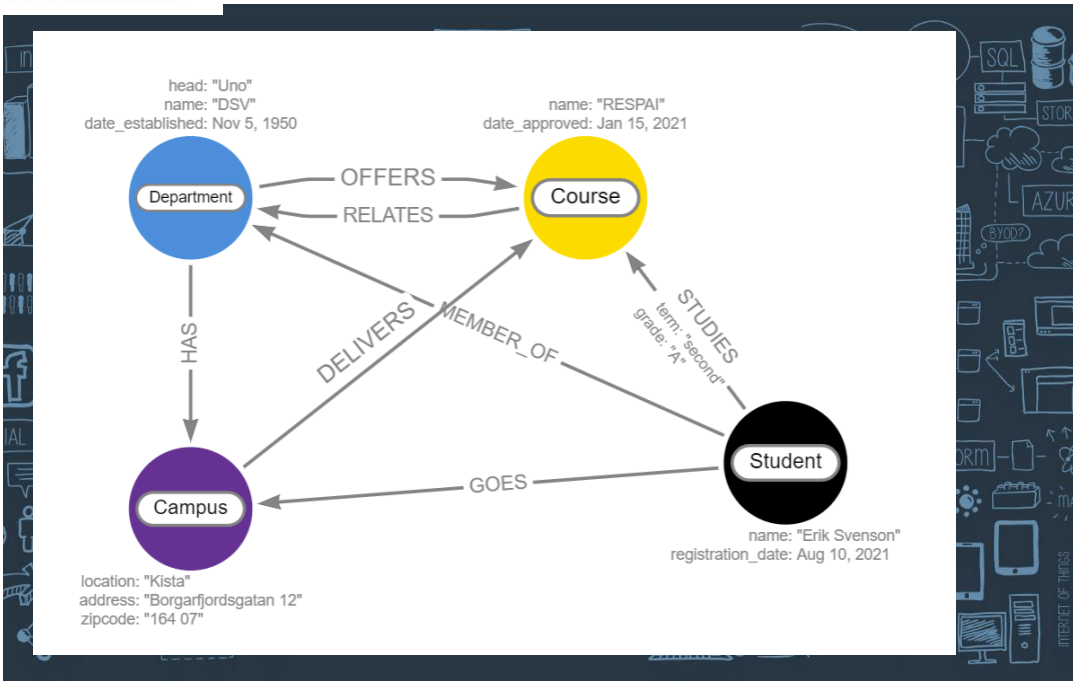
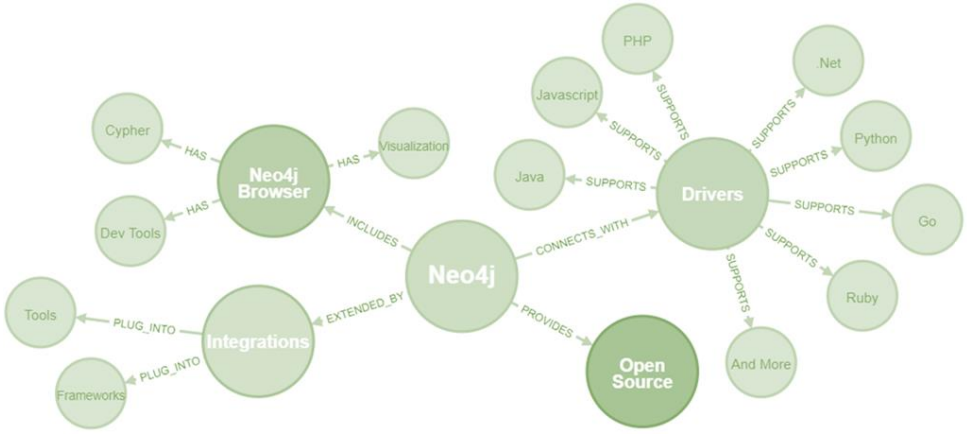
Stockholms universitet



Preparing for Assignment 1, Part 2 – Graph Algorithms

RESPAI

Scalable and Responsible AI in Organizations | VT 2024,
Compiled by Workneh Yilma Ayele



Compiled by - WYA

Table of Contents

Objectives of this tutorial

- Familiarize yourself with graph database
- Familiarize yourself with graph database basic concepts
- Learn about Neo4j Cypher to create and manage graph database
- Learn about Neo4j Bloom for more enhanced visualization of graph data analytics

1. Graphs and graph algorithms

Graph data analytics, also known as network analysis, deals with the study of graph data and the relationships it contains. We observe that natural and social networks are increasingly evolving in our daily lives. Examples of networks are – friendship networks on social media, research paper citation networks, road networks, air traffic networks, computing networks, networks of terrorist organizations, etc.

1.1. Graph types

There are different types of graphs having different forms of shapes and forms, such as directed vs. undirected, unweighted vs. weighted, and monopartite vs. multipartite.

Directed vs. undirected

When the direction of relationships between nodes is essential, we use a directed graph. For example, in social media networks, FOLLOWS, LIKES, DISLIKES, etc., are examples of relationships directed from one side, outgoing link, to the other side with the incoming link.

Example –

(Don)--[FOLLOWS]→(Mike),
(Don)--[Likes]→(Erik)
(Maradona)--[DISLIKES]→(Dog)

In undirected graphs, a single relationship represents both incoming and outgoing relationships in both directions.

Example –

(Svenson) ←[IS_FRIEND]→(Ali)
(Ruth) ←[IS_FRIEND]→(Yilma)

Unweighted vs. weighted

Weighted graphs contain weight or cost, or value in their relationships. Weighted graphs represent, for example, transportation networks where nodes represent source/destinations, and the distance between these nodes contains properties that have quantitative values such as cost for traveling from a source to a destination and the distance between nodes in the network. On the other hand, unweighted graphs are graphs without weight properties.

Monopartite vs. multipartite

Graph data is categorized into two depending on the set of nodes, monopartite and multipartite. A monopartite graph represents graphs having only a single set of nodes. For example, a monopartite graph can contain relationships of persons in social media without including other node types. A multipartite graph contains several more than one type of node.

1.2. Using graph algorithms

1.2.1. Graph Data Science Library (GDC)

The GDC is implemented to gain new insights from graph data in Neo4j. The Neo4j GDC is an analytics engine enabling engineers to unveil the possibilities of addressing complex challenges in understanding group behavior and system dynamics from graph data.

Here is an architectural design workflow where which works if the GDC is already installed:

1. Load data from Neo4j
2. Store in an efficient data structure
3. Run graph algorithm using the Graph API
4. Return a result

The graph algorithms packaged in GDC as procedures can be implemented and executed directly using cypher in cypher shell, client codes (C#, Java, etc.), or Neo4j Browser. GDC contains categories of graph algorithms such as pathfinding, centrality, community detection, similarity, and link prediction.

1.2.2. Types of graphs and graph projections

There are two types of graphs, anonymous graphs, and named graphs. Graph catalog manages named graphs, and when Neo4j is restarted, named graphs get lost stored in the graph catalog. Thus, named graphs serve as global variables holding the graph until deleted or removed by the user or the system. Anonymous graphs are created when algorithms operate, but they are not stored in the catalog.

