The Neo4j Graph Algorithms User Guide v3.4

Table of Contents

This is the user guide for Neo4j Graph Algorithms version 3.4, authored by *the Neo4j Team.*

The guide covers the following areas:

- Introduction — An introduction to Neo4j Graph Algorithms.
- The Yelp example — An illustration of how to use graph algorithms on a social network of friends.
- Procedures — A list of Neo4j Graph Algorithm procedures.
- Algorithms — A detailed guide to each of the Neo4j Graph Algorithms, including use-cases and examples.

Introduction

This chapter provides an introduction to the available graph algorithms, and instructions for installation and use.

This library provides efficiently implemented, parallel versions of common graph algorithms for Neo4j 3.x, exposed as Cypher procedures.

Releases are available here: https://github.com/neo4j-contrib/neo4j-graph-algorithms/releases

Algorithms

Centralities

These algorithms determine the importance of distinct nodes in a network:

- PageRank (`algo.pageRank`)
- Betweenness Centrality (`algo.betweenness`)
- Closeness Centrality (`algo.closeness`)

Community detection

These algorithms evaluate how a group is clustered or partitioned, as well as its tendency to strengthen or break apart:

- Louvain (`algo.louvain`)
- Label Propagation (`algo.labelPropagation`)
- (Weakly) Connected Components (`algo.unionFind`)
- Strongly Connected Components (`algo.scc`)
- Triangle Count / Clustering Coefficient (`algo.triangleCount`)

Path finding

These algorithms help find the shortest path or evaluate the availability and quality of routes:

- Minimum Weight Spanning Tree (`algo.mst`)
- All Pairs- and Single Source - Shortest Path (`algo.shortestPath`, `algo.allShortestPaths`)
- A* Algorithm (`algo.shortestPath.astar`)
- Yen’s K-Shortest Paths (`algo.kShortestPaths`)

These procedures work either on the whole graph, or on a subgraph filtered by label and relationship-type. You can also use filtering and projection using Cypher queries.

Installation

Download graph-algorithms-algo-[version].jar from the matching release and copy it into your $NEO4J_HOME/plugins directory.

Because the algorithms use the lower level Kernel API to read from, and to write to Neo4j, for security purposes you will also have to enable them in the configuration:

1. Add the following to your $NEO4J_HOME/conf/neo4j.conf file:

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

2. Restart Neo4j
3. To see a list of all the algorithms, run the following query:

   ```
   CALL algo.list()
   ```

Usage

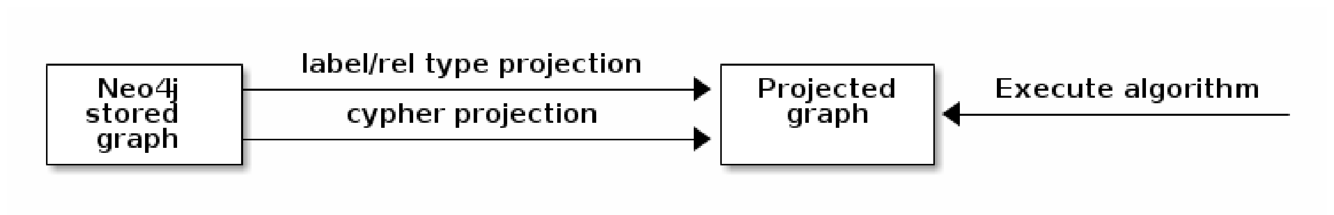
These algorithms are exposed as Neo4j procedures. They can be called directly from Cypher in your Neo4j Browser, from cypher-shell, or from your client code.

For most algorithms there are two procedures:

- algo.<name> - this procedure writes results back to the graph as node-properties, and reports statistics.
- algo.<name>.stream - this procedure returns a stream of data. For example, node-ids and computed values.

  For large graphs, the streaming procedure might return millions, or even billions of results. In this case it may be more convenient to store the results of the algorithm, and then use them with later queries.

We can project the graph we want to run algorithms on with either label and relationship-type projection, or cypher projection.



The projected graph model is separate from Neo4j's stored graph model to enable fast caching for the topology of the graph, containing only relevant nodes, relationships and weights. The projected graph model does not support multiple relationships between a single pair of nodes. During projection, only one relationship between a pair of nodes per direction (in, out) is allowed in the directed case, but two relationships are allowed for BOTH the undirected cases.

Label and relationship-type projection

We can project the subgraph we want to run the algorithm on by using the label parameter to describe nodes, and relationship-type to describe relationships.

The general call syntax is:

```
CALL algo.<name>('NodeLabel', "RelationshipType", {config})
```

For example, PageRank on DBpedia (11M nodes, 116M relationships):

```
CALL algo.pageRank('Page','Link',{iterations:5, dampingFactor:0.85, write: true,
writeProperty:'pagerank'});
// YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, dampingFactor,
write, writeProperty

CALL algo.pageRank.stream('Page','Link',{iterations:5, dampingFactor:0.85})
YIELD node, score
RETURN node.title, score
ORDER BY score DESC LIMIT 10;
```

Huge graph projection

The default label and relationship-type projection has a limitation of 2 billion nodes and 2 billion relationships, so if our project graph is bigger than this we need to use a huge graph projection. This can be enabled by setting `graph:'huge'` in the config.

The general call syntax is:

```
CALL algo.<name>('NodeLabel', "RelationshipType", {graph: "huge"})
```

For example, PageRank on DBpedia:

```
CALL algo.pageRank('Page','Link',{iterations:5, dampingFactor:0.85,
writeProperty:'pagerank',graph:'huge'});
YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, dampingFactor,
writeProperty
```

Cypher projection

If label and relationship-type projection is not selective enough to describe our subgraph to run the algorithm on, we can use Cypher statements to project subsets of our graph. Use a node-statement instead of the label parameter and a relationship-statement instead of the relationship-type, and use `graph:'cypher'` in the config.

Relationships described in the relationship-statement will only be projected if both source and target nodes are described in the node-statement. Relationships that don't have both source and target nodes described in the node-statement will be ignored.

We can also return a property value or weight (according to our config) in addition to the ids from these statements.

Cypher projection enables us to be more expressive in describing our subgraph that we want to analyse, but might take longer to project the graph with more complex cypher queries.

The general call syntax is:

```
CALL algo.<name>(
  'MATCH (n) RETURN id(n) AS id',
  "MATCH (n)-->(m) RETURN id(n) AS source, id(m) AS target",
  {graph: "cypher"})
```

For example, PageRank on DBpedia:

```
CALL algo.pageRank(
'MATCH (p:Page) RETURN id(p) as id',
'MATCH (p1:Page)-[:Link]->(p2:Page) RETURN id(p1) as source, id(p2) as target',
{graph:'cypher', iterations:5, write: true});
```

Cypher projection can also be used to project a virtual (non-stored) graph. Here is an example of how to project an undirected graph of people who visited the same web page and run the Louvain community detection algorithm on it, using the number of common visited web pages between pairs of people as relationship weight:

```
CALL algo.louvain(
'MATCH (p:Person) RETURN id(p) as id',
'MATCH (p1:Person)-[:Visit]->(:Page)<-[:Visit]-(p2:Person)
RETURN id(p1) as source, id(p2) as target, count(*) as weight',
{graph:'cypher', iterations:5, write: true});
```

Graph loading

As it can take some time to load large graphs into the algorithm data structures, you can pre-load graphs and then later refer to them by name for several graph algorithms. After usage they can be removed from memory to free resources used:

```
// Load graph
CALL algo.graph.load('my-graph','Label','REL_TYPE',{graph:'heavy',..other config...})
  YIELD name, graph, direction, undirected, sorted, nodes, loadMillis, alreadyLoaded,
        nodeWeight, relationshipWeight, nodeProperty, loadNodes, loadRelationships;

// Info on loaded graph
CALL algo.graph.info('my-graph')
  YIELD name, type, exists, removed, nodes;

// Use graph
CALL algo.pageRank(null,null,{graph:'my-graph',...})


// Remove graph
CALL algo.graph.remove('my-graph')
  YIELD name, type, exists, removed, nodes;
```

Building locally

Currently aiming at Neo4j 3.x (with a branch per version):

```
git clone https://github.com/neo4j-contrib/neo4j-graph-algorithms
cd neo4j-graph-algorithms
git checkout 3.3
mvn clean install
cp algo/target/graph-algorithms-*.jar $NEO4J_HOME/plugins/
$NEO4J_HOME/bin/neo4j restart
```

The Yelp example

This chapter introduces the Yelp Open Dataset that is used throughout to exemplify how the Neo4j Graph Algorithms work.

The Yelp Open Dataset

Yelp.com has been running the Yelp Dataset challenge since 2013; a competition that encourages people to explore and research Yelp's open dataset. As of Round 10 of the challenge, the dataset contained:

- almost 5 million reviews
- over 1.1 million users
- over 150,000 businesses
- 12 metropolitan areas

Since its launch, the dataset has become very popular, with hundreds of academic papers written about it. It has well-structured, and highly relational data, and is therefore a realistic dataset with which to showcase Neo4j and graph algorithms.

We will illustrate how to use graph algorithms on a social network of friends, and how to create and analyse an inferred graph (for example, projecting a review co-occurence graph, or similarity between users based on their reviews). For more information, it is also worth checking out past winners, and their work.

Data

In Round 10 of the challenge, the dataset included:

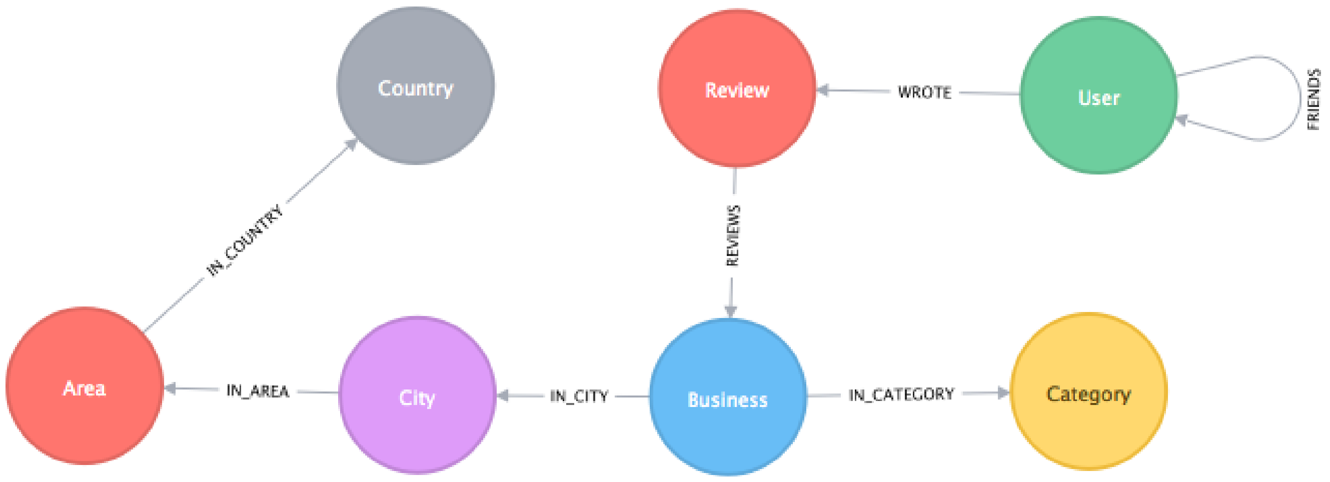
- 156,639 businesses
- 1,005,693 tips from users about businesses
- 4,736,897 reviews of businesses by users
- 9,489,337 users total
- 35,444,850 friend relationships

You can download the dataset in JSON format by filling out a form on Yelp's website. There are 6 JSON files available (detailed documentation). For the purposes of this example, we will ignore the photos and checkins files as they are not relevant for our analysis.

We will create a knowledge graph from the rest of the files, and will use the APOC plugin to help us with importing and batching data in Neo4j. Depending on your setup, import might take some time (the *user.json* file contains data for about a 10 million-person social network of friends). While *review.json* is even bigger in size, it is mostly made up of the text that represents the actual review, so the import will be faster. We also do not need the actual text, but only the meta-data about them.

For example, meta-data on who wrote the review and how a certain business was rated is imported, but the text itself will not be imported.

Graph model



Our graph contains `User` labelled nodes, that can have a `FRIEND` relationship with other users. Users also write reviews and tips about businesses. All of the meta-data is stored as properties of nodes, except for categories of the businesses, which are represented by separate nodes labeled `Category`.

Graph model always depends on the application we have in mind for it. Our application is to analyse (inferred) networks with graph algorithms. If we were to use our graph as a recommendation engine, we might construct a different graph model.

For further information on using Neo4j as a recommendation engine, check out this great guide or this educational video.

Import

Define graph schema (constraint/index)

```
CALL apoc.schema.assert(
{Category:['name']},
{Business:['id'],User:['id'],Review:['id']});
```

Load businesses

```
CALL apoc.periodic.iterate("
CALL apoc.load.json('file:///dataset/business.json') YIELD value RETURN value
","
MERGE (b:Business{id:value.business_id})
SET b += apoc.map.clean(value,
['attributes','hours','business_id','categories','address','postal_code'],[])
WITH b,value.categories as categories
UNWIND categories as category
MERGE (c:Category{id:category})
MERGE (b)-[:IN_CATEGORY]->(c)
",{batchSize: 10000, iterateList: true});
```

Load tips

```
CALL apoc.periodic.iterate("
CALL apoc.load.json('file:///dataset/tip.json') YIELD value RETURN value
","
MATCH (b:Business{id:value.business_id})
MERGE (u:User{id:value.user_id})
MERGE (u)-[:TIP{date:value.date,likes:value.likes}]->(b)
",{batchSize: 20000, iterateList: true});
```

Load reviews

```
CALL apoc.periodic.iterate("
CALL apoc.load.json('file:///dataset/review.json')
YIELD value RETURN value
","
MERGE (b:Business{id:value.business_id})
MERGE (u:User{id:value.user_id})
MERGE (r:Review{id:value.review_id})
MERGE (u)-[:WROTE]->(r)
MERGE (r)-[:REVIEWS]->(b)
SET r += apoc.map.clean(value, ['business_id','user_id','review_id','text'],[0])
",{batchSize: 10000, iterateList: true});
```

Load users

```
CALL apoc.periodic.iterate("
CALL apoc.load.json('file:///dataset/user.json')
YIELD value RETURN value
","
MERGE (u:User{id:value.user_id})
SET u += apoc.map.clean(value, ['friends','user_id'],[0])
WITH u,value.friends as friends
UNWIND friends as friend
MERGE (u1:User{id:friend})
MERGE (u)-[:FRIEND]-(u1)
",{batchSize: 100, iterateList: true});
```

Networks

Social network

A Social network is a theoretical construct, useful in the social sciences to study relationships between individuals, groups, organizations, or even entire societies. An axiom of the social network approach to understanding social interaction is that social phenomena should be primarily conceived and investigated through the properties of relationships between and within nodes, instead of the properties of these nodes themselves. Precisely because many different types of relations, singular or in combination, form these network configurations, network analytics are useful to a broad range of research enterprises.

Social network analysis is the process of investigating social structures through the use of networks and graph theory. It characterizes networked structures in terms of nodes (individual actors, people, or things within the network) and the ties, edges, or links (relationships or interactions) that connect them. Examples of social structures commonly visualized through social network analysis include social media networks, memes spread, friendship and acquaintance networks, collaboration graphs, kinship, and disease transmission.

Social network analysis has emerged as a key technique in modern sociology. It has also gained a significant following in anthropology, biology, demography, communication studies, economics, geography, history, information science, organizational studies, political science, social psychology, development studies, sociolinguistics, and computer science.

Yelp’s friendship network is an *undirected* graph with *unweighted* friend relationships between users. While there are over 500,000 users with no friends, they will be ignored in this analysis.

Global graph statistics:

Nodes : 8981389

Relationships : 35444850

Weakly connected components : 18512

Nodes in largest WCC : 8938630

Edges in largest WCC : 35420520

Triangle count :

Average clustering coefficient :

Graph diameter (longest shortest path):

Local graph statistics:

Use apoc to calculate local statistics

```
MATCH (u:User)
RETURN avg(apoc.node.degree(u,'FRIEND')) as average_friends,
       stdev(apoc.node.degree(u,'FRIEND')) as stdev_friends,
       max(apoc.node.degree(u,'FRIEND')) as max_friends,
       min(apoc.node.degree(u,'FRIEND')) as min_friends
```

Average number of friends : 7.47

Standard deviation of friends : 46.96

Minimum count of friends : 1

Maximum count of friends : 14995

Prior work:

- http://snap.stanford.edu/class/cs224w-2015/projects_2015/Predicting_Yelp_Ratings_From_Social_Network_Data.pdf
- https://arxiv.org/pdf/1512.06915.pdf
- http://trust.sce.ntu.edu.sg/wit-ec16/paper/davoust.pdf

Projecting a review co-occurence graph

We can try to find which businesses are often reviewed by the same users, by inferring a co-occurence network between them.

Co-occurrence networks are the collective interconnection of nodes, based on their paired presence within a specified domain. Our network is generated by connecting pairs of businesses using a set of criteria defining co-occurrence.

The co-occurrence criteria for this network is that any pair of businesses must have at least 5 common reviewers. We save the count of common reviewers as a property of the relationship that will be used as a weight in community detection analysis. Inferred graph is *undirected*, as changing the direction of the relationships does not imply any semantic difference. We will limit our network to those businesses, that have more than 10 reviews and project a co-occurrent relationship between businesses:

Project a review co-occurence between businesses

```
CALL apoc.periodic.iterate('
MATCH (b1:Business)
WHERE size((b1)<-[:REVIEWS]->()) > 10 AND b1.city="Las Vegas"
RETURN b1
','
MATCH (b1)<-[:REVIEWS]-(r1)
MATCH (r1)<-[:WROTE]-(u)
MATCH (u)-[:WROTE]->(r2)
MATCH (r2)-[:REVIEWS]->(b2)
WHERE id(b1) < id(b2) AND b2.city="Las Vegas"
WITH b1, b2, COUNT(*) AS weight where weight > 5
MERGE (b1)-[cr:CO_OCCURENT_REVIEWS]-(b2)
ON CREATE SET cr.weight = weight
',{batchSize: 1});
```

Projecting a review similarity graph

We can try to find similar groups of users by projecting a review similarity network between them. The idea is to start with users that have more than 10 reviews, and find all pairs of users who have reviewed more than 10 common businesses. We do this to filter out users with not enough data. We could do something similar to filter out users who have reviewed every business (probably a bot, or someone very bored!).

Once we find pairs of users, we calculate their similarity of reviews by using cosine similarity, and by only creating a relationship if cosine similarity is greater than 0; which is sometimes also called hard similarity. We do this so we do not end up with complete graph, where every pair of users is connected. Most community detection algorithms perform poorly in a complete graph. Cosine similarity between pairs of users is saved as a property of relationship and can be used as a weight in graph algorithms. Projected graph is modeled *undirected*, as the direction of the relationships have no semantic value.

Projecting a review similarity graph is often used in recommendations; similar users are calculated based on review ratings, so we can recommend to a user what similar users liked.

