class probability is >=80% in all cases.

=== Mutate

The `mutate` execution mode updates the named graph with a new node property containing the predicted class for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row including information about timings and how many properties were written. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

In this example we will show how to use a trained model to predict the class of a node in your in-memory graph. In addition to the predicted class, we will also produce the probability for each class in another node property. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the train example which we gave the name `'nc-model'`.

[role=query-example, group=mutate] — [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.nodeClassification.predict.mutate('myGraph', { nodeLabels: ['House', 'UnknownHouse'], modelName: 'nc-model', mutateProperty: 'predictedClass', predictedProbabilityProperty: 'predictedProbabilities' }) YIELD nodePropertiesWritten ----

.Results [opts="header"]

| nodePropertiesWritten | 28

— Since we specified also the `predictedProbabilityProperty` we are writing two properties for each of the 14 nodes. In order to analyse our predicted classes we stream the properties from the in-memory graph:

[role=query-example, no-result=true, group=mutate] — [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamNodeProperties('myGraph', ['predictedProbabilities', 'predictedClass'], ['UnknownHouse']) YIELD nodeId, nodeProperty, propertyValue RETURN gds.util.asNode(nodeId).color AS classifiedHouse, nodeProperty, propertyValue ORDER BY classifiedHouse, nodeProperty ----

.Results [opts="header"]

| classifiedHouse | nodeProperty | propertyValue | "Pink" | "predictedClass" | 0 | "Pink" | "predictedProbabilities" | [0.9866455686217779, 0.01311656378786989, 2.3786759035214687E-4] | "Tan" | "predictedClass" | 1 | "Tan" | "predictedProbabilities" | [0.01749164563726576, 0.9824922482993587, 1.610606337562594E-5] | "Yellow" | "predictedClass" | 2 | "Yellow" | "predictedProbabilities" | [0.0385634113659007, 0.16350471177895198, 0.7979318768551473]

— As we can see, the model was able to predict the pink house into class 0, tan house into class 1, and yellow house into class 2. This makes sense, as all houses in class 0 had three stories, class 1 two stories and class 2 one story, and the same is true of the pink, tan and yellow houses, respectively. Additionally, we see that the model is confident in these predictions, as the highest class probability is >75% in all cases.

=== Write

The `write` execution mode writes the predicted property for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row including information about timings and how many properties were written. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see Write.

In this example we will show how to use a trained model to predict the class of a node in your in-memory graph. In addition to the predicted class, we will also produce the probability for each class in another node property. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the train example which we gave the name `'nc-model'`.

[role=query-example, group=write] — [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.nodeClassification.predict.write('myGraph', { nodeLabels: ['House', 'UnknownHouse'], modelName: 'nc-model', writeProperty: 'predictedClass', predictedProbabilityProperty: 'predictedProbabilities' }) YIELD nodePropertiesWritten ----

.Results [opts="header"]

| nodePropertiesWritten | 28

— Since we specified also the `predictedProbabilityProperty` we are writing two properties for each of the 14 nodes. In order to analyse our predicted classes we stream the properties from the in-memory graph:

[role=query-example, no-result=true, group=write] — [source, cypher, role=noplay, indent=0] ---- MATCH (house:UnknownHouse) RETURN house.color AS classifiedHouse, house.predictedClass AS predictedClass, house.predictedProbabilities AS predictedProbabilities ----

.Results [opts="header"]

| classifiedHouse | predictedClass | predictedProbabilities | "Pink" | 0 | [0.9866455686217779, 0.01311656378786989, 2.3786759035214687E-4] | "Tan" | 1 | [0.01749164563726576, 0.9824922482993587, 1.610606337562594E-5] | "Yellow" | 2 | [0.0385634113659007, 0.16350471177895198, 0.7979318768551473]

—

As we can see, the model was able to predict the pink house into class 0, tan house into class 1, and yellow house into class 2. This makes sense, as all houses in class 0 had three stories, class 1 two stories and class 2 one story, and the same is true of the pink, tan and yellow houses, respectively. Additionally, we see that the model is confident in these predictions, as the highest class probability is >75% in all cases.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Link Prediction Model in the Neo4j Graph Data Science library. = Link Prediction :entity: relationship :result: relationships :algorithm: Link Prediction :modelType: Link Prediction

[abstract] — This section describes the Link Prediction Model in the Neo4j Graph Data Science library. —

== Introduction

Link prediction is a common machine learning task applied to graphs: training a model to learn, between pairs of nodes in a graph, where relationships should exist. The predicted links are undirected. You can think of this as building a model to predict missing relationships in your dataset or relationships that are likely to form in the future. Neo4j GDS trains supervised machine learning models based on the relationships and node properties in your graph to predict the existence - and probability - of relationships.

Link Prediction can be used favorably together with pre-processing algorithms.

The basic work flow of Link Prediction contains the following parts which are described below:

* Creating training and test graphs * Training and Evaluating model candidates * Applying a model for prediction

=== Training, Model Selection and Evaluation

When building a model, it is possible to specify multiple model configurations and a model selection metric. The train mode, `gds.alpha.ml.linkPrediction.train`, is responsible for training and evaluating the models, selecting the best model, and storing it in the model catalog.

The train mode takes as input two relationship types representing the training graph and test graph respectively. The relationship types must have an integer property, with values being either `0` or `1`. If the value is `0` the relationship represents a negative example, meaning a node pair which is not connected in the original graph. If the value is `1` the

R]-(b) WHERE a < b RETURN count(rel) AS r ----

When choosing a value for `negativeClassWeight`, two factors should be considered. First, the desired ratio of total probability mass of negative versus positive examples in the test set. Second, what the ratio of sampled negative examples to positive examples was in the test set. To be consistent with *traditional* evaluation, one should choose parameters so that `negativeSamplingRatio * negativeClassWeight = 1.0`, for example by setting the values to the true class ratio and its reciprocal, or both values to `1.0`.

Alternatively, one can aim for the ratio of total probability weight between the classes to be close to the true class ratio. That is, making sure `negativeSamplingRatio * negativeClassWeight` is close to the true class ratio. The reported metric (AUCPR) then better reflects the expected precision on unseen highly imbalanced data. With this type of evaluation one has to adjust expectations as the metric value then becomes much smaller.

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

WARNING: The named graphs must be projected in the `UNDIRECTED` orientation for the Link Prediction model.

.Link Prediction syntax per mode [.tabbed-example, caption =] ====

[.include-with-train] ====== .Run Link Prediction in train mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.linkPrediction.train(graphName: String, configuration: Map) YIELD trainMillis: Integer, modelInfo: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of the model to train, must not exist in the Model Catalog. | featureProperties | List of String | [] | yes | The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float. | nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | trainRelationshipType | String | n/a | no | Relationship type to use during model training. | testRelationshipType | String | n/a | no | Relationship type to use during model evaluation. | validationFolds | Integer | n/a | no | Number of divisions of the training graph used during model selection. | negativeClassWeight | Float | n/a | no | Weight of negative examples in model evaluation. Positive examples have weight 1. | params | List of Map | n/a | no | List of model configurations to be trained and compared. See next table for details. | randomSeed | Integer | n/a | yes | Seed for the random number generator used during training.

.Model configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | penalty | Float | 0.0 | yes | Penalty used for the logistic regression. By default, no penalty is applied. | linkFeatureCombiner | String | "L2" | yes | Link feature combiner is used to combine two node feature vectors into the feature vector for the training. Available combiners are `L2`, `HADAMARD` and `COSINE`. | batchSize | Integer | 100 | yes | Number of nodes per batch. | minEpochs | Integer | 1 | yes | Minimum number of training epochs. | maxEpochs | Integer | 100 | yes | Maximum number of training epochs. | patience | Integer | 1 | yes | Maximum number of unproductive consecutive epochs. | tolerance | Float | 0.001 | yes | The minimal improvement of the loss to be considered productive. | concurrency | Integer | see description | yes | Concurrency for training the model candidate. By default, the value of the top level `concurrency` parameter is used.

For hyperparameter tuning ideas, look here.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | trainMillis | Integer | Milliseconds used for training. | modelInfo | Map | Information about the training and the winning model. | configuration | Map | Configuration used for the train procedure.

The `modelInfo` can also be retrieved at a later time by using the Model List Procedure. The `modelInfo` return field has the following algorithm-specific subfields:

.Model info fields [opts="header",cols="1,1,6"]

| Name | Type | Description | bestParameters | Map | The model parameters which performed best on

average on validation folds according to the primary metric. | metrics | Map | Map from metric description to evaluated metrics for various models and subsets of the data, see below.

The structure of `modelInfo` is:

[listing] ---- { bestParameters: Map, // <1> metrics: { // <2> AUCPR: { test: Float, // <3> outerTrain: Float, // <4> train: [{ // <5> avg: Float, max: Float, min: Float, params: Map }, { avg: Float, max: Float, min: Float, params: Map }, ...], validation: [{ // <6> avg: Float, max: Float, min: Float, params: Map }, { avg: Float, max: Float, min: Float, params: Map }, ...] } } } ---- <1> The best scoring model candidate configuration. <2> The `metrics` map contains an entry for each metric description (currently only `AUCPR`) and the corresponding results for that metric. <3> Numeric value for the evaluation of the best model on the test set. <4> Numeric value for the evaluation of the best model on the outer train set. <5> The `train` entry lists the scores over the `train` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`. <6> The `validation` entry lists the scores over the `validation` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.

======

[.include-with-stream] ====== .Run Link Prediction in stream mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.linkPrediction.predict.stream(graphName: String, configuration: Map) YIELD node1: Integer, node2: Integer, probability: Float ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | topN | Integer | n/a | no | Limit on predicted relationships to output. | threshold | Float | n/a | no | Minimum predicted probability on relationships to output.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | node1 | Integer | Node ID of the first node. | node2 | Integer | Node ID of the second node. | probability | Float | Predicted probability of a link between the nodes.

======

[.include-with-mutate] ====== .Run Link Prediction in mutate mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.linkPrediction.predict.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, relationshipsWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of a Link Prediction model in the model catalog. | nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. | mutateRelationshipType | String | n/a | no | The relationship type used for the new relationships written to the in-memory graph. | mutateProperty | String | 'probability' | yes | The relationship property in the GDS graph to which the result is written.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | topN | Integer | n/a | no | Limit on predicted relationships to output. | threshold | Float | n/a | no | Minimum predicted probability on relationships to output.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | relationshipsWritten | Integer | Number of relationships created. | configuration | Map | Configuration used for running the algorithm.

======

[.include-with-write] ====== .Run Link Prediction in write mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.linkPrediction.predict.write(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, writeMillis: Integer, relationshipsWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of a Link Prediction model in the model catalog. | nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. | writeRelationshipType | String | n/a | no | The relationship type used to persist the computed relationships in the Neo4j database. | writeProperty | String | n/a | no | The relationship property in the Neo4j database to which the result is written.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | topN | Integer | n/a | no | Limit on predicted relationships to output. | threshold | Float | n/a | no | Minimum predicted probability on relationships to output.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | writeMillis | Integer | Milliseconds for writing result data back to Neo4j. | relationshipsWritten | Integer | Number of relationships created. | configuration | Map | Configuration used for running the algorithm.

====== ====

== Examples

:algorithm-name: {algorithm} :graph-description: social network :image-file: link-prediction.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (alice:Person {name: 'Alice', numberOfPosts: 38}), (michael:Person {name: 'Michael', numberOfPosts: 67}), (karin:Person {name: 'Karin', numberOfPosts: 30}), (chris:Person {name: 'Chris', numberOfPosts: 132}), (will:Person {name: 'Will', numberOfPosts: 6}), (mark:Person {name: 'Mark', numberOfPosts: 32}), (greg:Person {name: 'Greg', numberOfPosts: 29}), (veselin:Person {name: 'Veselin', numberOfPosts: 3}),

(alice)-[:KNOWS]→(michael), (michael)-[:KNOWS]→(karin), (michael)-[:KNOWS]→(chris), (michael)-[:KNOWS]→(greg), (will)-[:KNOWS]→(michael), (will)-[:KNOWS]→(chris), (mark)-[:KNOWS]→(michael), (mark)-[:KNOWS]→(will), (greg)-[:KNOWS]→(chris), (veselin)-[:KNOWS]→(chris), (karin)-[:KNOWS]→(veselin), (chris)-[:KNOWS]→(karin); ----

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. We will also project the `numberOfPosts` property, so we can use it as a model feature. For the relationships we must use the `UNDIRECTED` orientation. This is because the Link Prediction model is defined only for undirected graphs.

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', { Person: { properties: ['numberOfPosts'] } }, { KNOWS: { orientation: 'UNDIRECTED' } }) ----

WARNING: The Link Prediction model requires the graph to be created using the `UNDIRECTED` orientation for relationships.

In the following examples we will demonstrate using the Link Prediction model on this graph.

=== Train

First, we must do the test/train splits. For this we will make use of the `gds.alpha.ml.splitRelationships` procedure. We will do one split to generate the test graph. We note that in the example graph there are eight nodes and twelve directed relationships. Recall that we compute the class ratio as `(q - r) / q`, where we then have `q = 8(8-1)/2` and `r = 12` which gives us class ratio of `(28 - 12) / 12 ~= 1.33`. We use this to configure `negativeSampleRatio` to achieve a sampling proportional to the class ratio.

[role=query-example, group=lp] — [source, cypher, indent=0] ---- CALL gds.alpha.ml.splitRelationships.mutate('myGraph', { relationshipTypes: ['KNOWS'], remainingRelationshipType: 'KNOWS_REMAINING', holdoutRelationshipType: 'KNOWS_TESTGRAPH', holdoutFraction: 0.2, negativeSamplingRatio: 1.33, randomSeed: 1984 }) YIELD relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 25

— We will create copied relationships for each existing relationship, into either the `KNOWS_REMAINING` or the `KNOWS_TESTGRAPH` relationship types. All relationships in `KNOWS_TESTGRAPH` will have a `label` property. Additionally, a number of non-existing relationships will be created into the `KNOWS_TESTGRAPH` relationship type to be used as negative examples, with a `label` of `0`.

Next, we will create the train graph.

[role=query-example, group=lp] — [source, cypher, indent=0] ---- CALL gds.alpha.ml.splitRelationships.mutate('myGraph', { relationshipTypes: ['KNOWS_REMAINING'], remainingRelationshipType: 'KNOWS_IGNORED_FOR_TRAINING', holdoutRelationshipType: 'KNOWS_TRAINGRAPH', holdoutFraction: 0.2, negativeSamplingRatio: 1.33, randomSeed: 1984 }) YIELD relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 20

— With both training and test graphs, we are ready to train models. We will use 5 validation folds, meaning we will split the train graph into 5 pairs, using one part of each pair for training and one for validation. Since we set the `negativeSamplingRatio` to 1.33 (the class ratio of the graph) above, we'll set the `negativeClassWeight` during training to `1 / 1.33` to assign equal weight to both classes.

[role=query-example, group=lp] — .Train a Link Prediction model: [source, cypher, indent=0] ---- CALL gds.alpha.ml.linkPrediction.train('myGraph', { trainRelationshipType: 'KNOWS_TRAINGRAPH', testRelationshipType: 'KNOWS_TESTGRAPH', modelName: 'lp-numberOfPosts-model', featureProperties: ['numberOfPosts'], validationFolds: 5, negativeClassWeight: 1.0 / 1.33, randomSeed: 2, concurrency: 1, params: [{penalty: 0.5, maxEpochs: 1000}, {penalty: 1.0, maxEpochs: 1000}, {penalty: 0.0, maxEpochs: 1000}] }) YIELD modelInfo RETURN { maxEpochs: modelInfo.bestParameters.maxEpochs, penalty: modelInfo.bestParameters.penalty } AS winningModel, modelInfo.metrics.AUCPR.outerTrain AS trainGraphScore, modelInfo.metrics.AUCPR.test AS testGraphScore ----

.Results [opts="header"]

| winningModel | trainGraphScore | testGraphScore | {maxEpochs=1000, penalty=0.5} | 0.38525757517173825 | 0.46710171439292664

— Here we can observe that the model candidate with penalty 0.5 performed the best in the training phase, with a score of about 71% over the train graph. On the test graph, the model scored much lower at about 35%. This indicates that the model reacted fairly well to the train graph, but did not generalise very well to unseen data. In order to achieve a higher test score, we may need to use better features, a larger graph, or different model configuration.

=== Stream

In the `stream` execution mode, the algorithm returns the top predicted relationships. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example, group=lp] — [source, cypher, indent=0] ---- CALL gds.alpha.ml.linkPrediction.predict.stream('myGraph', { relationshipTypes: ['KNOWS'], modelName: 'lp-numberOfPosts-model', topN: 5, threshold: 0.45 }) YIELD node1, node2, probability MATCH (n), (m) WHERE id(n) = node1 AND id(m) = node2 RETURN n.name AS name1, m.name AS name2, probability ----

.Results [opts="header"]

| name1 | name2 | probability | "Karin" | "Greg" | 0.4991363247445545 | "Karin" | "Mark" | 0.49896977670628373 | "Mark" | "Greg" | 0.49869219716877955 | "Will" | "Veselin" |

0.49869219716877955 | "Alice" | "Mark" | 0.49719328593255546

— We specified `threshold` to filter out predictions with probability less than 45%, and `topN` to further limit output to the top 5 relationships. Note that the predicted link between the Karin and Greg nodes does not reflect any particular direction between them.

=== Mutate

In this example we will show how to use a trained model to predict new relationships in your in-memory graph. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the train example which we gave the name `'lp-numberOfPosts-model'`.