In Neo4j, algorithms operate on named graphs, which are created using projections. There are two types of projections, native projections, and cypher projection. One of the advantages of using projections is the separation of projected graphs from the Neo4j database, enabling fast caching and processing of graph data.

Creating graph using native projections

Graph catalog stores projected graphs using user-defined names. Thus, you can use the user-defined name to refer to the graph in algorithms procedure calls as a parameter allowing several algorithms to use the same graphs without having to create them.

It is recommended to have native projections to provide the best performance in development and even production phases.

Syntax

```
– note that graphName, nodeProjection, and relationshipProjection are mandatory while the rest of the parameters are optional
CALL gds.graph.create(
    graphName: String,
    nodeProjection: String or List or Map,
    relationshipProjection: String or List or Map,
    configuration: Map
)
YIELD
  graphName: String,
  nodeProjection: Map,
  nodeCount: Integer,
  relationshipProjection: Map,
  relationshipCount: Integer,
  createMillis: Integer
```

Example – Given graph data in Neo4j with node of type Person with relationship type 'FOLLOWS' - Create a graph named myProjection with nodes of type Person and relationship type 'FOLLOWS.

```
CALL gds.graph.create(
        'myProjection',
        'Person',
        'FOLLOWS'
)
YIELD
        graphName AS graph, nodeProjection, nodeCount AS nodes, relationshipProjection,
      relationshipCount AS rels
```

Example – Given graph data in Neo4j with node of type Person and Moving with relationship types 'FOLLOWS' and ACTED_IN - Create a graph with a graph named myProjection having nodes of type Person and Movies, and relationship typeS 'FOLLOWS' and 'ACTED_IN.'

```
CALL gds.graph.create(
        'myProjection',
        ['Person', 'Book'],
        ['FOLLOWS', 'ACTED_IN']
)
YIELD
      graphName AS graph, nodeProjection, nodeCount AS nodes, relationshipCount AS rels
```

Note – you can also create graphs by specifying all of the following – node properties, relationship properties, and relationship types.

Creating graph using cypher (cypher projections)

Graph catalog stores projected graphs using user-defined names. The graphs can be created using native projection as illustrated above or cypher projection, where cypher projection has a more expressive and flexible way of projection than native projection while penalizing performance. Thus, cypher projections are recommended for development phases rather than production phases.

Syntax

– note that graphName, nodeQuery, and relationshipQuery are mandatory while the rest of the parameters are optional

```
CALL gds.graph.create(
    graphQuery: String,
    nodeQuery: String or List or Map,
    relationshipQuery: String or List or Map,
    configuration: Map
)
YIELD
  graphName: String,
  nodeQuery: Map,
  nodeCount: Integer,
  relationshipQuery: Map,
  relationshipCount: Integer,
  createMillis: Integer
```

Example – creating named graph using cypher projection - Create a graph with a graph named myProjection having nodes of type Person and relationship type 'FOLLOWS.'

```
CALL gds.graph.create.cypher(
 'myProjection',
  MATCH (n:Person) RETURN id(n) AS id,
  MATCH (n:Person)-[r:FOLLOWS]->(n2:Person) RETURN id(n) AS source, id(n2) AS target'
)
YIELD
  graphName AS graph, nodeQuery, nodeCount AS nodes, relationshipQuery, relationshipCount AS rels
```

Note – you can also create graphs by specifying all of the following – node properties, relationship properties, and relationship types.

1.2.3. Using graph algorithms

Graph algorithms run on the top of graph projections, which is a graph data model itself, see Figure 1.

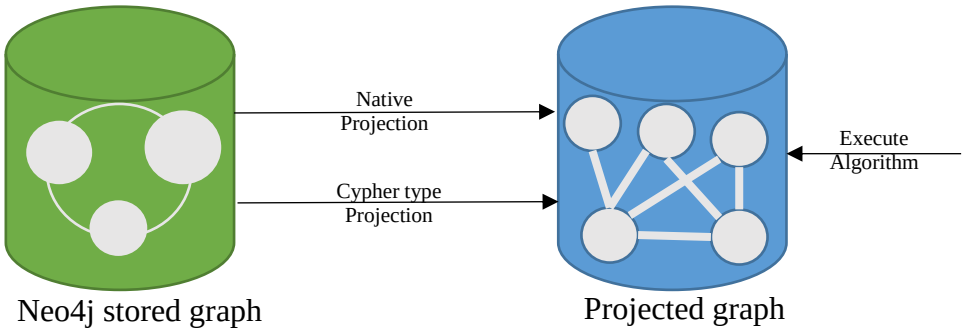


Figure 1.1 graph algorithms run on projected graphs, native projection and cypher projection.

1.3. Syntax of graph algorithms execution

There are two variants of graph algorithms, namely the named graph variant and the anonymous graph variant. The named graph variant uses a named graph stored in the graph catalog of Neo4j,

while the anonymous graph variant creates the graph on the fly and deletes it as the execution ends.

Syntax with anonymous graph

```
CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>](
      configuration: Map

 )
```

Syntax with named graph

```
CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>](
      graphName: String,
      configuration: Map
)
```

Tier [<tier>]

Here, the tier is optional, representing the algorithms' quality tier. If the algorithm is at the production stage, it will be omitted. Otherwise, it is at an alpha or beta tier under development. Production-quality tier support all execution modes and memory estimation procedures.

Algorithms <algorithm>

- The algorithm's name to be used is written.

Execution mode:

- **stream** – the algorithm returns the result as a stream of record as a cypher result. The results are algorithms computations for nodes, such as Page Rank score, node IDs, and similarity scores.
- **stats** – the algorithm returns a record showing summary statistics, but it does not write to the Neo4j database.
- **mutate** – the algorithm writes the result to the in-memory graph returning summary statistics. (works only with named graphs)
    - this option is useful in three scenarios – (1) when the current algorithms use the results of previously executed algorithms without being written to the Neo4j, (2) when the results of algorithms are written together, and (3) when the results of algorithms are queried via cypher without being written to Neo4j.
- **write** – the algorithm writes the result as a stream of records to the Neo4j graph database.
    - The written data include new relationship or relationship properties and node properties (see the example, in the advanced voluntary tutorial, Boom visualization).

Estimate [.<estimate>]

- If the estimate option is appended to the algorithm, then Neo4j will do a memory estimation for the execution.

2. Development environment

You can access Neo4j Desktop through your Remote Windows Desktop access from your laptops or DSV labs. At it is your first access to Neo4j, you will be prompted to register using your SU email.

TODO: Steps

- Start Neo4j Desktop
- Create a new project with the filename 'Intro algo' and a database name 'algo1'
- Enter a password – use 'graph1234' as your password.
- Install – Graph Data Science Library and APOC plugins
- Start the database (if you get an error, keep on trying to start)
- Start Neo4j browser ← this is where you will be working on

3. Pathfinding algorithms

The graph search and pathfinding algorithms reach a designation from the start at a node by expanding relationships. Search algorithms find a path that might not be the shortest, while pathfinding algorithms find the cheapest path based on weight or number of hopes.