Create a review similarity graph

```
CALL apoc.periodic.iterate(
"MATCH (p1:User) WHERE size((p1)-[:WROTE]->()) > 5 RETURN p1",
"
MATCH (p1)-[:WROTE]->(r1)-->()<--(r2)<-[:WROTE]-(p2)
WHERE id(p1) < id(p2) AND size((p2)-[:WROTE]->()) > 10
WITH p1,p2,count(*) as coop, collect(r1.stars) as s1, collect(r2.stars) as s2 where
coop > 10
WITH p1,p2, apoc.algo.cosineSimilarity(s1,s2) as cosineSimilarity WHERE
cosineSimilarity > 0
MERGE (p1)-[s:SIMILAR_REVIEWS]-(p2) SET s.weight = cosineSimilarity"
, {batchSize:100, parallel:false,iterateList:true});
```

Prior work:

- http://snap.stanford.edu/class/cs224w-2015/projects_2015/Predicting_Yelp_Ratings_Using_User_Friendship_Network_Information.pdf
- http://snap.stanford.edu/class/cs224w-2013/projects2013/cs224w-038-final.pdf

Procedures

***This chapter contains a reference of all the procedures in the Neo4j Graph** Algorithms library.*

qualified name	description
`algo.unionFind.mscoloring`	`CALL algo.unionFind.mscoloring(label:String, relationship:String, {property:'weight', threshold:0.42, defaultValue:1.0, write: true, partitionProperty:'partition', concurrency:4}) YIELD nodes, setCount, loadMillis, computeMillis, writeMillis`
`algo.unionFind.mscoloring.stream`	`CALL algo.unionFind.mscoloring.stream(label:String, relationship:String, {property:'propertyName', threshold:0.42, defaultValue:1.0, concurrency:4) YIELD nodeId, setId - yields a setId to each node id`
`algo.closeness.harmonic.stream`	`CALL algo.closeness.harmonic.stream(label:String, relationship:String{concurrency:4}) YIELD nodeId, centrality - yields centrality for each node`
`algo.closeness.harmonic`	`CALL algo.closeness.harmonic(label:String, relationship:String, {write:true, writeProperty:'centrality, concurrency:4'}) YIELD loadMillis, computeMillis, writeMillis, nodes] - yields evaluation details`
`algo.list`	`CALL algo.list - lists all algorithm procedures, their description and signature`
`algo.shortestPath.stream`	`CALL algo.shortestPath.stream(startNode:Node, endNode:Node, weightProperty:String{nodeQuery:'labelName', relationshipQuery:'relationshipName', direction:'BOTH', defaultValue:1.0}) YIELD nodeId, cost - yields a stream of {nodeId, cost} from start to end (inclusive)`
`algo.shortestPath`	`CALL algo.shortestPath(startNode:Node, endNode:Node, weightProperty:String{nodeQuery:'labelName', relationshipQuery:'relationshipName', direction:'BOTH', defaultValue:1.0, write:'true', writeProperty:'sssp'}) YIELD nodeId, cost, loadMillis, evalMillis, writeMillis - yields nodeCount, totalCost, loadMillis, evalMillis, writeMillis`
`algo.shortestPath.astar.stream`	`CALL algo.shortestPath.astar.stream(startNode:Node, endNode:Node, weightProperty:String, propertyKeyLat:String,propertyKeyLon:String, {nodeQuery:'labelName', relationshipQuery:'relationshipName', direction:'BOTH', defaultValue:1.0}) YIELD nodeId, cost - yields a stream of {nodeId, cost} from start to end (inclusive)`
`algo.louvain`	`CALL algo.louvain(label:String, relationship:String, {weightProperty:'weight', defaultValue:1.0, write: true, writeProperty:'community', concurrency:4}) YIELD nodes, communityCount, iterations, loadMillis, computeMillis, writeMillis`
`algo.louvain.stream`	`CALL algo.louvain.stream(label:String, relationship:String, {weightProperty:'propertyName', defaultValue:1.0, concurrency:4) YIELD nodeId, community - yields a setId to each node id`
`algo.unionFind.forkJoin`	`CALL algo.unionFind(label:String, relationship:String, {property:'weight', threshold:0.42, defaultValue:1.0, write: true, partitionProperty:'partition',concurrency:4}) YIELD nodes, setCount, loadMillis, computeMillis, writeMillis`
`algo.unionFind.forkJoin.stream`	`CALL algo.unionFind.stream(label:String, relationship:String, {property:'propertyName', threshold:0.42, defaultValue:1.0,concurrency:4}) YIELD nodeId, setId - yields a setId to each node id`

qualified name	description
algo.unionFind.queue	CALL algo.unionFind(label:String, relationship:String, {property:'weight', threshold:0.42, defaultValue:1.0, write: true, partitionProperty:'partition',concurrency:4}) YIELD nodes, setCount, loadMillis, computeMillis, writeMillis
algo.unionFind.queue.stream	CALL algo.unionFind.stream(label:String, relationship:String, {property:'propertyName', threshold:0.42, defaultValue:1.0, concurrency:4}) YIELD nodeId, setId - yields a setId to each node id
algo.labelPropagation	CALL algo.labelPropagation(label:String, relationship:String, direction:String, {iterations:1, weightProperty:'weight', partitionProperty:'partition', write:true, concurrency:4}) YIELD nodes, iterations, didConverge, loadMillis, computeMillis, writeMillis, write, weightProperty, partitionProperty - simple label propagation kernel
algo.labelPropagation.stream	CALL algo.labelPropagation.stream(label:String, relationship:String, config:Map<String, Object>) YIELD nodeId, label
algo.closeness.dangalchev.stream	CALL algo.closeness.dangalchev.stream(label:String, relationship:String{concurrency:4}) YIELD nodeId, centrality - yields centrality for each node
algo.closeness.dangalchev	CALL algo.closeness.dangalchev(label:String, relationship:String, {write:true, writeProperty:'centrality, concurrency:4'}) YIELD loadMillis, computeMillis, writeMillis, nodes] - yields evaluation details
algo.closeness.stream	CALL algo.closeness.stream(label:String, relationship:String{concurrency:4}) YIELD nodeId, centrality - yields centrality for each node
algo.closeness	CALL algo.closeness(label:String, relationship:String, {write:true, writeProperty:'centrality, concurrency:4'}) YIELD loadMillis, computeMillis, writeMillis, nodes] - yields evaluation details
algo.allShortestPaths.stream	CALL algo.allShortestPaths.stream(weightProperty:String{nodeQuery:'labelName', relationshipQuery:'relationshipName', defaultValue:1.0, concurrency:4}) YIELD sourceNodeId, targetNodeId, distance - yields a stream of {sourceNodeId, targetNodeId, distance}
algo.shortestPaths.stream	CALL algo.shortestPaths.stream(startNode:Node, weightProperty:String{nodeQuery:'labelName', relationshipQuery:'relationshipName', defaultValue:1.0}) YIELD nodeId, distance - yields a stream of {nodeId, cost} from start to end (inclusive)
algo.shortestPaths	CALL algo.shortestPaths(startNode:Node, weightProperty:String{write:true, targetProperty:'path', nodeQuery:'labelName', relationshipQuery:'relationshipName', defaultValue:1.0}) YIELD loadDuration, evalDuration, writeDuration, nodeCount, targetProperty - yields nodeCount, totalCost, loadDuration, evalDuration
algo.spanningTree.kmax	CALL algo.spanningTree.kmax(label:String, relationshipType:String, weightProperty:String, startNodeId:long, k:int, {writeProperty:String}) YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
algo.spanningTree.kmin	CALL algo.spanningTree.kmin(label:String, relationshipType:String, weightProperty:String, startNodeId:long, k:int, {writeProperty:String}) YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
algo.betweenness.sampled.stream	CALL algo.betweenness.sampled.stream(label:String, relationship:String, {strategy:{'random', 'degree'}, probability:double, maxDepth:int, direction:String, concurrency:int}) YIELD nodeId, centrality - yields centrality for each node
algo.betweenness.stream	CALL algo.betweenness.stream(label:String, relationship:String, {direction:'out', concurrency :4})YIELD nodeId, centrality - yields centrality for each node

qualified name	description
algo.betweenness	CALL algo.betweenness(label:String, relationship:String, {direction:'out',write:true, writeProperty:'centrality', stats:true, concurrency:4}) YIELD loadMillis, computeMillis, writeMillis, nodes, minCentrality, maxCentrality, sumCentrality - yields status of evaluation
algo.betweenness.sampled	CALL algo.betweenness.sampled(label:String, relationship:String, {strategy:'random', probability:double, maxDepth:5, direction:'out',write:true, writeProperty:'centrality', stats:true, concurrency:4}) YIELD loadMillis, computeMillis, writeMillis, nodes, minCentrality, maxCentrality, sumCentrality - yields status of evaluation
algo.scc	CALL algo.scc(label:String, relationship:String, config:Map<String, Object>) YIELD loadMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize
algo.scc.stream	CALL algo.scc.stream(label:String, relationship:String, config:Map<String, Object>) YIELD loadMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize
algo.scc.recursive.tarjan	CALL algo.scc.tarjan(label:String, relationship:String, config:Map<String, Object>) YIELD loadMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize
algo.scc.recursive.tunedTarjan	CALL algo.scc.recursive.tunedTarjan(label:String, relationship:String, config:Map<String, Object>) YIELD loadMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize
algo.scc.recursive.tunedTarjan.stream	CALL algo.scc.recursive.tunedTarjan.stream(label:String, relationship:String, config:Map<String, Object>) YIELD nodeId, partition
algo.scc.iterative	CALL algo.scc.iterative(label:String, relationship:String, config:Map<String, Object>) YIELD loadMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize
algo.scc.iterative.stream	CALL algo.scc.iterative.stream(label:String, relationship:String, config:Map<String, Object>) YIELD nodeId, partition
algo.scc.multistep	CALL algo.scc.multistep(label:String, relationship:String, {write:true, concurrency:4, cutoff:100000}) YIELD loadMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize
algo.scc.multistep.stream	CALL algo.scc.multistep.stream(label:String, relationship:String, {write:true, concurrency:4, cutoff:100000}) YIELD nodeId, partition
algo.scc.forwardBackward.stream	CALL algo.scc.forwardBackward.stream(long startNodeId, label:String, relationship:String, {write:true, concurrency:4}) YIELD nodeId, partition
algo.mst	CALL algo.mst(label:String, relationshipType:String, weightProperty:String, startNodeId:long, {writeProperty:String}) YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
algo.spanningTree	CALL algo.spanningTree(label:String, relationshipType:String, weightProperty:String, startNodeId:long, {writeProperty:String}) YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
algo.spanningTree.minimum	CALL algo.spanningTree.minimum(label:String, relationshipType:String, weightProperty:String, startNodeId:long, {writeProperty:String}) YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
algo.spanningTree.maximum	CALL algo.spanningTree.maximum(label:String, relationshipType:String, weightProperty:String, startNodeId:long, {writeProperty:String}) YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
algo.graph.load	CALL algo.graph.load(name:String, label:String, relationship:String{direction:'OUT/IN/BOTH', undirected:true/false, sorted:true/false, nodeProperty:'value', nodeWeight:'weight', relationshipWeight: 'weight', graph:'heavy/huge/cypher'}) YIELD nodes, relationships, loadMillis, computeMillis, writeMillis, write, nodeProperty, nodeWeight, relationshipWeight - load named graph

qualified name	description
algo.graph.remove	CALL algo.graph.remove(name:String
algo.graph.info	CALL algo.graph.info(name:String
algo.triangle.stream	CALL algo.triangle.stream(label, relationship, {concurrency:4}) YIELD nodeA, nodeB, nodeC - yield nodeA, nodeB and nodeC which form a triangle
algo.triangleCount.stream	CALL algo.triangleCount.stream(label, relationship, {concurrency:8}) YIELD nodeId, triangles - yield nodeId, number of triangles
algo.triangleCount.forkJoin.stream	CALL algo.triangleCount.forkJoin.stream(label, relationship, {concurrency:8}) YIELD nodeId, triangles - yield nodeId, number of triangles
algo.triangleCount	CALL algo.triangleCount(label, relationship, {concurrency:4, write:true, writeProperty:'triangles', clusteringCoefficientProperty:'coefficient'}) YIELD loadMillis, computeMillis, writeMillis, nodeCount, triangleCount, averageClusteringCoefficient
algo.triangleCount.forkJoin	CALL algo.triangleCount.forkJoin(label, relationship, {concurrency:4, write:true, writeProperty:'triangles', clusteringCoefficientProperty:'coefficient'}) YIELD loadMillis, computeMillis, writeMillis, nodeCount, triangleCount, averageClusteringCoefficient
algo.pageRank	CALL algo.pageRank(label:String, relationship:String, {iterations:5, dampingFactor:0.85, write: true, writeProperty:'pagerank', concurrency:4}) YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, dampingFactor, write, writeProperty - calculates page rank and potentially writes back
algo.pageRank.stream	CALL algo.pageRank.stream(label:String, relationship:String, {iterations:20, dampingFactor:0.85, concurrency:4}) YIELD node, score - calculates page rank and streams results
algo.shortestPath.deltaStepping.stream	CALL algo.shortestPath.deltaStepping.stream(startNode:Node, weightProperty:String, delta:Double{label:'labelName', relationship:'relationshipName', defaultValue:1.0, concurrency:4}) YIELD nodeId, distance - yields a stream of {nodeId, distance} from start to end (inclusive)
algo.shortestPath.deltaStepping	CALL algo.shortestPath.deltaStepping(startNode:Node, weightProperty:String, delta:Double{label:'labelName', relationship:'relationshipName', defaultValue:1.0, write:true, writeProperty:'sssp'}) YIELD loadDuration, evalDuration, writeDuration, nodeCount
algo.kShortestPaths	CALL algo.kShortestPaths(startNode:Node, endNode:Node, k:int, weightProperty:String{nodeQuery:'labelName', relationshipQuery:'relationshipName', direction:'OUT', defaultValue:1.0, maxDepth:42, write:'true', writePropertyPrefix:'PATH_'}) YIELD resultCount, loadMillis, evalMillis, writeMillis - yields resultCount, loadMillis, evalMillis, writeMillis
algo.unionFind.forkJoinMerge	CALL algo.unionFind(label:String, relationship:String, {property:'weight', threshold:0.42, defaultValue:1.0, write: true, partitionProperty:'partition', concurrency:4}) YIELD nodes, setCount, loadMillis, computeMillis, writeMillis
algo.unionFind.forkJoinMerge.stream	CALL algo.unionFind.stream(label:String, relationship:String, {property:'propertyName', threshold:0.42, defaultValue:1.0, concurrency:4}) YIELD nodeId, setId - yields a setId to each node id
algo.unionFind	CALL algo.unionFind(label:String, relationship:String, {weightProperty:'weight', threshold:0.42, defaultValue:1.0, write: true, partitionProperty:'partition'}) YIELD nodes, setCount, loadMillis, computeMillis, writeMillis

qualified name	description
algo.unionFind.stream	CALL algo.unionFind.stream(label:String, relationship:String, {weightProperty:'propertyName', threshold:0.42, defaultValue:1.0) YIELD nodeId, setId - yields a setId to each node id
algo.isFinite	CALL algo.isFinite(value) - return true iff the given argument is a finite value (not ±Infinity, NaN, or null), false otherwise.
algo.isInfinite	CALL algo.isInfinite(value) - return true iff the given argument is not a finite value (±Infinity, NaN, or null), false otherwise.
algo.Infinity	CALL algo.Infinity() - returns Double.POSITIVE_INFINITY as a value.
algo.NaN	CALL algo.NaN() - returns Double.NaN as a value.

Algorithms

This chapter provides explanations and examples for each of the algorithms in the Neo4j Graph Algorithms library.

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 PageRank algorithm

This section describes the PageRank algorithm in the Neo4j Graph Algorithms library.

PageRank is an algorithm that measures the **transitive** influence or connectivity of nodes.

It can be computed by either iteratively distributing one node's rank (originally based on degree) over its neighbours or by randomly traversing the graph and counting the frequency of hitting each node during these walks.

History and explanation

PageRank is named after Google co-founder Larry Page, and is used to rank websites in Google's search results. It counts the number, and quality, of links to a page which determines an estimation of how important the page is. The underlying assumption is that pages of importance are more likely to receive a higher volume of links from other pages.

PageRank is defined in the original Google paper as follows:

```
PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))
```

where,

- we assume that a page `A` has pages `T1` to `Tn` which point to it (i.e., are citations).
- `d` is a damping factor which can be set between 0 and 1. It is usually set to 0.85.

- `C(A)` is defined as the number of links going out of page `A`.

Use-cases - when to use the PageRank algorithm

PageRank can be applied across a wide range of domains. The following are some notable use-cases:

- Personalized PageRank is used by Twitter to present users with recommendations of other accounts that they may wish to follow. The algorithm is run over a graph which contains shared interests and common connections. Their approach is described in more detail in "WTF: The Who to Follow Service at Twitter".
- PageRank has been used to rank public spaces or streets, predicting traffic flow and human movement in these areas. The algorithm is run over a graph which contains intersections connected by roads, where the PageRank score reflects the tendency of people to park, or end their journey, on each street. This is described in more detail in "Self-organized Natural Roads for Predicting Traffic Flow: A Sensitivity Study".
- PageRank can be used as part of an anomaly or fraud detection system in the healthcare and insurance industries. It can help find doctors or providers that are behaving in an unusual manner, and then feed the score into a machine learning algorithm.

There are many more use cases, which you can read about in David Gleich’s "PageRank beyond the web"

Constraints - when not to use the PageRank algorithm

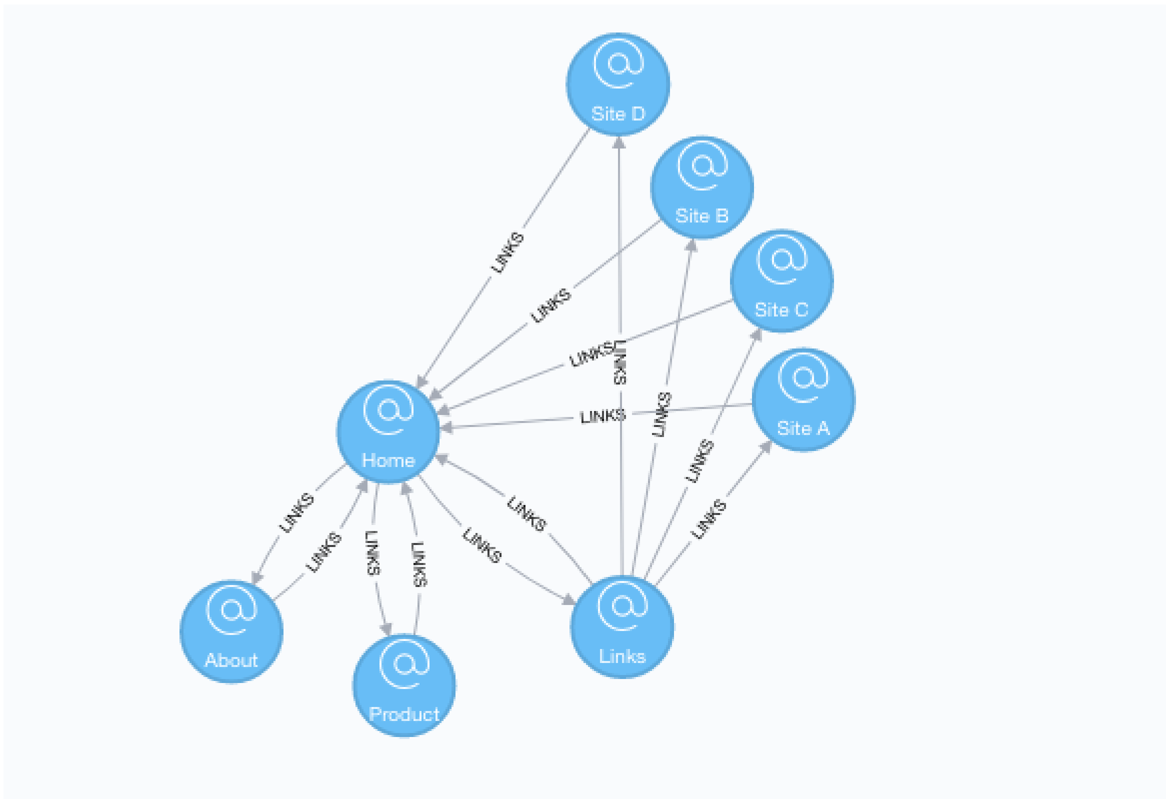
There are some things to be aware of when using the PageRank algorithm:

- If there are no links from within a group of pages to outside of the group, then the group is considered a spider trap.
- Rank sink can occur when a network of pages form an infinite cycle.
- Dead-ends occur when pages have no out-links. If a page contains a link to another page which has no out-links, the link would be known as a dangling link.

If you see unexpected results from running the algorithm, it is worth doing some exploratory analysis of the graph to see if any of these problems are the cause. You can read The Google PageRank Algorithm and How It Works to learn more.

PageRank algorithm sample

This sample will explain the PageRank algorithm, using a simple graph:



The following will create a sample graph:

```
MERGE (home:Page {name:'Home'})
MERGE (about:Page {name:'About'})
MERGE (product:Page {name:'Product'})
MERGE (links:Page {name:'Links'})
MERGE (a:Page {name:'Site A'})
MERGE (b:Page {name:'Site B'})
MERGE (c:Page {name:'Site C'})
MERGE (d:Page {name:'Site D'})

MERGE (home)-[:LINKS]->(about)
MERGE (about)-[:LINKS]->(home)
MERGE (product)-[:LINKS]->(home)
MERGE (home)-[:LINKS]->(product)
MERGE (links)-[:LINKS]->(home)
MERGE (home)-[:LINKS]->(links)
MERGE (links)-[:LINKS]->(a)
MERGE (a)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(b)
MERGE (b)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(c)
MERGE (c)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(d)
MERGE (d)-[:LINKS]->(home)
```

The following will run the algorithm and stream results:

```
CALL algo.pageRank.stream('Page', 'LINKS', {iterations:20, dampingFactor:0.85})
YIELD nodeId, score

MATCH (node) WHERE id(node) = nodeId

RETURN node.name AS page,score
ORDER BY score DESC
```

The following will run the algorithm and write back results:

```
CALL algo.pageRank('Page', 'LINKS',
  {iterations:20, dampingFactor:0.85, write: true,writeProperty:"pagerank"})
YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, dampingFactor, write,
writeProperty
```

Table 1. Results

Name	PageRank
Home	3.232
Product	1.059
Links	1.059
About	1.059
Site A	0.328
Site B	0.328
Site C	0.328
Site D	0.328

As we might expect, the Home page has the highest PageRank because it has incoming links from all other pages. We can also see that it's not only the number of incoming links that is important, but also the importance of the pages behind those links.

Example usage

In this example we will run PageRank on Yelp's social network to find potential influencers.

When importing the Yelp dataset we stored the social network as a undirected graph. Relationships in Neo4j always have a direction, but in this domain the direction is irrelevant. If `Person A` is a `FRIEND` with `Person B`, we can say that `Person B` is also a `FRIEND` with `Person A`.

The default label and relationship-type selection syntax won't work for us here, because it will project a directed social network. Instead, we can project our undirected social network using **Cypher loading**. We can also apply this approach to other algorithms that use **Cypher loading**.

The following will run the algorithm on Yelp social network:

```
CALL algo.pageRank.stream(
  'MATCH (u:User) WHERE exists( (u)-[:FRIENDS]-() ) RETURN id(u) as id',
  'MATCH (u1:User)-[:FRIENDS]-(u2:User) RETURN id(u1) as source, id(u2) as target',
  {graph:'cypher'}
) YIELD node,score with node,score order by score desc limit 10
RETURN node {.name, .review_count, .average_stars,.useful,.yelping_since,.funny},
score
```

Syntax

The following will run the algorithm and write back results:

```
CALL algo.pageRank(label:String, relationship:String,
    {iterations:20, dampingFactor:0.85, write: true, writeProperty:'pagerank',
concurrency:4})
YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, dampingFactor, write,
writeProperty
```

Table 2. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
iterations	int	20	yes	How many iterations of PageRank to run
concurrency	int	available CPUs	yes	The number of concurrent threads
dampingFactor	float	0.85	yes	The damping factor of the PageRank calculation
write	boolean	true	yes	Specify if the result should be written back as a node property
writeProperty	string	'pagerank'	yes	The property name written back to
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 3. Results

Name	Type	Description
nodes	int	The number of nodes considered
iterations	int	The number of iterations run

Name	Type	Description
dampingFactor	float	The damping factor used
writeProperty	string	The property name written back to
write	boolean	Specifies if the result was written back as node property
loadMillis	int	Milliseconds for loading data
computeMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the algorithm and stream results:

```
CALL algo.pageRank.stream(label:String, relationship:String,
    {iterations:20, dampingFactor:0.85, concurrency:4})
YIELD node, score
```

Table 4. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all nodes
iterations	int	20	yes	Specify how many iterations of PageRank to run
concurrency	int	available CPUs	yes	The number of concurrent threads
dampingFactor	float	0.85	yes	The damping factor of the PageRank calculation
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 5. Results

Name	Type	Description
node	long	Node ID
score	float	PageRank weight

Huge graph projection

If our projected graph contains more than 2 billion nodes or relationships, we need to use huge graph projection, as the default label and relationship-type projection has a limitation of 2 billion nodes and 2 billion relationships.