We must also make sure that we do not include any of the relationships from the train or test graphs, which we do by specifying a relationship filter for the original relationship type `'KNOWS'`.

[role=query-example, group=lp] — [source, cypher, indent=0] ---- CALL gds.alpha.ml.linkPrediction.predict.mutate('myGraph', { relationshipTypes: ['KNOWS'], modelName: 'lp-numberOfPosts-model', mutateRelationshipType: 'KNOWS_PREDICTED', topN: 5, threshold: 0.45 }) YIELD relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 10

— We specified `threshold` to filter out predictions with probability less than 45%, and `topN` to further limit output to the top 5 relationships. Because we are using the `UNDIRECTED` orientation, we will write twice as many relationships to the in-memory graph.

=== Write

In this example we will show how to use a trained model to predict new relationships in your in-memory graph, and write the predictions back to Neo4j. We will again use the model `'lp-numberOfPosts-model'`, as in the mutate example.

[role=query-example, group=lp] — [source, cypher, indent=0] ---- CALL gds.alpha.ml.linkPrediction.predict.write('myGraph', { relationshipTypes: ['KNOWS'], modelName: 'lp-numberOfPosts-model', writeRelationshipType: 'KNOWS_PREDICTED', topN: 5, threshold: 0.45 }) YIELD relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 10

— The end result looks like this:

image::example-graphs/link-prediction-mutate.svg[align="center"]

In yellow we highlight the predicted relationships.

:leveloffset: 2

:leveloffset: +3

:description: This section describes Link Prediction Pipelines in the Neo4j Graph Data Science library. = Link Prediction Pipelines :entity: relationship :result: relationships :modelType: Link prediction pipeline

[abstract] — This section describes Link Prediction Pipelines in the Neo4j Graph Data Science library. —

== Introduction

Link prediction is a common machine learning task applied to graphs: training a model to learn, between pairs of nodes in a graph, where relationships should exist. More precisely, the input of the machine learning model are *examples* of node pairs which are labeled as connected or not connected. The GDS library provides Link prediction, see here. Here we describe an additional method that provides an end-to-end Link prediction experience. In addition to managing a predictive model, it also manages:

* splitting relationships into subsets for `test`, `train` and `feature input` * a pipeline of processing steps that supply custom features for the model

The motivation for using pipelines are:

* easier to get splits right and prevent data leakage * ensuring that the same feature creation steps are applied at predict and train time * applying the trained model with a single procedure call * persisting the pipeline as a whole

The rest of this page is divided as follows:

* Creating a pipeline * Adding node properties * Adding link features * Configuring the relationship splits * Configuring the model parameters * Training the pipeline * Applying a trained model for prediction

== Creating a pipeline

The first step of building a new pipeline is to create one using `gds.alpha.ml.pipeline.linkPrediction.create`. This stores a trainable model object in the model catalog of type `Link prediction training pipeline`. This represents a configurable pipeline that can later be invoked for training, which in turn creates a trained pipeline. The latter is also a model which is stored in the catalog with type `Link prediction pipeline`.

=== Syntax

[.pipeline-create-syntax] — .Create pipeline syntax [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.create(pipelineName: String) YIELD name: String, nodePropertySteps: List of Map, featureSteps: List of Map, splitConfig: Map, parameterSpace: List of Map ----

.Parameters [opts="header",cols="1,1,4"]

| Name | Type | Description | pipelineName | String | The name of the created pipeline.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | name | String | Name of the pipeline. | nodePropertySteps | List of Map | List of configurations for node property steps. | featureSteps | List of Map | List of configurations for feature

steps. | splitConfig | Map | Configuration to define the split before the model training. | parameterSpace | List of Map | List of parameter configurations for models which the train mode uses for model selection.

— === Example

[role=query-example,group=lp] — .The following will create a pipeline: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.create('pipe') ----

.Results [opts="header",cols="1,1,1,1,1"]

| name | nodePropertySteps | featureSteps | splitConfig | parameterSpace | "pipe" | [] | [] | {negativeSamplingRatio=1.0, testFraction=0.1, validationFolds=3, trainFraction=0.1} | [{useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001, concurrency=4}]

— This show that the newly created pipeline does not contain any steps yet, and has defaults for the split and train parameters.

== Adding node properties

A link prediction pipeline can execute one or several GDS algorithms in mutate mode that create node properties in the in-memory graph. Such steps producing node properties can be chained one after another and created properties can also be used to add features. Moreover, the node property steps that are added to the pipeline will be executed both when training a model and when the trained model is applied for prediction.

The name of the procedure that should be added can be a fully qualified GDS procedure name ending with `.mutate`. The ending `.mutate` may be omitted and one may also use shorthand forms such as `node2vec` instead of `gds.beta.node2vec.mutate`.

For example, pre-processing algorithms can be used as node property steps.

=== Syntax

[.pipeline-add-node-property-syntax] — .Add node property syntax [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.addNodeProperty(pipelineName: String, procedureName: String, procedureConfiguration: Map) YIELD name: String, nodePropertySteps: List of Map, featureSteps: List of Map, splitConfig: Map, parameterSpace: List of Map ----

.Parameters [opts="header",cols="1,1,4"]

| Name | Type | Description | pipelineName | String | The name of the pipeline. | procedureName | String | The name of the procedure to be added to the pipeline. | procedureConfiguration | Map | The configuration of the procedure, excluding `graphName`, `nodeLabels` and `relationshipTypes`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | name | String | Name of the pipeline. | nodePropertySteps | List of Map | List of configurations for node property steps. | featureSteps | List of Map | List of configurations for feature steps. | splitConfig | Map | Configuration to define the split before the model training. | parameterSpace | List of Map | List of parameter configurations for models which the train mode uses for model selection.

— === Example

[role=query-example,group=lp] — .The following will add a node property step to the pipeline: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.addNodeProperty('pipe', 'fastRP', { mutateProperty: 'embedding', embeddingDimension: 256, randomSeed: 42 }) ----

.Results [opts="header",cols="1,1,1,1,1"]

| name | nodePropertySteps | featureSteps | splitConfig | parameterSpace | "pipe" | [{name=gds.fastRP.mutate, config={randomSeed=42, embeddingDimension=256, mutateProperty=embedding}}] | [] | {negativeSamplingRatio=1.0, testFraction=0.1, validationFolds=3, trainFraction=0.1} | [{useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001, concurrency=4}]

The pipeline will now execute the fastRP algorithm in mutate mode both before training a model, and when the trained model is applied for prediction. This ensures the `embedding` property can be used as an input for link features. —

== Adding link features

A Link Prediction pipeline executes a sequence of steps to compute the features used by a machine learning model. A feature step computes a vector of features for given node pairs. For each node pair, the results are concatenated into a single *link feature vector*. The order of the features in the link feature vector follows the order of the feature steps. Like with node property steps, the feature steps are also executed both at training and prediction time. The supported methods for obtaining features are described below.

=== Syntax

[.pipeline-add-feature-syntax] — .Adding a link feature to a pipeline syntax [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.addFeature(pipelineName: String, featureType: String, configuration: Map) YIELD name: String, nodePropertySteps: List of Map, featureSteps: List of Map, splitConfig: Map, parameterSpace: List of Map ----

.Parameters [opts="header",cols="1,1,4"]

| Name | Type | Description | pipelineName | String | The name of the pipeline. | featureType | String | The featureType determines the method used for computing the link feature. See supported types. | configuration | Map | Configuration for splitting the relationships.

.Configuration [opts="header",cols="1,1,1,4"]

| Name | Type | Default | Description | nodeProperties | List of String | no | The names of the node properties that should be used as input.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | name | String | Name of the pipeline. | nodePropertySteps | List of Map | List of configurations for node property steps. | featureSteps | List of Map | List of configurations for feature steps. | splitConfig | Map | Configuration to define the split before the model training. | parameterSpace | List of Map | List of parameter configurations for models which the train mode uses for model selection.

— === Supported feature types

A feature step can use node properties that exist in the input graph or are added by the pipeline. For each node in a node pair of interest, the values of `nodeProperties` are concatenated, in the configured order, into a vector. We denote the entries of the vectors of a pair by `a[i]` and `b[i]`, and we take `f[i]` to be the `i`-th entry of the output of a feature step.

The supported types of features can then be described as follows:

.Supported feature types [opts="header",cols="1,4"]

| Feature Type | Formula / Description | L2 | f[i] = (a[i] - b[i])^2 | HADAMARD | f[i] = a[i] * b[i] | COSINE | f[0] = cosine similarity of vectors a and b

=== Example

[role=query-example,group=lp] — .The following will add a feature step to the pipeline: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.addFeature('pipe', 'hadamard', { nodeProperties: ['embedding', 'numberOfPosts'] }) YIELD featureSteps ----

.Results [opts="header",cols="1"]

| featureSteps | [{name=HADAMARD, config={nodeProperties=[embedding, numberOfPosts]}}]

When executing the pipeline, the `nodeProperties` must be either present in the input graph, or created by a previous node property step. For example, the `embedding` property could be created by the previous example, and we expect `numberOfPosts` to already be present in the in-memory graph used as input, at train and predict time. — == Configuring the relationship splits

Link Prediction pipelines manage splitting the relationships into several sets and add sampled negative relationships to some of these sets. Configuring the splitting is optional, and if omitted, splitting will be done using default settings.

The splitting configuration of a pipeline can be inspected by using `gds.beta.model.list` and possibly only yielding `splitConfig`.

The splitting of relationships proceeds internally in the following steps:

1. The graph is filtered according to specified `nodeLabels` and `relationshipTypes`, which are configured at train time. 2. The relationships remaining after filtering we call *positive*, and they are split into a `test` set and remaining relationships. * The `test` set contains a `testFraction` fraction of the positive relationships. * Random negative relationships are added to the `test` set. The number of negative relationships is the number of positive ones multiplied by the `negativeSamplingRatio`. * The negative relationships do not coincide with positive relationships. 3. The remaining positive relationships are split into a `train` set and a `feature input` set. * The `train` set contains a `trainFraction` fraction of *all* the positive relationships. **Therefore we require `trainFraction + testFraction < 1.0`.** The `feature input` set contains the remaining `1.0 - (trainFraction + testFraction)` fraction of the positive relationships. * Random negative relationships are added to the `train` set. The number of negative relationships is the number of positive ones multiplied by the `negativeSamplingRatio`. * The negative relationships do not coincide with positive relationships, nor with test relationships.

The sampled positive and negative relationships are given relationship weights of `1.0` and `0.0` respectively so that they can be distinguished.

The `feature input` graph is used, both in training and testing, for computing node properties and therefore also features which depend on node properties.

The `train` and `test` relationship sets are used for:

* determining the label (positive or negative) for each training or test example * identifying the node pair for which link features are to be computed

However, they are not used by the algorithms run in the node property steps. The reason for this is that otherwise the model would use the prediction target (existence of a relationship) as a feature.

=== Syntax

[.pipeline-configure-split-syntax] — .Configure the relationship split syntax [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.configureSplit(pipelineName: String, configuration: Map) YIELD name: String, nodePropertySteps: List of Map, featureSteps: List of Map, splitConfig: Map, parameterSpace: List of Map ----

.Parameters [opts="header",cols="1,1,4"]

| Name | Type | Description | pipelineName | String | The name of the pipeline. | configuration | Map | Configuration for splitting the relationships.

.Configuration [opts="header",cols="1,1,1,4"]

| Name | Type | Default | Description | validationFolds | Integer | 3 | Number of divisions of the training graph used during model selection. | testFraction | Double | 0.1 | Portion of the graph reserved for testing. Must be in the range (0, 1). | trainFraction | Double | 0.1 | Portion of the graph reserved for training. Must be in the range (0, 1). | negativeSamplingRatio | Double | 1.0 | The desired ratio of negative to positive samples in the test and train set.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | name | String | Name of the pipeline. | nodePropertySteps | List of Map | List of configurations for node property steps. | featureSteps | List of Map | List of configurations for feature steps. | splitConfig | Map | Configuration to define the split before the model training. | parameterSpace | List of Map | List of parameter configurations for models which the train mode uses for model selection.

— === Example

[role=query-example,group=lp] — .The following will configure the splitting of the pipeline: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.configureSplit('pipe', { testFraction: 0.3, trainFraction: 0.3, validationFolds: 7 }) YIELD splitConfig ----

.Results [opts="header",cols="1"]

| splitConfig | {negativeSamplingRatio=1.0, testFraction=0.3, validationFolds=7, trainFraction=0.3}

We now reconfigured the splitting of the pipeline, which will be applied during training. — == Configuring the model parameters

The `gds.alpha.ml.pipeline.linkPrediction.configureParams` mode is used to set up the train mode with a list of configurations of logistic regression models. The set of model configurations is called the *parameter space* which parametrizes a set of model candidates. The parameter space can be configured by passing this procedure a list of maps, where each map configures the training of one logistic regression model. In Training the pipeline, we explain further how the configured model candidates are trained, evaluated and compared.

The allowed model parameters are listed in the table Model configuration.

If `configureParams` is not used, then a single model with defaults for all the model parameters is used. The parameter space of a pipeline can be inspected using `gds.beta.model.list` and optionally yielding only `parameterSpace`.

=== Syntax

[.pipeline-configure-params-syntax] — .Configure the train parameters syntax [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.configureParams(pipelineName: String, parameterSpace: List of Map) YIELD name: String, nodePropertySteps: List of Map, featureSteps: List of Map, splitConfig: Map, parameterSpace: List of Map ----

.Parameters [opts="header",cols="1,1,4"]

| Name | Type | Description | pipelineName | String | The name of the pipeline. | parameterSpace | List of Map | The parameter space used to select the best model from. Each Map corresponds to potential model. The allowed parameters for a model are defined in the next table.

.Model configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | penalty | Float | 0.0 | yes | Penalty used for the logistic regression. By default, no penalty is applied. | batchSize | Integer | 100 | yes | Number of nodes per batch. | minEpochs | Integer | 1 | yes | Minimum number of training epochs. | maxEpochs | Integer | 100 | yes | Maximum number of training epochs. | patience | Integer | 1 | yes | Maximum number of unproductive consecutive epochs. | tolerance | Float | 0.001 | yes | The minimal improvement of the loss to be considered productive. | useBiasFeature | Boolean | true | yes | Whether the logistic regression model uses a bias feature. | concurrency | Integer | see description | yes | Concurrency for training the model candidate. By default, the value of `concurrency` defined at training is used.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | name | String | Name of the pipeline. | nodePropertySteps | List of Map | List of configurations for node property steps. | featureSteps | List of Map | List of configurations for feature steps. | splitConfig | Map | Configuration to define the split before the model training. | parameterSpace | List of Map | List of parameter configurations for models which the train mode uses for model selection.

— === Example

[role=query-example,group=lp] — .The following will configure the parameter space of the pipeline: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.configureParams('pipe', [{tolerance: 0.001}, {tolerance: 0.01}, {maxEpochs: 500}]) YIELD parameterSpace ----

.Results [opts="header",cols="1"]

| parameterSpace | [{useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001}, {useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.01}, {useBiasFeature=true, maxEpochs=500, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001}]

The `parameterSpace` in the pipeline now contains the three different model parameters, expanded with the default values. Each specified model configuration will be tried out during the model selection in training. — == Training the pipeline

The train mode, `gds.alpha.ml.pipeline.linkPrediction.train`, is responsible for splitting data, feature extraction, model selection, training and storing a model for future use. Running this mode results in a `Link prediction pipeline` model being stored in the model catalog along with metrics collected during training. The trained pipeline can be applied to a possibly different graph which produces a relationship type of predicted links, each having a predicted probability stored as a property.

More precisely, the procedure will in order:

. apply `nodeLabels` and `relationshipType` filters to the graph. All subsequent graphs have the same node set. . create a relationship split of the graph into `test`, `train` and `feature input` sets as described in Configuring the relationship splits. These graphs are internally managed and exist only for the duration of the training. . apply the node property steps, added according to Adding node properties, on the `feature input` graph. . apply the feature steps, added according to Adding link features, to the `train` graph, which yields for each `train` relationship an *instance*, that is, a feature vector and a binary label. . split the training instances using stratified k-fold crossvalidation. The number of folds `k` can be configured using `validationFolds` in `gds.alpha.ml.pipeline.linkPrediction.configureSplit`. . train each model candidate given by the parameter space for each of the folds and evaluate the model on the respective validation set. The training process uses a logistic regression algorithm, and the evaluation uses the AUCPR metric. . declare as winner the model with the highest average metric across the folds. . re-train the winning model on the whole training set and evaluate it on both the `train` and `test` sets. In order to evaluate on the `test` set, the feature pipeline is first applied again as for the `train` set. . register the winning model in the Model Catalog.

NOTE: The above steps describe what the procedure does logically. The actual steps as well as their ordering in the implementation may differ.

NOTE: A step can only use node properties that are already present in the input graph or produced by steps, which were added before.

=== Syntax

[.include-with-train] — .Run Link Prediction in train mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.train(graphName: String, configuration: Map) YIELD trainMillis: Integer, modelInfo: Map, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.Configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of the model to train, must not exist in the Model Catalog. | pipeline | String | n/a | no | The name of the pipeline to execute. | negativeClassWeight | Float | 1.0 | yes | Weight of negative examples in model evaluation. Positive examples have weight 1. | randomSeed | Integer | n/a | yes | Seed for the random number generator used during training. | nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | trainMillis | Integer | Milliseconds used for training. | modelInfo | Map |

Information about the training and the winning model. | configuration | Map | Configuration used for the train procedure.