3.1. Shortest path algorithms

The shortest path algorithms are used to calculate the shortest path between nodes using their weight between nodes in a path. The most common shortest path algorithm is Dijkstra's algorithm which is still used in mobile and web applications. In this exercise, we will learn two types of shortest path algorithms:

- Shortest path (the shortest path between a pair of nodes)
- Single-source shortest path

3.1.1. Shortest path

The shortest path is an algorithm that calculates the shortest path between a pair of nodes. The most popular shortest path algorithm is Dijkstra's algorithm.

Extended syntax for execution mode stream and on a named graph

```
CALL gds.shortestPath.dijkstra.stream(
  graphName: String,
  configuration: Map
)
YIELD
  index: Integer,
  sourceNode: Integer,
  targetNode: Integer,
  totalCost: Float,
  nodeIds: List of Integer,
  costs: List of Float,
```

```
path: Path
```

Where

Parameters:

- **graphName** (string) – is mandatory and is stored in catalog
- **configuration** (Map) is optional, it contains algorithm specific configurations.

Configuration for named graph:

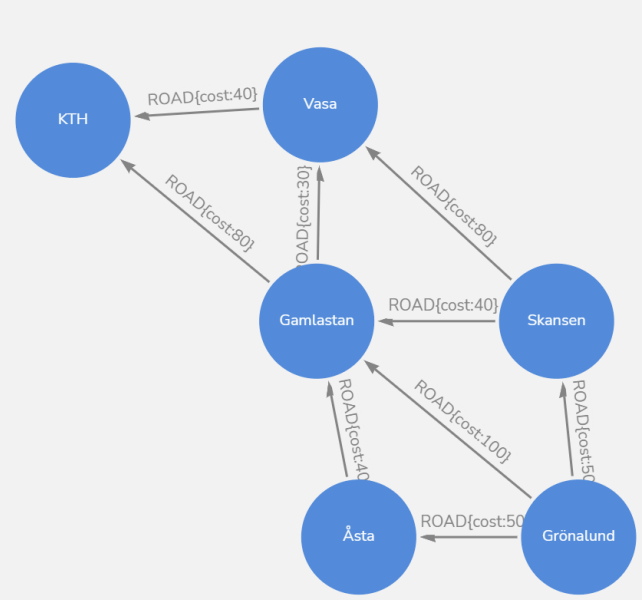
- **nodeLabels** (List of String, optional, includes all labels by default) – is used to filter named graph using labels
- **relationshipType** (List of String, optional, includes all labels by default) – is used to filter named graph using relationship type
- **concurrency** (Integer, optional) – is used to specify the number of concurrent threads for the execution of the algorithm, and its default value is 4.
- **sourceNode** (Integer, mandatory) – is source node id
- **targetNode** (Integer, mandatory) – is source node id
- **relationshipWeightProperty** (String, optional) – is the name of the relationship to be used as weights (if this is not specified the algorithms runs unweighted)

Results:

- index (Integer), returns index of the path identified, sourceNode (Integer), targetNode (Integer), totalCost (Float), nodeIds (List of Integer), costs (List of Float), and path.

Example:

Diagram -



Cypher to create this graph

```
MERGE (n1:Loc {name:"Grönalund"})
MERGE (n2:Loc {name:"Åsta "})
MERGE (n3:Loc {name:"Skansen"})
MERGE (n4:Loc {name:"Gamlastan "})
MERGE (n5:Loc {name:"Vasa"})
```

```
MERGE (n6:Loc {name:"KTH"})

MERGE (n1)-[:ROAD {cost:50}]->(n2)
MERGE (n1)-[:ROAD {cost:50}]->(n3)
MERGE (n1)-[:ROAD {cost:100}]->(n4)
MERGE (n2)-[:ROAD {cost:40}]->(n4)
MERGE (n3)-[:ROAD {cost:40}]->(n4)
MERGE (n3)-[:ROAD {cost:80}]->(n5)
MERGE (n4)-[:ROAD {cost:30}]->(n5)
MERGE (n4)-[:ROAD {cost:80}]->(n6)
MERGE (n5)-[:ROAD {cost:40}]->(n6);
```

Create a named projection graph called ‘’

```
CALL gds.graph.create(
    'myProjectedGraph',
    'Loc',
    'ROAD',
    {
        relationshipProperties: 'cost'
    }
)
```

To do memory estimation of the algorithm

```
MATCH (source:Loc {name: 'Grönalund'}), (target:Loc {name: 'KTH'})
CALL gds.shortestPath.dijkstra.write.estimate('myProjectedGraph', {
    sourceNode: source,
    targetNode: target,
    relationshipWeightProperty: 'cost',
    writeRelationshipType: 'PATH'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

You will now see how many bytes are required by the memory to run the algorithm

To calculate the shortest path between Grönalund and KTH. use the following query using stream mode

```
MATCH (source:Loc {name: 'Grönalund'}), (target:Loc {name: 'KTH'})
CALL gds.shortestPath.dijkstra.stream('myProjectedGraph', {
   sourceNode: source,
   targetNode: target,
   relationshipWeightProperty: 'cost'
})
YIELD nodeIds, costs, path, totalCost
RETURN
   nodeIds,[nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
costs,totalCost
```

The fasterst path with the smallest cost is shown below

nodeIds	nodeNames	costs	totalCost
[0, 1, 3, 4, 5]	["Grönalund", "Ästa ", "Gamlastan ", "Vasa", "KTH"]	[0.0, 50.0, 90.0, 120.0, 160.0]	160.0

3.1.2. Single source shortest path

Single source shortest path (SSSP) algorithms calculate the shortest path from a given node to all other nodes. SSSP and shortest path algorithms are implemented using Dijkstra's algorithms. However, according to (Needham and Hodler, 2018), Neo4j implements a delta-stepping algorithm, a variation of SSSP. Delta-stepping algorithm runs in both parallel and sequential settings for a variety of graph types, and it outperforms Dijkstra's algorithms (Needham and Hodler, 2018). SSSP is applicable in detecting link failures in IP networks through topology changes.

Memory estimation

```
MATCH (source:Loc {name: 'Grönalund'})
CALL gds.allShortestPaths.dijkstra.write.estimate('myProjectedGraph', {
    sourceNode: source,
    relationshipWeightProperty: 'cost',
    writeRelationshipType: 'PATH'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

You will now see how many bytes are required by the memory to run the algorithm

Extended syntax for execution mode stream and on a named graph

```
CALL gds.allShortestPaths.dijkstra.stream(
  graphName: String,
  configuration: Map
)
YIELD
  index: Integer,
  sourceNode: Integer,
  targetNode: Integer,
```

```
totalCost: Float,
nodeIds: List of Integer,
costs: List of Float,
path: Path
```

Where

Parameters:

- **graphName** (string) – is mandatory and is stored in catalog
- **configuration** (Map) is optional, it contains algorithm specific configurations.