Set `graph:'huge'` in the config:

```
CALL algo.pageRank('Page','LINKS',
  {graph:'huge'})
YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, dampingFactor,
writeProperty;
```

Cypher projection

If label and relationship-type are not selective enough to describe a subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. You must ensure that `graph:'cypher'` is set in the config:

```
CALL algo.pageRank(
  'MATCH (p:Page) RETURN id(p) as id',
  'MATCH (p1:Page)-[:Link]->(p2:Page) RETURN id(p1) as source, id(p2) as target',
  {graph:'cypher', iterations:5, write: true}
)
```

Versions

We support the following versions of the PageRank algorithm:

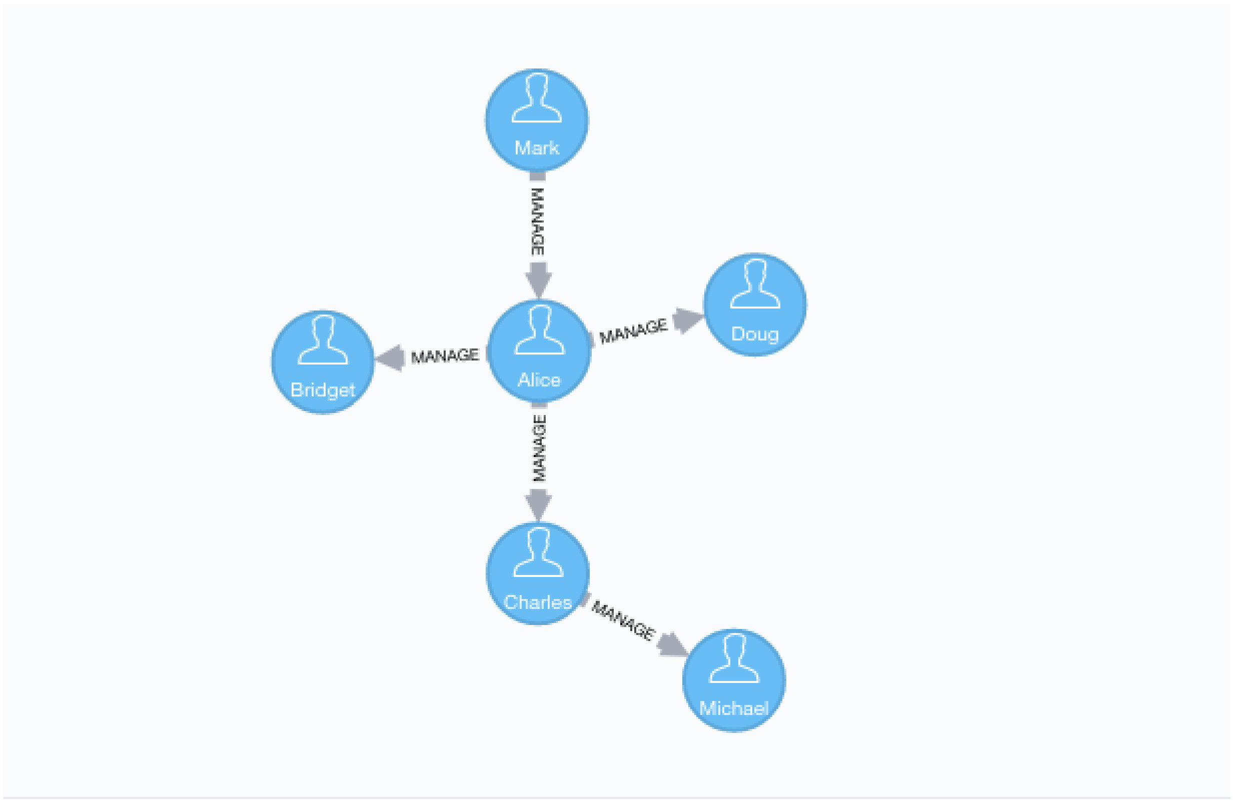
- ✓ directed, unweighted
- ☐ directed, weighted
- ✓ undirected, unweighted
  - Only with cypher projection
- ☐ undirected, weighted

The Betweenness Centrality algorithm

This section describes the Betweenness Centrality algorithm in the Neo4j Graph Algorithms library.

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.

In the following example, Alice is the main connection in the graph:



If Alice is removed, all connections in the graph would be cut off. This makes Alice important, because she ensures that no nodes are isolated.

History and explanation

The Betweenness Centrality algorithm calculates the shortest (weighted) path between every pair of nodes in a connected graph, using the breadth-first search algorithm. Each node receives a score, based on the number of these shortest paths that pass through the node. Nodes that most frequently lie on these shortest paths will have a higher betweenness centrality score.

The algorithm was given its first formal definition by Linton Freeman, in his 1971 paper "A Set of Measures of Centrality Based on Betweenness". It was considered to be one of the "three distinct intuitive conceptions of centrality".

Use-cases - when to use the Betweenness Centrality algorithm

- Betweenness centrality is used to research the network flow in a package delivery process, or telecommunications network. These networks are characterized by traffic that has a known target and takes the shortest path possible. This, and other scenarios, are described by Stephen P. Borgatti in "Centrality and network flow".
- Betweenness centrality is used to identify influencers in legitimate, or criminal, organizations. Studies show that influencers in organizations are not necessarily in management positions, but instead can be found in brokerage positions of the organizational network. Removal of such influencers could seriously destabilize the organization. More detail can be found in "Brokerage qualifications in ringing operations", by Carlo Morselli and Julie Roy.
- Betweenness centrality can be used to help microbloggers spread their reach on Twitter, with a recommendation engine that targets influencers that they should interact with in the future. This approach is described in "Making Recommendations in a Microblog to Improve the Impact

of a Focal User".

Constraints - when not to use the Betweenness Centrality algorithm

- Betweeness centrality makes the assumption that all communication between nodes happens along the shortest path and with the same frequency, which isn't the case in real life. Therefore, it doesn't give us a perfect view of the most influential nodes in a graph, but rather a good representation. Newman explains this in more detail on page 186 of Networks: An Introduction.
- For large graphs, exact centrality computation isn't practical. The fastest known algorithm for exactly computing betweenness of all the nodes requires at least `O(nm)` time for unweighted graphs, where `n` is the number of nodes and `m` is the number of relationships. Instead, we can use an approximation algorithm that works with a subset of nodes.

Betweenness Centrality algorithm sample

People with high betweenness tend to be the innovators and brokers in social networks. They combine different perspectives, transfer ideas between groups, and get power from their ability to make introductions and pull strings.

The following will create a sample graph:

```
MERGE (nAlice:User {id:'Alice'})
MERGE (nBridget:User {id:'Bridget'})
MERGE (nCharles:User {id:'Charles'})
MERGE (nDoug:User {id:'Doug'})
MERGE (nMark:User {id:'Mark'})
MERGE (nMichael:User {id:'Michael'})

MERGE (nAlice)-[:MANAGE]->(nBridget)
MERGE (nAlice)-[:MANAGE]->(nCharles)
MERGE (nAlice)-[:MANAGE]->(nDoug)
MERGE (nMark)-[:MANAGE]->(nAlice)
MERGE (nCharles)-[:MANAGE]->(nMichael);
```

The following will run the algorithm and stream results:

```
CALL algo.betweenness.stream('User','MANAGE',{direction:'out'})
YIELD nodeId, centrality

MATCH (user:User) WHERE id(user) = nodeId

RETURN user.id AS user,centrality
ORDER BY centrality DESC;
```

The following will run the algorithm and write back results:

```
CALL algo.betweenness('User','MANAGE', {direction:'out',write:true,
writeProperty:'centrality'})
YIELD nodes, minCentrality, maxCentrality, sumCentrality, loadMillis, computeMillis,
writeMillis;
```

Table 6. Results

Name	Centrality weight
Alice	4
Charles	2
Bridget	0
Michael	0
Doug	0
Mark	0

We can see that Alice is the main broker in this network, and Charles is a minor broker. The others don't have any influence, because all the shortest paths between pairs of people go via Alice or Charles.

Approximation of Betweenness Centrality

As mentioned above, calculating the exact betweenness centrality on large graphs can be very time consuming. Therefore, you might choose to use an approximation algorithm that will run much quicker, and still provide useful information.

RA-Brandes algorithm

The RA-Brandes algorithm is the best known algorithm for calculating an approximate score for betweenness centrality. Rather than calculating the shortest path between every pair of nodes, the RA-Brandes algorithm considers only a subset of nodes. Two common strategies for selecting the subset of nodes are:

random

Nodes are selected uniformly, at random, with defined probability of selection. The default probability is `log10(N) / e^2`. If the probability is 1, then the algorithm works the same way as the normal Betweenness Centrality algorithm, where all nodes are loaded.

degree

First, the mean degree of the nodes is calculated, and then only the nodes whose degree is higher than the mean are visited (i.e. only dense nodes are visited).

As a further optimisation, you can choose to limit the depth used by the shortest path algorithm. This can be controlled by the `maxDepth` parameter.

The following will run the algorithm and stream results:

```
CALL algo.betweenness.sampled.stream('User','MANAGE',
  {strategy:'random', probability:1.0, maxDepth:1, direction: "out"})

YIELD nodeId, centrality

MATCH (user) WHERE id(user) = nodeId
RETURN user.id AS user,centrality
ORDER BY centrality DESC;
```

The following will run the algorithm and write back results:

```
CALL algo.betweenness.sampled('User','MANAGE',
  {strategy:'random', probability:1.0, writeProperty:'centrality', maxDepth:1,
direction: "out"})
YIELD nodes, minCentrality, maxCentrality
```

Table 7. Results

Name	Centrality weight
Alice	3
Charles	1
Bridget	0
Michael	0
Doug	0
Mark	0

Alice is still the main broker in the network, and Charles is a minor broker, although their centrality score has reduced as the algorithm only considers relationships at a depth of 1. The others don’t have any influence, because all the shortest paths between pairs of people go via Alice or Charles.

Example usage

Syntax

The following will run the Brandes algorithm and write back results:

```
CALL algo.betweenness(label:String, relationship:String,
  {direction:'out',write:true, stats:true, writeProperty:'centrality',concurrency:1})
YIELD nodes, minCentrality, maxCentrality, sumCentrality, loadMillis, computeMillis,
writeMillis
- calculates betweenness centrality and potentially writes back
```

Table 8. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected
write	boolean	true	yes	Specifies if the result should be written back as a node property
stats	boolean	true	yes	Specifies if stats about centrality should be returned
writeProperty	string	'centrality'	yes	The property name written back to
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement
concurrency	int	available CPUs	yes	The number of concurrent threads

Table 9. Results

Name	Type	Description
nodes	int	The number of nodes considered
minCentrality	int	The minimum centrality value
maxCentrality	int	The maximum centrality value
sumCentrality	int	The sum of all centrality values
loadMillis	int	Milliseconds for loading data
evalMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the Brandes algorithm and stream results:

```
CALL algo.betweenness.stream(label:String, relationship:String,
{direction:'out',concurrency:1})
YIELD nodeId, centrality - yields centrality for each node
```

Table 10. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
concurrency	int	available CPUs	yes	The number of concurrent threads
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected

Table 11. Results

Name	Type	Description
node	long	Node ID
centrality	float	Betweenness centrality weight

The following will run the RA-Brandes algorithm and write back results:

```
CALL algo.betweenness.sampled(label:String, relationship:String,
  {direction:'out', strategy:'random', probability: 1, maxDepth: 4, stats:true,
 writeProperty:'centrality',concurrency:1})
YIELD nodes, minCentrality, maxCentrality, sumCentrality, loadMillis, computeMillis,
writeMillis
- calculates betweenness centrality and potentially writes back
```

Table 12. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all nodes
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected
write	boolean	true	yes	Specifies if the result should be written back as a node property
strategy	string	'random'	yes	The node selection strategy
probability	float	log10(N) / e^2	yes	The probability a node is selected. Values between 0 and 1. If 1, selects all nodes and works like original Brandes algorithm
maxDepth	int	Integer.MAX	yes	The depth of the shortest paths traversal
stats	boolean	true	yes	Specifies if stats about centrality should be returned

Name	Type	Default	Optional	Description
writeProperty	string	'centrality'	yes	The property name written back to
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement
concurrency	int	available CPUs	yes	The number of concurrent threads

Table 13. Results

Name	Type	Description
nodes	int	The number of nodes considered
minCentrality	int	The minimum centrality value
maxCentrality	int	The maximum centrality value
sumCentrality	int	The sum of all centrality values
loadMillis	int	Milliseconds for loading data
evalMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the RA-Brandes algorithm and stream results:

```
CALL algo.betweenness.sampled.stream(label:String, relationship:String,
  {direction:'out',concurrency:1, strategy:'random', probability: 1, maxDepth: 4})
YIELD nodeId, centrality - yields centrality for each node
```

Table 14. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
concurrency	int	available CPUs	yes	The number of concurrent threads
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected
strategy	string	'random'	yes	The node selection strategy

Name	Type	Default	Optional	Description
probability	float	log10(N) / e^2	yes	The probability a node is selected. Values between 0 and 1. If 1, selects all nodes and works like original Brandes algorithm
maxDepth	int	Integer.MAX	yes	The depth of the shortest paths traversal

Table 15. Results

Name	Type	Description
node	long	Node ID
centrality	float	Betweenness centrality weight

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` *in the config:*

```
CALL algo.betweenness(
  'MATCH (p:User) RETURN id(p) as id',
  'MATCH (p1:User)-[:MANAGE]->(p2:User) RETURN id(p1) as source, id(p2) as target',
  {graph:'cypher', write: true}
);
```

Versions

We support the following versions of the betweenness centrality algorithm:

- ☑ directed, unweighted
  - loading incoming relationships: 'INCOMING','IN','I' or '<'
  - loading outgoing relationships: 'OUTGOING','OUT','O' or '>'
- ☐ directed, weighted
- ☑ undirected, unweighted
  - direction:'both' or '<>'
- ☐ undirected, weighted

Implementations

`algo.betweenness()`

- Implementation of brandes-bc algorithm and nodePartitioning extension.
- If concurrency parameter is set (and >1), ParallelBetweennessCentrality is used.

- ParallelBC spawns N(given by the concurrency param) concurrent threads for calculation, where each one calculates the BC for one node at a time.

`algo.betweenness.exp1()`

- Brandes-like algorithm, which uses successor sets instead of predecessor sets.
- The algorithm is based on Brandes definition, but with some changes regarding the dependency-accumulation step.
- Does not support undirected graph

`algo.betweenness.sampled()`

- Calculates betweenness-dependencies on a subset of pivot nodes (instead of all nodes). 2 randomization strategies are implemented, which can be set using the optional argument strategy: `random selection(default): strategy:'random':` (takes optional argument probability:double(0-1) or log10(N) / e^2 as default)
- Degree based randomization: `strategy:'degree':` (makes dense nodes more likely)
- Optional Arguments: `maxDepth:int`

The Closeness Centrality algorithm

This section describes the Closeness Centrality algorithm in the Neo4j Graph Algorithms 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.

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

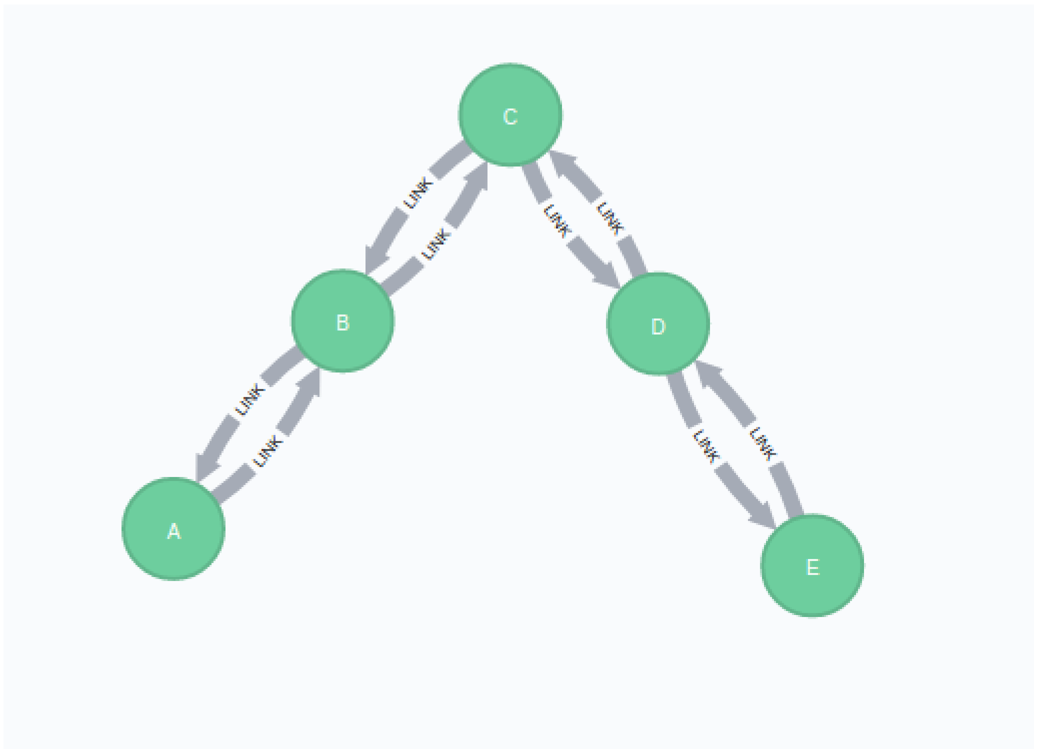
- Closeness centrality is used to research organizational networks, where individuals with high closeness centrality are in a favourable position to control and acquire vital information and resources within the organization. One such study is "Mapping Networks of Terrorist Cells" by Valdis E. Krebs.
- 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.

Closeness Centrality algorithm sample



The following will create a sample graph:

```
MERGE (a:Node{id:"A"})
MERGE (b:Node{id:"B"})
MERGE (c:Node{id:"C"})
MERGE (d:Node{id:"D"})
MERGE (e:Node{id:"E"})

MERGE (a)-[:LINK]->(b)
MERGE (b)-[:LINK]->(a)
MERGE (b)-[:LINK]->(c)
MERGE (c)-[:LINK]->(b)
MERGE (c)-[:LINK]->(d)
MERGE (d)-[:LINK]->(c)
MERGE (d)-[:LINK]->(e)
MERGE (e)-[:LINK]->(d);
```

The following will run the algorithm and stream results:

```
CALL algo.closeness.stream('Node', 'LINK')
YIELD nodeId, centrality

MATCH (n:Node) WHERE id(n) = nodeId

RETURN n.id AS node, centrality
ORDER BY centrality DESC
LIMIT 20;
```

The following will run the algorithm and write back results:

```
CALL algo.closeness('Node', 'LINK', {write:true, writeProperty:'centrality'})
YIELD nodes,loadMillis, computeMillis, writeMillis;
```

Table 16. Results

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.