The `modelInfo` can also be retrieved at a later time by using the Model List Procedure. The `modelInfo` return field has the following algorithm-specific subfields:

.Model info fields [opts="header",cols="1,1,6"]

| Name | Type | Description | bestParameters | Map | The model parameters which performed best on average on validation folds according to the primary metric. | metrics | Map | Map from metric description to evaluated metrics for various models and subsets of the data, see below. | trainingPipeline | Map | The pipeline used for the training.

The structure of `modelInfo` is:

[listing] ---- { bestParameters: Map, // <1> trainingPipeline: Map // <2> metrics: { // <3> AUCPR: { test: Float, // <4> outerTrain: Float, // <5> train: [{ // <6> avg: Float, max: Float, min: Float, params: Map }, { avg: Float, max: Float, min: Float, params: Map }, ...], validation: [{ // <7> avg: Float, max: Float, min: Float, params: Map }, { avg: Float, max: Float, min: Float, params: Map }, ...] } } } ---- <1> The best scoring model candidate configuration. <2> The pipeline used for the training. <3> The `metrics` map contains an entry for each metric description (currently only `AUCPR`) and the corresponding results for that metric. <4> Numeric value for the evaluation of the best model on the test set. <5> Numeric value for the evaluation of the best model on the outer train set. <6> The `train` entry lists the scores over the `train` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`. <7> The `validation` entry lists the scores over the `validation` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`. — === Example

In this example we will create a small graph and train the pipeline we have built up thus far. The graph consists of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/link-prediction.svg[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, group=lp, indent=0] ---- CREATE (alice:Person {name: 'Alice', numberOfPosts: 38}), (michael:Person {name: 'Michael', numberOfPosts: 67}), (karin:Person {name: 'Karin', numberOfPosts: 30}), (chris:Person {name: 'Chris', numberOfPosts: 132}), (will:Person {name: 'Will', numberOfPosts: 6}), (mark:Person {name: 'Mark', numberOfPosts: 32}), (greg:Person {name: 'Greg', numberOfPosts: 29}), (veselin:Person {name: 'Veselin', numberOfPosts: 3}),

(alice)-[:KNOWS]→(michael), (michael)-[:KNOWS]→(karin), (michael)-[:KNOWS]→(chris), (michael)-[:KNOWS]→(greg), (will)-[:KNOWS]→(michael), (will)-[:KNOWS]→(chris), (mark)-[:KNOWS]→(michael), (mark)-[:KNOWS]→(will), (greg)-[:KNOWS]→(chris), (veselin)-[:KNOWS]→(chris), (karin)-[:KNOWS]→(veselin), (chris)-[:KNOWS]→(karin); ----

With the graph in Neo4j we can now project it into the graph catalog. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. We will also project the `numberOfPosts` property, so it can be used when creating link features. For the relationships we must use the `UNDIRECTED` orientation. This is because the Link Prediction pipelines are defined only for undirected graphs.

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, group=lp, indent=0] ---- CALL gds.graph.create('myGraph', { Person: { properties: ['numberOfPosts'] } }, { KNOWS: { orientation: 'UNDIRECTED' } }) ----

WARNING: The Link Prediction model requires the graph to be created using the `UNDIRECTED` orientation for relationships.

[role=query-example,group=lp] — .The following will train a model using a pipeline: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.train('myGraph', { pipeline: 'pipe', modelName: 'lp-pipeline-model', randomSeed: 42 }) YIELD modelInfo RETURN modelInfo.bestParameters AS winningModel, modelInfo.metrics.AUCPR.outerTrain AS trainGraphScore, modelInfo.metrics.AUCPR.test AS testGraphScore ----

.Results [opts="header", cols="6, 2, 2"]

| winningModel | trainGraphScore | testGraphScore | {useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001, concurrency=4} | 0.41666666666666663 | 0.7638888888888888

We can see the model configuration with `tolerance = 0.001` (and defaults filled for remaining parameters) was selected, and has a score of `0.76` on the test set. The score computed as the AUCPR metric, which is in the range [0, 1]. A model which gives higher score to all links than non-links will have a score of 1.0, and a model that assigns random scores will on average have a score of 0.5. — == Applying a trained model for prediction

In the previous sections we have seen how to build up a Link Prediction training pipeline and train it to produce a predictive model. After training, the runnable model is of type `Link prediction pipeline` and resides in the model catalog.

The trained model can then be applied to a graph in the graph catalog to create a new relationship type containing the predicted links. The relationships also have a property which stores the predicted probability of the link, which can be seen as a relative measure of the model's prediction confidence.

Since the model has been trained on features which are created using the feature pipeline, the same feature pipeline is stored within the model and executed at prediction time. As during training, intermediate node properties created by the node property steps in the feature pipeline are transient and not visible after execution.

When using the model for prediction, the relationships of the input graph are used in two ways. First, the input graph is fed into the feature pipeline and therefore influences predictions if there is at least one step in the pipeline which uses the input relationships (typically any node property step does). Second, predictions are carried out on each node pair that is not connected in the input graph.

The predicted links are sorted by score before the ones having score below the configured `threshold` are discarded. Finally, the configured `topN` predictions are stored back to the in-memory graph.

It is necessary that the predict graph contains the properties that the pipeline requires and that the used array properties have the same dimensions as in the train graph. If the predict and train graphs are distinct, it is also beneficial that they have similar origins and semantics, so that the model is able to generalize well.

=== Syntax

.Link Prediction syntax per mode [.tabbed-example, caption =] ==== [.include-with-mutate] ====== .Run Link Prediction in mutate mode on a named graph: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.predict.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, postProcessingMillis: Integer, mutateMillis: Integer, relationshipsWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.Configuration [opts="header",cols="1,1,2m,1,5"]

| Name | Type | Default | Optional | Description | modelName | String | n/a | no | The name of a Link Prediction model in the model catalog. | nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm. | mutateRelationshipType | String | n/a | no | The relationship type used for the new relationships written to the in-memory graph. | mutateProperty | String | 'probability' | yes | The relationship property in the GDS graph to which the result is written.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | topN | Integer | n/a | no | Limit on predicted relationships

to output. | threshold | Float | 0.0 | yes | Minimum predicted probability on relationships to output.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for creating the graph. | computeMillis | Integer | Milliseconds for running the algorithm. | postProcessingMillis | Integer | Milliseconds for computing the global metrics. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | relationshipsWritten | Integer | Number of relationships created. | configuration | Map | Configuration used for running the algorithm.

====== ====

=== Example

In this example we will show how to use a trained model to predict new relationships in your in-memory graph. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the train example which we gave the name `lp-pipeline-model`. The algorithm excludes predictions for existing relationships in the graph.

[role=query-example, group=lp] — [source, cypher, indent=0] ---- CALL gds.alpha.ml.pipeline.linkPrediction.predict.mutate('myGraph', { modelName: 'lp-pipeline-model', mutateRelationshipType: 'KNOWS_PREDICTED', topN: 5, threshold: 0.45 }) YIELD relationshipsWritten ----

.Results [opts="header"]

| relationshipsWritten | 10

— We specified `threshold` to filter out predictions with probability less than 45%, and `topN` to further limit output to the top 5 relationships. Because we are using the `UNDIRECTED` orientation, we will write twice as many relationships to the in-memory graph.

In the following, we will inspect the predicted relationships:

[role=query-example, group=lp] — .Stream the predicted relationships: [source, cypher, role=noplay, indent=0] ---- CALL gds.graph.streamRelationshipProperty('myGraph', 'probability', ['KNOWS_PREDICTED']) YIELD sourceNodeId, targetNodeId, propertyValue WHERE sourceNodeId < targetNodeId RETURN gds.util.asNode(sourceNodeId).name as source, gds.util.asNode(targetNodeId).name as target, propertyValue AS probability ORDER BY source ASC, target ASC ----

.Results [opts="header"]

| source | target | probability | "Alice" | "Chris" | 0.5422350772807373 | "Alice" | "Greg" | 0.51204718863418 | "Alice" | "Karin" | 0.5123040606165334 | "Alice" | "Mark" | 0.5130009448848327 | "Chris" | "Mark" | 0.5364414066546659

We can see, that our model predicts the most likely link is between Alice and Chris. — :leveloffset: 2

:leveloffset: +2

:description: This chapter provides explanations and examples for auxiliary procedures in the Neo4j Graph Data Science library. = Auxiliary procedures

[abstract] — This chapter provides explanations and examples for auxiliary procedures in the Neo4j Graph Data Science library. — Auxiliary procedures are extra tools that can be useful in your workflow.
The Neo4j GDS library includes the following auxiliary procedures, grouped by quality tier:

* Beta Graph Generation * Alpha Collapse Path Scale Properties One Hot Encoding ** Split Relationships

:leveloffset: 2

:leveloffset: +3

:description: This section describes how random graphs can be generated in the Neo4j Graph Data Science library. [.beta] = Graph Generation

[abstract] — This section describes how random graphs can be generated in the Neo4j Graph Data Science library. — In certain use cases it is useful to generate random graphs, for example, for testing or benchmarking purposes. For that reason the Neo4j Graph Algorithm library comes with a set of built-in graph generators. The generator stores the resulting graph in the graph catalog. That graph can be used as input for any algorithm in the library.

[.beta-symbol] [.tier-note] This algorithm is in the beta tier. For more information on algorithm tiers, see Algorithms.

[NOTE] ==== It is currently not possible to persist these graphs in Neo4j. Running an algorithm in write mode on a generated graph will lead to unexpected results. ====

The graph generation is parameterized by three dimensions:

* node count - the number of nodes in the generated graph * average degree - describes the average out-degree of the generated nodes * relationship distribution function - the probability distribution method used to connect generated nodes

== Syntax

.The following describes the API for running the algorithm [source, cypher, role=noplay, indent=0] ---- CALL gds.beta.graph.generate(graphName: String, nodeCount: Integer, averageDegree: Integer, { relationshipDistribution: String, relationshipProperty: Map }) YIELD name, nodes, relationships, generateMillis, relationshipSeed, averageDegree, relationshipDistribution, relationshipProperty ----

.Parameters [opts="header",cols="2m,1,1m,1,5"]

| Name | Type | Default | Optional | Description | graphName | String | null | no | The name under which the generated graph is stored. | nodeCount | Integer | null | no | The number of generated nodes. | averageDegree | Integer | null | no | The average out-degree of generated nodes. | configuration | Map | {} | yes | Additional configuration, see below.

.Configuration [opts="header",cols="3m,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipDistribution | String | UNIFORM | yes | The probability distribution method used to connect generated nodes. For more information see Relationship Distribution. | relationshipSeed | Integer | null | yes | The seed used for generating relationships. | relationshipProperty | Map | {} | yes | Describes the method used to generate a relationship property. By default no relationship property is generated. For more information see Relationship Property. |

aggregation | String | NONE | yes | The relationship aggregation method cf. Relationship Projection. | orientation | String | NATURAL | yes | The method of orienting edges. Allowed values are NATURAL, REVERSE and UNDIRECTED. | allowSelfLoops | Boolean | false | yes | Whether to allow relationships with identical source and target node.

.Results [opts="header",cols="3m,1,6"]

| Name | Type | Description | name | String | The name under which the stored graph was stored. | nodes | Integer | The number of nodes in the graph. | relationships | Integer | The number of relationships in the graph. | generateMillis | Integer | Milliseconds for generating the graph. | relationshipSeed | Integer | The seed used for generating relationships. | averageDegree | Float | The average out degree of the generated nodes. | relationshipDistribution | String | The probability distribution method used to connect generated nodes. | relationshipProperty | String | The configuration of the generated relationship property.

== Relationship Distribution

The `relationshipDistribution` parameter controls the statistical method used for the generation of new relationships. Currently there are three supported methods:

* `UNIFORM` - Distributes the outgoing relationships evenly, i.e., every node has exactly the same out degree (equal to the average degree). The target nodes are selected randomly. * `RANDOM` - Distributes the outgoing relationships using a normal distribution with an average of `averageDegree` and a standard deviation of `2 * averageDegree`. The target nodes are selected randomly. * `POWER_LAW` - Distributes the incoming relationships using a power law distribution. The out degree is based on a normal distribution.

== Relationship Seed

The `relationshipSeed` parameter allows, to generate graphs with the same relationships, if they have no property. Currently the `relationshipProperty` is not seeded, therefore the generated graphs can differ in their property values. Hence generated graphs based on the same `relationshipSeed` are not identical.

== Relationship Property

The graph generator is capable of generating a relationship property. This can be controlled using the `relationshipProperty` parameter which accepts the following parameters:

.Configuration [opts="header",cols="1m,1,1,1,4"]

| Name | Type | Default | Optional | Description | name | String | null | no | The name under which the property values are stored. | type | String | null | no | The method used to generate property values. | min | Float | 0.0 | yes | Minimal value of the generated property (only supported by `RANDOM`). | max | Float | 1.0 | yes | Maximum value of the generated property (only supported by `RANDOM`). | value | Float | null | yes | Fixed value assigned to every relationship (only supported by `FIXED`).

Currently, there are two supported methods to generate relationship properties:

* `FIXED` - Assigns a fixed value to every relationship. The `value` parameter must be set. * `RANDOM` - Assigns a random value between the lower (`min`) and upper (`max`) bound.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Collapse Path algorithm in the Neo4j Graph Data Science library. [.alpha] = Collapse Path

[abstract] — This section describes the Collapse Path algorithm in the Neo4j Graph Data Science library. —

== Introduction

The Collapse Path algorithm is a traversal algorithm capable of creating relationships between the start and end nodes of a traversal. In other words, the path between the start node and the end node is collapsed into a single relationship (a direct path). The algorithm is intended to support the creation of monopartite graphs required by many graph algorithms.

The main input for the algorithm is a list of relationship types. Starting from every node in the specified graph, these relationship types are traversed one after the other using the order specified in the configuration. Only nodes reached after traversing every relationship type specified are used as end nodes. Exactly one relationship is created for every pair of nodes for which at least one path from start to end node exists.

== Syntax

.Collapse Path syntax per mode [.tabbed-example, caption =] ====

[.include-with-mutate] ====== .Run Collapse Path in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.collapsePath.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, relationshipsWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | relationshipTypes | List of String | n/a | no | Ordered list of relationship types used for the traversal. The same relationship type can be added multiple times, in order to traverse them as indicated. | mutateRelationshipType | String | n/a | no | Relationship type of the newly created relationships. | allowSelfLoops | Boolean | false | yes | Indicates whether it is possible to create self referencing relationships, i.e. relationships where the start and end node are identical.

.Results [opts="header",cols="1m,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | relationshipsWritten | Integer | The number of relationships created by the algorithm. | configuration | Map | The configuration used for running the algorithm.

====== ====

== Examples

Consider the graph created by the following Cypher statement:

[source, cypher, role=noplay setup-query, indent=0] ---- CREATE (Dan:Person), (Annie:Person), (Matt:Person), (Jeff:Person),

(Guitar:Instrument), (Flute:Instrument),

(Dan)-[:PLAYS]→(Guitar), (Annie)-[:PLAYS]→(Guitar),

(Matt)-[:PLAYS]→(Flute), (Jeff)-[:PLAYS]→(Flute) ----

In this example we want to create a relationship, called `PLAYS_SAME_INSTRUMENT`, between `Person` nodes that play the same instrument. To achieve that we have to traverse a path specified by the following Cypher pattern:

[source, cypher, role=noplay, indent=0] ---- (p1:Person)-[:PLAYS]→(:Instrument)-[:PLAYED_BY]→(p2:Person) ----

In our source graph only the `PLAYS` relationship type exists. The `PLAYED_BY` relationship type can be created by loading the `PLAYS` relationship type in `REVERSE` direction. The following query will create such a graph:

[source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('persons', ['Person', 'Instrument'], { PLAYS: { type: 'PLAYS', orientation: 'NATURAL' }, PLAYED_BY: { type: 'PLAYS', orientation: 'REVERSE' } }) ----

Now we can run the algorithm by specifying the traversal `PLAYS, PLAYED_BY` in the `relationshipTypes` option.

[role=query-example] — [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.collapsePath.mutate('persons', { relationshipTypes: ['PLAYS', 'PLAYED_BY'], allowSelfLoops: false, mutateRelationshipType: 'PLAYS_SAME_INSTRUMENT' }) YIELD relationshipsWritten ----

[opts=header] .Results

| relationshipsWritten | 4

— .The mutated graph will look like the following graph when filtered by the PLAYS_SAME_INSTRUMENT relationship [source, cypher, role=noplay, indent=0] ---- CREATE (Dan:Person), (Annie:Person), (Matt:Person), (Jeff:Person),

(Guitar:Instrument), (Flute:Instrument),

(Dan)-[:PLAYS_SAME_INSTRUMENT]→(Annie), (Annie)-[:PLAYS_SAME_INSTRUMENT]→(Dan),

(Matt)-[:PLAYS_SAME_INSTRUMENT]→(Jeff), (Jeff)-[:PLAYS_SAME_INSTRUMENT]→(Matt), ----

:leveloffset: 2

:leveloffset: +3

:description: This section describes the Scale Properties algorithm in the Neo4j Graph Data Science library. [.alpha] = Scale Properties

:algorithm: Scale Properties :entity: node

[abstract] — This section describes the Scale Properties algorithm in the Neo4j Graph Data Science library. — == Introduction

The Scale Properties algorithm is a utility algorithm that is used to pre-process node properties for model training or post-process algorithm results such as PageRank scores. It scales the node properties based on the specified scaler. Multiple properties can be scaled at once and are returned in a list property.

The input properties must be numbers or lists of numbers. The lists must all have the same size. The output property will always be a list. The size of the output list is equal to the sum of length of the input properties. That is, if the input properties are two scalar numeric properties and one list property of length three, the output list will have a total length of five.