Configuration for named graph:

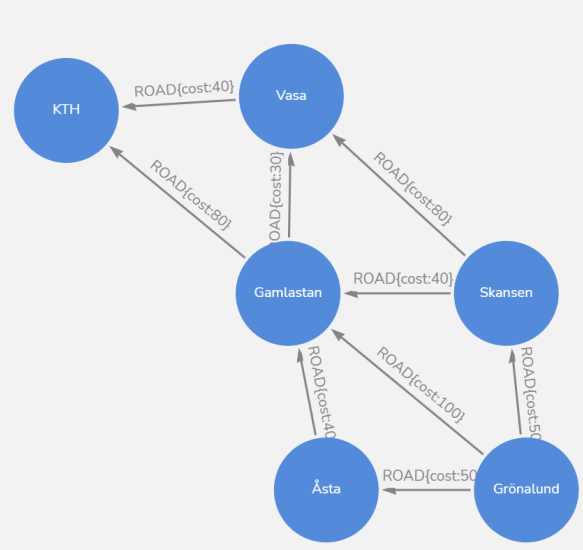
- **nodeLabels** (List of String, optional, includes all labels by default) – is used to filter named graph using labels
- **relationshipType** (List of String, optional, includes all labels by default) – is used to filter named graph using relationship type
- **concurrency** (Integer, optional) – is used to specify the number of concurrent threads for the execution of the algorithm, and its default value is 4.
- **sourceNode** (Integer, mandatory) – is source node id
- **relationshipWeightProperty** (String, optional) – is the name of the relationship to be used as weights (if this is not specified the algorithms runs unweighted)

Results:

- index (Integer), returns index of the path identified, sourceNode (Integer), totalCost (Float), nodeIds (List of Integer), costs (List of Float), and path.

Example: - single source shortest path

Diagram -



Cypher to create this graph – as presented earlier:

```
MERGE (n1:Loc {name:"Grönalund"})
MERGE (n2:Loc {name:"Åsta "})
MERGE (n3:Loc {name:"Skansen"})
MERGE (n4:Loc {name:"Gamlastan "})
MERGE (n5:Loc {name:"Vasa"})
MERGE (n6:Loc {name:"KTH"})
```

```
MERGE (n1)-[:ROAD {cost:50}]->(n2)
MERGE (n1)-[:ROAD {cost:50}]->(n3)
MERGE (n1)-[:ROAD {cost:100}]->(n4)
MERGE (n2)-[:ROAD {cost:40}]->(n4)
MERGE (n3)-[:ROAD {cost:40}]->(n4)
MERGE (n3)-[:ROAD {cost:80}]->(n5)
MERGE (n4)-[:ROAD {cost:30}]->(n5)
MERGE (n4)-[:ROAD {cost:80}]->(n6)
MERGE (n5)-[:ROAD {cost:40}]->(n6);
```

To calculate the single source shortest path between Grönalund to all other locations. use the following query:

```
MATCH (source:Loc {name: 'Grönalund'})
CALL gds.allShortestPaths.dijkstra.stream('myProjectedGraph', {
    sourceNode: source,
    relationshipWeightProperty: 'cost'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
    index,
    gds.util.asNode(sourceNode).name AS sourceNodeName,
    gds.util.asNode(targetNode).name AS targetNodeName,
    totalCost,
    [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
    costs,
    nodes(path) as path
ORDER BY index
```

"index"	"sourceNodeName"	"targetNodeName"	"totalCost"	"nodeNames"	"costs"	"path"
0	"Grönalund"	"Grönalund"	0.0	["Grönalund"]	[0.0]	[{"name":"Grönalund"}]
1	"Grönalund"	"Åsta "	50.0	["Grönalund","Åsta "]	[0.0,50.0]	[{"name":"Grönalund"},{"name":"Åsta "}]
2	"Grönalund"	"Skansen"	50.0	["Grönalund","Skansen"]	[0.0,50.0]	[{"name":"Grönalund"},{"name":"Skansen"}]
3	"Grönalund"	"Gamlastan "	90.0	["Grönalund","Åsta ","Gamlastan "]	[0.0,50.0,90.0]	[{"name":"Grönalund"},{"name":"Åsta "},{"name":"Gamlastan "}]
4	"Grönalund"	"Vasa"	120.0	["Grönalund","Åsta ","Gamlastan ","Vasa"]	[0.0,50.0,90.0,120.0]	[{"name":"Grönalund"},{"name":"Åsta "},{"name":"Gamlastan "},{"name":"Vasa"}]
5	"Grönalund"	"KTH"	160.0	["Grönalund","Åsta ","Gamlastan ","Vasa","KTH"]	[0.0,50.0,90.0,120.0,160.0]	[{"name":"Grönalund"},{"name":"Åsta "},{"name":"Gamlastan "},{"name":"Vasa"},{"name":"KTH"}]

4. Community detection algorithms

Community detection algorithms are used to evaluate how a group of nodes have the tendency to break apart or strengthen, as well as partitioning or clustering of the group of nodes.

4.1. Label Propagation algorithm (LPA)

This algorithm is a fast algorithm that enables you to find communities in a graph using network structure alone without requiring prior information about communities. You have the option to assign preliminary labels to narrow down the volume of generated solutions; hence it can be considered as a semi-supervised way of community detection. Thus, you have the option to handpick initial communities. LPA was first proposed by Raghavan et al. (2007), and LPA propagates labels throughout the graph, forming communities.

How does it work?

- Initialization of each node with unique labels (identifier)
- Labels propagate through the network
- On every iteration of propagation, based on the maximum number of its neighbors that a node belongs it updates its label
- LPA converges when each node has the majority label of neighbor nodes

Example: - Label propagation

Extended syntax for execution mode stream and on a named graph

```
CALL gds.labelPropagation.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  communityId: Integer,
```

Where

Parameters:

- **graphName** (string) – is mandatory and is stored in catalog
- **configuration** (Map) is optional, it contains algorithm specific configurations.
- **nodeLabels** (List of String, optional, includes all labels by default) – is used to filter named graph using labels
- **relationshipType** (List of String, optional, includes all labels by default) – is used to filter named graph using relationship types
- **concurrency** (Integer, optional) – is used to specify the number of concurrent threads for the execution of the algorithm, and its default value is 4.
- **nodeWeightProperty** (String, optional) – is the name of the node property that contains weights
- **relationshipWeightProperty** (String, optional) – is the name of the relationship to be used as weights (if this is not specified the algorithms runs unweighted)
- **seedProperty** (String, Optional) – is the initial component for a node, where the value must be a number.
- **maxIterations** (Integer, Optional, default value = 10) – is the maximum value for the number of iterations that the algorithm runs.

- **consecutiveIds** (Boolean, Optional, default value = false) – this flag indicates if component identifiers are mapped or not with consecutive id space (this option takes additional memory.

Results:

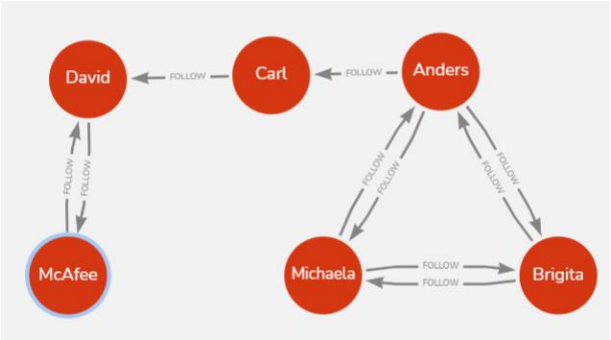
- nodeId, communityId.