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
```

Syntax

The following will run the algorithm and write back results:

```
CALL algo.closeness(label:String, relationship:String,
    {write:true, writeProperty:'centrality',graph:'heavy', concurrency:4})
YIELD nodes, loadMillis, computeMillis, writeMillis
```

Table 17. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
write	boolean	true	yes	Specifies if the result should be written back as a node property
concurrency	int	available CPUs	yes	The number of concurrent threads
writeProperty	string	'centrality'	yes	The property name written back to
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter,. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 18. Results

Name	Type	Description
nodes	int	The number of nodes considered
loadMillis	int	Milliseconds for loading data
evalMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the algorithm and stream results:

```
CALL algo.closeness.stream(label:String, relationship:String, {concurrency:4})
YIELD nodeId, centrality
```

Table 19. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
concurrency	int	available CPUs	yes	The number of concurrent threads
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 20. Results

Name	Type	Description
node	long	Node ID
centrality	float	Closeness centrality weight

Huge graph projection

If our projected graph contains more than 2 billion nodes or relationships, we need to use huge graph projection, as the default label and relationship-type projection has a limitation of 2 billion nodes and 2 billion relationships.

Set `graph:'huge'` in the config:

```
CALL algo.closeness('Node', 'LINK', {graph:'huge'})
YIELD nodes,loadMillis, computeMillis, writeMillis;
```

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` *in the config:*

```
CALL algo.closeness(
  'MATCH (p:Node) RETURN id(p) as id',
  'MATCH (p1:Node)-[:LINK]->(p2:Node) RETURN id(p1) as source, id(p2) as target',
  {graph:'cypher', write: true}
);
```

Versions

We support the following versions of the closeness centrality algorithm:

- [x] directed, unweighted
- [ ] directed, weighted
- [x] undirected, unweighted
    - Only with cypher projection
- [ ] undirected, weighted

The Harmonic Centrality algorithm

This section describes the Harmonic Centrality algorithm in the Neo4j Graph Algorithms library.

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 unconnected graphs. As with many of the centrality algorithms, it originates from the field of social network analysis.

History and explanation

Harmonic centrality was proposed by Marchiori and Latora in Harmony in the Small World while trying to come up with a sensible notion of "average shortest path".

They suggested a different way of calculating the average distance to that used in the Closeness Centrality algorithm. Rather than summing the distances of a node to all other nodes, the harmonic centrality algorithm sums the inverse of those distances. This enables it deal with infinite values.

The **raw harmonic centrality** for a node is calculated using the following formula:

`raw harmonic centrality(node) = sum(1 / distance from node to every other node excluding itself)`

As with closeness centrality, we can also calculate a **normalized harmonic centrality** with the following formula:

`normalized harmonic centrality(node) = sum(1 / distance from node to every other node excluding itself) / (number of nodes - 1)`

In this formula, ∞ values are handled cleanly.

Use-cases - when to use the Harmonic Centrality algorithm

Harmonic centrality was proposed as an alternative to closeness centrality, and therefore has similar use cases.

For example, we might use it if we're trying to identify where in the city to place a new public service so that it's easily accessible for residents. If we're trying to spread a message on social media we could use the algorithm to find the key influencers that can help us achieve our goal.

Harmonic Centrality algorithm sample

The following will create a sample graph:

```
MERGE (a:Node{id:"A"})
MERGE (b:Node{id:"B"})
MERGE (c:Node{id:"C"})
MERGE (d:Node{id:"D"})
MERGE (e:Node{id:"E"})

MERGE (a)-[:LINK]->(b)
MERGE (b)-[:LINK]->(c)
MERGE (d)-[:LINK]->(e);
```

The following will run the algorithm and stream results:

```
CALL algo.closeness.harmonic.stream('Node', 'LINKS') YIELD nodeId, centrality
RETURN nodeId,centrality
ORDER BY centrality DESC
LIMIT 20;
```

The following will run the algorithm and write back results:

```
CALL algo.closeness.harmonic('Node', 'LINK', {writeProperty:'centrality'})
YIELD nodes,loadMillis, computeMillis, writeMillis;
```

Calculation:

`k = N-1 = 4`

```
     A     B     C     D     E
 ---|-----------------------------
 A  | 0     1     2     -     -    // distance between each pair of nodes
 B  | 1     0     1     -     -    // or infinite if no path exists
 C  | 2     1     0     -     -
 D  | -     -     -     0     1
 E  | -     -     -     1     0
 ---|------------------------------
 A  | 0     1    1/2    0     0    // inverse
 B  | 1     0     1     0     0
 C  |1/2    1     0     0     0
 D  | 0     0     0     0     1
 E  | 0     0     0     1     0
 ---|------------------------------
sum |1.5    2    1.5    1     1
 ---|------------------------------
 *k |0.37  0.5  0.37  0.25  0.25
```

Instead of calculating the farness, we sum the inverse of each cell and multiply by `1/(n-1)`.

Syntax

The following will run the algorithm and write back results:

```
CALL algo.closeness.harmonic(label:String, relationship:String,
    {write:true, writeProperty:'centrality', graph:'heavy', concurrency:4})
YIELD nodes, loadMillis, computeMillis, writeMillis
```

Table 21. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
write	boolean	true	yes	Specifies if the result should be written back as a node property
concurrency	int	available CPUs	yes	The number of concurrent threads
writeProperty	string	'centrality'	yes	The property name written back to
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 22. Results

Name	Type	Description
nodes	int	The number of nodes considered
loadMillis	int	Milliseconds for loading data
evalMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the algorithm and stream results:

```
CALL algo.closeness.harmonic.stream(label:String, relationship:String,
{concurrency:4})
YIELD nodeId, centrality
```

Table 23. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
concurrency	int	available CPUs	yes	The number of concurrent threads

Table 24. Results

Name	Type	Description
node	long	Node ID
centrality	float	Closeness centrality weight

Huge graph projection

If our projected graph contains more than 2 billion nodes or relationships, we need to use huge graph projection, as the default label and relationship-type projection has a limitation of 2 billion nodes and 2 billion relationships.

Set `graph:'huge'` in the config:

```
CALL algo.closeness.harmonic('Node', 'LINK', {graph:'huge'})
YIELD nodes,loadMillis, computeMillis, writeMillis;
```

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` in the config:

```
CALL algo.closeness.harmonic(
  'MATCH (p:Node) RETURN id(p) as id',
  'MATCH (p1:Node)-[:LINK]-(p2:Node) RETURN id(p1) as source, id(p2) as target',
  {graph:'cypher', writeProperty: 'centrality'}
);
```

Versions

We support the following versions of the harmonic centrality algorithm:

- ☑ undirected, unweighted
- ☐ undirected, weighted

The Minimum Weight Spanning Tree algorithm

***This section describes the Minimum Weight Spanning Tree algorithm in the** Neo4j Graph Algorithms 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.

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

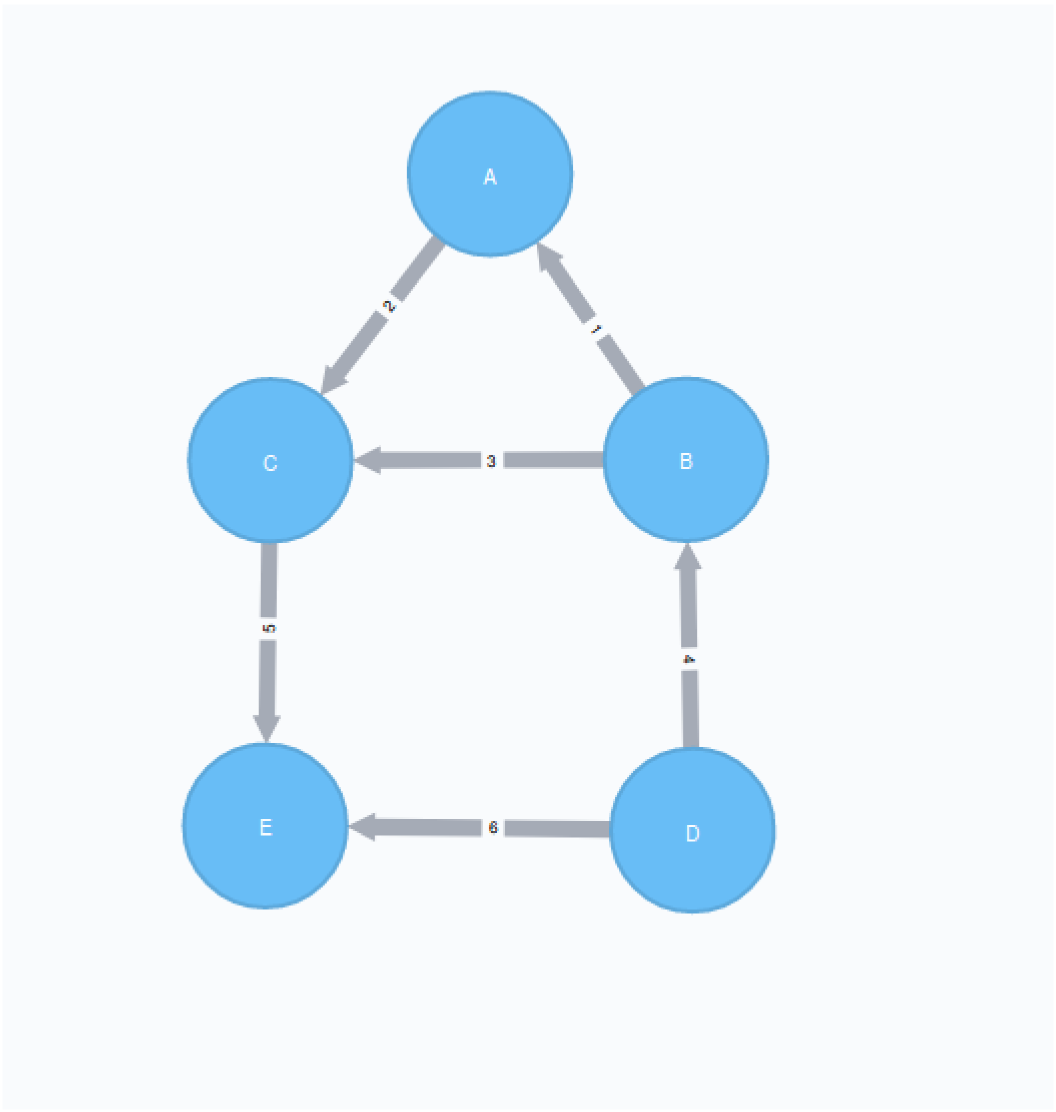
- Minimum spanning tree was applied to analyze airline and sea connections of Papua New Guinea, and minimize the travel cost of exploring the country. It could be used to help design low-cost tours that visit many destinations across the country. The research mentioned can be found in "An Application 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.

Minimum Weight Spanning Tree algorithm sample



The following will create a sample graph:

```
MERGE (a:Place {id:"A"})
MERGE (b:Place {id:"B"})
MERGE (c:Place {id:"C"})
MERGE (d:Place {id:"D"})
MERGE (e:Place {id:"E"})
MERGE (f:Place {id:"F"})
MERGE (g:Place {id:"G"})

MERGE (d)-[:LINK {cost:4}]->(b)
MERGE (d)-[:LINK {cost:6}]->(e)
MERGE (b)-[:LINK {cost:1}]->(a)
MERGE (b)-[:LINK {cost:3}]->(c)
MERGE (a)-[:LINK {cost:2}]->(c)
MERGE (c)-[:LINK {cost:5}]->(e)
MERGE (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:

```
MATCH (n:Place {id:"D"})
CALL algo.spanningTree.minimum('Place', 'LINK', 'cost', id(n),
  {write:true, writeProperty:"MINST"})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis, computeMillis, writeMillis, effectiveNodeCount;
```

The following will query minimum spanning tree:

```
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.cost AS cost
```

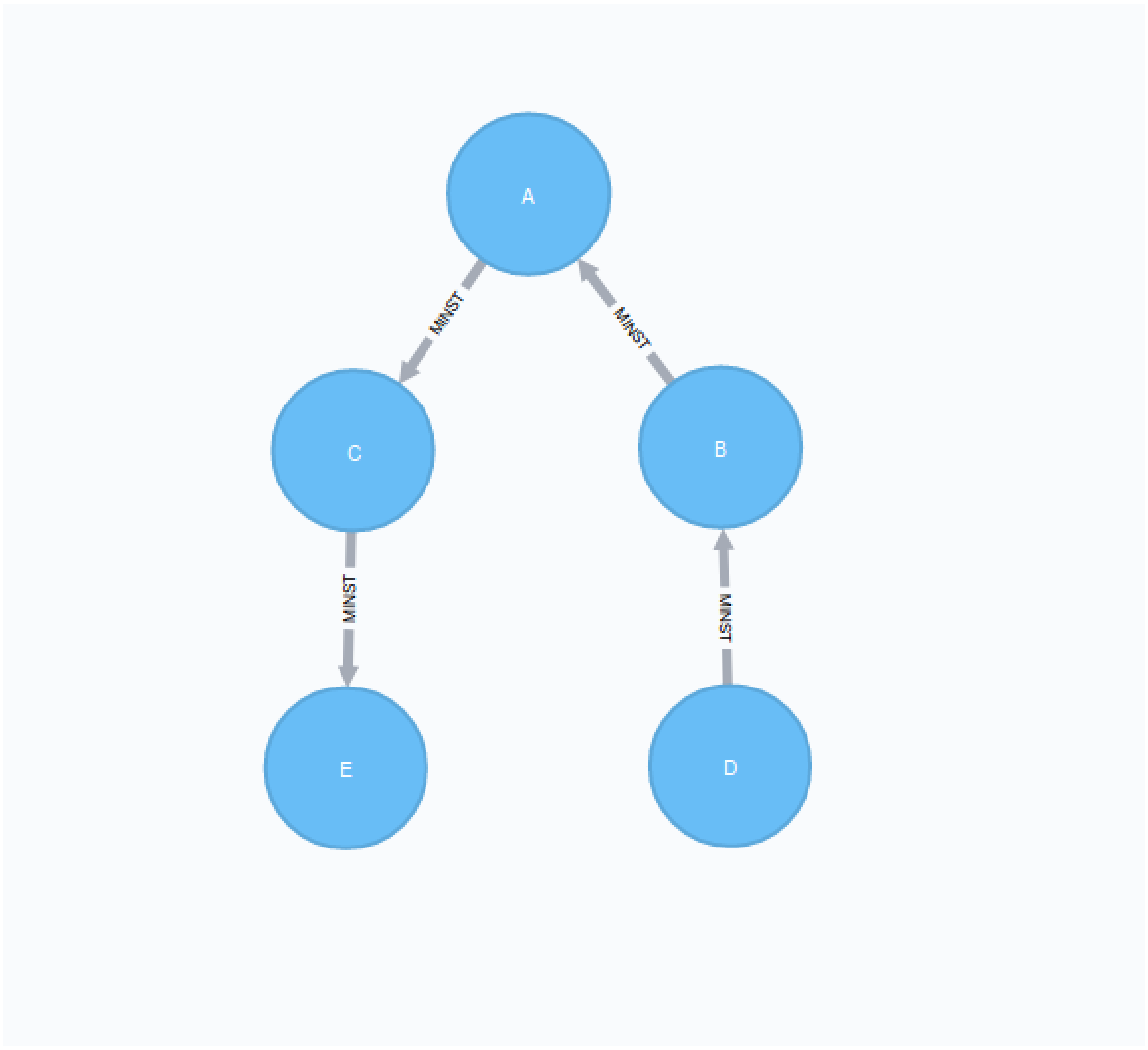


Figure 1. Results

To find all pairs of nodes included in our minimum spanning tree, run the following query:

Table 25. Results

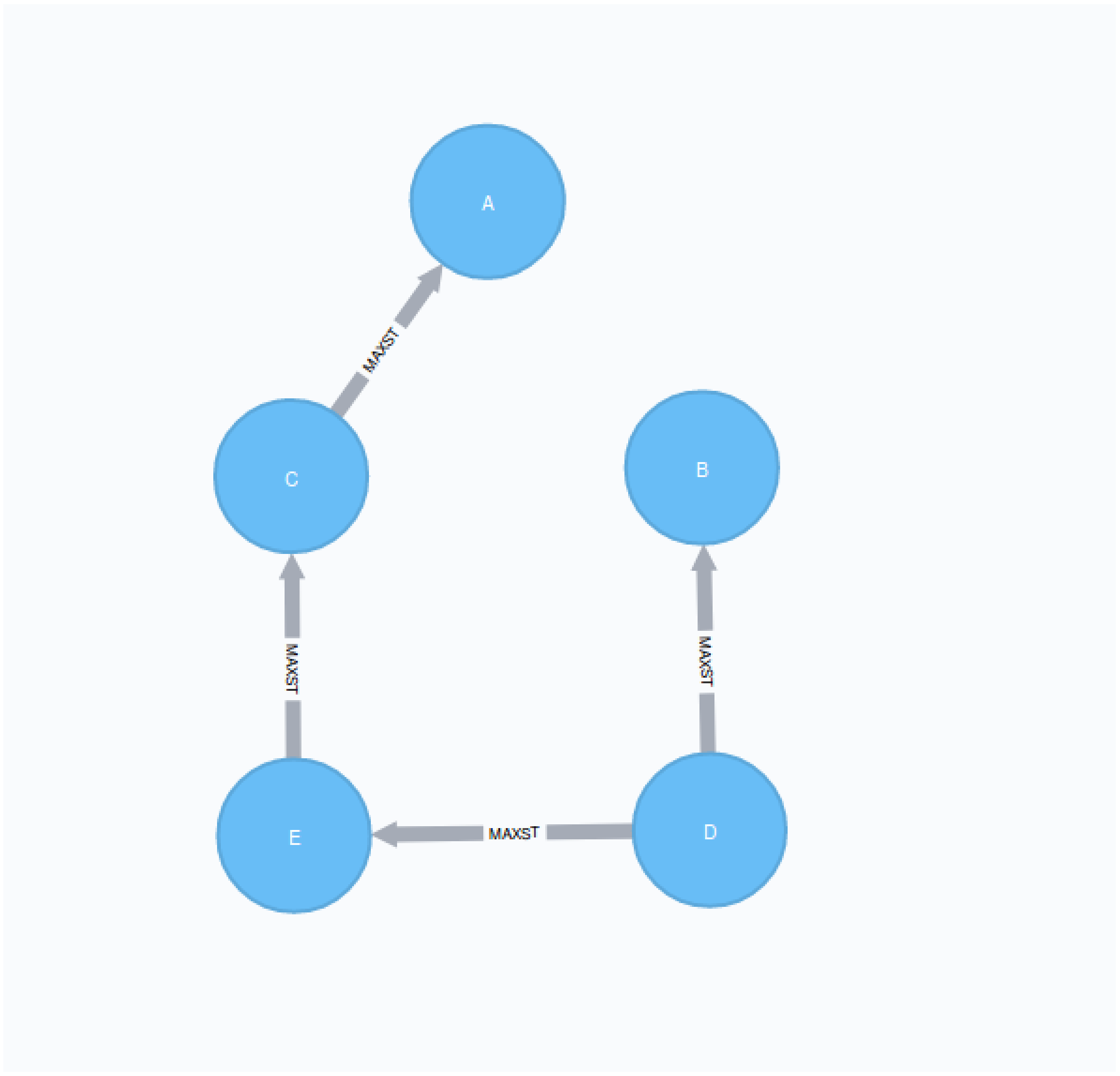
Source	Destination	Cost
D	B	4
B	A	1
A	C	2
C	E	5

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:

```
MATCH (n:Place{id:"D"})
CALL algo.spanningTree.maximum('Place', 'LINK', 'cost', id(n),
  {write:true, writeProperty:"MAXST"})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis,computeMillis, writeMillis, effectiveNodeCount;
```



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:

```
MATCH (n:Place{id:"D"})
CALL algo.spanningTree.kmin('Place', 'LINK', 'cost',id(n), 3,
  {writeProperty:"kminst"})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis,computeMillis,writeMillis, effectiveNodeCount;
```

Table 26. Results

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:

```
MATCH (n:Place{id:"D"})
CALL algo.spanningTree.kmax('Place', 'LINK', 'cost', id(n), 3,
  {writeProperty:"kmaxst"})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis,computeMillis,writeMillis, effectiveNodeCount;
```

Table 27. Results

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.

When we run this algorithm on a bigger graph, we can use the following query to find nodes that belong to our k-spanning tree result:

Find nodes that belong to our k-spanning tree result:

```
MATCH (n:Place)
WITH n.partition AS partition, count(*) as count
WHERE count = k
RETURN n
```

Syntax

The following will run the algorithm and write back results:

```
CALL algo.spanningTree(label:String, relationshipType:String, weightProperty:String,
startNodeId:int, {writeProperty:String})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
```

Table 28. Parameters

Name	Type	Default	Optional	Description
label	String	null	no	The label to load from the graph. If null, load all nodes
relationshipType	String	null	no	The relationship-type to load from the graph. If null, load all nodes
weightProperty	string	null	no	The property name that contains weight. Must be numeric.
startNodeId	long	null	no	The start node ID
write	boolean	true	yes	Specify if the result should be written back as relationships
writeProperty	string	'mst'	yes	The relationship-type written back as result

Table 29. Results

Name	Type	Description
effectiveNodeCount	int	The number of visited nodes
loadMillis	int	Milliseconds for loading data
computeMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the k-spanning tree algorithm and write back results:

```
CALL algo.spanningTree.k*(label:String, relationshipType:String,
weightProperty:String, startNodeId:int, k:int, {writeProperty:String})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
```

Table 30. Parameters

Name	Type	Default	Optional	Description
label	String	null	no	The label to load from the graph. If null, load all nodes
relationshipType	String	null	no	The relationship type
weightProperty	string	null	no	The property name that contains weight. Must be numeric.
startNodeId	int	null	no	The start node ID
k	int	null	no	The result is a tree with k nodes and $k - 1$ relationships
write	boolean	true	yes	Specifies if the result should be written back as a node property
writeProperty	string	'mst'	yes	The relationship-type written back as result

Table 31. Results

Name	Type	Description
effectiveNodeCount	int	The number of visited nodes
loadMillis	int	Milliseconds for loading data
computeMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

Versions

We support the following versions of the Minimum Weight Spanning Tree algorithm:

✓ undirected, weighted

The Shortest Path algorithm

This section describes the Shortest Path algorithm in the Neo4j Graph Algorithms library.

The Shortest Path algorithm calculates the shortest (weighted) path between a pair of nodes. In this category, Dijkstra’s algorithm is the most well known. It is a real time graph algorithm, and can be used as part of the normal user flow in a web or mobile application.

History and explanation

Path finding has a long history, and is considered to be one of the classical graph problems; it has been researched as far back as the 19th century. It gained prominence in the early 1950s in the context of ‘alternate routing’, i.e. finding a second shortest route if the shortest route is blocked.

Dijkstra came up with his algorithm in 1956 while trying to come up with something to show off the new ARMAC computers. He needed to find a problem and solution that people not familiar with computing would be able to understand, and designed what is now known as Dijkstra’s algorithm. He later implemented it for a slightly simplified transportation map of 64 cities in the Netherlands.