There are a number of supported scalers for the Scale Properties algorithm. These can be configured using the `scaler` configuration parameter.

List properties are scaled index-by-index. See the list example for more details.

In the following equations, p denotes the vector containing all property values for a single property across all nodes in the graph.

=== Min-max scaler

Scales all property values into the range `[0, 1]` where the minimum value(s) get the scaled value `0` and the maximum value(s) get the scaled value `1`, according to this formula:

image::equations/scale-properties/minmax.svg[scaled p equals p minus minimum of p divided by maximum of p minus minimum of p,align="center"]

=== Max scaler

Scales all property values into the range `[-1, 1]` where the absolute maximum value(s) get the scaled value `1`, according to this formula:

image::equations/scale-properties/max.svg[scaled p equals p divided by the absolute maximum of p,align="center"]

=== Mean scaler

Scales all property values into the range `[-1, 1]` where the average value(s) get the scaled value `0`.

image::equations/scale-properties/mean.svg[scaled p equals p minus average of p divided by maximum of p minus minimum of p,align="center"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProperties | List of String | n/a | no | The names of the node properties that are to be scaled. All property names must exist in the in-memory graph. | scaler | String | n/a | no | The name of the scaler applied for the properties. Supported values are `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | nodeId | Integer | Node ID. | scaledProperty | List of Float | Scaled values for each input node property.

======

[.include-with-mutate] ====== .Run Scale Properties in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.scaleProperties.mutate(graphName: String, configuration: Map) YIELD createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, postProcessingMillis: Integer, nodePropertiesWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | Filter the named graph using the given node labels. | relationshipTypes | List of String | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]

| Name | Type | Default | Optional | Description | nodeProperties | List of String | n/a | no | The names of the node properties that are to be scaled. All property names must exist in the in-memory graph. | scaler | String | n/a | no | The name of the scaler applied for the properties. Supported values are `MinMax`, `Max`, `Mean`, `Log`, `L1Norm`, `L2Norm` and `StdScore`.

.Results [opts="header",cols="1,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | postProcessingMillis | Integer | Unused. | nodePropertiesWritten | Integer | Number of node properties written. | configuration | Map | Configuration used for running the algorithm.

====== ====

== Examples

:algorithm-name: {algorithm} :graph-description: hotel :image-file: scale-properties.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

.The following Cypher statement will create the example graph in the Neo4j database: [source, cypher, role=noplay setup-query, indent=0] ---- CREATE (:Hotel {avgReview: 4.2, buildYear: 1978, storyCapacity: [32, 32, 0], name: 'East'}), (:Hotel {avgReview: 8.1, buildYear: 1958, storyCapacity: [18, 20, 0], name: 'Plaza'}), (:Hotel {avgReview: 19.0, buildYear: 1999, storyCapacity: [100, 100, 70], name: 'Central'}), (:Hotel {avgReview: -4.12, buildYear: 2005, storyCapacity: [250, 250, 250], name: 'West'}), (:Hotel {avgReview: 0.01, buildYear: 2020, storyCapacity: [1250, 1250, 900], name: 'Polar'}), (:Hotel {avgReview: 3.3, buildYear: 1981, storyCapacity: [240, 240, 0], name: 'Beach'}), (:Hotel {avgReview: 6.7, buildYear: 1984, storyCapacity: [80, 0, 0], name: 'Mountain'}), (:Hotel {avgReview: -1.2, buildYear: 2010, storyCapacity: [55, 20, 0], name: 'Forest'}) ----

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Hotel` nodes, including their properties. Note that no relationships are necessary to scale the node properties. Thus we use a star projection (**'') for relationships.**

[NOTE] ==== In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or Cypher projections can also be used. ====

.The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'. [source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('myGraph', 'Hotel', '', { nodeProperties: ['avgReview', 'buildYear', 'storyCapacity'] }) ----

In the following examples we will demonstrate how to scale the node properties of this graph.

=== Stream

:result: scaled properties :entity: node :stream-details: Note that the output is always a single list property, containing all scaled node properties in the input order. In the `stream` execution mode, the algorithm returns the {result} for each {entity}. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see Stream.

[role=query-example] — .The following will run the algorithm in `stream` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.scaleProperties.stream('myGraph', { nodeProperties: ['buildYear', 'avgReview'], scaler: 'MinMax' }) YIELD nodeId, scaledProperty RETURN gds.util.asNode(nodeId).name AS name, scaledProperty ORDER BY name ASC ----

.Results [opts="header"]

| name | scaledProperty | "Beach" | [0.3709677419354839, 0.3209342560553633] | "Central" | [0.6612903225806451, 1.0] | "East" | [0.3225806451612903, 0.35986159169550175] | "Forest" |

[0.8387096774193549, 0.12629757785467127] | "Mountain" | [0.41935483870967744, 0.4679930795847751] | "Plaza" | [0.0, 0.5285467128027681] | "Polar" | [1.0, 0.17863321799307957] | "West" | [0.7580645161290323, 0.0]

— In the results we can observe that the first element in the resulting `scaledProperty` we get the min-max-scaled values for `buildYear`, where the `Plaza` hotel has the minimum value and is scaled to zero, while the `Polar` hotel has the maximum value and is scaled to one. This can be verified with the example graph. The second value in the `scaledProperty` result are the scaled values of the `avgReview` property.

=== Mutate

The `mutate` execution mode enables updating the named graph with a new node property containing the scaled properties for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row containing metrics from the computation. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see Mutate.

In this example we will scale the two hotel properties of `buildYear` and `avgReview` using the Mean scaler. The output is a list property which we will call `hotelFeatures`, imagining that we will use this as input for a machine learning model later on.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.scaleProperties.mutate('myGraph', { nodeProperties: ['buildYear', 'avgReview'], scaler: 'Mean', mutateProperty: 'hotelFeatures' }) YIELD nodePropertiesWritten ----

.Results [opts="header"]

| nodePropertiesWritten | 8

— The result shows that there are now eight new node properties in the in-memory graph. These contain the scaled values from the input properties, where the scaled `buildYear` values are in the first list position and scaled `avgReview` values are in the second position. To find out how to inspect the new schema of the in-memory graph, see Listing graphs in the catalog.

=== List properties

The `storyCapacity` property models the amount of rooms on each story of the hotel. The property is normalized so that hotels with fewer stories have a zero value. This is because the Scale Properties algorithm requires that all values for the same property have the same length. In this example we will show how to scale the values in these lists using the Scale Properties algorithm. We imagine using the output as feature vector to input in a machine learning algorithm. Additionally, we will include the `avgReview` property in our feature vector.

[role=query-example] — .The following will run the algorithm in `mutate` mode: [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.scaleProperties.stream('myGraph', { nodeProperties: ['avgReview', 'storyCapacity'], scaler: 'StdScore' }) YIELD nodeId, scaledProperty RETURN gds.util.asNode(nodeId).name AS name, scaledProperty AS features ORDER BY name ASC ----

.Results [opts="header"]

| name | features | "Beach" | [-0.17956547594003253, -0.03401933556831381, 0.00254261210704973, -0.5187592498702616] | "Central" | [2.172199255871029, -0.3968922482969945, -0.3534230828799124, -0.2806402499298136] | "East" | [-0.0447509371737933, -0.5731448059080679, -0.526320706159294, -0.5187592498702616] | "Forest" | [-0.8536381697712284, -0.513529970245499, -0.5568320514438908, -0.5187592498702616] | "Mountain" | [0.32973389273242665, -0.4487312358296632, -0.6076842935848854, -0.5187592498702616] | "Plaza" | [0.5394453974799097,

-0.609432097180936, -0.5568320514438908, -0.5187592498702616] | "Polar" | [-0.672387512096618, 2.583849534831454, 2.5705808402272767, 2.542770749364069] | "West" | [-1.2910364511016934, -0.00809984180197948, 0.027968733177547028, 0.3316657499170525]

— The resulting feature vector contains the standard-score scaled value for the `avgReview` property in the first list position. We can see that some values are negative and that the maximum value sticks out for the `Central` hotel.

The other three list positions are the scaled values for the `storyCapacity` list property. Note that each list item is scaled only with respect to the corresponding item in the other lists. Thus, the `Polar` hotel has the greatest scaled value in all list positions.

:leveloffset: 2

:leveloffset: +3

:description: This section describes the One Hot Encoding function in the Neo4j Graph Data Science library. [.alpha] = One Hot Encoding

[abstract] — This section describes the One Hot Encoding function in the Neo4j Graph Data Science library. — The One Hot Encoding function is used to convert categorical data into a numerical format that can be used by Machine Learning libraries.

[.alpha-symbol] [.tier-note] This algorithm is in the alpha tier. For more information on algorithm tiers, see Algorithms.

== One Hot Encoding sample

One hot encoding will return a list equal to the length of the `available values`. In the list, `selected values` are represented by `1`, and `unselected values` are represented by `0`.

.The following will run the algorithm on hardcoded lists: [source, cypher, role=noplay, indent=0] ---- RETURN gds.alpha.ml.oneHotEncoding(['Chinese', 'Indian', 'Italian'], ['Italian']) AS embedding ----

.Results [opts="header",cols="1"]

| embedding | [0,0,1]

.The following will create a sample graph: [source, cypher, role=noplay, indent=0] ---- CREATE (french:Cuisine {name:'French'}), (italian:Cuisine {name:'Italian'}), (indian:Cuisine {name:'Indian'}),

(zhen:Person {name: "Zhen"}), (praveena:Person {name: "Praveena"}), (michael:Person {name: "Michael"}), (arya:Person {name: "Arya"}),

(praveena)-[:LIKES]→(indian), (zhen)-[:LIKES]→(french), (michael)-[:LIKES]→(french), (michael)-[:LIKES]→(italian) ----

.The following will return a one hot encoding for each user and the types of cuisine that they like: [source, cypher, role=noplay, indent=0] ---- MATCH (cuisine:Cuisine) WITH cuisine ORDER BY cuisine.name WITH collect(cuisine) AS cuisines MATCH (p:Person) RETURN p.name AS name, gds.alpha.ml.oneHotEncoding(cuisines, [(p)-[:LIKES]→(cuisine) cuisine]) AS embedding ORDER BY name ----

.Results [opts="header",cols="1, 1"]

| name | embedding | Arya | [0,0,0] | Michael | [1,0,1] | Praveena | [0,1,0] | Zhen | [1,0,0]

```
.Parameters [opts="header",cols="1,1,1,1,4"]
```

| Name | Type | Default | Optional | Description | `availableValues` | list | null | yes | The available values. If null, the function will return an empty list. | `selectedValues` | list | null | yes | The selected values. If null, the function will return a list of all 0's.

```
.Results [opts="header",cols="1,6"]
```

| Type | Description | `list` | One hot encoding of the selected values.

```
:leveloffset: 2

:leveloffset: +3

:description: This section describes the Split Relationships algorithm in the Neo4j Graph Data Science library.  [.alpha] = Split Relationships

:algorithm: Split Relationships

[abstract] — This section describes the Split Relationships algorithm in the Neo4j Graph Data Science library. —

== Introduction

The Split relationships algorithm is a utility algorithm that is used to pre-process a graph for model training. It splits the relationships into a holdout set and a remaining set. The holdout set is divided into two classes: positive, i.e., existing relationships, and negative, i.e., non-existing relationships. The class is indicated by a `label` property on the relationships. This enables the holdout set to be used for training or testing a machine learning model. Both, the holdout and the remaining relationships are added to the in-memory graph.

== Syntax

This section covers the syntax used to execute the {algorithm} algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see Syntax overview.

.Split Relationships syntax per mode [.tabbed-example, caption = ] ====

[.include-with-mutate] ====== .Run Split Relationships in mutate mode on a named graph. [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.splitRelationships.mutate( graphName: String, configuration: Map ) YIELD createMillis: Integer, computeMillis: Integer, mutateMillis: Integer, relationshipsWritten: Integer, configuration: Map ----

.Parameters [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | graphName | String | n/a | no | The name of a graph stored in the catalog. | configuration | Map | {} | yes | Configuration for algorithm-specifics and/or graph filtering.

```
.General configuration for algorithm execution on a named graph. [opts="header",cols="2,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description

| nodeLabels | List of String | ['*'] | yes | **Filter the named graph using the given node labels.** | relationshipTypes | **List of String** | ['*'] | yes | Filter the named graph using the given relationship types. | concurrency | Integer | 4 | yes | The number of concurrent threads used for running the algorithm.

```
.Algorithm specific configuration [opts="header",cols="1,1,1m,1,4"]
```

| Name | Type | Default | Optional | Description | holdoutFraction | Float | n/a | no | The fraction of all relationships being used as holdout set. | negativeSamplingRatio | Float | n/a | no | The desired ratio of negative to positive samples in holdout set. | holdoutRelationshipType | String | n/a | no | Relationship type used for the holdout set. Each relationship has a property `label` indicating whether it is a positive or negative sample. | remainingRelationshipType | String | n/a | no | Relationship type used for the remaining set. | nonNegativeRelationshipTypes | List of String | n/a | yes | Additional relationship types that are used for negative sampling. | randomSeed | Integer | n/a | yes | An optional seed value for the random selection of relationships.

.Results [opts="header",cols="1m,1,6"]

| Name | Type | Description | createMillis | Integer | Milliseconds for loading data. | computeMillis | Integer | Milliseconds for running the algorithm. | mutateMillis | Integer | Milliseconds for adding properties to the in-memory graph. | relationshipsWritten | Integer | The number of relationships created by the algorithm. | configuration | Map | The configuration used for running the algorithm.

====== ====

== Examples

:algorithm-name: {algorithm} :graph-description: :image-file: split-relationships.svg In this section we will show examples of running the {algorithm-name} algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small {graph-description} graph of a handful nodes connected in a particular pattern. The example graph looks like this:

image::example-graphs/{image-file}[Visualization of the example graph,align="center"]

Consider the graph created by the following Cypher statement:

[source, cypher, role=noplay setup-query, indent=0] ---- CREATE (n0:Label), (n1:Label), (n2:Label), (n3:Label), (n4:Label), (n5:Label),

(n0)-[:TYPE]→(n1), (n1)-[:TYPE]→(n2), (n2)-[:TYPE]→(n3), (n3)-[:TYPE]→(n4), (n4)-[:TYPE]→(n5) ----

Given the above graph, we want to use 20% of the relationships as holdout set. The holdout set will be split into two same-sized classes: positive and negative. Positive relationships will be randomly selected from the existing relationships and marked with a property `label: 1`. Negative relationships will be randomly generated, i.e., they do not exist in the input graph, and are marked with a property `label: 0`.

[source, cypher, role=noplay graph-create-query, indent=0] ---- CALL gds.graph.create('graph', 'Label', { TYPE: { orientation: 'UNDIRECTED' } }) ----

Now we can run the algorithm by specifying the appropriate ratio and the output relationship types. We use a random seed value in order to produce deterministic results.

[role=query-example] — [source, cypher, role=noplay, indent=0] ---- CALL gds.alpha.ml.splitRelationships.mutate('graph', { holdoutRelationshipType: 'TYPE_HOLDOUT', remainingRelationshipType: 'TYPE_REMAINING', holdoutFraction: 0.2, negativeSamplingRatio: 1.0, randomSeed: 1337 }) YIELD relationshipsWritten ----

[opts=header] .Results

| relationshipsWritten | 10

— The input graph consists of 5 relationships. We use 20% (1 relationship) of the relationships to create the 'TYPE_HOLDOUT' relationship type (holdout set). This creates 1 relationship with positive label. Because of the `negativeSamplingRatio`, one relationship with negative label is also created. Finally, the `TYPE_REMAINING` relationship type is formed with the remaining 80% (4 relationships). These are written as orientation `UNDIRECTED` which counts as writing 8 relationships.

.The mutated graph will look like the following graph when filtered by the `TEST` and `TRAIN` relationship. [source, cypher, role=noplay, indent=0] ---- CREATE (n0:Label), (n1:Label), (n2:Label), (n3:Label), (n4:Label), (n5:Label),

(n2)-[:TYPE_HOLDOUT { label: 0 }]→(n5), // negative, non-existing (n3)-[:TYPE_HOLDOUT { label: 1 }]→(n2), // positive, existing

(n0)←[:TYPE_REMAINING]-(n1), (n1)←[:TYPE_REMAINING]-(n2), (n3)←[:TYPE_REMAINING]-(n4), (n4)←[:TYPE_REMAINING]-(n5), (n0)-[:TYPE_REMAINING]→(n1), (n1)-[:TYPE_REMAINING]→(n2), (n3)-[:TYPE_REMAINING]→(n4), (n4)-[:TYPE_REMAINING]→(n5) ----

:leveloffset: 2

:leveloffset: +2

:description: This chapter provides documentation for the Pregel API in the Neo4j Graph Data Science library. = Pregel API

[abstract] — This chapter provides documentation for the Pregel API in the Neo4j Graph Data Science library. — == Introduction

Pregel is a vertex-centric computation model to define your own algorithms via a user-defined *compute* function. Node values can be updated within the compute function and represent the algorithm result. The input graph contains default node values or node values from a graph projection.

The compute function is executed in multiple iterations, also called *supersteps*. In each superstep, the compute function runs for each node in the graph. Within that function, a node can receive messages from other nodes, typically its neighbors. Based on the received messages and its currently stored value, a node can compute a new value. A node can also send messages to other nodes, typically its neighbors, which are received in the next superstep. The algorithm terminates after a fixed number of supersteps or if no messages are being sent between nodes.

A Pregel computation is executed in parallel. Each thread executes the compute function for a batch of nodes.

For more information about Pregel, have a look at https://kowshik.github.io/JPregel/pregel_paper.pdf.