Delete the graph in catalog(myProjectedGraph)

- CALL gds.graph.drop('myProjectedGraph')

- MATCH(n) DETACH DELETE n

Example: - Label propagation

Diagram - 

Cypher to create this graph

```
MERGE (nAnders:User {id:"Anders", seed: 52})
MERGE (nBrigita:User {id:"Brigita", seed: 21})
MERGE (nCarl:User {id:"Carl", seed: 43})
MERGE (nDavid:User {id:"David", seed: 21})
MERGE (nMcAfee:User {id:"McAfee", seed: 19})
MERGE (nMichaela:User {id:"Michaela", seed: 52})

MERGE (nAnders)-[:FRIEND {weight: 1}]->( nBrigita)
MERGE (nAnders)-[:FRIEND {weight: 10}]->( nCarl)
MERGE (nMcAfee)-[:FRIEND {weight: 1}]->( nDavid)
MERGE (nBrigita)-[:FRIEND {weight: 1}]->(nMichaela)
MERGE (nDavid)-[:FRIEND {weight: 1}]->( nMcAfee)
MERGE (nMichaela)-[:FRIEND {weight: 1}]->( nAnders)
MERGE (nAnders)-[:FRIEND {weight: 1}]->( nMichaela)
MERGE (nBrigita)-[:FRIEND {weight: 1}]->( nAnders)
MERGE (nMichaela)-[:FRIEND {weight: 1}]->(nBrigita)
MERGE (nCarl)-[:FRIEND {weight: 1}]->(nDavid);
```

Create subgraph named 'myProjectedGraph' - The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myProjectedGraph'.

CALL gds.graph.create(

```
    'myProjectedGraph',
    'User',
    'FRIEND',
    {
      nodeProperties: 'seed',
      relationshipProperties: 'weight'
    }
)
```

To estimate the memory estimation Label propagation, use the cypher:

```
CALL gds.labelPropagation.write.estimate('myProjectedGraph', { writeProperty:
'community' })
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

To find communities among users using Louvian, use the cypher:

```
CALL gds.labelPropagation.stream('myProjectedGraph')
YIELD nodeId, communityId AS Community
RETURN gds.util.asNode(nodeId).id AS Name, Community
ORDER BY Community, Name
```

"Name"	"Community"
"Anders"	1
"Brigita"	1
"Michaela"	1
"Carl"	4
"David"	4
"McAfee"	4

To get statistics about the algorithm you can run it in **stats** mode, run the following cypher.

```
CALL gds.labelPropagation.stats('myProjectedGraph')
YIELD communityCount, ranIterations, didConverge
```

"communityCount"	"ranIterations"	"didConverge"
2	3	true

4.2. Triangle count

Triangle Count is a community detection graph algorithm that identifies all possible triangles in a graph network, where a triangle is a set of three connected nodes. In social network analysis, triangle counting is a popular practice for detecting communities and measuring the cohesiveness

of these communities. Also, triangle count algorithms use clustering coefficients to measure the stability of a graph.

There are two types of clustering coefficients:

Local clustering coefficient

- The likelihood that neighboring nodes are connected is measured using the local clustering coefficient for each node through counting triangles.

Global clustering coefficient

- The normalized sum of the local clustering coefficient is referred to as global clustering.

For more on triangle count refer to (Schank and Wagner, 2005).

Extended syntax for execution mode stream and on a named graph

```
CALL gds.triangleCount.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  triangleCount: Integer
```

Where

Parameters:

- **graphName** (string) – is the name of the named graph, it is mandatory and is stored in catalog
- **configuration** (Map) is optional, it contains algorithm specific configurations.
- **nodeLabels** (List of String, optional, includes all labels by default) – is used to filter named graph using labels.
- **relationshipType** (List of String, optional, includes all labels by default) – is used to filter named graph using relationship type.
- **concurrency** (Integer, optional) – is used to specify the number of concurrent threads for the execution of the algorithm, and its default value is 4.
- **maxDegree** (Integer, Optional, default value = 2^{63}-1) – if the degree of a node is higher than the maxDegree value it will be excluded by the algorithm. The triangle count for these nodes will be -1.

Results:

- nodeId, trinagleId.

Delete the graph in catalog(myProjectedGraph)

- CALL gds.graph.drop('myProjectedGraph')
- MATCH(n) DETACH DELETE n

In addition to the triangle count algorithm, there are triangle listing algorithms, which is an alpha procedure enabling the listing of triangles found in the named graph.

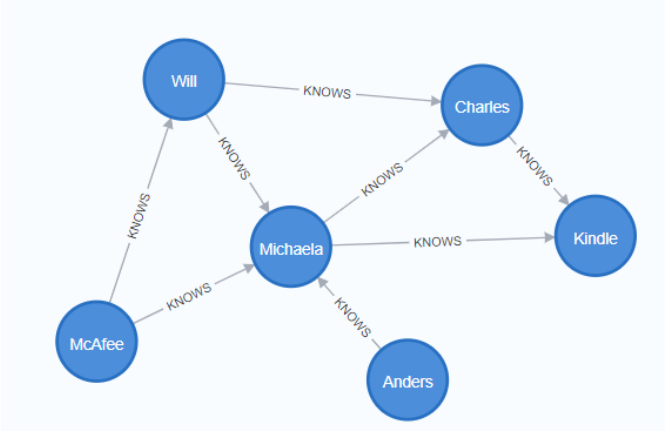
Triangle listing algorithms

```
CALL gds.alpha.triangles(
  graphName: String,
  configuration: Map
)
YIELD nodeA, nodeB, nodeC
```

Example:

Example: - Tringale count

Diagram -



Cypher to create this graph

```
MERGE (anders:Person{id:"Anders"})
MERGE (michaela:Person{id:"Michaela"})
MERGE (kindle:Person{id:"Kindle"})
MERGE (charles:Person{id:"Charles"})
MERGE (will:Person{id:"Will"})
MERGE (mcAfee:Person{id:"McAfee "})

MERGE (michaela)-[:KNOWS]->(kindle)
MERGE (michaela)-[:KNOWS]->(charles)
MERGE (will)-[:KNOWS]->(michaela)
MERGE (mcAfee)-[:KNOWS]->(michaela)
MERGE (mcAfee)-[:KNOWS]->(will)
MERGE (anders)-[:KNOWS]->(michaela)
MERGE (will)-[:KNOWS]->( charles)
MERGE (charles)-[:KNOWS]->(kindle);
```

Create named graph – the triangle count requires UNDIRECTED orientation graph

```
CALL gds.graph.create(
```

```
  'myProjectedGraph',
  'Person',
  {
    KNOWS: {
      orientation: 'UNDIRECTED'
    }
  }
)
```

Memory estimation

```
CALL gds.triangleCount.write.estimate('myProjectedGraph', { writeProperty:
'triangleCount' })
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

To find communities of the KNOWS triangles between users, use the cypher:

```
CALL gds.triangleCount.stream('myProjectedGraph')
YIELD nodeId, triangleCount
RETURN gds.util.asNode(nodeId).id AS name, triangleCount
ORDER BY triangleCount DESC
```

"name"	"triangleCount"
"Michaela"	3
"Charles"	2
"Will"	2
"Kindle"	1
"McAfee "	1
"Anders"	0

What do you understand from the result?