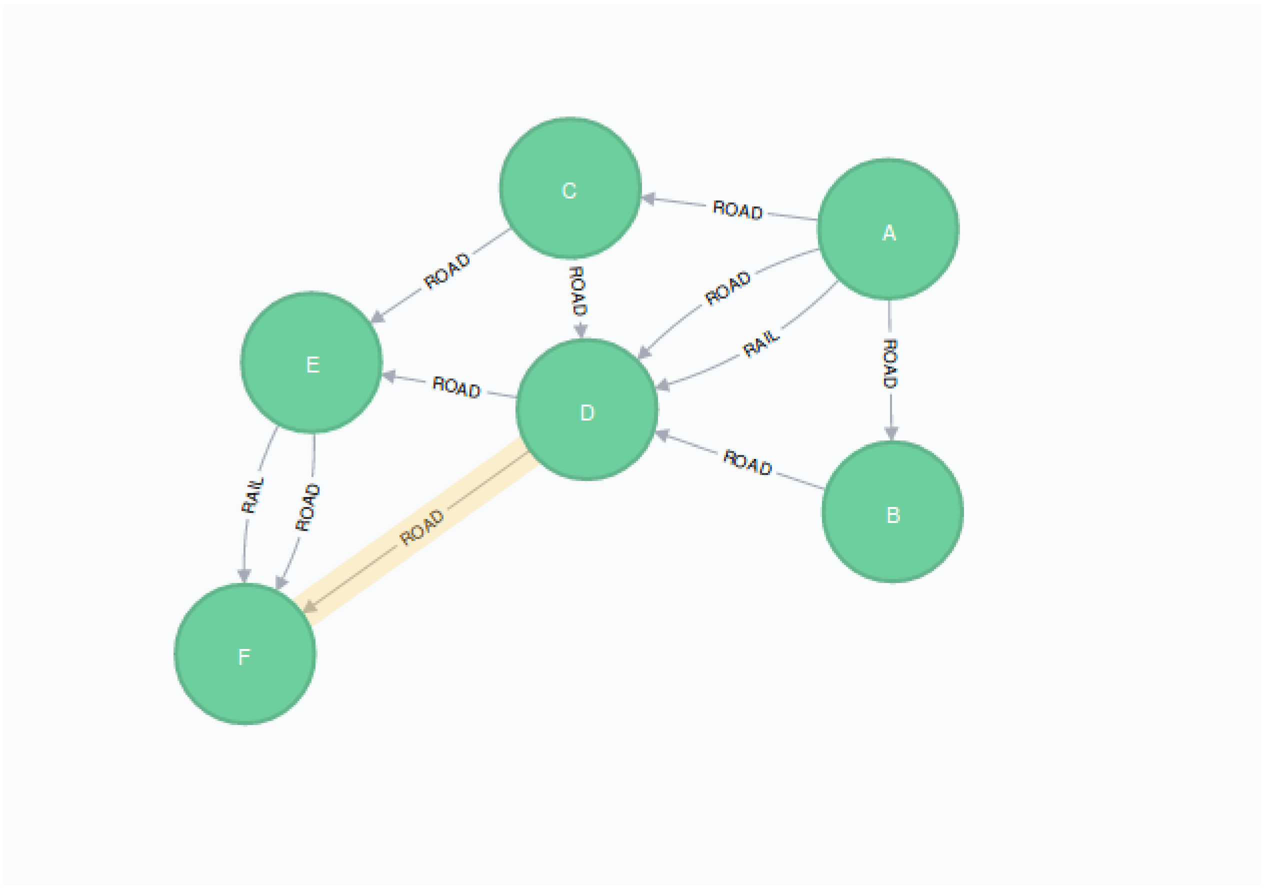
Use-cases - when to use the Shortest Path algorithm

- Finding directions between physical locations. This is the most common usage, and web mapping tools such as Google Maps use the shortest path algorithm, or a variant of it, to provide driving directions.
- Social networks can use the algorithm to find the degrees of separation between people. For example, when you view someone’s profile on LinkedIn, it will indicate how many people separate you in the connections graph, as well as listing your mutual connections.

Constraints - when not to use the Shortest Path algorithm

Dijkstra 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.

Shortest Path algorithm sample



The following will create a sample graph:

```
MERGE (a:Loc {name:'A'})
MERGE (b:Loc {name:'B'})
MERGE (c:Loc {name:'C'})
MERGE (d:Loc {name:'D'})
MERGE (e:Loc {name:'E'})
MERGE (f:Loc {name:'F'})

MERGE (a)-[:ROAD {cost:50}]->(b)
MERGE (a)-[:ROAD {cost:50}]->(c)
MERGE (a)-[:ROAD {cost:100}]->(d)
MERGE (b)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:80}]->(e)
MERGE (d)-[:ROAD {cost:30}]->(e)
MERGE (d)-[:ROAD {cost:80}]->(f)
MERGE (e)-[:ROAD {cost:40}]->(f);
```

The Dijkstra Shortest Path algorithm

The following will run the algorithm and stream results:

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.shortestPath.stream(start, end, 'cost')
YIELD nodeId, cost
MATCH (other:Loc) WHERE id(other) = nodeId
RETURN other.name AS name, cost
```

The following will run the algorithm and write back results:

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.shortestPath(start, end, 'cost',{write:true,writeProperty:'sssp'})
YIELD writeMillis,loadMillis,nodeCount, totalCost
RETURN writeMillis,loadMillis,nodeCount,totalCost
```

Table 32. Results

Name	Cost
A	0
C	50
D	90
E	120
F	160

The quickest route takes us from A to F, via C, D, and E, at a total cost of 160:

- First, we go from A to C, at a cost of 50.

- Then, we go from C to D, for an additional 40.
- Then, from D to E, for an additional 30.
- Finally, from E to F, for a further 40.

Syntax

The following will run the algorithm and write back results:

```
CALL algo.shortestPath(startNode:Node, endNode:Node, weightProperty:String
    {nodeQuery:'labelName', relationshipQuery:'relationshipName', defaultValue:1.0,
write:'true', writeProperty:'sssp', direction:'OUTGOING'})
YIELD nodeCount, totalCost, loadMillis, evalMillis, writeMillis
```

Table 33. Parameters

Name	Type	Default	Optional	Description
startNode	node	null	no	The start node
endNode	node	null	no	The end node
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
defaultValue	float	null	yes	The default value of the weight in case it is missing or invalid
write	boolean	true	yes	Specifies if the result should be written back as a node property
writeProperty	string	'sssp'	yes	The property name written back to the node sequence of the node in the path
nodeQuery	string	null	yes	The label to load from the graph. If null, load all nodes
relationshipQuery	string	null	yes	The relationship-type to load from the graph. If null, load all nodes
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected

Table 34. Results

Name	Type	Description
nodeCount	int	The number of nodes considered
totalCost	float	The sum of all weights along the path
loadMillis	int	Milliseconds for loading data
evalMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the algorithm and stream results:

```
CALL algo.shortestPath.stream(startNode:Node, endNode:Node, weightProperty:String
    {nodeQuery:'labelName', relationshipQuery:'relationshipName', defaultValue:1.0,
direction:'OUTGOING'})
 YIELD nodeId, cost
```

Table 35. Parameters

Name	Type	Default	Optional	Description
startNode	node	null	no	The start node
endNode	node	null	no	The end node
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
nodeQuery	string	null	yes	The label to load from the graph. If null, load all nodes
relationshipQuery	string	null	yes	The relationship-type to load from the graph. If null, load all nodes
defaultValue	float	null	yes	The default value of the weight in case it is missing or invalid
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected

Table 36. Results

Name	Type	Description
nodeId	int	Node ID
cost	int	The cost it takes to get from start node to specific node

Versions

We support the following versions of the shortest path algorithms:

- ✓ directed, unweighted:
  - direction: 'OUTGOING' or INCOMING, weightProperty: null
- ✓ directed, weighted
  - direction: 'OUTGOING' or INCOMING, weightProperty: 'cost'
- ✓ undirected, unweighted
  - direction: 'BOTH', weightProperty: null
- ✓ undirected, weighted
  - direction: 'BOTH', weightProperty: 'cost'

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` *in the config:*

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.shortestPath(start, end, 'cost',{
nodeQuery:'MATCH(n:Loc) WHERE not n.name = "c" RETURN id(n) as id',
relationshipQuery:'MATCH(n:Loc)-[r:ROAD]->(m:Loc) RETURN id(n) as source, id(m) as
target, r.cost as weight',
graph:'cypher'})
YIELD writeMillis,loadMillis,nodeCount, totalCost
RETURN writeMillis,loadMillis,nodeCount,totalCost
```

Implementations

`algo.shortestPath`

- Specify start and end node, find the shortest path between them.
- Dijkstra single source shortest path algorithm.
- There may be more then one shortest path, algorithm returns only one.
- If initialized with an non-existing weight-property, it will treat the graph as unweighted.

`algo.shortestPaths`

- Specify start node, find the shortest paths to all other nodes.
- Dijkstra single source shortest path algorithm.
- If initialized with an non-existing weight-property, it will treat the graph as unweighted.

The Single Source Shortest Path algorithm

This section describes the Single Source Shortest Path algorithm in the Neo4j Graph Algorithms library.

The Single Source Shortest Path (SSSP) algorithm calculates the shortest (weighted) path from a node to all other nodes in the graph.

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.

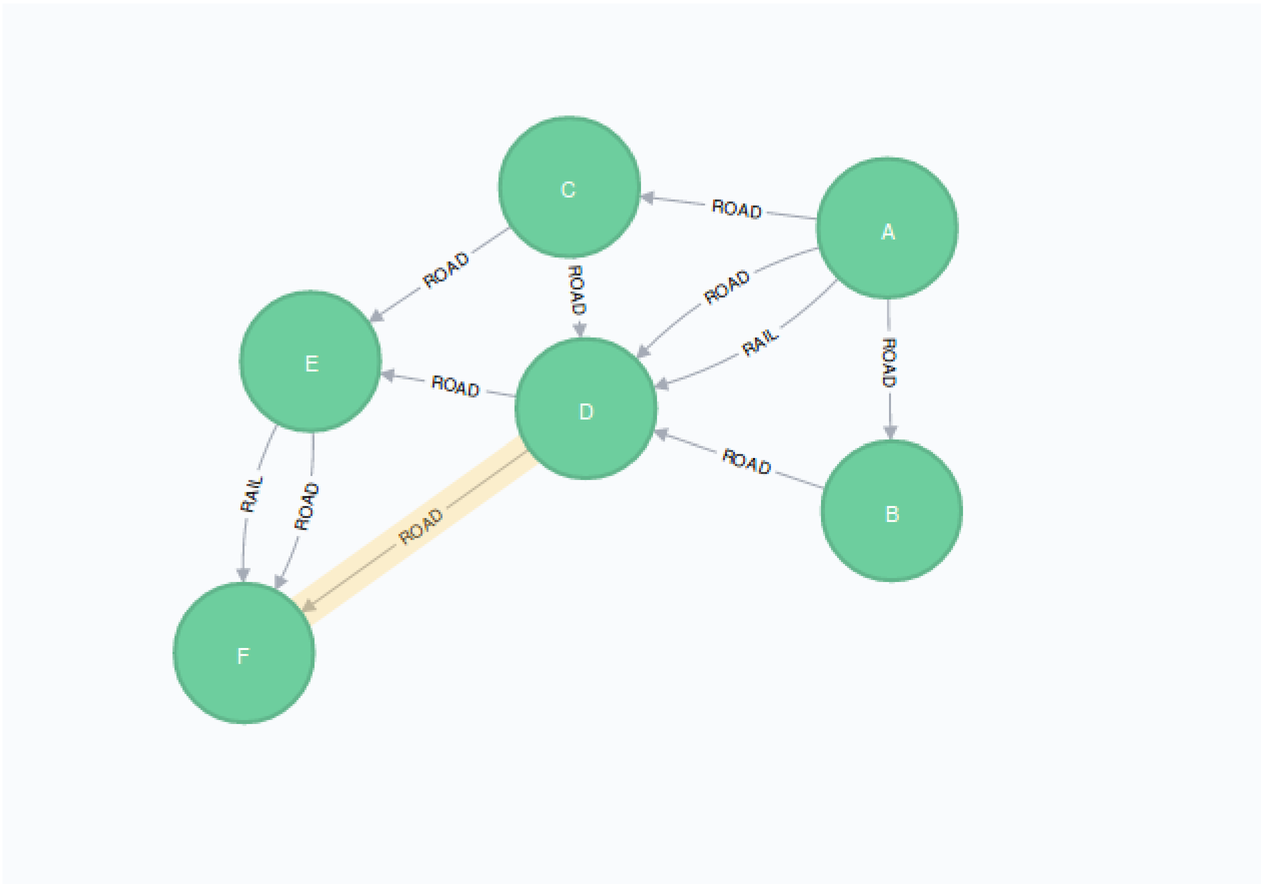
Use-cases - when to use the Single Source Shortest Path algorithm

- 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.

Single Source Shortest Path algorithm sample



The following will create a sample graph:

```
MERGE (a:Loc {name:'A'})
MERGE (b:Loc {name:'B'})
MERGE (c:Loc {name:'C'})
MERGE (d:Loc {name:'D'})
MERGE (e:Loc {name:'E'})
MERGE (f:Loc {name:'F'})

MERGE (a)-[:ROAD {cost:50}]->(b)
MERGE (a)-[:ROAD {cost:50}]->(c)
MERGE (a)-[:ROAD {cost:100}]->(d)
MERGE (b)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:80}]->(e)
MERGE (d)-[:ROAD {cost:30}]->(e)
MERGE (d)-[:ROAD {cost:80}]->(f)
MERGE (e)-[:ROAD {cost:40}]->(f);
```

Delta stepping algorithm

The following will run the algorithm and stream results:

```
MATCH (n:Loc {name:'A'})
CALL algo.shortestPath.deltaStepping.stream(n, 'cost', 3.0)
YIELD nodeId, distance

MATCH (destination) WHERE id(destination)  = nodeId

RETURN destination.name AS destination, distance
```

The following will run the algorithm and write back results:

```
MATCH (n:Loc {name:'A'})
CALL algo.shortestPath.deltaStepping(n, 'cost', 3.0, {defaultValue:1.0, write:true,
writeProperty:'sssp'})
YIELD nodeCount, loadDuration, evalDuration, writeDuration
RETURN nodeCount, loadDuration, evalDuration, writeDuration
```

Table 37. Results

Name	Cost
A	0
B	50
C	50
D	90
E	120

Name	Cost
F	160

The above table shows the cost of going from A to each of the other nodes, including itself at a cost of 0.

Syntax

The following will run the algorithm and write back results:

```
CALL algo.shortestPath.deltaStepping(startNode:Node, weightProperty:String,
delta:Float,
    {defaultValue:1.0, write:true, writeProperty:'sssp'})
YIELD nodeCount, loadDuration, evalDuration, writeDuration
RETURN nodeCount, loadDuration, evalDuration, writeDuration
```

Table 38. Parameters

Name	Type	Default	Optional	Description
startNode	node	null	no	The start node
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
delta	float	null	yes	The grade of concurrency to use.
write	boolean	true	yes	Specifies if the result should be written back as a node property
writeProperty	string	'sssp'	yes	The property name written back to the node sequence of the node in the path
nodeQuery	string	null	yes	The label to load from the graph. If null, load all nodes
relationshipQuery	string	null	yes	The relationship-type to load from the graph. If null, load all nodes
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected

Table 39. Results

Name	Type	Description
nodeCount	int	The number of nodes considered
totalCost	float	The sum of all weights along the path
loadMillis	int	Milliseconds for loading data
evalMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the algorithm and stream results:

```
CALL algo.shortestPath.deltaStepping.stream(startNode:Node, weightProperty:String,
delta: Float,
    {nodeQuery:'labelName', relationshipQuery:'relationshipName', defaultValue:1.0,
direction:'OUTGOING'})
YIELD nodeId, cost
```

Table 40. Parameters

Name	Type	Default	Optional	Description
startNode	node	null	no	The start node
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
delta	float	null	yes	The grade of concurrency to use.
nodeQuery	string	null	yes	The label to load from the graph. If null, load all nodes
relationshipQuery	string	null	yes	The relationship-type to load from the graph. If null, load all nodes
defaultValue	float	null	yes	The default value of the weight in case it is missing or invalid
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected

Table 41. Results

Name	Type	Description
nodeId	int	Node ID
cost	int	The cost it takes to get from start node to specific node

Versions

We support the following versions of the shortest path algorithms:

- ✓ directed, unweighted:
  - direction: 'OUTGOING' or INCOMING, weightProperty: null
- ✓ directed, weighted
  - direction: 'OUTGOING' or INCOMING, weightProperty: 'cost'
- ✓ undirected, unweighted
  - direction: 'BOTH', weightProperty: null
- ✓ undirected, weighted
  - direction: 'BOTH', weightProperty: 'cost'

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` in the config:

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.shortestPath(start, end, 'cost',{
nodeQuery:'MATCH(n:Loc) WHERE not n.name = "c" RETURN id(n) as id',
relationshipQuery:'MATCH(n:Loc)-[r:ROAD]->(m:Loc) RETURN id(n) as source, id(m) as
target, r.cost as weight',
graph:'cypher'})
YIELD writeMillis,loadMillis,nodeCount, totalCost
RETURN writeMillis,loadMillis,nodeCount,totalCost
```

Implementations

`algo.shortestPath.deltaStepping`

- Specify start node, find the shortest paths to all other nodes.
- Parallel non-negative single source shortest path algorithm for weighted graphs.
- It can be tweaked using the delta-parameter which controls the grade of concurrency.
- If initialized with an non-existing weight-property, it will treat the graph as unweighted.

The A* algorithm

This section describes the A* algorithm in the Neo4j Graph Algorithms library.

The A* (pronounced “A-star”) algorithm improves on the classic Dijkstra algorithm. It is based upon the observation that some searches are informed, and that by being informed we can make better choices over which paths to take through the graph.

History and explanation

The A* algorithm was first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael. For more information, see A Formal Basis for the Heuristic Determination of Minimum Cost Paths.

In A*, we split the path cost into two parts:

`g(n)`
: This is the cost of the path from the starting point to some node n.

`h(n)`
: This represents the estimated cost of the path from the node n to the destination node, as

computed by a heuristic (an intelligent guess).

The A* algorithm balances `g(n)` and `h(n)` as it iterates the graph, thereby ensuring that at each iteration it chooses the node with the lowest overall cost `f(n) = g(n) + h(n)`.

In our implementation, geospatial distance is used as heurestic.

Use-cases - when to use the A* algorithm

- The A* algorithm can be used to find shortest paths between single pairs of locations, where GPS coordinates are known.

A* algorithm sample

The following will create a sample graph:

```
MERGE (a:Station{name:"King's Cross St. Pancras"})
SET a.latitude = 51.5308,a.longitude = -0.1238
MERGE (b:Station{name:"Euston"})
SET b.latitude = 51.5282, b.longitude = -0.1337
MERGE (c:Station{name:"Camden Town"})
SET c.latitude = 51.5392, c.longitude = -0.1426
MERGE (d:Station{name:"Mornington Crescent"})
SET d.latitude = 51.5342, d.longitude = -0.1387
MERGE (e:Station{name:"Kentish Town"})
SET e.latitude = 51.5507, e.longitude = -0.1402
MERGE (a)-[:CONNECTION{time:2}]->(b)
MERGE (b)-[:CONNECTION{time:3}]->(c)
MERGE (b)-[:CONNECTION{time:2}]->(d)
MERGE (d)-[:CONNECTION{time:2}]->(c)
MERGE (c)-[:CONNECTION{time:2}]->(e);
```

The following will run the algorithm and stream results:

```
MATCH (start:Station{name:"King's Cross St. Pancras"}),(end:Station{name:"Kentish
Town"})
CALL algo.shortestPath.astar.stream(start, end, 'time', 'latitude', 'longitude',
{defaultValue:1.0})
YIELD nodeId, cost
MATCH (n) where id(n) = nodeId
RETURN n.name as station,cost
```

Table 42. Results

Name	Cost
King’s Cross St. Pancras	0
Euston	2
Camden Town	5

Name	Cost
Kentish Town	7

Syntax

The following will run the algorithm and stream results:

```
CALL algo.shortestPath.astar.stream((startNode:Node, endNode:Node,
weightProperty:String, propertyKeyLat:String, propertyKeyLon:String,
    {nodeQuery:'labelName', relationshipQuery:'relationshipName', direction:'BOTH',
defaultValue:1.0})
YIELD nodeId, cost
```

Table 43. Parameters

Name	Type	Default	Optional	Description
startNode	node	null	no	The start node
endNode	node	null	no	The end node
weightProperty	string	null	yes	The property name that contains weight
propertyKeyLat	string	null	no	The property name that contains latitude coordinate
propertyKeyLon	string	null	no	The property name that contains longitude coordinate
nodeQuery	string	null	yes	The label to load from the graph. If null, load all nodes
relationshipQuery	string	null	yes	The relationship-type to load from the graph. If null, load all nodes
defaultValue	float	null	yes	The default value of the weight in case it is missing or invalid
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected

Table 44. Results

Name	Type	Description
nodeId	int	Node ID
cost	int	The cost it takes to get from start node to specific node

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` in the config:

```
MATCH (start:Station{name:"King's Cross St. Pancras"}),(end:Station{name:"Kentish
Town"})
CALL algo.shortestPath.astar.stream(start, end, 'time','latitude','longitude',{
nodeQuery:'MATCH (p:Station) RETURN id(p) as id',
relationshipQuery:'MATCH (p1:Station)-[r:CONNECTION]->(p2:Station) RETURN id(p1) as
source, id(p2) as target,r.time as weight',
graph:'cypher'})
YIELD nodeId, cost
RETURN nodeId,cost
```

Versions

We support the following versions of the shortest path algorithms:

- ✓ directed, unweighted:
  - direction: 'OUTGOING' or INCOMING, weightProperty: null
- ✓ directed, weighted
  - direction: 'OUTGOING' or INCOMING, weightProperty: 'cost'
- ✓ undirected, unweighted
  - direction: 'BOTH', weightProperty: null
- ✓ undirected, weighted
  - direction: 'BOTH', weightProperty: 'cost'

Implementations

`algo.shortestPath.astar.stream()`

- Implementation of A* heuristic function is for geospatial distances.

The Yen’s K-shortest paths algorithm

This section describes the Yen’s K-shortest paths algorithm in the Neo4j Graph Algorithms library.

Yen’s K-shortest paths algorithm computes single-source K-shortest loopless paths for a graph with non-negative relationship weights.

History and explanation

Algorithm was defined in 1971 by Jin Y. Yen in the research paper Finding the K Shortest Loopless Paths in a Network. Our implementation uses Dijkstra algorithm to find the shortest path and then proceeds to find k-1 deviations of the shortest paths.