To implement your own Pregel algorithm, the Graph Data Science library provides a Java API, which is described below.

The introduction of a new Pregel algorithm can be separated in two main steps. First, we need to implement the algorithm using the Pregel Java API. Second, we need to expose the algorithm via a Cypher procedure to make use of it.

For an example on how to expose a custom Pregel computation via a Neo4j procedure, have a look at the Pregel examples.

== Pregel Java API

The Pregel Java API allows us to easily build our own algorithm by implementing several interfaces.

=== Computation

The first step is to implement the `org.neo4j.gds.beta.pregel.PregelComputation` interface. It is the main interface to express user-defined logic using the Pregel framework.

.The Pregel computation [source, java, indent=0] ---- public interface PregelComputation<C extends PregelConfig> { // The schema describes the node property layout. PregelSchema schema(); // Called in the first superstep and allows initializing node state. default void init(PregelContext.InitContext<C> context) {} // Called in each superstep for each node and contains the main logic. void compute(PregelContext.ComputeContext<C> context, Pregel.Messages messages); // Called exactly

| Name | Type | Default Value | Description | maxIterations | Integer | - | Maximum number of supersteps after which the computation will terminate. | isAsynchronous | Boolean | false | Flag indicating if messages can be sent and received in the same superstep. | partitioning | String | "range" | Selects the partitioning of the input graph, can be either "range", "degree" or "auto". | relationshipWeightProperty | String | null | Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted. | concurrency | Integer | 4 | Concurrency used when executing the Pregel computation. | writeConcurrency | Integer | concurrency | Concurrency used when writing computation results to Neo4j. | writeProperty | String | "pregel_" | Prefix string that is prepended to node schema keys in write mode. | mutateProperty | String | "pregel_" | Prefix string that is prepended to node schema keys in mutate mode.

For some algorithms, we want to specify additional configuration options.

Typically, these options are algorithm specific arguments, such as thresholds. Another reason for a custom config relates to the initialization phase of the computation. If we want to init the node state based on a graph property, we need to access that property via its key. Since those keys are dynamic properties of the graph, we need to provide them to the computation. We can achieve that by declaring an option to set that key in a custom configuration.

If a user-defined Pregel computation requires custom options a custom configuration can be created by extending the `PregelConfig`.

.A custom configuration and how it can be used in the init phase. [source, java, indent=0] ---- @ValueClass @Configuration public interface CustomConfig extends PregelConfig { // A property key that refers to a seed property. String seedProperty(); // An algorithm specific parameter. int minDegree(); }

public class CustomComputation implements PregelComputation<CustomConfig> {

@Override public void init(PregelContext.InitContext<CustomConfig> context) { // Use the custom config key to access a graph property. var seedProperties = context.nodeProperties(context.config().seedProperty()); // Init the node state with the graph property for that node. context.setNodeValue("state", seedProperties.doubleValue(context.nodeId())); }

@Override public void compute(PregelContext.ComputeContext<CustomConfig> context, Pregel.Messages messages) { if (context.degree() >= context.config().minDegree()) { // ... } }

// ... } ----

== Run Pregel via Cypher

To make a custom Pregel computation accessible via Cypher, it needs to be exposed via the procedure API. The Pregel framework in GDS provides an easy way to generate procedures for all the default modes.

=== Procedure generation

To generate procedures for a computation, it needs to be annotated with the `@org.neo4j.gds.beta.pregel.annotation.PregelProcedure` annotation. In addition, the config parameter of the custom computation must be a subtype of `org.neo4j.gds.beta.pregel.PregelProcedureConfig`.

.Using the `@PregelProcedure` annotation to configure code generation. [source, java, indent=0] ---- @PregelProcedure(name = "custom.pregel.proc", modes = {GDSMode.STREAM, GDSMode.WRITE}, description = "My custom Pregel algorithm") public class CustomComputation implements PregelComputation<PregelProcedureConfig> { // ... } ----

The annotation provides a number of configuration options for the code generation.

.Configuration [opts="header",cols="1,1,1,6"]

| Name | Type | Default Value | Description | name | String | - | The prefix of the generated procedure name. It is appended by the mode. | modes | List | `[STREAM, WRITE, MUTATE, STATS]` | A procedure is generated

for each of the specified modes. | description | String | `""` | Procedure description that is printed in `dbms.listProcedures()`.

For the above Code snippet, we generate four procedures:

* `custom.pregel.proc.stream` * `custom.pregel.proc.stream.estimate` * `custom.pregel.proc.write` * `custom.pregel.proc.write.estimate`

=== Building and installing a Neo4j plugin

In order to use a Pregel algorithm in Neo4j via a procedure, we need to package it as Neo4j plugin. The pregel-bootstrap project is a good starting point. The `build.gradle` file within the project contains all the dependencies necessary to implement a Pregel algorithm and to generate corresponding procedures.

Make sure to change the `gdsVersion` and `neo4jVersion` according to your setup. GDS and Neo4j are runtime dependencies. Therefore, GDS needs to be installed as a plugin on the Neo4j server.

To build the project and create a plugin jar, just run:

[source, bash, indent=0] ---- ./gradlew shadowJar ----

You can find the `pregel-bootstrap.jar` in `build/libs`. The jar needs to be placed in the `plugins` directory within your Neo4j installation alongside a GDS plugin jar. In order to have access to the procedure in Cypher, its namespace potentially needs to be added to the `neo4j.conf` file.

.Enabling an example procedure in `neo4j.conf` [source, bash, indent=0] ---- dbms.security.procedures.unrestricted=custom.pregel.proc.* dbms.security.procedures.allowlist=custom.pregel.proc.* ----

NOTE: Before `Neo4j 4.2`, the configuration setting is called `dbms.security.procedures.whitelist`

== Examples

The pregel-examples module contains a set of examples for Pregel algorithms. The algorithm implementations demonstrate the usage of the Pregel API. Along with each example, we provide test classes that can be used as a guideline on how to write tests for custom algorithms. To play around, we recommend copying one of the algorithms into the `pregel-bootstrap` project, build it and setup the plugin in Neo4j.

:leveloffset: 2

:leveloffset: +1

:description: This chapter explains advanced details with regards to common Neo4j components. = Production deployment

[abstract] — This chapter explains advanced details with regards to common Neo4j components. — This chapter is divided into the following sections:

* Transaction Handling * Using GDS and Fabric * GDS Feature Toggles

:leveloffset: 2

:leveloffset: +2

:description: This section describes the usage of transactions during the execution of an algorithm. = Transaction Handling

[abstract] — This section describes the usage of transactions during the execution of an algorithm. When an algorithm procedure is called from Cypher, the procedure call is executed within the same transaction as the Cypher statement. —

== During graph projection

During graph projection, new transactions are used that do not inherit the transaction state of the Cypher transaction. This means that changes from the Cypher transaction state are not visible to the graph projection transactions.

For example, the following statement will only create an empty graph (assuming the `MyLabel` label was not already present

| nodeCount | 0

— The situation is the same when using an anonymous projection with an algorithm procedure:

[role=query-example] — [source, cypher, role=noplay, indent=0] ---- CREATE (n:MyWccLabel) // the new node is part of Cypher transaction state WITH * CALL gds.wcc.stats({nodeProjection: 'MyWccLabel', relationshipProjection:'*'}) YIELD componentCount RETURN componentCount ----

.Results [opts="header"]

| componentCount | 0

—

== During results writing

Results from algorithms (node properties, for example) are written to the graph in new transactions. The number of transactions used depends on the size of the results and the `writeConcurrency` configuration parameter (for more details, please refer to sections Write and Common Configuration parameters). These transactions are committed independently from the Cypher transaction. This means, if the Cypher transaction is terminated (either by the user or by the database system), already committed write transactions will *not* be rolled back.

:leveloffset: 2

= Transaction writing examples

[NOTE] ==== The code in this section is for illustrative purposes. The goal is to demonstrate correct usage of the GDS library write functionality with Cypher Shell and Java API. ====

== Cypher Shell

Example for incorrect use.

[source, cypher, role=noplay, indent=0] ---- :BEGIN

CALL gds.graph.create.cypher('test', 'MATCH (n) WHERE n:Artist OR n:Genre RETURN id(n) AS id', 'MATCH (a:Artist)<-[:RELEASED_BY]-(:Album)-[:HAS_GENRE]->(g:Genre) RETURN id(g) AS source, id(a) AS target, "IS_ASSOCIATED_WITH" AS type');

MATCH ()-[r:SIMILAR_TO]->() DELETE r;

CALL gds.nodeSimilarity.write('test', { writeRelationshipType: 'SIMILAR_TO', writeProperty: 'score' });

:COMMIT ----

The issue with the above statement is that all the queries run in the same transaction.

A correct handling of the above statement would be to run each statement in its own transaction, which is shown below. Notice the reordering of the statements, this ensures that the in-memory graph will have the most recent changes after the removal of the relationships.

First remove the unwanted relationships.

[source, cypher, role=noplay, indent=0] ---- :BEGIN

MATCH ()-[r:SIMILAR_TO]->() DELETE r;

:COMMIT ----

Create the in-memory graph.

[source, cypher, role=noplay, indent=0] ---- :BEGIN

CALL gds.graph.create.cypher('test', 'MATCH (n) WHERE n:Artist OR n:Genre RETURN id(n) AS id', 'MATCH (a:Artist)<-[:RELEASED_BY]-(:Album)-[:HAS_GENRE]->(g:Genre) RETURN id(g) AS source, id(a) AS target, "IS_ASSOCIATED_WITH" AS type');

:COMMIT ----

Run the algorithm.

[source, cypher, role=noplay, indent=0] ---- :BEGIN

|Description | Operation .4+<.^|Create Graph | `gds.graph.create` | `gds.graph.create.estimate` | `gds.graph.create.cypher` | `gds.graph.create.cypher.estimate` .2+<.^|Check if a named graph exists | `gds.graph.exists` | `gds.graph.exists` |List graphs | `gds.graph.list` |Remove node properties from a named graph | `gds.graph.removeNodeProperties` |Delete relationships from a named graph | `gds.graph.deleteRelationships` |Remove a named graph from memory | `gds.graph.drop` |Stream a single node property to the procedure caller | `gds.graph.streamNodeProperty` |Stream node properties to the procedure caller | `gds.graph.streamNodeProperties` |Stream a single relationship property to the procedure caller | `gds.graph.streamRelationshipProperty` |Stream relationship properties to the procedure caller | `gds.graph.streamRelationshipProperties` |Write node properties to Neo4j | `gds.graph.writeNodeProperties` |Write relationships to Neo4j | `gds.graph.writeRelationship` |Graph Export | `gds.graph.export`

```
== Beta Tier

.List of all beta graph operations in the GDS library. Functions are written in _italic_. [role=procedure-listing]
[opts=header,cols="1, 1"]
```

|Description | Operation |Create a graph from a named graph | `gds.beta.graph.create.subgraph` |Generate Random Graph| `gds.beta.graph.generate` .2+<.^|CSV Export | `gds.beta.graph.export.csv` | `gds.beta.graph.export.csv.estimate`

```
:leveloffset: 2

:leveloffset: +3

= Model Catalog

== Beta Tier

.List of all beta model catalog operations in the GDS library. Functions are written in _italic_. [role=procedure-listing]
[opts=header,cols="1, 1"]
```

| Description | Operation | Check if a model exists | `gds.beta.model.exists` | Remove a model from memory | `gds.beta.model.drop` | List models | `gds.beta.model.list`

```
== Alpha Tier

.List of all alpha model catalog operations in the GDS library. Functions are written in _italic_. [role=procedure-listing]
[opts=header,cols="1, 1"]
```

| Description | Operation | Store a model | `gds.alpha.model.store` | Load a stored model | `gds.alpha.model.load` | Delete a stored model | `gds.alpha.model.delete` | Publish a model | `gds.alpha.model.publish`

:leveloffset: 2

:leveloffset: +3

= Graph Algorithms

Algorithms exist in one of three tiers of maturity:

* Production-quality **Indicates that the algorithm has been tested with regards to stability and scalability.** Algorithms in this tier are prefixed with `gds.<algorithm>`. * Beta **Indicates that the algorithm is a candidate for the production-quality tier.** Algorithms in this tier are prefixed with `gds.beta.<algorithm>`. * Alpha **Indicates that the algorithm is experimental and might be changed or removed at any time.** Algorithms in this tier are prefixed with `gds.alpha.<algorithm>`.

== Production-quality tier

.List of all production-quality algorithms in the GDS library. Functions are written in *italic*. [role=procedure-listing] [opts=header,cols="1, 1"]

| Algorithm name | Operation .8+<.^|Label Propagation | `gds.labelPropagation.mutate` | `gds.labelPropagation.mutate.estimate` | `gds.labelPropagation.write` | `gds.labelPropagation.write.estimate` | `gds.labelPropagation.stream` | `gds.labelPropagation.stream.estimate` | `gds.labelPropagation.stats` | `gds.labelPropagation.stats.estimate` .8+<.^|Louvain | `gds.louvain.mutate` | `gds.louvain.mutate.estimate` | `gds.louvain.write` | `gds.louvain.write.estimate` | `gds.louvain.stream` | `gds.louvain.stream.estimate` | `gds.louvain.stats` | `gds.louvain.stats.estimate` .8+<.^|Node Similarity | `gds.nodeSimilarity.mutate` | `gds.nodeSimilarity.mutate.estimate` | `gds.nodeSimilarity.write` | `gds.nodeSimilarity.write.estimate` | `gds.nodeSimilarity.stream` | `gds.nodeSimilarity.stream.estimate` | `gds.nodeSimilarity.stats` | `gds.nodeSimilarity.stats.estimate` .8+<.^|PageRank | `gds.pageRank.mutate` | `gds.pageRank.mutate.estimate` | `gds.pageRank.write` | `gds.pageRank.write.estimate` | `gds.pageRank.stream` | `gds.pageRank.stream.estimate` | `gds.pageRank.stats` | `gds.pageRank.stats.estimate` .8+<.^|Weakly Connected Components | `gds.wcc.mutate` | `gds.wcc.mutate.estimate` | `gds.wcc.write` | `gds.wcc.write.estimate` | `gds.wcc.stream` | `gds.wcc.stream.estimate` | `gds.wcc.stats` | `gds.wcc.stats.estimate` .8+<.^| Triangle Count | `gds.triangleCount.stream` | `gds.triangleCount.stream.estimate` | `gds.triangleCount.stats` | `gds.triangleCount.stats.estimate` | `gds.triangleCount.write` | `gds.triangleCount.write.estimate` | `gds.triangleCount.mutate` | `gds.triangleCount.mutate.estimate` .8+<.^| Local Clustering Coefficient | `gds.localClusteringCoefficient.stream` | `gds.localClusteringCoefficient.stream.estimate` | `gds.localClusteringCoefficient.stats` | `gds.localClusteringCoefficient.stats.estimate` | `gds.localClusteringCoefficient.write` | `gds.localClusteringCoefficient.write.estimate` | `gds.localClusteringCoefficient.mutate` | `gds.localClusteringCoefficient.mutate.estimate` .8+<.^| Betweenness Centrality | `gds.betweenness.stream` | `gds.betweenness.stream.estimate` | `gds.betweenness.stats` | `gds.betweenness.stats.estimate` | `gds.betweenness.mutate` | `gds.betweenness.mutate.estimate` | `gds.betweenness.write` | `gds.betweenness.write.estimate` .8+<.^| Fast Random Projection | `gds.fastRP.mutate` | `gds.fastRP.mutate.estimate` | `gds.fastRP.stats` | `gds.fastRP.stats.estimate` | `gds.fastRP.stream` | `gds.fastRP.stream.estimate` | `gds.fastRP.write` | `gds.fastRP.write.estimate` .8+<.^| Degree Centrality | `gds.degree.mutate` | `gds.degree.mutate.estimate` | `gds.degree.stats` | `gds.degree.stats.estimate` | `gds.degree.stream` | `gds.degree.stream.estimate` | `gds.degree.write` | `gds.degree.write.estimate` .8+<.^|ArticleRank | `gds.articleRank.mutate` | `gds.articleRank.mutate.estimate` | `gds.articleRank.write` | `gds.articleRank.write.estimate` | `gds.articleRank.stream` | `gds.articleRank.stream.estimate` | `gds.articleRank.stats` | `gds.articleRank.stats.estimate` .8+<.^|Eigenvector | `gds.eigenvector.mutate` |

gds.eigenvector.mutate.estimate | gds.eigenvector.write | gds.eigenvector.write.estimate |
gds.eigenvector.stream | gds.eigenvector.stream.estimate | gds.eigenvector.stats |
gds.eigenvector.stats.estimate .6+<.^| Shortest Path Dijkstra | gds.shortestPath.dijkstra.stream |
gds.shortestPath.dijkstra.stream.estimate | gds.shortestPath.dijkstra.write |
gds.shortestPath.dijkstra.write.estimate | gds.shortestPath.dijkstra.mutate |
gds.shortestPath.dijkstra.mutate.estimate .6+<.^| All Shortest Paths Dijkstra |
gds.allShortestPaths.dijkstra.stream | gds.allShortestPaths.dijkstra.stream.estimate |
gds.allShortestPaths.dijkstra.write | gds.allShortestPaths.dijkstra.write.estimate |
gds.allShortestPaths.dijkstra.mutate | gds.allShortestPaths.dijkstra.mutate.estimate .6+<.^|
Shortest Paths Yens | gds.shortestPath.yens.stream | gds.shortestPath.yens.stream.estimate |
gds.shortestPath.yens.write | gds.shortestPath.yens.write.estimate | gds.shortestPath.yens.mutate
| gds.shortestPath.yens.mutate.estimate .6+<.^| Shortest Path AStar | gds.shortestPath.astar.stream |
gds.shortestPath.astar.stream.estimate | gds.shortestPath.astar.write |
gds.shortestPath.astar.write.estimate | gds.shortestPath.astar.mutate |
gds.shortestPath.astar.mutate.estimate