To get the stats or stats execution mode, where you will be able to see a summary of the result in a row, use the following cypher.

```
CALL gds.triangleCount.stats('myProjectedGraph')
YIELD globalTriangleCount, nodeCount
```

"globalTriangleCount"	"nodeCount"
3	6

Here we can see that the graph has six nodes with a total number of three triangles. Comparing this to the **stream example,** we can see that the 'Michael' node has a triangle count equal to the global triangle count. In other words, that node is part of all of the triangles in the graph and thus has a very central position in the graph.

5. Centrality algorithms

Centrality algorithms, which are mainly invented in social network analysis, are used to find the most influential nodes from a graph database.

5.1. PageRank

Most centrality algorithms measure the direct influence of nodes, while PageRank measures the transitive or directional influence of nodes. PageRank is used to estimate importance and influence.

For more academic and online resources refer to the articles by (Brin and Page, 1998), (Manaskasemsak and Rungsawang, 2005), and (Gleich, 2015).

Syntax

```
CALL gds.pageRank.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  score: Float
```

Where

Parameters:

- **graphName** (string) – is the name of the named graph, it is mandatory and is stored in catalog
- **configuration** (Map) is optional, it contains algorithm specific configurations.
- **nodeLabels** (List of String, optional, includes all labels by default) – is used to filter named graph using labels.
- **relationshipType** (List of String, optional, includes all labels by default) – is used to filter named graph using relationship type.
- **concurrency** (Integer, optional) – is used to specify the number of concurrent threads for the execution of the algorithm, and its default value is 4.

- **dampingFactor** (Float, Optional, default value = 0.85) – is used to set the damping factor and its value must be between o and 1
- **maxIterations** (Integer, Optional, default value = 20) – is used to the maximum iteration value
- **tolerance** (Float, Optional, default value = 0.0000001) – is the minimum change of scores between iterations, and if these change is less than the value of the tolerance the algorithm returns as the result is considered stable.
- **relationshipWeightProperty** (String, Optional) – is the name of the relationship to be used as weights (if this is not specified the algorithms runs unweighted)
- **sourceNodes** (List or Node or Number, Optional) – represents node ids or nodes that is used for computing Page Rank.
- **scaler** (String, Optional) – is the name of scaler used for the final scores, and it supports the following values - None, StdScore, MinMax, Mean, Max, Log, L1Norm, and L2Norm

Results:

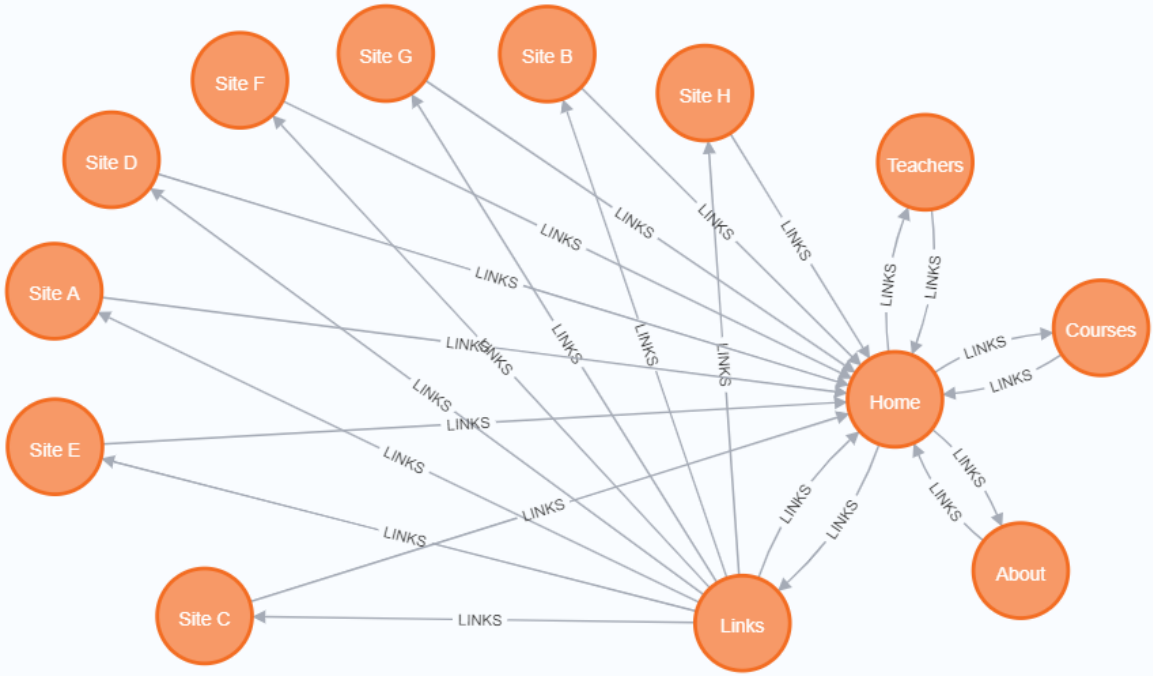
- nodeId, score.

Delete the graph in catalog(myProjectedGraph)

```
CALL gds.graph.drop('myProjectedGraph')
MATCH(n) DETACH DELETE n
```

Example: - PageRank

Diagram -



Compiled by - WYA

Cypher to create this graph

```
MERGE (home:Page {name:"Home"})
MERGE (about:Page {name:"About"})
MERGE (contuctUs:Page {name:"ContactUs"})
MERGE (courses:Page {name:"Courses"})
MERGE (teachers:Page {name:"Teachers"})

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 (e:Page {name:"Site E"})
MERGE (f:Page {name:"Site F"})
MERGE (g:Page {name:"Site G"})
MERGE (h:Page {name:"Site H"})

MERGE (home)-[:LINKS]->(teachers)
MERGE (teachers)-[:LINKS]->(home)
MERGE (home)-[:LINKS]->(about)
MERGE (about)-[:LINKS]->(home)
MERGE (home)-[:LINKS]->(ContactUs)
MERGE (ContactUs)-[:LINKS]->(home)
MERGE (courses)-[:LINKS]->(home)
MERGE (home)-[:LINKS]->(courses)
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)
MERGE (links)-[:LINKS]->(e)
MERGE (e)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(f)
MERGE (f)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(g)
MERGE (g)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(h)
MERGE (h)-[:LINKS]->(home)
```

Create named graph with native projections

```
CALL gds.graph.create(
  'myProjectedGraph',
  'Page',
  'LINKS',
  {
    relationshipProperties: 'weight'
  }
)
```

Memory estimation

```
CALL gds.pageRank.write.estimate('myProjectedGraph', {
  writeProperty: 'pageRank',
  maxIterations: 20,
  dampingFactor: 0.85
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

To calculate influential pages, use the following cypher:

```
CALL gds.pageRank.stream('myProjectedGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

"name"	"score"
"Home"	5.287499395254957
"About"	1.2650835808649896
"Courses"	1.2650835808649896
"Links"	1.2650835808649896
"Teachers"	1.2650835808649896
"Site A"	0.2685365471482999
"Site B"	0.2685365471482999
"Site C"	0.2685365471482999
"Site D"	0.2685365471482999
"Site E"	0.2685365471482999
"Site F"	0.2685365471482999
"Site G"	0.2685365471482999
"Site H"	0.2685365471482999
"ContactUs"	0.15000000000000002

What do you understand from the result?