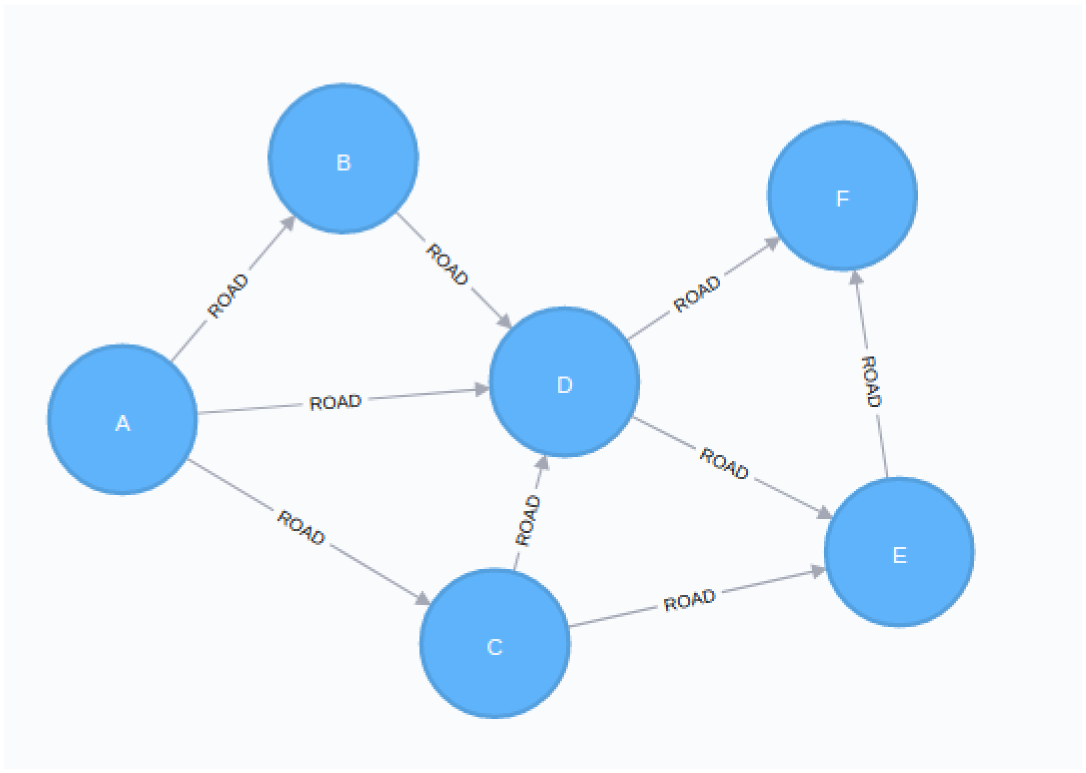
Use-cases - when to use the Yen's K-shortest paths algorithm

- K-shortest paths algorithm has been used to optimize multiple object tracking by formalizing the motions of targets as flows along the relationships of the spatial graph. Find more in Multiple Object Tracking using K-Shortest Paths Optimization
- K-shortest paths algorithm is used to study alternative routing on road networks and to recommend top k-paths to the user. Find this study in Alternative Routing: k-Shortest Paths with Limited Overlap
- K-shortest paths algorithm has been used as part of Finding Diverse High-Quality Plans for Hypothesis Generation process.

Constraints - when not to use the Yen's K-shortest paths algorithm

Yen's K-Shortest paths algorithm 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.

Yen's K-shortest paths algorithm sample



The following will create a sample graph:

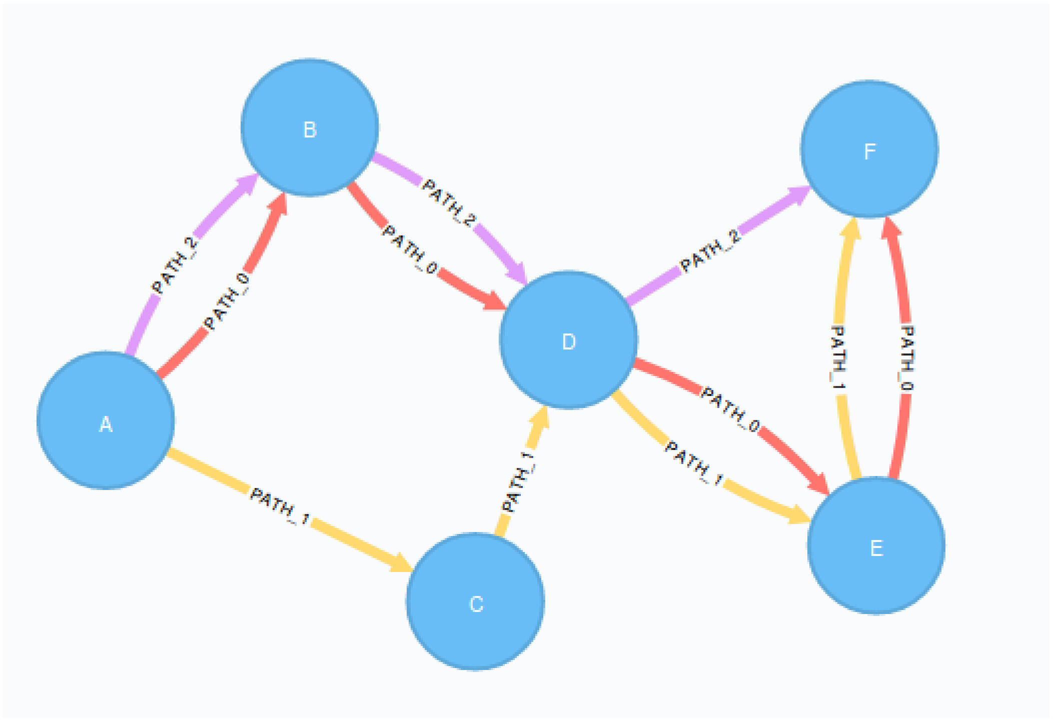
```
MERGE (a:Loc {name:'A'})
MERGE (b:Loc {name:'B'})
MERGE (c:Loc {name:'C'})
MERGE (d:Loc {name:'D'})
MERGE (e:Loc {name:'E'})
MERGE (f:Loc {name:'F'})

MERGE (a)-[:ROAD {cost:50}]->(b)
MERGE (a)-[:ROAD {cost:50}]->(c)
MERGE (a)-[:ROAD {cost:100}]->(d)
MERGE (b)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:80}]->(e)
MERGE (d)-[:ROAD {cost:30}]->(e)
MERGE (d)-[:ROAD {cost:80}]->(f)
MERGE (e)-[:ROAD {cost:40}]->(f);
```

The following will run the algorithm and write back results:

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.kShortestPaths(start, end, 3, 'cost' ,{})
YIELD resultCount
RETURN resultCount
```

The following will return all 3 of the shortest path:

```
MATCH p=()-[r:PATH_0|:PATH_1|:PATH_2]->() RETURN p LIMIT 25
```



The quickest route takes us from A to B, via D and E and is saved as `PATH_0`. Second quickest path is saved as `PATH_1` and third one is saved as`PATH_2`

Syntax

The following will run the algorithm and write back results:

```
CALL algo.kShortestPaths(startNode:Node, endNode:Node, k:int, weightProperty:String,
    {nodeQuery:'labelName', relationshipQuery:'relationshipName', direction:'OUT',
defaultValue:1.0,
    maxDepth:42, write:'true', writePropertyPrefix:'PATH_'})
YIELD resultCount, loadMillis, evalMillis, writeMillis
```

Table 45. Parameters

Name	Type	Default	Optional	Description
startNode	node	null	no	The start node
endNode	node	null	no	The end node
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
nodeQuery	string	null	yes	The label to load from the graph. If null, load all nodes
relationshipQuery	string	null	yes	The relationship-type to load from the graph. If null, load all nodes
direction	string	outgoing	yes	The relationship direction to load from the graph. If 'both', treats the relationships as undirected
defaultValue	float	null	yes	The default value of the weight in case it is missing or invalid
maxDepth	int	Integer.MAX	yes	The depth of the shortest paths traversal
write	boolean	true	yes	Specifies if the result should be written back as a node property
writePropertyPrefix	string	'PATH_'	yes	The relationship-type prefix written back to the graph

Table 46. Results

Name	Type	Description
resultCount	int	The number of shortest paths results
loadMillis	int	Milliseconds for loading data
evalMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

Versions

We support the following versions of the shortest path algorithms:

- ☑ directed, unweighted:
  - direction: 'OUTGOING' or INCOMING, weightProperty: null
- ☑ directed, weighted
  - direction: 'OUTGOING' or INCOMING, weightProperty: 'cost'
- ☑ undirected, unweighted
  - direction: 'BOTH', weightProperty: null
- ☑ undirected, weighted
  - direction: 'BOTH', weightProperty: 'cost'

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` in the config:

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.kShortestPaths(start, end, 3, 'cost',{
nodeQuery:'MATCH(n:Loc) WHERE not n.name = "C" RETURN id(n) as id',
relationshipQuery:'MATCH (n:Loc)-[r:ROAD]->(m:Loc) RETURN id(n) as source, id(m) as
target, r.cost as weight',
graph:'cypher',writePropertyPrefix:'cypher_'})
YIELD resultCount
RETURN resultCount
```

Implementations

`algo.kShortestPaths`

- Specify start and end node, find the k-shortest path between them.
- If initialized with an non-existing weight-property, it will treat the graph as unweighted.

The All Pairs Shortest Path algorithm

This section describes the All Pairs Shortest Path algorithm in the Neo4j Graph Algorithms library.

The All Pairs Shortest Path (APSP) calculates the shortest (weighted) path between all pairs of nodes. This algorithm has optimisations that make it quicker than calling the Single Source Shortest Path algorithm for every pair of nodes in the graph.

History and explanation

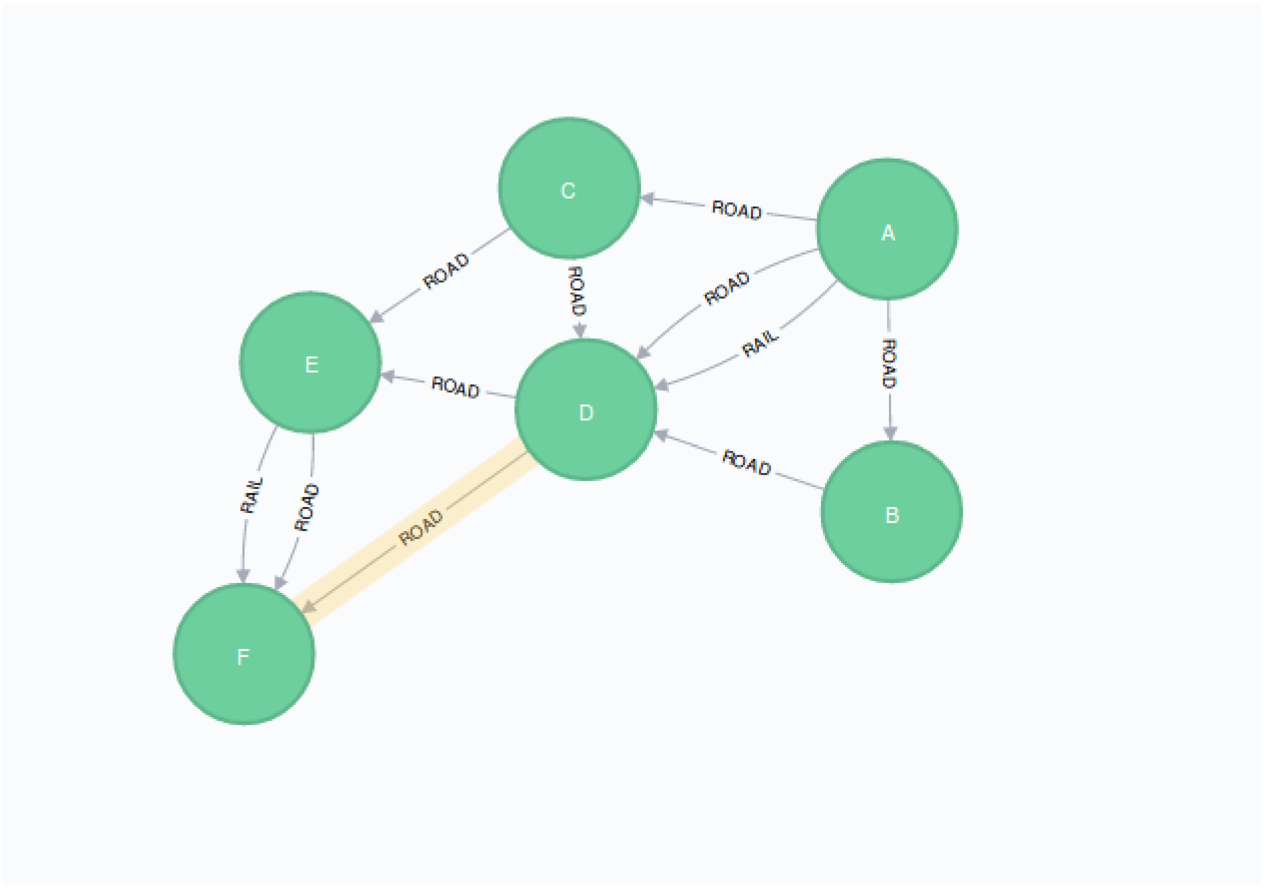
Some pairs of nodes might not be reachable between each other, so no shortest path exists between these pairs. In this scenario, the algorithm will return `Infinity` value as a result between these pairs of nodes.

Plain cypher does not support filtering `Infinity` values, so `algo.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"

All Pairs Shortest Path algorithm sample



The following will create a sample graph:

```
MERGE (a:Loc {name:'A'})
MERGE (b:Loc {name:'B'})
MERGE (c:Loc {name:'C'})
MERGE (d:Loc {name:'D'})
MERGE (e:Loc {name:'E'})
MERGE (f:Loc {name:'F'})

MERGE (a)-[:ROAD {cost:50}]->(b)
MERGE (a)-[:ROAD {cost:50}]->(c)
MERGE (a)-[:ROAD {cost:100}]->(d)
MERGE (b)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:80}]->(e)
MERGE (d)-[:ROAD {cost:30}]->(e)
MERGE (d)-[:ROAD {cost:80}]->(f)
MERGE (e)-[:ROAD {cost:40}]->(f);
```

The following will run the algorithm and stream results:

```
CALL algo.allShortestPaths.stream('cost',{nodeQuery:'Loc',defaultValue:1.0})
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE algo.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
LIMIT 10
```

Table 47. Results

Source	Target	Cost
A	F	100
C	F	90
B	F	90
A	E	80
C	E	70
B	E	80
A	B	50
D	F	50
A	C	50

Source	Target	Cost
A	D	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:

```
CALL algo.allShortestPaths.stream('cost', {
nodeQuery:'MATCH (n:Loc) RETURN id(n) as id',
relationshipQuery:'MATCH (n:Loc)-[r]-(p:Loc) RETURN id(n) as source, id(p) as target,
r.cost as weight',
graph:'cypher', defaultValue:1.0})
```

Huge graph projection

If our projected graph contains more than 2 billion nodes or relationships, we need to use huge graph projection, as the default label and relationship-type projection has a limitation of 2 billion nodes and 2 billion relationships.

Set `graph:'huge'` in the config:

```
CALL
algo.allShortestPaths.stream('cost',{nodeQuery:'Loc',defaultValue:1.0,graph:'huge'})
YIELD sourceNodeId, targetNodeId, distance
RETURN sourceNodeId, targetNodeId, distance LIMIT 10
```

Implementations

`algo.allShortestPaths.stream`

- Find shortest paths between all pairs of nodes.
- Returns a stream of source-target node to distance tuples for each pair of nodes.
- Writeback is not supported.
- If initialized with an non-existing weight-property, it will treat the graph as unweighted.

The Triangle Counting / Clustering Coefficient algorithm

This section describes the Triangle Count or Clustering Coefficient algorithm in the Neo4j Graph Algorithms library.

Triangle counting is a community detection graph algorithm that is used to determine the number of triangles passing through each node in the graph. A triangle is a set of three nodes, where each node has a relationship to all other nodes.

History and explanation

Triangle counting 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 the clustering coefficient.

There are two types of clustering coefficient:

Local clustering coefficient

The local clustering coefficient of a node is the likelihood that its neighbours are also connected. The computation of this score involves triangle counting.

Global clustering coefficient

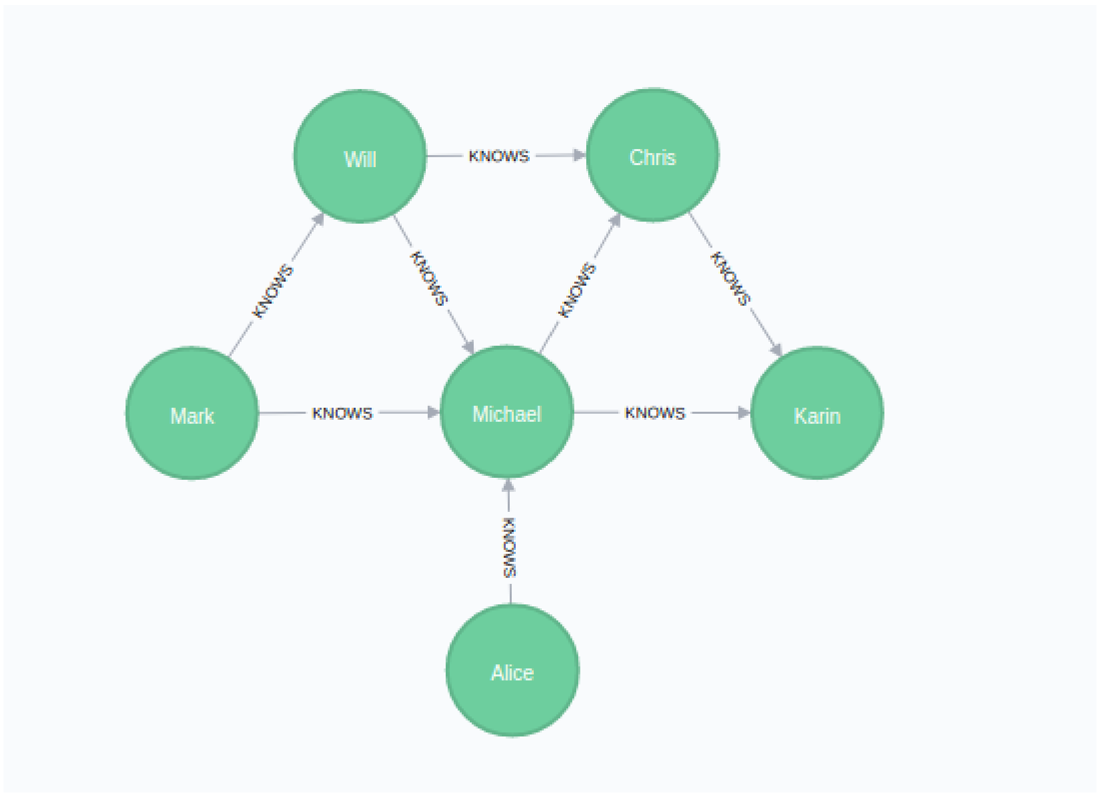
The global clustering coefficient is the normalized sum of those local clustering coefficients.

The transitivity coefficient of a graph is sometimes used, which is three times the number of triangles divided by the number of triples in the graph. For more information, see "Finding, Counting and Listing all Triangles in Large Graphs, An Experimental Study".

Use-cases - when to use the Triangle Counting / Clustering Coefficient algorithm

- 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".
- Clustering coefficient has been used to investigate the community structure of Facebook's social graph, where they found dense neighbourhoods of users in an otherwise sparse global graph. Find this study in "The Anatomy of the Facebook Social Graph".
- Clustering coefficient has been proposed to help explore thematic structure of the web, and detect communities of pages with a common topic based on the reciprocal links between them. For more information, see Curvature of co-links uncovers hidden thematic layers in the World Wide Web.

Triangle Counting / Clustering Coefficient algorithm sample



The following will create a sample graph:

```
MERGE (alice:Person{id:"Alice"})
MERGE (michael:Person{id:"Michael"})
MERGE (karin:Person{id:"Karin"})
MERGE (chris:Person{id:"Chris"})
MERGE (will:Person{id:"Will"})
MERGE (mark:Person{id:"Mark"})

MERGE (michael)-[:KNOWS]->(karin)
MERGE (michael)-[:KNOWS]->(chris)
MERGE (will)-[:KNOWS]->(michael)
MERGE (mark)-[:KNOWS]->(michael)
MERGE (mark)-[:KNOWS]->(will)
MERGE (alice)-[:KNOWS]->(michael)
MERGE (will)-[:KNOWS]->(chris)
MERGE (chris)-[:KNOWS]->(karin);
```

The following will return a stream of triples, with `nodeId` for each triangle:

```
CALL algo.triangle.stream('Person','KNOWS')
YIELD nodeA,nodeB,nodeC

MATCH (a:Person) WHERE id(a) = nodeA
MATCH (b:Person) WHERE id(b) = nodeB
MATCH (c:Person) WHERE id(c) = nodeC

RETURN a.id AS nodeA, b.id AS nodeB, c.id AS nodeC
```

Table 48. Results

nodeA	nodeB	nodeC
Will	Michael	Chris
Will	Mark	Michael
Michael	Karin	Chris

We can see that there are `KNOWS` triangles containing "Will, Michael, and Chris", "Will, Mark, and Michael", and "Michael, Karin, and Chris". This means that everybody in the triangle knows each other.

The following will count the number of triangles that a node is member of, and write it back. It will return the total triangle count and average clustering coefficient of the given graph:

```
CALL algo.triangleCount('Person', 'KNOWS',
  {concurrency:4, write:true,
writeProperty:'triangles',clusteringCoefficientProperty:'coefficient'})
YIELD loadMillis, computeMillis, writeMillis, nodeCount, triangleCount,
averageClusteringCoefficient;
```

The following will count the number of triangles that a node is member of, and return a stream with `nodeId` and `triangleCount`:

```
CALL algo.triangleCount.stream('Person', 'KNOWS', {concurrency:4})
YIELD nodeId, triangles, coefficient

MATCH (p:Person) WHERE id(p) = nodeId

RETURN p.id AS name, triangles, coefficient
ORDER BY coefficient DESC
```

Table 49. Results

Name	Triangles	Coefficient
Karin	1	1
Mark	1	1
Chris	2	0.6666666666666666

Name	Triangles	Coefficient
Will	2	0.6666666666666666
Michael	3	0.3
Alice	0	0

We learn that Michael is part of the most triangles, but it's Karin and Mark who are the best at introducing their friends - all of the people who know them, know each other!