== Beta tier

.List of all beta algorithms in the GDS library. Functions are written in *italic*. [role=procedure-listing] [opts=header,cols="1, 1"]

|Algorithm name | Operation .8+<.^|FastRPExtended | gds.beta.fastRPExtended.mutate |
gds.beta.fastRPExtended.mutate.estimate | gds.beta.fastRPExtended.stats |
gds.beta.fastRPExtended.stats.estimate | gds.beta.fastRPExtended.stream |
gds.beta.fastRPExtended.stream.estimate | gds.beta.fastRPExtended.write |
gds.beta.fastRPExtended.write.estimate .8+<.^|GraphSAGE | gds.beta.graphSage.stream |
gds.beta.graphSage.stream.estimate | gds.beta.graphSage.mutate |
gds.beta.graphSage.mutate.estimate | gds.beta.graphSage.write | gds.beta.graphSage.write.estimate
| gds.beta.graphSage.train | gds.beta.graphSage.train.estimate .8+<.^|K1Coloring |
gds.beta.k1coloring.mutate | gds.beta.k1coloring.mutate.estimate | gds.beta.k1coloring.stats |
gds.beta.k1coloring.stats.estimate | gds.beta.k1coloring.stream |
gds.beta.k1coloring.stream.estimate | gds.beta.k1coloring.write |
gds.beta.k1coloring.write.estimate .8+<.^|K-Nearest Neighbors | gds.beta.knn.mutate |
gds.beta.knn.mutate.estimate | gds.beta.knn.stats | gds.beta.knn.stats.estimate |
gds.beta.knn.stream | gds.beta.knn.stream.estimate | gds.beta.knn.write |
gds.beta.knn.write.estimate .6+<.^| Modularity Optimization |
gds.beta.modularityOptimization.mutate | gds.beta.modularityOptimization.mutate.estimate |
gds.beta.modularityOptimization.stream | gds.beta.modularityOptimization.stream.estimate |
gds.beta.modularityOptimization.write | gds.beta.modularityOptimization.write.estimate
.6+<.^|Node2Vec | gds.beta.node2vec.mutate | gds.beta.node2vec.mutate.estimate |
gds.beta.node2vec.stream | gds.beta.node2vec.stream.estimate | gds.beta.node2vec.write |
gds.beta.node2vec.write.estimate

== Alpha tier

.List of all alpha algorithms in the GDS library. Functions are written in *italic*. [role=procedure-listing] [opts=header,cols="1, 1"]

|Algorithm name | Operation .1+<.^|All Shortest Paths | gds.alpha.allShortestPaths.stream

.4+<.^|Approximate Maximum k-cut | gds.alpha.maxkcut.mutate | gds.alpha.maxkcut.mutate.estimate | gds.alpha.maxkcut.stream | gds.alpha.maxkcut.stream.estimate .1+<.^|Breadth First Search | gds.alpha.bfs.stream .4+<.^|Closeness Centrality | gds.alpha.closeness.stream | gds.alpha.closeness.write | gds.alpha.closeness.harmonic.stream | gds.alpha.closeness.harmonic.write .1+<.^|Collapse Path | gds.alpha.collapsePath.mutate .1+<.^|Depth First Search | gds.alpha.dfs.stream .8+<.^|HITS | gds.alpha.hits.mutate | gds.alpha.hits.mutate.estimate | gds.alpha.hits.stats | gds.alpha.hits.stats.estimate | gds.alpha.hits.stream | gds.alpha.hits.stream.estimate | gds.alpha.hits.write | gds.alpha.hits.write.estimate .1+<.^|Random Walk | gds.alpha.randomWalk.stream .2+<.^|Strongly Connected Components | gds.alpha.scc.stream | gds.alpha.scc.write .2+<.^|Single Source Shortest Path | gds.alpha.shortestPath.deltaStepping.write | gds.alpha.shortestPath.deltaStepping.stream .2+<.^|Scale Properties | gds.alpha.scaleProperties.mutate | gds.alpha.scaleProperties.stream .4+<.^|Cosine Similarity | gds.alpha.similarity.cosine.stats | gds.alpha.similarity.cosine.stream | gds.alpha.similarity.cosine.write | gds.alpha.similarity.cosine .5+<.^|Euclidean Similarity | gds.alpha.similarity.euclidean.stats | gds.alpha.similarity.euclidean.stream | gds.alpha.similarity.euclidean.write | gds.alpha.similarity.euclidean | gds.alpha.similarity.euclideanDistance | Jaccard Similarity | gds.alpha.similarity.jaccard .4+<.^|Overlap Similarity | gds.alpha.similarity.overlap.stats | gds.alpha.similarity.overlap.stream | gds.alpha.similarity.overlap.write | gds.alpha.similarity.overlap .4+<.^|Pearson Similarity | gds.alpha.similarity.pearson.stats | gds.alpha.similarity.pearson.stream | gds.alpha.similarity.pearson.write | gds.alpha.similarity.pearson .8+<.^|Speaker-Listener Label Propagation | gds.alpha.sllpa.mutate | gds.alpha.sllpa.mutate.estimate | gds.alpha.sllpa.stats | gds.alpha.sllpa.stats.estimate | gds.alpha.sllpa.stream | gds.alpha.sllpa.stream.estimate | gds.alpha.sllpa.write | gds.alpha.sllpa.write.estimate .5+<.^|Spanning Tree | gds.alpha.spanningTree.write | gds.alpha.spanningTree.kmax.write | gds.alpha.spanningTree.kmin.write | gds.alpha.spanningTree.maximum.write | gds.alpha.spanningTree.minimum.write .2+<.^|Approximate Nearest Neighbours | gds.alpha.ml.ann.stream | gds.alpha.ml.ann.write .8+<.^| Link Prediction | gds.alpha.ml.linkPrediction.predict.mutate | gds.alpha.ml.linkPrediction.predict.mutate.estimate | gds.alpha.ml.linkPrediction.predict.stream | gds.alpha.ml.linkPrediction.predict.stream.estimate | gds.alpha.ml.linkPrediction.predict.write | gds.alpha.ml.linkPrediction.predict.write.estimate | gds.alpha.ml.linkPrediction.train | gds.alpha.ml.linkPrediction.train.estimate .7+<.^| Link Prediction Pipeline | gds.alpha.ml.pipeline.linkPrediction.create | gds.alpha.ml.pipeline.linkPrediction.addNodeProperty | gds.alpha.ml.pipeline.linkPrediction.addFeature | gds.alpha.ml.pipeline.linkPrediction.configureParams | gds.alpha.ml.pipeline.linkPrediction.configureSplit | gds.alpha.ml.pipeline.linkPrediction.train | gds.alpha.ml.pipeline.linkPrediction.predict.mutate | Adamic Adar | gds.alpha.linkprediction.adamicAdar | Common Neighbors | gds.alpha.linkprediction.commonNeighbors | Preferential Attachment | gds.alpha.linkprediction.preferentialAttachment | Preferential Attachment | gds.alpha.linkprediction.resourceAllocation | Same Community | gds.alpha.linkprediction.sameCommunity | Total Neighbors | gds.alpha.linkprediction.totalNeighbors .8+<.^| Node Classification | gds.alpha.ml.nodeClassification.predict.mutate | gds.alpha.ml.nodeClassification.predict.mutate.estimate | gds.alpha.ml.nodeClassification.predict.stream |

gds.alpha.ml.nodeClassification.predict.stream.estimate |
gds.alpha.ml.nodeClassification.predict.write |
gds.alpha.ml.nodeClassification.predict.write.estimate | gds.alpha.ml.nodeClassification.train |
gds.alpha.ml.nodeClassification.train.estimate .1+<.^| Split Relationships |
gds.alpha.ml.splitRelationships.mutate .1+<.^| Triangle Listing | gds.alpha.triangles .1+<.^| Influence Maximization - Greedy | gds.alpha.influenceMaximization.greedy.stream .1+<.^| Influence Maximization - CELF | gds.alpha.influenceMaximization.celf.stream

```
:leveloffset: 2

:leveloffset: +3

= Additional Operations

.List of all additional operations. Functions are written in _italic_. [role=procedure-listing] [opts=header,cols="1, 1"]
```

| Description | Operation | List all operations in GDS | gds.list | List logged progress |
gds.beta.listProgress | The version of the installed GDS | gds.version .2+<.^| Node id functions |
gds.util.asNode | gds.util.asNodes .4+<.^| Numeric Functions | gds.util.NaN | gds.util.infinity |
gds.util.isFinite | gds.util.isInfinite | Accessing a node property in a named graph |
gds.util.nodeProperty | One Hot Encoding | gds.alpha.ml.oneHotEncoding | Status of the system |
gds.debug.sysInfo | Create an impermanent database backed by a named in-memory graph |
gds.alpha.create.cypherdb | Get an overview of the system's workload and available resources |
gds.alpha.systemMonitor

:leveloffset: 2

:leveloffset: +2

:description: If you have previously used Graph Algorithm v3.5, you can find the information you will need to migrate to using the Graph Data Science library in this section. [appendix] = Migration from Graph Algorithms v3.5

[abstract] — If you have previously used Graph Algorithm v3.5, you can find the information you will need to migrate to using the Graph Data Science library in this section. — == Who should read this guide

This documentation is intended for users who are familiar with the Graph Algorithms library. We assume that most of the mentioned operations and concepts can be understood with little explanation. Thus we are intentionally brief in the examples and comparisons. Please see the dedicated chapters in this manual for details on all the features in the Graph Data Science library.

== Syntax Changes

In this section we will focus on side-by-side examples of operations using the syntax of the Graph Algorithms library and Graph Data Science library, respectively.

This section is divided into the following sub-sections:

* Common Changes * Memory estimation * Graph creation - Named Graph * Graph creation - Cypher Queries * Graph listing * Graph info * Graph removal * Production-quality algorithms

:leveloffset: 2

:leveloffset: +3

= Common changes

This section describes changes between Graph Algorithms library and Graph Data Science library that are common to all procedures.

.Namespace [opts=header,cols="1,1"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 |algo.* | gds.*

.Changes in Parameters [opts=header,cols="1,1,1"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 Named Graph |Graph Data Science v1.0 Anonymous Graph | - | `graphName` | `graphConfiguration` | node label [3] | - | - | relationship type [4] | - | - | `direction` | - | - | `config` | `configuration` | -

.Changes in configuration parameter map [opts=header,cols="1,1"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 |`write: true` | Replaced by dedicated `write` mode |`graph: 'cypher'\|'huge'` | Removed. Always using `huge` graph [5] |`direction` | Replaced by `projection` parameter of `relationshipProjection` |`direction: 'OUTGOING'` | `orientation: 'NATURAL'` |`direction: 'INCOMING'` | `orientation: 'REVERSE'` |`direction: 'BOTH'` | Removed [6] |`undirected: true` | Replaced by `orientation: 'UNDIRECTED'` parameter of `relationshipProjection` |`duplicateRelationships` | Replaced by `aggregation` parameter of `relationshipProjection` |`duplicateRelationships: 'SKIP'` | `aggregation: 'SINGLE'` |`iterations` | `maxIterations`

:leveloffset: 2

:leveloffset: +3

= Memory estimation

.Changes in the YIELD fields [opts=header,cols="1,1"]

|Graph Algorithms v3.5 |Graph Data Science v1.0
| `requiredMemory` | `requiredMemory`
| `bytesMin` | `bytesMin`
| `bytesMax` | `bytesMax`
| `mapView` | `mapView`
| - | `treeView`
| - | `nodeCount`
| - | `relationshipCount`

The most significant change in memory estimation is that in GDS to estimate an operation you suffix it with `.estimate` while in GA the operation had to be passed as parameter to `algo.memrec`.

.Estimating the memory requirements of loading a named graph: [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| Native Projections: |

```
CALL algo.memrec(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  'graph.load'
)
```

|

```
CALL gds.graph.create.estimate(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

2+| Cypher Projections: |

```
CALL algo.memrec(
  'MATCH (n:MyLabel) RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
   RETURN id(s) AS source, id(t) AS target',
  'graph.load',
  {
    graph: 'cypher'
  }
)
```

|

```
CALL gds.graph.create.cypher.estimate(
  'MATCH (n:MyLabel) RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
   RETURN id(s) AS source, id(t) AS target'
)
```

:leveloffset: 2

:leveloffset: +3

= Graph creation - Named Graph

.Changes in the YIELD fields [opts=header, cols="1,1"]

====

Graph Algorithms v3.5

Graph Data Science v1.0

`name`

`graphName`

`graph`

-

`direction`

-

`undirected`

-

`sorted`

-

`nodes`

`nodesCount`

`loadMillis`

`createMillis`

`alreadyLoaded`

-

`nodeProperties`

-

`relationshipProperties`

`relationshipCount`

`relationshipWeight`

-

`loadNodes`

-

`loadRelationships`

-

-

`nodeProjection`

-
`relationshipProjection`
====

.Loading a named graph in the default way: [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| Minimal Native Projection: |

```
CALL algo.graph.load(
  'myGraph',
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

|

```
CALL gds.graph.create(
  'myGraph',
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

2+| Native Projection with additional properties: |

```
CALL algo.graph.load(
  'myGraph',
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  {
    concurrency: 4,
    graph: 'huge',
    direction: 'INCOMING'
  }
)
```

|

```
CALL gds.graph.create(
  'myGraph',
  'MyLabel',
  {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'REVERSE'
    }
  },
  {
    readConcurrency: 4
  }
)
```

2+|Native Projection with `direction: 'BOTH'`: |

```
CALL algo.graph.load(
  'myGraph',
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  {
    graph: 'huge',
    direction: 'BOTH'
  }
)
```

|

```
CALL gds.graph.create(
  'myGraph',
  'MyLabel',
  {
    MY_RELATIONSHIP_TYPE_NATURAL: {
      type: 'MY_RELATIONSHIP_TYPE',
      orientation: 'NATURAL'
    },
    MY_RELATIONSHIP_TYPE_REVERSE: {
      type: 'MY_RELATIONSHIP_TYPE',
      orientation: 'REVERSE'
    }
  }
)
```

2+| Undirected Native Projection: |

```
CALL algo.graph.load(
  'myGraph',
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  {
    graph: 'huge',
    undirected: true
  }
)
```

|

```
CALL gds.graph.create(
  'myGraph',
  'MyLabel',
  {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'UNDIRECTED'
    }
  }
)
```

```
:leveloffset: 2

:leveloffset: +3

= Graph creation - Cypher Queries

.Loading a named graph using Cypher queries: [opts=header,cols="1a,1a"]
```

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| Basic Cypher queries, defining source and target: |

```
CALL algo.graph.load(
  'myGraph',
  'MATCH (n:MyLabel)
   RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
   RETURN id(s) AS source, id(t) AS target',
  {
    graph: 'cypher'
  }
)
```

|

```
CALL gds.graph.create.cypher(
  'myGraph',
  'MATCH (n:MyLabel)
   RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
   RETURN id(s) AS source, id(t) AS target'
)
```

2+| With concurrency property and Cypher query with relationship property: |

```
CALL algo.graph.load(
  'myGraph',
  'MATCH (n:MyLabel)
   RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
   RETURN
     id(s) AS source,
     id(t) AS target,
     r.myProperty AS weight',
  {
    concurrency: 4,
    graph: 'cypher'
  }
)
```

|

```
CALL gds.graph.create.cypher(
  'myGraph',
  'MATCH (n:MyLabel)
   RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
   RETURN
     id(s) AS source,
     id(t) AS target,
     r.myProperty AS weight',
  {
    readConcurrency: 4
  }
)
```

2+| Parallel loading: |

```
CALL algo.graph.load(
  'myGraph',
  'MATCH (n:MyLabel)
   WITH * SKIP $skip LIMIT $limit
   RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
   WITH * SKIP $skip LIMIT $limit
   RETURN
     id(s) AS source,
     id(t) AS target,
     r.myProperty AS weight',
  {
    concurrency: 4,
    graph: 'cypher'
  }
)
```

| -

```
:leveloffset: 2

:leveloffset: +3

= Graph listing

.Changes in the YIELD fields [opts=header,cols="1,1"]
```

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `name` | `graphName` | `nodes` | `nodeCount` | `relationships` | `relationshipCount` | `type` | - | `direction` | - | - | `nodeProjection` [7] | - | `relationshipProjection` [7] | - | `nodeQuery` [8] | - | `relationshipQuery` [8] | - | `degreeDistribution` [9]

```
.Listing named graphs: [opts=header,cols="1a,1a"]
```

|Graph Algorithms v3.5 |Graph Data Science v1.0 |

```
CALL algo.graph.list()
```

|

```
CALL gds.graph.list()
```

```
:leveloffset: 2

:leveloffset: +3

= Graph info

.Changes in the YIELD fields [opts=header,cols="1,1"]
```

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `name` | `graphName` | `nodes` | `nodeCount` | `relationships` | `relationshipCount` | `exists` | - | `removed` | - | `type` | - | `direction` | - | - | `nodeProjection` [10] | - | `relationshipProjection` [10] | - | `nodeQuery` [11] | - | `relationshipQuery` [11] | - | `degreeDistribution` [12] | `min, max, mean, p50, p75, p90, p95, p99, p999` [13] | -

```
.Viewing information about a specific named graph: [opts=header,cols="1a,1a"]
```

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| View information for a Named graph: |