5.2. Degree Centrality

Degree Centrality, which was first proposed by (Freeman 1978), is the simplest of all the centrality algorithms which measure the number of o. It measures the number of incoming and outgoing relationships outgoing and incoming relationships of nodes. The algorithms help you find the most popular node in graph data.

Supports – Directed, Undirected, Homogeneous, and Weighted graphs, while failing to support Heterogeneous graphs.

Syntax

```
CALL gds.degree.stream(
  graphName: String,
  configuration: Map
) YIELD
  nodeId: Integer,
  score: Float
```

Where

Parameters:

- **graphName** (string) – is the name of the named graph, it is mandatory and is stored in catalog
- **configuration** (Map) is optional, it contains algorithm specific configurations.
- **nodeLabels** (List of String, Optional, includes all labels by default) – is used to filter named graph using labels.
- **relationshipType** (List of String, Optional, includes all labels by default) – is used to filter named graph using relationship type.
- **concurrency** (Integer, Optional) – is used to specify the number of concurrent threads for the execution of the algorithm, and its default value is 4.
- **orientation** (String, Optional, default = NATURAL) – is used to specify the orientation for computing node degrees, and degree centrality algorithms supported NATURAL, REVERSE, and UNDIRECTED orentations.
- **relationshipWeightProperty** (String, Optional) – is the name of the relationship to be used as weights (if this is not specified the algorithms runs unweighted)

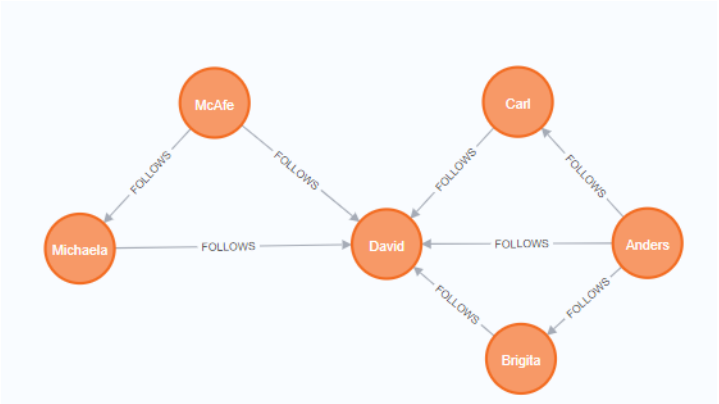
Results:

- nodeId, score.

Delete the graph in catalog(myProjectedGraph)

- CALL gds.graph.drop('myProjectedGraph')
- MATCH(n) DETACH DELETE n

Example: - degree centrality

Diagram - 

Cypher to create this graph

```
MERGE (nAnders:User {id:"Anders"})
MERGE (nBrigita:User {id:"Brigita"})
MERGE (nCarl:User {id:"Carl"})
MERGE (nDavid:User {id:"David"})
MERGE (nMcAfe:User {id:"McAfe"})
MERGE (nMichaela:User {id:"Michaela"})

MERGE (nAnders)-[:FOLLOWS {score: 1}]->(nDavid)
MERGE (nAnders)-[:FOLLOWS {score: -2}]->(nBrigita)
MERGE (nAnders)-[:FOLLOWS {score: 5}]->(nCarl)
MERGE (nMcAfe)-[:FOLLOWS {score: 1.5}]->(nDavid)
MERGE (nMcAfe)-[:FOLLOWS {score: 4.5}]->(nMichaela)
MERGE (nBrigita)-[:FOLLOWS {score: 1.5}]->(nDavid)
MERGE (nCarl)-[:FOLLOWS {score: 2}]->(nDavid)
MERGE (nMichaela)-[:FOLLOWS {score: 1.5}]->(nDavid)
```

Create named graph with native projection

```
CALL gds.graph.create(
  'myProjectedGraph',
  'User',
  {
    FOLLOWS: {
      orientation: 'REVERSE',
      properties: ['score']
    }
  }
)
```

Memory estimation

```
CALL gds.degree.write.estimate('myProjectedGraph', { writeProperty: 'degree' })
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

To count the number of people that every user follows and followed by:

```
MATCH (u:User)
RETURN u.id AS name,
size((u)-[:FOLLOWS]->()) AS follows,
size((u)<-[:FOLLOWS]-()) AS followers
```

"name"	"follows"	"followers"
"Anders"	3	0
"Brigita"	1	1
"Carl"	1	1
"David"	0	5
"McAfe"	2	0
"Michaela"	1	1

To calculate degree of centrality

```
CALL gds.degree.stream('myProjectedGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).id AS name, score AS followers
ORDER BY followers DESC, name DESC
```

"name"	"followers"
"David"	5.0
"Michaela"	1.0
"Carl"	1.0
"Brigita"	1.0
"McAfe"	0.0
"Anders"	0.0

What do you underestand?

6. Similarity

6.1. Node Similarity

In Neo4j, Node similarity is based on Jaccard Similarity Score. The algorithm supports Directed, Undirected, Homogeneous, Heterogeneous, and Weighted graphs. Jackard similarity is calculated using the formula:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

The input to the algorithm is a connected graph, which is bipartite, containing two disjoint sets of nodes.

For more information on this algorithm, see: Bipartite graph[1], Jaccard Index[2], and Structural Equivalence[3]

Note: Node similarity algorithms requires sufficient computational resource, more specifically available memory. Thus, it is recommended to check memory requirements before running it.

Syntax – named graph variant