Example usage

In graph theory, a clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. Evidence suggests that in most real-world networks, and in particular social networks, nodes tend to create tightly knit groups characterised by a relatively high density of ties; this likelihood tends to be greater than the average probability of a tie randomly established between two nodes.

We check if this holds true for Yelp's social network of friends:

```
CALL algo.triangleCount('User', 'FRIEND',
  {concurrency:4, write:true,
writeProperty:'triangles',clusteringCoefficientProperty:'coefficient'})
YIELD loadMillis, computeMillis, writeMillis, nodeCount, triangleCount,
averageClusteringCoefficient;
```

Average clustering coefficient is 0.0523, which is really low for a social network. This indicates that groups of friends are not tightly knit together, but rather sparse. We can assume that users are not on Yelp for finding and creating friends, like Facebook for example, but rather something else, like finding good restaurant recommendations.

Local triangle count and clustering coefficient of nodes can be used as features in finding influencers in social networks.

Syntax

The following will return a stream of triples with `nodeId` *for each triangle:*

```
CALL algo.triangle.stream(label:String, relationship:String, {concurrency:4})
YIELD nodeA, nodeB, nodeC
```

Table 50. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all nodes

Name	Type	Default	Optional	Description
concurrency	int	available CPUs	yes	The number of concurrent threads

Table 51. Results

Name	Type	Description
nodeA	int	The ID of node in the given triangle
nodeB	int	The ID of node in the given triangle
nodeC	int	The ID of node in the given triangle

The following will count the number of triangles that a node is a member of, and return a stream with `nodeId` *and* `triangleCount`*:*

```
CALL algo.triangleCount.stream(label:String, relationship:String, {concurrency:4})
YIELD nodeId, triangles
```

Table 52. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
concurrency	int	available CPUs	yes	The number of concurrent threads

Table 53. Results

Name	Type	Description
nodeId	int	The ID of node
triangles	int	The number of triangles a node is member of

The following will count the number of triangles that a node is a member of, and write it back. It will return the total triangle count and average clustering coefficient of the given graph:

```
CALL algo.triangleCount(label:String, relationship:String,
    {concurrency:4, write:true, writeProperty:'triangles',
clusteringCoefficientProperty:'coefficient'})
YIELD loadMillis, computeMillis, writeMillis, nodeCount, triangleCount,
averageClusteringCoefficient
```

Table 54. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes

Name	Type	Default	Optional	Description
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
concurrency	int	available CPUs	yes	The number of concurrent threads
write	boolean	true	yes	Specifies if the result should be written back as a node property
writeProperty	string	'triangles'	yes	The property name the number of triangles a node is member of is written to
clusteringCoefficientProperty	string	'coefficient'	yes	The property name clustering coefficient of the node is written to

Table 55. Results

Name	Type	Description
nodeCount	int	The number of nodes considered
loadMillis	int	Milliseconds for loading data
evalMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back
triangleCount	int	The number of triangles in the given graph
averageClusteringCoefficient	float	The average clustering coefficient of the given graph

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` *in the config:*

```
CALL algo.triangleCount(
  'MATCH (p:Person) RETURN id(p) as id',
  'MATCH (p1:Person)-[:KNOWS]->(p2:Person) RETURN id(p1) as source,id(p2) as target',
  {concurrency:4, write:true, writeProperty:'triangle',graph:'cypher',
clusteringCoefficientProperty:'coefficient'})
YIELD loadMillis, computeMillis, writeMillis, nodeCount, triangleCount,
averageClusteringCoefficient
```

Versions

We support the following versions of the triangle count algorithms:

✓ undirected, unweighted

The Label Propagation algorithm

***This section describes the Label Propagation algorithm in the Neo4j Graph** Algorithms library.*

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.

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.

History and explanation

LPA is a relatively new algorithm, and was only proposed by Raghavan et al in 2007, in "Near linear time algorithm to detect community structures in large-scale networks". It 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 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 uniformly and randomly.
- LPA reaches convergence when each node has the majority label of its neighbours.

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 label at convergence are said to belong to the same community.

Use-cases - when to use the Label Propagation algorithm

- Label propagation has been used to assign polarity of tweets, as a part of semantic analysis which uses seed labels from a classifier trained to detect positive and negative emoticons in combination with Twitter follower graph. For more information, see Twitter polarity classification with label propagation over lexical links and the follower graph
- Label propagation has been used to estimate potentially dangerous combinations of drugs to co-prescribe to a patient, based on the chemical similarity and side effect profiles. The study can be

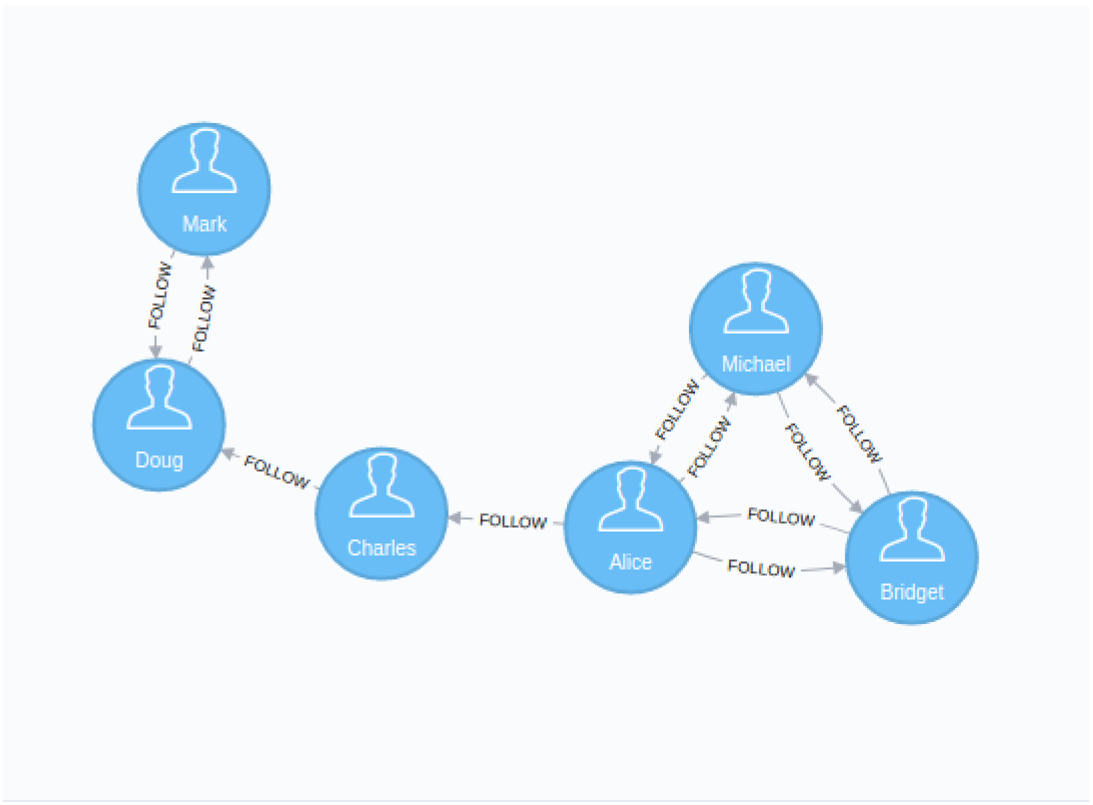
found in Label Propagation Prediction of Drug-Drug Interactions Based on Clinical Side Effects

- Label propagation has been used to infer features of utterances in a dialogue, for a machine learning model to track user intention with the help of Wikidata knowledge graph of concepts and their relations. For more information, see "Feature Inference Based on Label Propagation on Wikidata Graph for DST"

Constraints - when not to use the Label Propagation algorithm

In contrast with other algorithms, label propagation can result in different community structures when run multiple times on the same graph. The range of solutions can be narrowed if some nodes are given preliminary labels, while others are unlabelled. Unlabelled nodes will be more likely to adapt the preliminary labels.

Label Propagation algorithm sample



The following will create a sample graph:

```
MERGE (nAlice:User {id:'Alice'}) SET nAlice.seed_label=52
MERGE (nBridget:User {id:'Bridget'}) SET nBridget.seed_label=21
MERGE (nCharles:User {id:'Charles'}) SET nCharles.seed_label=43
MERGE (nDoug:User {id:'Doug'}) SET nDoug.seed_label=21
MERGE (nMark:User {id:'Mark'}) SET nMark.seed_label=19
MERGE (nMichael:User {id:'Michael'}) SET nMichael.seed_label=52

MERGE (nAlice)-[:FOLLOW]->(nBridget)
MERGE (nAlice)-[:FOLLOW]->(nCharles)
MERGE (nMark)-[:FOLLOW]->(nDoug)
MERGE (nBridget)-[:FOLLOW]->(nMichael)
MERGE (nDoug)-[:FOLLOW]->(nMark)
MERGE (nMichael)-[:FOLLOW]->(nAlice)
MERGE (nAlice)-[:FOLLOW]->(nMichael)
MERGE (nBridget)-[:FOLLOW]->(nAlice)
MERGE (nMichael)-[:FOLLOW]->(nBridget)
MERGE (nCharles)-[:FOLLOW]->(nDoug);
```

The following will run the algorithm and stream results:

```
CALL algo.labelPropagation.stream("User", "FOLLOW",
  {direction: "OUTGOING", iterations: 10})
```

The following will run the algorithm and write back results:

```
CALL algo.labelPropagation('User', 'FOLLOW','OUTGOING',
  {iterations:10,partitionProperty:'partition', write:true})
YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, write,
partitionProperty;
```

Table 56. Results

Name	**Partition**
Alice	5
Charles	4
Bridget	5
Michael	5
Doug	4
Mark	4

Our algorithm found two communities, with 3 members each.

It appears that Michael, Bridget, and Alice belong together, as do Doug and Mark. Only Charles doesn’t strongly fit into either side, but ends up with Doug and Mark.

Using seed labels

At the beginning of the algorithm, every node is initialized with unique label (called as identifier) and the labels propagate through the network.

It is possible to define preliminary labels (identifiers) of nodes using the `partition` parameter. We need to save a preliminary set of labels that we would like to run the Label Propagation algorithm with as a property of nodes (must be a number). In our example graph we saved them as the property `seed_label`.

The algorithm first checks if there is a seed label assigned to the node, and loads it if there is one. If there isn't one, it assigns the node new unique label (node ID is used). Using this preliminary set of labels (identifiers), it then sequentially updates each node's label to a new one, which is the most frequent label among its neighbors at every label propagation step (iteration).

The following will run the algorithm with pre-defined labels:

```
CALL algo.labelPropagation('User', 'FOLLOW','OUTGOING',
  {iterations:10,partitionProperty:'seed_label', write:true})
YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, write,
partitionProperty;
```

Syntax

The following will run the algorithm and write back results:

```
CALL algo.labelPropagation(label:String, relationship:String, direction:String,
{iterations:1,
    weightProperty:'weight', partitionProperty:'partition', write:true,
concurrency:4})
YIELD nodes, iterations, didConverge, loadMillis, computeMillis, writeMillis, write,
weightProperty, partitionProperty
```

Table 57. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
direction	string	'OUTGOING'	yes	The relationship-direction to use in the algorithm
concurrency	int	available CPUs	yes	The number of concurrent threads
iterations	int	1	yes	The maximum number of iterations to run
weightProperty	string	'weight'	yes	The property name of node and/or relationship that contain weight. Must be numeric.

Name	Type	Default	Optional	Description
partitionProperty	string	'partition'	yes	The property name written back to the partition of the graph in which the node reside. Can be used to define initial set of labels (must be a number)
write	boolean	true	yes	Specifies if the result should be written back as a node property
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 58. Results

Name	Type	Description
nodes	int	The number of nodes considered
iterations	int	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
loadMillis	int	Milliseconds for loading data
computeMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back
weightProperty	string	The property name that contains weight
partitionProperty	string	The property name written back to
write	boolean	Specifies if the result was written back as a node property

The following will run the algorithm and stream back results:

```
CALL algo.labelPropagation.stream(label:String, relationship:String, {iterations:1,
    weightProperty:'weight', partitionProperty:'partition', concurrency:4,
direction:'OUTGOING'})
YIELD nodeId, label
```

Table 59. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
direction	string	'OUTGOING'	yes	The relationship-direction to use in the algorithm

Name	Type	Default	Optional	Description
concurrency	int	available CPUs	yes	The number of concurrent threads
iterations	int	1	yes	The maximum number of iterations to run
weightProperty	string	'weight'	yes	The property name of node and/or relationship that contain weight. Must be numeric.
partitionProperty	string	'partition'	yes	The property name written back to the partition of the graph in which the node reside. Can be used to define initial set of labels (must be a number)
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 60. Results

Name	Type	Description
nodeId	int	Node ID
label	int	Community ID

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` in the config:

```
CALL algo.labelPropagation(
  'MATCH (p:User) RETURN id(p) as id, p.weight as weight, id(p) as value',
  'MATCH (p1:User)-[f:FRIEND]->(p2:User)
   RETURN id(p1) as source, id(p2) as target, f.weight as weight',
  "OUT",
  {graph:'cypher',write:true});
```

Versions

We support the following versions of the Label Propagation algorithm:

- ✓ directed, unweighted:
  - direction: 'INCOMING' or 'OUTGOING', weightProperty: null
- ✓ directed, weighted
  - direction: 'INCOMING' or 'OUTGOING', weightProperty : 'weight'
- ✓ undirected, unweighted

    - direction: 'BOTH', weightProperty: null
- ✓ undirected, weighted
    - direction: 'BOTH', weightProperty: 'weight'

The Louvain algorithm

***This section describes the Louvain algorithm in the Neo4j Graph** Algorithms library.*

The Louvain method of community detection is an algorithm for detecting communities in networks. It maximizes a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities by 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 one of the fastest modularity-based algorithms, and works well with large graphs. It also reveals a hierarchy of communities at different scales, which can be useful for understanding the global functioning of a network.

History and explanation

The "Louvain algorithm" was proposed in 2008 by authors from the University of Louvain.

The method consists of repeated application of two steps. The first step is a "greedy" assignment of nodes to communities, favoring local optimizations of modularity. The second step is the definition of a new coarse-grained network, based on the communities found in the first step. These two steps are repeated until no further modularity-increasing reassignments of communities are possible.

The algorithm is initialized with each node in its own community.

In the first stage we iterate through each of the nodes in the network. We take each node, remove it from its current community and replace it in the community of ones of its neighbors. We compute the modularity change for each of the node’s neighbors. If none of these modularity changes are positive, the node stays in its current community. If some of the modularity changes are positive, the node moves into the community where the modularity change is most positive. Ties are resolved arbitrarily. We repeat this process for each node until one pass through all nodes yields no community assignment changes.

The second stage in the Louvain method uses the communities that were discovered in the community reassignment stage, to define a new coarse-grained network. In this network, the newly discovered communities are the nodes. The relationship weight between the nodes representing two communities is the sum of the relationship weights between the lower-level nodes of each community.

The rest of the Louvain method consists of the repeated application of stages 1 and 2. By applying stage 1 (the community reassignment phase) to the coarse-grained graph, we find a second tier of communities of communities of nodes. Then, in the next application of stage 2, we define a new coarse-grained graph at this higher-level of the hierarchy. We keep going like this until an application of stage 1 yields no reassignments. At that point, repeated application of stages 1 and 2 will not yield any more modularity-optimizing changes, so the process is complete.

Use-cases - when to use the Louvain algorithm

- The Louvain method has been proposed to provide recommendations for Reddit users to find similar subreddits, based on the general user behavior. Find more details, see "Subreddit Recommendations within Reddit Communities".
- The Louvain method has been used to extract topics from online social platforms, such as Twitter and Youtube, based on the co-occurence graph of terms in documents, as a part of Topic Modeling process. This process is described in "Topic Modeling based on Louvain method in Online Social Networks".
- The Louvain method has been used to investigate the human brain, and find hierarchical community structures within the brain's functional network. The study mentioned is "Hierarchical Modularity in Human Brain Functional Networks".

Constraints - when not to use the Louvain algorithm

Although the Louvain method, and modularity optimization algorithms more generally, have found wide application across many domains, some problems with these algorithms have been identified:

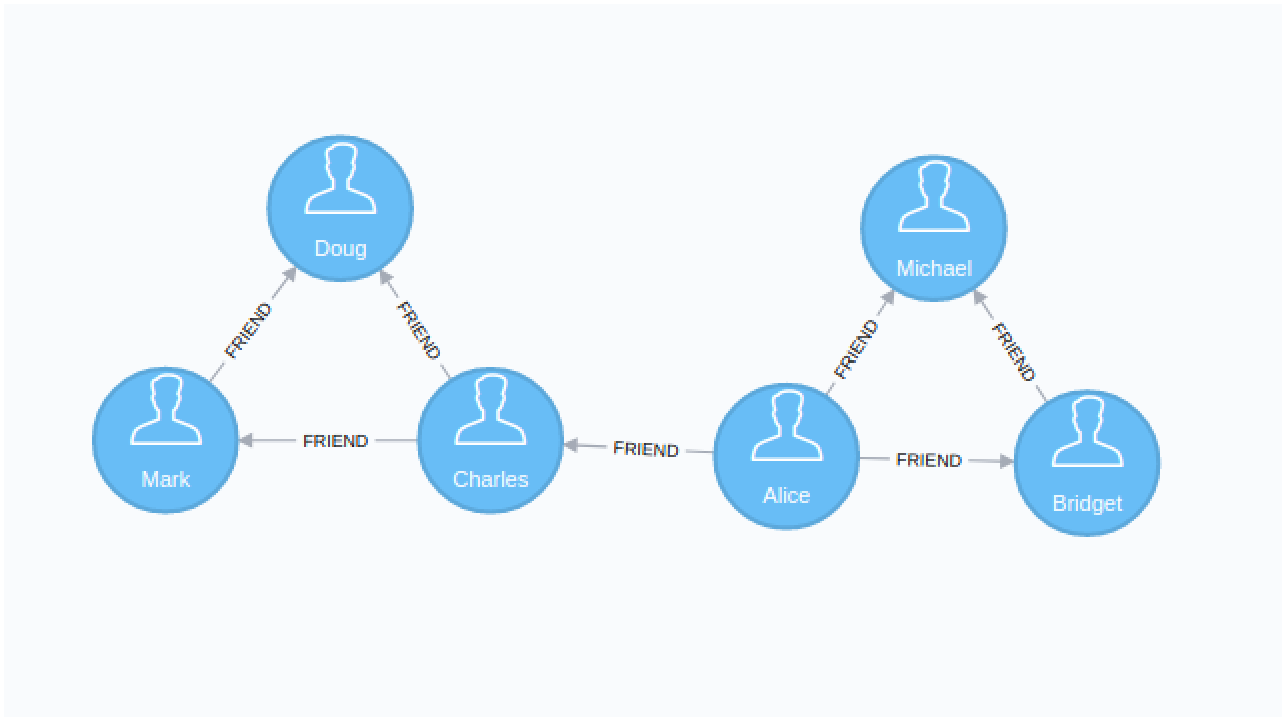
The *resolution* limit

For larger networks, the Louvain method doesn't stop with the "intuitive" communities. Instead, there's a second pass through the community modification and coarse-graining stages, in which several of the intuitive communities are merged together. This is a general problem with modularity optimization algorithms; they have trouble detecting small communities in large networks. It's a virtue of the Louvain method that something close to the intuitive community structure is available as an intermediate step in the process.

The *degeneracy* problem

There is typically an exponentially large (in network size) number of community assignments with modularities close to the maximum. This can be a severe problem because, in the presence of a large number of high modularity solutions, it's hard to find the global maximum, and difficult to determine if the global maximum is truly more scientifically important than local maxima that achieve similar modularity. Research undertaken at Universite Catholique de Louvain showed that the different locally optimal community assignments can have quite different structural properties. For more information, see "The performance of modularity maximization in practical contexts"

Louvain algorithm sample



The following will create a sample graph:

```
MERGE (nAlice:User {id:'Alice'})
MERGE (nBridget:User {id:'Bridget'})
MERGE (nCharles:User {id:'Charles'})
MERGE (nDoug:User {id:'Doug'})
MERGE (nMark:User {id:'Mark'})
MERGE (nMichael:User {id:'Michael'})

MERGE (nAlice)-[:FRIEND]->(nBridget)
MERGE (nAlice)-[:FRIEND]->(nCharles)
MERGE (nMark)-[:FRIEND]->(nDoug)
MERGE (nBridget)-[:FRIEND]->(nMichael)
MERGE (nCharles)-[:FRIEND]->(nMark)
MERGE (nAlice)-[:FRIEND]->(nMichael)
MERGE (nCharles)-[:FRIEND]->(nDoug);
```

The following will run the algorithm and stream results:

```
CALL algo.louvain.stream('User', 'FRIEND', {})
YIELD nodeId, community

MATCH (user:User) WHERE id(user) = nodeId

RETURN user.id AS user, community
ORDER BY community;
```

The following will run the algorithm and write back results:

```
CALL algo.louvain('User', 'FRIEND',
  {write:true, writeProperty:'community'})
YIELD nodes, communityCount, iterations, loadMillis, computeMillis, writeMillis;
```

Table 61. Results

Name	Community
Alice	5
Bridget	5
Michael	5
Charles	4
Doug	4
Mark	4

Our algorithm found two communities with 3 members each.

Mark, Doug, and Charles are all friends with each other, as are Bridget, Alice, and Michael. Charles is the only one who has friends in both communities, but he has more in community 4 so he fits better in that one.