```
CALL algo.graph.info('myGraph')
```

|

```
CALL gds.graph.list('myGraph')
```

2+| Check graph existence: |

```
CALL algo.graph.info('myGraph') YIELD exists
```

|

```
CALL gds.graph.exists('myGraph') YIELD exists
```

2+| View graph statistics: |

```
CALL algo.graph.info('myGraph', true)
YIELD min, max, mean, p50
```

|

```
CALL gds.graph.list('myGraph')
YIELD degreeDistribution AS dd
RETURN dd.min, dd.max, dd.mean, dd.p50
```

:leveloffset: 2

:leveloffset: +3

= Removing named graphs

.Changes in the YIELD fields [opts=header,cols="1,1"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `name` | `graphName` | `nodes` | `nodeCount` | `relationships` | `relationshipCount` | `exists` | - | `removed` | - | `type` | - | `direction` | - | - | `nodeProjection` [14] | - | `relationshipProjection` [14] | - | `nodeQuery` [15] | - | `relationshipQuery` [15] | - | `degreeDistribution`

.Removing a named graph: [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 |

```
CALL algo.graph.remove('myGraph')
```

|

```
CALL gds.graph.drop('myGraph')
```

:leveloffset: 2

:leveloffset: +3

= Production-ready algorithms

This section covers all algorithms that have been migrated to the production-ready tier of the Neo4j Graph Data Science library. Syntax changes in configuration, return columns, and execution modes are illustrated with side-by-side examples of queries.

* Label Propagation * Louvain * Node Similarity * PageRank * Weakly Connected Components * Triangle Count / Clustering Coefficient * Betweenness Centrality (exact and sampled)

:leveloffset: 2

:leveloffset: +4

= Label Propagation

.Changes in Configuration [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `direction` | - | `iterations` | `maxIterations` | `concurrency` | `concurrency` | `readConcurrency` | `readConcurrency` [16] | `writeConcurrency` | `writeConcurrency` [17] | `weightProperty` [18] | - | - | `nodeWeightProperty` | - | `relationshipWeightProperty` | `seedProperty` | `seedProperty` | `partitionProperty` | - | `writeProperty` | `writeProperty` [17] | `write` | - | `graph` | -

.Changes in YIELD fields [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 |`loadMillis` |`createMillis` |`computeMillis` |`computeMillis` |`writeMillis` |`writeMillis` |`postProcessingMillis` |`postProcessingMillis` |`nodes` |`nodePropertiesWritten` |`communityCount` |`communityCount` |`didConverge` |`didConverge` |- |`ranIterations` |`write` |- |- |`communityDistribution` |- |`configuration` [19] |`writeProperty` [20] |- |`weightProperty` [21] |- | `min`, `max`, `mean`, `p50`, `p75`, `p90`, `p95`, `p99`, `p999` [22] | -

.Label Propagation Stream Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| Streaming over a named graph: |

```
CALL algo.labelPropagation.stream(null, null, {graph: 'myGraph'})
YIELD nodeId, label
```

|

```
CALL gds.labelPropagation.stream('myGraph')
YIELD nodeId, communityId
```

2+| Streaming over a named graph using configuration for iterations and relationship weight property: |

```
CALL algo.labelPropagation.stream(
  null,
  null,
  {
    graph: 'myGraph',
    iterations: 15,
    weightProperty: 'myWeightProperty'
  }
)
```

|

```
CALL gds.labelPropagation.stream(
  'myGraph',
  {
    maxIterations: 15,
    relationshipWeightProperty: 'myWeightProperty'
  }
)
```

2+| Streaming over anonymous graph: |

```
CALL algo.labelPropagation.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

|

```
CALL gds.labelPropagation.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: 'MY_RELATIONSHIP_TYPE'
})
```

2+| Streaming over anonymous graph using relationship with `REVERSE` orientation: |

```
CALL algo.labelPropagation.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'INCOMING' }
)
```

|

```
CALL gds.labelPropagation.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'REVERSE'
    }
  }
})
```

2+| Streaming over anonymous graph using two way relationships [23]: |

```
CALL algo.labelPropagation.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'BOTH' }
)
```

|

```
CALL gds.labelPropagation.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE_NATURAL: {
      type: 'MY_RELATIONSHIP_TYPE',
      orientation: 'NATURAL'
    },
    MY_RELATIONSHIP_TYPE_REVERSE: {
      type: 'MY_RELATIONSHIP_TYPE',
      orientation: 'REVERSE'
    }
  }
})
```

.Label Propagation Write Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+|Minimalistic write: |

```
CALL algo.labelPropagation(
  null,
  null,
  {
    graph: 'myGraph',
    writeProperty: 'myWriteProperty',
    write: true
  }
)
YIELD
  writeMillis,
  iterations,
  p1,
  writeProperty
```

|

```
CALL gds.labelPropagation.write(
  'myGraph',
  { writeProperty: 'myWriteProperty' }
)
YIELD
  writeMillis,
  ranIterations,
  communityDistribution AS cd,
  configuration AS conf
RETURN
  writeMillis,
  ranIterations,
  cd.p1 AS p1,
  conf.writeProperty AS writeProperty
```

2+|Write using weight properties [24]: |

```
CALL algo.labelPropagation(
  null,
  null,
  {
    graph: 'myGraph',
    writeProperty: 'myWriteProperty',
    weightProperty: 'myRelationshipWeightProperty',
    write: true
  }
)
```

|

```
CALL gds.labelPropagation.write(
  'myGraph',
  {
    writeProperty: 'myWriteProperty',
    relationshipWeightProperty: 'myRelationshipWeightProperty',
    nodeWeightProperty: 'myNodeWeightProperty'
  }
)
```

2+|Memory estimation of the algorithm: |

```
CALL algo.memrec(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  'labelPropagation',
  {
    writeProperty: 'myWriteProperty',
    weightProperty: 'myRelationshipWeightProperty',
    write: true
  }
)
```

|

```
CALL gds.labelPropagation.write.estimate(
  {
    nodeProjection: 'MyLabel',
    relationshipProjection: 'MY_RELATIONSHIP_TYPE',
    writeProperty: 'myWriteProperty',
    relationshipWeightProperty: 'myRelationshipWeightProperty',
    nodeWeightProperty: 'myNodeWeightProperty'
  }
)
```

:leveloffset: 2

:leveloffset: +4

= Louvain

.Changes in Configuration [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `direction` | - | `levels` | `maxLevels` | `concurrency` | `concurrency` | `readConcurrency` | `readConcurrency` [25] | `writeConcurrency` | `writeConcurrency` [26] | `weightProperty` | `relationshipWeightProperty` | `seedProperty` | `seedProperty` | `innerIterations` | `maxIterations` | `includeIntermediateCommunities` | `includeIntermediateCommunities` | `tolerance` | `tolerance` | `writeProperty` | `writeProperty` [26] | `write` | - | `graph` | -

.Changes in YIELD fields [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `loadMillis` | `createMillis` | `computeMillis` | `computeMillis` | `writeMillis` | `writeMillis` | `postProcessingMillis` | `postProcessingMillis` | `nodes` | `nodePropertiesWritten` | `communityCount` | `communityCount` | `levels` | `ranLevels` | `nodeId` | `nodeId` [27] | `community` | `communityId` [27] | `communities` | `intermediateCommunityIds` [27] | `modularity` | `modularity` [28] | `modularities` | `modularities` [28] | `write` | - | - | `communityDistribution` | - | `configuration` [29] | `includeIntermediateCommunities` [30] | - | `writeProperty` [30] | - | `weightProperty` [31] | - | `min, max, mean, p50, p75, p90, p95, p99, p999` [32] | -

.Louvain Stream Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+|Minimalistic streaming over named graph: |

```
CALL algo.beta.louvain.stream(null, null, {graph: 'myGraph'})
YIELD nodeId, community, communities
```

|

```
CALL gds.louvain.stream('myGraph')
YIELD nodeId, communityId, intermediateCommunityIds
```

2+|Streaming over named graph using additional properties - `maxLevels` and `maxIterations`: |

```
CALL algo.beta.louvain.stream(
  null,
  null,
  {
    graph: 'myGraph',
    levels: 15,
    innerIterations: 30
  }
)
```

|

```
CALL gds.louvain.stream(
  'myGraph',
  {
    maxLevels: 15,
    maxIterations: 30
  }
)
```

2+| Streaming over named graph with weight property: |

```
CALL algo.beta.louvain.stream(
  null,
  null,
  {
    graph: 'myGraph',
    weightProperty: 'myWeightProperty'
  }
)
```

|

```
CALL gds.louvain.stream(
  'myGraph',
  {
    relationshipWeightProperty: 'myWeightProperty'
  }
)
```

2+| Minimalistic streaming over anonymous graph: |

```
CALL algo.beta.louvain.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

|

```
CALL gds.louvain.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: 'MY_RELATIONSHIP_TYPE'
})
```

2+| Streaming over anonymous graph with REVERSE relationship orientation: |

```
CALL algo.beta.louvain.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'INCOMING' }
)
```

|

```
CALL gds.louvain.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'REVERSE'
    }
  }
})
```

2+| Streaming over anonymous graph using two way relationships [33]: |

```
CALL algo.louvain.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'BOTH' }
)
```

|

```
CALL gds.louvain.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE_NATURAL: {
      type: 'MY_RELATIONSHIP_TYPE',
      orientation: 'NATURAL'
    },
    MY_RELATIONSHIP_TYPE_REVERSE: {
      type: 'MY_RELATIONSHIP_TYPE',
      orientation: 'REVERSE'
    }
  }
})
```

.Louvain Write Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| Minimalistic write with just `writeProperty`: |

```
CALL algo.beta.louvain(
  null,
  null,
  {
    graph: 'myGraph',
    writeProperty: 'myWriteProperty',
    write: true
  }
)
YIELD
  nodes,
  writeMillis,
  levels,
  iterations,
  p1,
  writeProperty
```

|

```
CALL gds.louvain.write(
  'myGraph',
  { writeProperty: 'myWriteProperty' }
)
YIELD
  nodePropertiesWritten,
  writeMillis,
  ranLevels,
  ranIterations,
  communityDistribution AS cd,
  configuration AS conf
RETURN
  nodePropertiesWritten,
  writeMillis,
  ranLevels,
  ranIterations,
  cd.p1 AS p1,
  conf.writeProperty AS writeProperty
```

2+| Running in `write` mode over weighted graph: |

```
CALL algo.beta.louvain(
  null,
  null,
  {
    graph: 'myGraph',
    writeProperty: 'myWriteProperty',
    weightProperty: 'myWeightProperty',
    write: true
  }
)
```

|

```
CALL gds.louvain.write(
  'myGraph',
  {
    writeProperty: 'myWriteProperty',
    relationshipWeightProperty: 'myWeightProperty'
  }
)
```

2+| Memory estimation of the algorithm: |

```
CALL algo.memrec(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  'beta.louvain',
  {
    writeProperty: 'myWriteProperty',
    weightProperty: 'myRelationshipWeightProperty',
    write: true
  }
)
```

|

```
CALL gds.louvain.write.estimate(
  {
    nodeProjection: 'MyLabel',
    relationshipProjection: 'MY_RELATIONSHIP_TYPE',
    writeProperty: 'myWriteProperty',
    relationshipWeightProperty: 'myWeightProperty'
  }
)
```

:leveloffset: 2

:leveloffset: +4

= Node Similarity

.Changes in Configuration [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `direction` | - | `concurrency` | `concurrency` | `readConcurrency` | `readConcurrency` [34] | `writeConcurrency` | `writeConcurrency` [35] | `topK` | `topK` | `bottomK` | `bottomK` | `topN` | `topN` | `bottomN` | `bottomN` | `similarityCutoff` | `similarityCutoff` | `degreeCutoff` | `degreeCutoff` | `writeProperty` | `writeProperty` [35] | `writeRelationshipType` | `writeRelationshipType` [35] | `write` | - | `graph` | -

.Changes in YIELD fields [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `loadMillis` | `createMillis` | `computeMillis` | `computeMillis` | `writeMillis` | `writeMillis` | `postProcessingMillis` | `postProcessingMillis` | `node1` | `node1` [36] | `node2` | `node2` [36] | `similarity` | `similarity` [36] | `nodesCompared` | `nodesCompared` [37] | `relationships` | `relationshipsWritten` [37] | `write` | - | - | `similarityDistribution` | - | `configuration` [38] | `writeProperty` [39] | - | `writeRelationshipType` [39] | - | `min, max, mean, p50, p75, p90, p95, p99, p999` [40] | -

.Node Similarity Stream Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| Minimalistic streaming over named graph: |

```
CALL algo.nodeSimilarity.stream(null, null, {graph: 'myGraph'})
YIELD node1, node2, similarity
```

|

```
CALL gds.nodeSimilarity.stream('myGraph')
YIELD node1, node2, similarity
```

2+| Streaming over named graph using `topK` and `similarityCutoff` configuration properties: |

```
CALL algo.nodeSimilarity.stream(
  null,
  null,
  {
    graph: 'myGraph',
    topK: 1,
    similarityCutoff: 0.5
  }
)
```

|

```
CALL gds.nodeSimilarity.stream(
  'myGraph',
  {
    topK: 1,
    similarityCutoff: 0.5
  }
)
```

2+| Streaming over named graph using `bottomK` configuration property: |

```
CALL algo.nodeSimilarity.stream(
  null,
  null,
  {
    graph: 'myGraph',
    bottomK: 15
  }
)
```

|

```
CALL gds.nodeSimilarity.stream(
  'myGraph',
  {
    bottomK: 15
  }
)
```

2+| Minimalistic streaming over anonymous graph: |

```
CALL algo.nodeSimilarity.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

|

```
CALL gds.nodeSimilarity.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: 'MY_RELATIONSHIP_TYPE'
})
```

2+| Streaming over anonymous graph using `REVERSE` relationship projection: |

```
CALL algo.nodeSimilarity.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'INCOMING' }
)
```

|

```
CALL gds.nodeSimilarity.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'REVERSE'
    }
  }
})
```

2+| Streaming over anonymous graph using two way relationships [41]: |

```
CALL algo.nodeSimilarity.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'BOTH' }
)
```

|

```
CALL gds.nodeSimilarity.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE_NATURAL: {
      type: 'MY_RELATIONSHIP_TYPE',
      orientation: 'NATURAL'
    },
    MY_RELATIONSHIP_TYPE_REVERSE: {
      type: 'MY_RELATIONSHIP_TYPE',
      orientation: 'REVERSE'
    }
  }
})
```

.Node Similarity Write Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+|Minimalistic `write` with `writeRelationshipType` and `writeProperty`: |

```
CALL algo.nodeSimilarity(
  null,
  null,
  {
    graph: 'myGraph',
    writeRelationshipType: 'MY_WRITE_REL_TYPE',
    writeProperty: 'myWriteProperty',
    write: true
  }
)
YIELD
  nodesCompared,
  relationships,
  writeMillis,
  iterations,
  p1,
  writeProperty
```

|

```
CALL gds.nodeSimilarity.write(
  'myGraph',
  {
    writeRelationshipType: 'MY_WRITE_REL_TYPE',
    writeProperty: 'myWriteProperty'
  }
)
YIELD
  nodesCompared,
  relationships,
  writeMillis,
  ranIterations,
  similarityDistribution AS sd,
  configuration AS conf
RETURN
  nodesCompared,
  relationships,
  writeMillis,
  ranIterations,
  sd.p1 AS p1,
  conf.writeProperty AS writeProperty
```

2+| Memory estimation of the algorithm: |

```
CALL algo.memrec(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  'nodeSimilarity',
  {
    writeRelationshipType: 'MY_WRITE_REL_TYPE',
    writeProperty: 'myWriteProperty',
    write: true
  }
)
```

|

```
CALL gds.nodeSimilarity.write.estimate(
  {
    nodeProjection: 'MyLabel',
    relationshipProjection: 'MY_RELATIONSHIP_TYPE',
    writeRelationshipType: 'MY_WRITE_REL_TYPE',
    writeProperty: 'myWriteProperty'
  }
)
```

:leveloffset: 2

:leveloffset: +4

= PageRank

.Changes in Configuration [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `direction` | - | `iterations` | `maxIterations` | `tolerance` | `tolerance` | `dampingFactor` | `dampingFactor` | `concurrency` | `concurrency` | `readConcurrency` | `readConcurrency` [42] | `writeConcurrency` | `writeConcurrency` [43] | `writeProperty` | `writeProperty` [43] | `weightProperty` | `relationshipWeightProperty` | `write` | - | `graph` | -

.Changes in YIELD fields [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `loadMillis` | `createMillis` | `computeMillis` | `computeMillis` | `writeMillis` | `writeMillis` | `postProcessingMillis` | `postProcessingMillis` | `node` | `nodeId` [44] | `score` | `score` [44] | `nodes` | `nodePropertiesWritten` [45] | `iterations` | `ranIterations` | `write` | - | - | `configuration` [46] | `writeProperty` [47] | - | `dampingFactor` [47] | - | `tolerance` [47] | - | `weightProperty` [48] | -

.PageRank Stream Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+|Minimalistic stream over named graph: |

```
CALL algo.pageRank.stream(null, null, {graph: 'myGraph'})
YIELD nodeId, score
```

|

```
CALL gds.pageRank.stream('myGraph')
YIELD nodeId, score
```

2+| Streaming over named graph with iteration limit: |

```
CALL algo.pageRank.stream(
  null,
  null,
  {
    graph: 'myGraph',
    iterations: 20
  }
)
```

|

```
CALL gds.pageRank.stream(
  'myGraph',
  {
    maxIterations: 20
  }
)
```

2+| Minimalistic streaming over anonymous graph: |

```
CALL algo.pageRank.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

|