Syntax

The following will run the algorithm and write back results:

```
CALL algo.louvain(label:String, relationship:String,
    {weightProperty:'weight', defaultValue:1.0, write: true,
writeProperty:'community', concurrency:4})
YIELD nodes, communityCount, iterations, loadMillis, computeMillis, writeMillis
```

Table 62. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
write	boolean	true	yes	Specifies if the result should be written back as a node property
writeProperty	string	'community'	yes	The property name written back to the ID of the community that particular node belongs to

Name	Type	Default	Optional	Description
defaultValue	float	null	yes	The default value of the weight in case it is missing or invalid
concurrency	int	available CPUs	yes	The number of concurrent threads
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 63. Results

Name	Type	Description
nodes	int	The number of nodes considered
communityCount	int	The number of communities found
iterations	int	The number of iterations run
loadMillis	int	Milliseconds for loading data
computeMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the algorithm and stream results:

```
CALL algo.louvain.stream(label:String, relationship:String,
    {weightProperty:'propertyName', defaultValue:1.0, concurrency:4})
YIELD nodeId, community - yields a community to each node id
```

Table 64. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
defaultValue	float	1.0	yes	The default value of the weight if it is missing or invalid
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 65. Results

Name	Type	Description
nodeId	int	Node ID
community	int	Community ID

Huge graph projection

If our projected graph contains more than 2 billion nodes or relationships, we need to use huge graph projection, as the default label and relationship-type projection has a limitation of 2 billion nodes and 2 billion relationships.

Set `graph:'huge'` in the config:

```
CALL algo.louvain('User', 'FRIEND',{graph:'huge'})
YIELD nodes, communityCount, iterations, loadMillis, computeMillis, writeMillis;
```

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` in the config:

```
CALL algo.louvain(
  'MATCH (p:User) RETURN id(p) as id',
  'MATCH (p1:User)-[f:FRIEND]-(p2:User)
   RETURN id(p1) as source, id(p2) as target, f.weight as weight',
  {graph:'cypher',write:true});
```

Versions

- ✓ undirected, unweighted
    - weightProperty: null
- ✓ undirected, weighted
    - weightProperty : 'weight'

The Connected Components algorithm

***This section describes the Connected Components algorithm in the Neo4j** Graph Algorithms library.*

The Connected Components, or `Union Find`, algorithm finds sets of connected nodes in an undirected graph where each node is reachable from any other node in the same set. It differs from the Strongly Connected Components algorithm (SCC) because it only needs a path to exist between pairs of nodes in one direction, whereas SCC needs a path to exist in both directions. As with SCC, `UnionFind` is often used early in an analysis to understand a graph’s structure.

History and explanation

The algorithm was first described by Bernard A. Galler and Michael J. Fischer in 1964. The components in a graph are computed using either the breadth-first search or depth-first search algorithms.

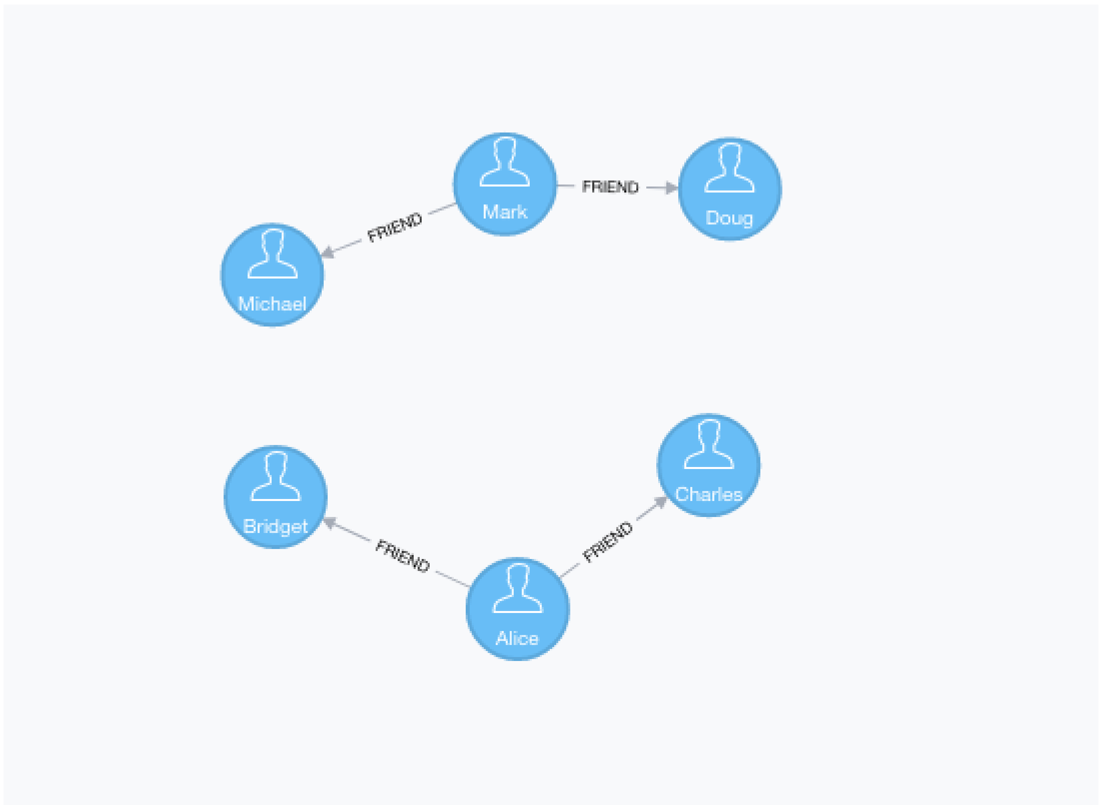
Use-cases - when to use the Connected Components algorithm

- Testing whether a graph is connected is an essential pre-processing step for every graph algorithm. Such tests can be performed so quickly, and easily, that you should always verify that your input graph is connected, even when you know it has to be. Subtle, difficult-to-detect, bugs often result when your algorithm is run only on one component of a disconnected graph.
- `Union Find` can be used to keep track of clusters of database records, as part of the de-duplication process - an important task in master data management applications. Read more in "An efficient domain-independent algorithm for detecting approximately duplicate database records".
- Weakly connected components (WCC) can be used to analyse citation networks. 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. Read more in "Characterizing and Mining Citation Graph of Computer Science Literature".

Connected Components algorithm sample

If we recall that an undirected graph is connected if, for every pair of vertices there is a path in the graph between those vertices. A connected component of an undirected graph is a maximal connected subgraph of the graph. That means that the direction of the relationships in our graph are ignored - we treat the graph as undirected.

We have two implementations of the Connected Components algorithm. The first treats the graph as unweighted and the second treats it as weighted, where you can define the threshold of the weight above which relationships are included.



The following will create a sample graph:

```
MERGE (nAlice:User {id:'Alice'})
MERGE (nBridget:User {id:'Bridget'})
MERGE (nCharles:User {id:'Charles'})
MERGE (nDoug:User {id:'Doug'})
MERGE (nMark:User {id:'Mark'})
MERGE (nMichael:User {id:'Michael'})

MERGE (nAlice)-[:FRIEND {weight:0.5}]->(nBridget)
MERGE (nAlice)-[:FRIEND {weight:4}]->(nCharles)
MERGE (nMark)-[:FRIEND {weight:1}]->(nDoug)
MERGE (nMark)-[:FRIEND {weight:2}]->(nMichael);
```

Unweighted version

The following will run the algorithm and stream results:

```
CALL algo.unionFind.stream('User', 'FRIEND', {})
YIELD nodeId,setId

MATCH (u:User) WHERE id(u) = nodeId

RETURN u.id AS user, setId
```

The following will run the algorithm and write back results:

```
CALL algo.unionFind('User', 'FRIEND', {write:true, partitionProperty:"partition"})
YIELD nodes, setCount, loadMillis, computeMillis, writeMillis;
```

Table 66. Results

Name	Partition
Alice	0
Charles	0
Bridget	0
Michael	4
Doug	4
Mark	4

We have two distinct group of users, that have no link between them.

The first group contains Alice, Charles, and Bridget, while the second group contains Michael, Doug, and Mark.

The following will check the number and size of partitions, using Cypher:

```
MATCH (u:User)
RETURN u.partition as partition,count(*) as size_of_partition
ORDER by size_of_partition DESC
LIMIT 20;
```

Weighted version

If you define the property that holds the weight (`weightProperty`) and the threshold, it means the nodes are only connected, if the threshold on the weight of the relationship is high enough, otherwise the relationship is thrown away.

The following will run the algorithm and stream results:

```
CALL algo.unionFind.stream('User', 'FRIEND', {weightProperty:'weight',
defaultValue:0.0, threshold:1.0, concurrency: 1})
YIELD nodeId,setId

MATCH (u:User) WHERE id(u) = nodeId

RETURN u.id AS user, setId
```

The following will run the algorithm and write back results:

```
CALL algo.unionFind('User', 'FRIEND', {write:true,
partitionProperty:"partition",weightProperty:'weight', defaultValue:0.0,
threshold:1.0, concurrency: 1})
YIELD nodes, setCount, loadMillis, computeMillis, writeMillis;
```

Table 67. Results

Name	Partition
Alice	0
Charles	0
Bridget	1
Michael	4
Doug	4
Mark	4

In this case we can see that, because the weight of the relationship between Bridget and Alice is only 0.5, the relationship is ignored by the algorithm, and Bridget ends up in her own component.

Example usage

As mentioned above, connected components are an essential step in preprocessing your data. One reason is that most centralities suffer from disconnected components, or you just want to find disconnected groups of nodes. Int his example, Yelp's social network will be used to demonstrate how to proceed when dealing with real world data. A typical social network consists of one big component and a number of small disconnected components.

The following will get the count of connected components:

```
CALL algo.unionFind.stream('User', 'FRIEND', {})
YIELD nodeId,setId
RETURN count(distinct setId) as count_of_components;
```

We get back the count of disconnected components being 18512 if we do not count users without friends. Let's now check the size of top 20 components to get a better picture:

The following will get the size of top 20 components:

```
CALL algo.unionFind.stream('User', 'FRIEND', {})
YIELD nodeId,setId
RETURN setId,count(*) as size_of_component
ORDER BY size_of_component
LIMIT 20;
```

The biggest component has 8938630 out of total 8981389 (99,5%). It is quite high, but not shocking, as we have a friendship social network where we can expect small world effect and 6 degree of separation rule, where you can get to any person in a social network, just depends how long is the path.

We can now move on to next step of analysis and run centralities on only the biggest components, so that our results will be more accurate. We will write back the results to the node, and use centralities with Cypher loading, or set a new label for the biggest component.

Syntax

The following will run the algorithm and write back results:

```
CALL algo.unionFind(label:String, relationship:String, {threshold:0.42,
    defaultValue:1.0, write: true, partitionProperty:'partition',
weightProperty:'weight', graph:'heavy', concurrency:4})
YIELD nodes, setCount, loadMillis, computeMillis, writeMillis
```

Table 68. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
write	boolean	true	yes	Specifies if the result should be written back as a node property
partitionProperty	string	'partition'	yes	The property name written back the ID of the partition particular node belongs to
threshold	float	null	yes	The value of the weight above which the relationship is not thrown away
defaultValue	float	null	yes	The default value of the weight in case it is missing or invalid
concurrency	int	available CPUs	yes	The number of concurrent threads
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 69. Results

Name	Type	Description
nodes	int	The number of nodes considered
setCount	int	The number of partitions found

Name	Type	Description
loadMillis	int	Milliseconds for loading data
computeMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the algorithm and stream results:

```
CALL algo.unionFind.stream(label:String, relationship:String,
    {weightProperty:'weight', threshold:0.42, defaultValue:1.0, concurrency:4})
YIELD nodeId, setId - yields a setId to each node id
```

Table 70. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
concurrency	int	available CPUs	yes	The number of concurrent threads
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
threshold	float	null	yes	The value of the weight above which the relationship is not thrown away
defaultValue	float	null	yes	The default value of the weight in case it is missing or invalid
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 71. Results

Name	Type	Description
nodeId	int	Node ID
setId	int	Partition ID

Huge graph projection

If our projected graph contains more than 2 billion nodes or relationships, we need to use huge graph projection, as the default label and relationship-type projection has a limitation of 2 billion nodes and 2 billion relationships.

Set `graph:'huge'` in the config:

```
CALL algo.unionFind('User', 'FRIEND', {graph:'huge'})
YIELD nodes, setCount, loadMillis, computeMillis, writeMillis;
```

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` in the config:

```
CALL algo.unionFind(
  'MATCH (p:User) RETURN id(p) as id',
  'MATCH (p1:User)-[f:FRIEND]->(p2:User)
   RETURN id(p1) as source, id(p2) as target, f.weight as weight',
  {graph:'cypher',write:true}
);
```

Implementations

`algo.unionFind`

- If a threshold configuration parameter is supplied, only relationships with a property value higher than the threshold are merged.

  `algo.unionFind.queue`

- Parallel `Union Find`, using `ExecutorService` only.
- Algorithm based on the idea that `DisjointSetStruct` can be built using just a partition of the nodes, which are then merged pairwise.
- The implementation is based on a queue which acts as a buffer for each computed `DisjointSetStruct`. As long as there are more elements on the queue, the algorithm takes two, merges them, and adds its result to the queue until only 1 element remains.

`algo.unionFind.forkJoinMerge`

- Like in `algo.unionFind.queue`, the resulting `DisjointSetStruct` of each node-partition is merged by the `ForkJoin` pool, while the calculation of the `DisjointSetStruct` is done by the `ExecutorService`.

`algo.unionFind.forkJoin`

- Calculation and merge using `forkJoinPool`

`algo.unionFind.mscoloring`

- Coloring based parallel algorithm

The Strongly Connected Components algorithm

This section describes the Strongly Connected Components algorithm in the Neo4j Graph Algorithms library.

The Strongly Connected Components (SCC) algorithm finds sets of connected nodes in a directed graph where each node is reachable in both directions from any other node in the same set. It is often used early in a graph analysis process to help us get an idea of how our graph is structured.

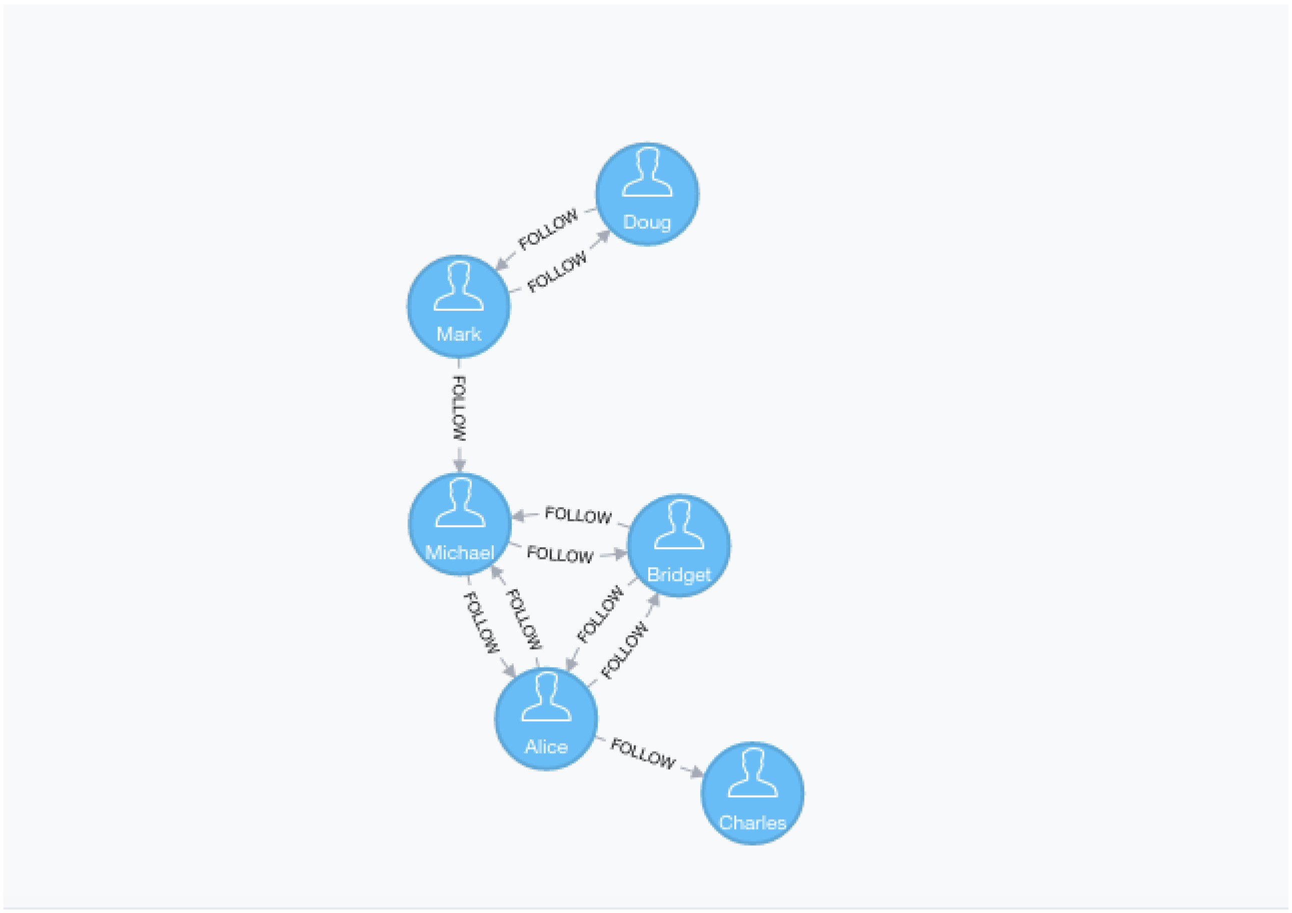
History and explanation

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.

Strongly Connected Components algorithm sample



A directed graph is strongly connected if there is a path between all pairs of vertices. This algorithm treats the graph as directed, which means that the direction of the relationship is important. A strongly connected component only exists if there are relationships between nodes in both direction.

The following will create a sample graph:

```
MERGE (nAlice:User {id:'Alice'})
MERGE (nBridget:User {id:'Bridget'})
MERGE (nCharles:User {id:'Charles'})
MERGE (nDoug:User {id:'Doug'})
MERGE (nMark:User {id:'Mark'})
MERGE (nMichael:User {id:'Michael'})

MERGE (nAlice)-[:FOLLOW]->(nBridget)
MERGE (nAlice)-[:FOLLOW]->(nCharles)
MERGE (nMark)-[:FOLLOW]->(nDoug)
MERGE (nMark)-[:FOLLOW]->(nMichael)
MERGE (nBridget)-[:FOLLOW]->(nMichael)
MERGE (nDoug)-[:FOLLOW]->(nMark)
MERGE (nMichael)-[:FOLLOW]->(nAlice)
MERGE (nAlice)-[:FOLLOW]->(nMichael)
MERGE (nBridget)-[:FOLLOW]->(nAlice)
MERGE (nMichael)-[:FOLLOW]->(nBridget);
```

The following will run the algorithm and write back results:

```
CALL algo.scc('User','FOLLOW', {write:true,partitionProperty:'partition'})
YIELD loadMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize;
```

Table 72. Results

Name	Partition
Alice	1
Bridget	1
Michael	1
Charles	0
Doug	2
Mark	2

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:

```
MATCH (u:User)
RETURN u.partition as partition,count(*) as size_of_partition
ORDER by size_of_partition DESC
LIMIT 1
```

Syntax

The following will run the algorithm and write back results:

```
CALL algo.scc(label:String, relationship:String,
    {write:true,partitionProperty:'partition',concurrency:4, graph:'heavy'})
YIELD loadMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize
```

Table 73. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
write	boolean	true	yes	Specifies if the result should be written back as a node property

Name	Type	Default	Optional	Description
partitionProperty	string	'partition'	yes	The property name written back to
concurrency	int	available CPUs	yes	The number of concurrent threads
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 74. Results

Name	Type	Description
setCount	int	The number of partitions found
maxSetSize	int	The number of members in biggest partition
minSetSize	int	The number of members in smallest partition
loadMillis	int	Milliseconds for loading data
computeMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the algorithm and stream results:

```
CALL algo.scc.stream(label:String, relationship:String, {concurrency:4})
YIELD nodeId, partition
```

Table 75. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes
relationship	string	null	yes	The relationship-type to load from the graph. If null, load all relationships
concurrency	int	available CPUs	yes	The number of concurrent threads
graph	string	'heavy'	yes	Use 'heavy' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node-statement and relationship-statement

Table 76. Results

Name	Type	Description
nodeId	int	Node ID

Name	Type	Description
partition	int	Partition ID

Huge graph projection

If our projected graph contains more than 2 billion nodes or relationships, we need to use huge graph projection, as the default label and relationship-type projection has a limitation of 2 billion nodes and 2 billion relationships.

Set `graph:'huge'` in the config:

```
CALL algo.scc('User','FOLLOW', {graph:'huge'})
YIELD loadMillis, computeMillis, writeMillis, setCount;
```

Cypher projection

If label and relationship-type are not selective enough to describe your subgraph to run the algorithm on, you can use Cypher statements to load or project subsets of your graph. This can also be used to run algorithms on a virtual graph.

Set `graph:'cypher'` in the config:

```
CALL algo.scc(
  'MATCH (u:User) RETURN id(u) as id',
  'MATCH (u1:User)-[:FOLLOW]->(u2:User) RETURN id(u1) as source,id(u2) as target',
  {write:true,graph:'cypher'})
YIELD loadMillis, computeMillis, writeMillis;
```

Implementations

`algo.scc`

- **Iterative** adaptation (same as `algo.scc.iterative`).

`algo.scc.recursive.tarjan`

- Original **recursive** tarjan implementation.

`algo.scc.recursive.tunedTarjan`

- Also a **recursive** tarjan implementation.

`algo.scc.iterative`

- **Iterative** adaption of tarjan algorithm.

`algo.scc.multistep`

- Parallel SCC algorithm.