```
CALL gds.pageRank.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: 'MY_RELATIONSHIP_TYPE'
})
```

2+| Streaming over anonymous graph with `REVERSE` relationship orientation: |

```
CALL algo.pageRank.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'INCOMING' }
)
```

|

```
CALL gds.pageRank.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'REVERSE'
    }
  }
})
```

2+| Streaming over anonymous graph with relationship weight property, assigning it a default value in case the property doesn't have value: |

```
CALL algo.pageRank.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  {
    weightProperty: 'myWeightProperty',
    defaultValue: 1.5
  }
)
```

|

```
CALL gds.pageRank.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      properties: {
        myWeightProperty: {
          defaultValue: 1.5
        }
      }
    }
  }
})
```

.PageRank Write Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| Running `write` mode on named graph: |

```
CALL algo.pageRank(
  null,
  null,
  {
    graph: 'myGraph',
    writeProperty: 'myWriteProperty',
    write: true
  }
)
YIELD
  nodes,
  loadMillis,
  iterations,
  p1,
  writeProperty
```

|

```
CALL gds.pageRank.write(
  'myGraph',
  {
    writeProperty: 'myWriteProperty'
  }
)
YIELD
  nodePropertiesWritten,
  createMillis,
  ranIterations,
  configuration AS conf
RETURN
  nodePropertiesWritten,
  writeMillis,
  ranIterations,
  conf.writeProperty AS writeProperty
```

2+| Memory estimation of the algorithm: |

```
CALL algo.memrec(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  'pageRank',
  {
    writeProperty: 'myWriteProperty',
    write: true
  }
)
```

|

```
CALL gds.pageRank.write.estimate(
  {
    nodeProjection: 'MyLabel',
    relationshipProjection: 'MY_RELATIONSHIP_TYPE',
    writeProperty: 'myWriteProperty'
  }
)
```

:leveloffset: 2

:leveloffset: +4

= Weakly Connected Components

.Changes in Configuration [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `direction` | - | `concurrency` | `concurrency` | `readConcurrency` | `readConcurrency` [49] | `writeConcurrency` | `writeConcurrency` [50] | `writeProperty` | `writeProperty` [50] | `weightProperty` | `relationshipWeightProperty` | `defaultValue` | `defaultValue` | `seedProperty` | `seedProperty` | `threshold` | `threshold` | `consecutiveIds` | `consecutiveIds` | `write` | - | `graph` | -

.Changes in YIELD fields [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.0 | `loadMillis` | `createMillis` | `computeMillis` | `computeMillis` | `writeMillis` | `writeMillis` | `postProcessingMillis` | `postProcessingMillis` | `nodeId` | `nodeId` [51] | `setId` | `componentId` [51] | `nodes` | `nodePropertiesWritten` [52] | - | `relationshipPropertiesWritten` [52] | `write` | - | - | `componentDistribution` | - | `configuration` [53] | `writeProperty` [54] | - | `weightProperty` [55] | - | `min, max, mean, p50, p75, p90, p95, p99, p999` [56] | -

.Weakly Connected Components Stream Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| Minimalistic stream over named graph: |

```
CALL algo.unionFind.stream(null, null, {graph: 'myGraph'})
YIELD nodeId, setId
```

|

```
CALL gds.wcc.stream('myGraph')
YIELD nodeId, componentId
```

2+| Streaming over weighted named graph: |

```
CALL algo.unionFind.stream(
  null,
  null,
  {
    graph: 'myGraph',
    weightProperty: 'myWeightProperty'
  }
)
```

|

```
CALL gds.wcc.stream(
  'myGraph',
  {
    relationshipWeightProperty: 'myWeightProperty'
  }
)
```

2+| Minimalistic streaming over anonymous graph: |

```
CALL algo.unionFind.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

|

```
CALL gds.wcc.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: 'MY_RELATIONSHIP_TYPE'
})
```

2+| Streaming over anonymous graph with `REVERSE` relationship orientation: |

```
CALL algo.unionFind.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'INCOMING' }
)
```

|

```
CALL gds.wcc.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'REVERSE'
    }
  }
})
```

2+| Streaming over anonymous graph with relationship specifying default value for the weight property: |

```
CALL algo.unionFind.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  {
    graph: 'myGraph',
    weightProperty: 'myWeightProperty',
    defaultValue: 2.0
  }
)
```

|

```
CALL gds.wcc.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      properties: {
        myWeightProperty: {
          defaultValue: 2
        }
      }
    }
  }
})
```

.Weakly Connected Components Write Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.0 2+| Minimalistic `write` mode: |

```
CALL algo.unionFind(
  null,
  null,
  {
    graph: 'myGraph',
    writeProperty: 'myWriteProperty',
    write: true
  }
)
YIELD
  nodes,
  loadMillis,
  p1,
  writeProperty
```

|

```
CALL gds.wcc.write(
  'myGraph',
  { writeProperty: 'myWriteProperty' }
)
YIELD
  nodePropertiesWritten,
  createMillis,
  componentDistribution AS cd,
  configuration AS conf
RETURN
  nodePropertiesWritten,
  createMillis,
  cd.p1 AS p1,
  conf.writeProperty AS writeProperty
```

2+| Running `write` mode over weighted named graph: |

```
CALL algo.unionFind(
  null,
  null,
  {
    graph: 'myGraph',
    writeProperty: 'myWriteProperty',
    weightProperty: 'myWeightProperty',
    write: true
  }
)
```

|

```
CALL gds.wcc.write(
  'myGraph',
  {
    writeProperty: 'myWriteProperty',
    relationshipWeightProperty: 'myWeightProperty'
  }
)
```

2+| Memory estimation of the algorithm: |

```
CALL algo.memrec(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  'unionFind',
  {
    writeProperty: 'myWriteProperty',
    weightProperty: 'myRelationshipWeightProperty',
    write: true
  }
)
```

|

```
CALL gds.wcc.write.estimate(
  {
    nodeProjection: 'MyLabel',
    relationshipProjection: 'MY_RELATIONSHIP_TYPE',
    writeProperty: 'myWriteProperty',
    relationshipWeightProperty: 'myWeightProperty'
  }
)
```

:leveloffset: 2

:leveloffset: +4

= Triangle Counting / Clustering Coefficient

The `alpha` procedures from the namespace `algo.triangleCount` are being replaced by a pair of procedure namespaces:

* gds.triangleCount * gds.localClusteringCoefficient

Everything relating to clustering coefficients has been extracted into a separate algorithm backing `gds.localClusteringCoefficient` procedures. To compute both triangle count and local clustering coefficient values multiple procedures will be necessary.

The triangle enumeration procedure `algo.triangles.stream()` has been renamed to `gds.alpha.triangles()`.

.Common changes in Configuration [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.2 | `direction` | - | `concurrency` | `concurrency` | `readConcurrency` | `readConcurrency` [57] | `writeConcurrency` | `writeConcurrency` [58] | `writeProperty` | `writeProperty` [58] | `write` | - | `graph` | -

.Changes in YIELD fields of `algo.triangleCount` [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.2 | `nodeId` | `nodeId` [59] | `triangles` | `triangleCount` [60] | `triangleCount` | `globalTriangleCount` [61] | `nodeCount` | `nodeCount` [61] | `averageClusteringCoefficient` [62] | - | `clusteringCoefficientProperty` [63] | - | `loadMillis` | `createMillis` | `computeMillis` | `computeMillis` | `writeMillis` | `writeMillis` | `write` | - | - | `configuration` [64] | `writeProperty` [65] | - | `min, max, mean, p50, p75, p90, p95, p99, p999` | -

.TriangleCount Stream Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.2

2+| Streaming triangle counts over named graph: |

```
CALL algo.triangleCount.stream(null, null, {graph: 'myGraph'})
YIELD nodeId, triangles
```

|

```
CALL gds.triangleCount.stream('myGraph')
YIELD nodeId, triangleCount
```

2+| Streaming local clustering coefficients over named graph: |

```
CALL algo.triangleCount.stream(null, null, {graph: 'myGraph'})
YIELD nodeId, coefficient
```

|

```
CALL gds.localClusteringCoefficient.stream('myGraph')
YIELD nodeId, localClusteringCoefficient
```

2+| Streaming both triangle counts and local clustering coefficients: |

```
CALL algo.triangleCount.stream(null, null, {graph: 'myGraph'})
YIELD nodeId, triangles, coefficient
```

|

```
CALL gds.triangleCount.mutate('myGraph', {mutateProperty: 'tc'})
YIELD globalTriangleCount
CALL gds.localClusteringCoefficient.stream(
  'myGraph', {
    triangleCountProperty: 'tc'
}) YIELD nodeId, localClusteringCoefficient
WITH
  nodeId,
  localClusteringCoefficient,
  gds.util.nodeProperty('myGraph', nodeId, 'tc') AS triangleCount
RETURN nodeId, triangleCount, localClusteringCoefficient
```

2+| Streaming triangle counts over anonymous graph: |

```
CALL algo.triangleCount.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

|

```
CALL gds.triangleCount.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'UNDIRECTED'
    }
  }
})
```

.TriangleCount Write Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.2

2+| Writing triangle counts from named graph: |

```
CALL algo.triangleCount(null, null, {
  graph: 'myGraph',
  write: true,
  writeProperty: 'tc'
}) YIELD nodeCount, triangleCount
```

|

```
CALL gds.triangleCount.write('myGraph', {
  writeProperty: 'tc'
}) YIELD nodeCount, globalTriangleCount
```

2+| Writing local clustering coefficients from named graph: |

```
CALL algo.triangleCount(null, null, {
  graph: 'myGraph',
  write: true,
  clusteringCoefficientProperty: 'lcc'
}) YIELD nodeCount, averageClusteringCoefficient
```

|

```
CALL gds.localClusteringCoefficient.write('myGraph', {
  writeProperty: 'lcc'
}) YIELD nodeCount, averageClusteringCoefficient
```

2+| Writing both triangle counts and local clustering coefficients: |

```
CALL algo.triangleCount(null, null, {
  graph: 'myGraph',
  write: true,
  writeProperty: 'tc',
  clusteringCoefficientProperty: 'lcc'
}) YIELD nodeCount, triangleCount, averageClusteringCoefficient
```

|

```
CALL gds.triangleCount.mutate('myGraph', {
  mutateProperty: 'tc'
}) YIELD globalTriangleCount
CALL gds.localClusteringCoefficient.write('myGraph', {
  triangleCountProperty: 'tc',
  writeProperty: 'lcc'
}) YIELD nodeCount, averageClusteringCoefficient
CALL gds.graph.writeNodeProperties('myGraph', ['tc'])
YIELD propertiesWritten
RETURN nodeCount, globalTriangleCount, averageClusteringCoefficient
```

:leveloffset: 2

:leveloffset: +4

= Betweenness Centrality

In Graph Algorithms v3.5, Betweenness Centrality was surfaced in two different procedures:

* `algo.betweenness` and `algo.betweenness.stream` * `algo.betweenness.sampled` and `algo.betweenness.sampled.stream`

These two have been merged into a single procedure `gds.betweenness` with all four execution modes supported. The sampling is controlled via configuration parameter rather than explicit procedures. Setting the sampling size to the node count will produce exact results.

.Changes in Configuration [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.3 | `stats` | - | `strategy` | - | `probability` | - | - |

samplingSize | - | samplingSeed | maxDepth | - | direction | - | concurrency | concurrency | readConcurrency | readConcurrency [66] | writeConcurrency | writeConcurrency [67] | writeProperty | writeProperty [67] | write | - | graph | -

.Changes in YIELD fields [opts=header]

|Graph Algorithms v3.5 |Graph Data Science v1.3 | centrality | score [68] | nodes | - | minCentrality | minimumScore | maxCentrality | maximumScore | sumCentrality | scoreSum | loadMillis | createMillis | evalMillis | computeMillis | writeMillis | writeMillis | - | postProcessingMillis | - | configuration [69] | - | nodePropertiesWritten [70] | write | - | writeProperty [71] | -

.Betweenness Centrality Stream Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.3
2+|Minimalistic stream over named graph:
|

```
CALL algo.betweenness.stream(null, null, {graph: 'myGraph'})
YIELD nodeId, centrality
```

|

```
CALL gds.betweenness.stream('myGraph')
YIELD nodeId, score
```

2+|Minimalistic stream over named graph, sampled:
|

```
CALL algo.betweenness.sampled.stream(null, null, {graph: 'myGraph', probability: 0.5})
YIELD nodeId, centrality
```

|

```
CALL gds.betweenness.stream('myGraph', {samplingSize: 1000}) // assume 2000 nodes
YIELD nodeId, score
```

2+| Minimalistic streaming over anonymous graph:
|

```
CALL algo.betweenness.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

|

```
CALL gds.betweenness.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: 'MY_RELATIONSHIP_TYPE'
})
```

2+| Streaming over anonymous graph with UNDIRECTED relationship orientation:
|

```
CALL algo.betweenness.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { undirected: true }
)
```

|

```
CALL gds.betweenness.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'UNDIRECTED'
    }
  }
})
```

.Betweenness Centrality Write Mode [opts=header,cols="1a,1a"]

|Graph Algorithms v3.5 |Graph Data Science v1.3 2+| Running `write` mode on named graph: |

```
CALL algo.betweenness(
  null,
  null,
  {
    graph: 'myGraph',
    writeProperty: 'myWriteProperty',
    write: true
  }
)
YIELD
  nodes,
  minCentrality,
  maxCentrality,
  sumCentrality,
  loadMillis,
  evalMillis,
  writeMillis
```

|

```
CALL gds.betweenness.write(
  'myGraph',
  {
    writeProperty: 'myWriteProperty'
  }
)
YIELD
  nodePropertiesWritten,
  minimumScore,
  maximumScore,
  scoreSum,
  createMillis,
  computeMillis,
  writeMillis,
  configuration
```

:leveloffset: 2

<<< :leveloffset: +1

[discrete] = License

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

.You are free to

Share:: copy and redistribute the material in any medium or format Adapt:: remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

.Under the following terms

Attribution:: You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. NonCommercial:: You may not use the material for commercial purposes. ShareAlike:: If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions:: You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

.Notices You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

See https://creativecommons.org/licenses/by-nc-sa/4.0/ for further details. The full license text is available at https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.

:leveloffset: 2

[3] Moved to `graphConfiguration` as `nodeProjection`
[4] Moved to `graphConfiguration` as `relationshipProjection`
[5] Graph creation with cypher queries has dedicate `gds.graph.create.cypher` procedure. There are parameters `nodeQuery` and `relationshipQuery` for anonymous graphs
[6] This behaviour can be achieved by creating two relationship projections - one with `orientation: 'NATURAL'` and one with `orientation: 'REVERSE'`
[7] Field will be `null` if a Cypher projection was used
[8] Field will be `null` unless a Cypher projection was used
[9] Graph statistics map, i.e. min, max, percentiles, etc.
[10] Field will be `null` if a Cypher projection was used
[11] Field will be `null` unless a Cypher projection was used
[12] Graph statistics map, i.e. min, max, percentiles, etc.
[13] Inlined into `degreeDistribution`
[14] Field will be `null` if a Cypher projection was used
[15] Field will be `null` unless a Cypher projection was used
[16] Only when using anonymous graph
[17] Only for `write` mode
[18] Can be configured separately by using `nodeWeightProperty` and `relationshipWeightProperty`
[19] The configuration used to run the algorithm
[20] Inlined into `configuration`
[21] Inlined into `configuration` as `nodeWeightProperty` and/or `relationshipWeightProperty`
[22] Inlined into `communityDistribution`
[23] Not algorithm specific. This is also possible with named graphs. The graph has to be created with two relationship projections - one for each direction

[24] In GA the weightProperty is for Nodes and Relationships while in GDS there are separate properties
[25] Only when using anonymous graph
[26] Only for `write` mode
[27] Only for `stream` mode
[28] Only for `write` mode
[29] The configuration used to run the algorithm
[30] Inlined into `configuration`
[31] Inlined into `configuration` as `relationshipWeightProperty`
[32] Inlined into `communityDistribution`
[33] Not algorithm specific. This is also possible with named graphs. The graph has to be created with two relationship projections - one for each direction
[34] Only when using anonymous graph
[35] Only for `write` mode
[36] Only for `stream` mode
[37] Only for `write` mode
[38] The configuration used to run the algorithm
[39] Inlined into `configuration`
[40] Inlined into `similarityDistribution`
[41] Not algorithm specific. This is also possible with named graphs. The graph has to be created with two relationship projections - one for each direction
[42] Only when using anonymous graph
[43] Only for `write` mode
[44] Only for `stream` mode
[45] Only for `write` mode
[46] The configuration used to run the algorithm
[47] Inlined into `configuration`
[48] Inlined into `configuration` as `relationshipWeightProperty`
[49] Only when using anonymous graph
[50] Only for `write` mode
[51] Only for `stream` mode
[52] Only for `write` mode
[53] The configuration used to run the algorithm
[54] Inlined into `configuration`
[55] Inlined into `configuration` as `relationshipWeightProperty`
[56] Inlined into `componentDistribution`
[57] Only when using anonymous graph
[58] Only for `write` mode
[59] Only for `stream` mode
[60] Only for `stream` mode
[61] Not present in `stream` mode
[62] Moved to `gds.localClusteringCoefficient`
[63] Moved as `writeProperty` to `gds.localClusteringCoefficient`
[64] The configuration used to run the algorithm
[65] Inlined into `configuration`
[66] Only when using anonymous graph
[67] Only for `write` mode
[68] Only for `stream` mode
[69] The configuration used to run the algorithm
[70] Only for `write` mode
[71] Inlined into `configuration`