```
CALL gds.nodeSimilarity.stream(
  graphName: String,
  configuration: Map
) YIELD
  node1: Integer,
  node2: Integer,
  similarity: Float
```

Where

Parameters:

- **graphName** (string) – is the name of the named graph, it is mandatory and is stored in catalog
- **configuration** (Map) is optional, it contains algorithm specific configurations.
- **nodeLabels** (List of String, Optional, includes all labels by default) – is used to filter named graph using labels.
- **relationshipType** (List of String, Optional, includes all labels by default) – is used to filter named graph using relationship type.
- **concurrency** (Integer, Optional) – is used to specify the number of concurrent threads for the execution of the algorithm, and its default value is 4.
- **similarityCutoff** (Float, Optional, default = 1E-42) – is used to specify the lower limit for the similarity score to be used in the result, and its value is between 0 and 1.

[1] https://en.wikipedia.org/wiki/Bipartite_graph
[2] https://en.wikipedia.org/wiki/Jaccard_index
[3] https://en.wikipedia.org/wiki/Similarity_(network_science)#Structural_equivalence

- **degreeCutoff** (Integer, Optional, default = 1) – is used to specify the lower limit for the node degree for consideration in the comparison, and its value must be greater than or equal to 1.
- **topK** (Integer, Optional, default = 10) – is used to specify the lower limit for the number of scores for each node, where the K largest results are returned, and its value must be greater than or equal to 1.
- **bottomK** (Integer, Optional, default = 10) – is used to specify the lower limit for the number of scores for each node, where the K smallest results are returned, and its value must be greater than or equal to 1.
- **topN** (Integer, Optional, default = 0) – is used to specify the global limit on the number of scores computed, where N largest total results are returned, and this value must be non-negative. Also, if the value is 0 this means there is no global limit.
- **bottomN** (Integer, Optional, default = 10) – is used to specify the global limit on the number of scores computed, where N smallest total results are returned, and this value must be non-negative. Also, if the value is 0 this means there is no global limit
- **relationshipWeightProperty** (String, Optional) – is the name of the relationship to be used as weights (if this is not specified the algorithms runs unweighted)

Results:

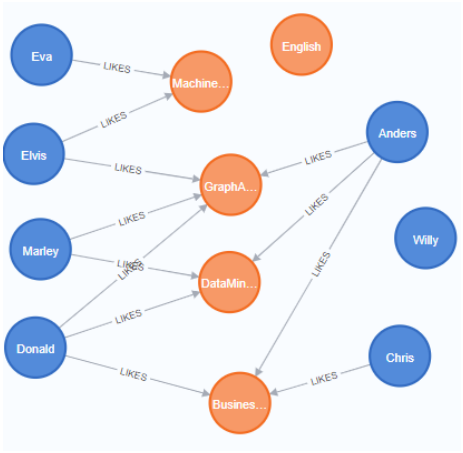
- node1, node2, similarity

Delete the graph in catalog(myProjectedGraph)

- CALL gds.graph.drop('myProjectedGraph')
- MATCH(n) DETACH DELETE n

Example: - node similarity

Diagram -



Cypher to create this graph

```
MERGE  (anders:Person {name: 'Anders'})
MERGE  (marley:Person {name: 'Marley'})
MERGE  (chris:Person {name: 'Chris'})
MERGE  (donald:Person {name: 'Donald'})
MERGE  (eva:Person {name: 'Eva'})
MERGE  (elvis:Person {name: 'Elvis'})

MERGE  (graphAnalytics:Course {name: 'GraphAnalytics'})
MERGE  (dataMining:Course {name: 'DataMining'})
MERGE  (businessIntelligence:Course {name: 'BusinessIntelligence'})
MERGE  (english:Course {name: 'English'})
MERGE  (machineLearning:Course {name: 'MachineLearning'})

MERGE  (anders)-[:LIKES]->(graphAnalytics)
MERGE  (anders)-[:LIKES {strength: 0.25}]->(dataMining)
MERGE  (anders)-[:LIKES {strength: 0.5}]->(businessIntelligence)
MERGE  (marley)-[:LIKES {strength: 0.25}]->(graphAnalytics)
MERGE  (marley)-[:LIKES {strength: 0.25}]->(dataMining)
MERGE  (chris)-[:LIKES {strength: 0.25}]->(businessIntelligence)
MERGE  (donald)-[:LIKES {strength: 0.25}]->(graphAnalytics)
MERGE  (donald)-[:LIKES]->(dataMining)
MERGE  (donald)-[:LIKES {strength: 0.25}]->(businessIntelligence)
MERGE  (eva)-[:LIKES {strength: 0.25}]->(machineLearning)
MERGE  (elvis)-[:LIKES {strength: 0.25}]->(graphAnalytics)
MERGE  (elvis)-[:LIKES {strength: 0.25}]->(machineLearning);
```

This bipartite graph in this example is consists of a set of two node types connected via LIKES relationships, namely Person and instrument. Thus, the example illustrates how we can compare people based on their preferences to music instruments.

Create named graph with native projection

```
CALL gds.graph.create(
  'myProjectedGraph',
  ['Person', 'Course'],
  {
    LIKES: {
      type: 'LIKES',
      properties: {
        strength: {
          property: 'strength',
          defaultValue: 1.0
        }
      }
    }
  }
);
```

Memory estimation

```
CALL gds.nodeSimilarity.write.estimate('myProjectedGraph', {
  writeRelationshipType: 'SIMILAR',
  writeProperty: 'score'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

To calculate the number similarity of people's preferences towards courses:

```
CALL gds.nodeSimilarity.stream('myProjectedGraph')
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS
Person2, similarity
ORDER BY similarity DESCENDING, Person1, Person2
```

"Person1"	"Person2"	"similarity"
"Anders"	"Donald"	1.0
"Donald"	"Anders"	1.0
"Anders"	"Marley"	0.6666666666666666
"Donald"	"Marley"	0.6666666666666666
"Marley"	"Anders"	0.6666666666666666
"Marley"	"Donald"	0.6666666666666666
"Elvis"	"Eva"	0.5
"Eva"	"Elvis"	0.5
"Anders"	"Chris"	0.3333333333333333
"Chris"	"Anders"	0.3333333333333333
"Chris"	"Donald"	0.3333333333333333
"Donald"	"Chris"	0.3333333333333333
"Elvis"	"Marley"	0.3333333333333333
"Marley"	"Elvis"	0.3333333333333333
"Anders"	"Elvis"	0.25
"Donald"	"Elvis"	0.25
"Elvis"	"Anders"	0.25
"Elvis"	"Donald"	0.25

6.2. K-Nearest Neighbors

In Neo4j, K-Nearest Neighbors algorithms create new relationships between each and every node and its K-Nearest neighbors after computing distance values for each node pair in the graph. The K-Nearest distance is calculated based on the properties of nodes.

Inputs to the algorithm and how it is done!

- a monopartite graph (where the relationship between nodes is ignored, is irrelevant)
- numeric properties of nodes are used for calculating and comparing the distance between nodes
- the initial set of neighbors are randomly picked and refined in iterations, and the **maxIterations** is configured as a parameter
- the algorithm will stop before the limited configuration, which is controlled by **deltaThreshold**
- The algorithm compares a sample of possible neighbors on each iteration (controlled by the configuration parameter - **sampleRate**), sample rate varies between 0 (exclusive) and 1 (inclusive), where the default value is 0.5
- SampleRate – controls the trade-off between runtime performance and accuracy
    - A higher sample rate boosts higher accuracy with a penalty on computational resources.
    - A lower sample rate boosts runtime performance, but some nodes could be missed during comparison.

Outputs of the algorithms

- New relationships will be created between each node
- K nearest neighbors between nodes.

For more information on this algorithm, see: (Wei Dong et al. 2011), and the article in Wikipedia Nearest neighbor graph[4]

Note: K-Nearest Neighbor algorithm requires sufficient computational resources, more specifically available memory. Thus, it is recommended to check memory requirements before running it.

6.2.1. Similarity computation

The similarity measure used in the KNN algorithm depends on the type of the configured node property. KNN supports both scalar numeric values as well as lists of numbers. The similarity is computed as discussed below,

Where p_s – is property of source node and p_t – is property of target node

[4] https://en.wikipedia.org/wiki/Nearest_neighbor_graph

Scalar numeric property

- If the property is a scalar number, the similarity is calculated as $1 / (1 + [p_s – p_t])$

List of integers

- If the property is a list of integers, the similarity is calculated as $1 / (1 + \triangle(p_s, p_t))$,

Here, $\triangle(p_s, p_t)$ is the number of unequal numbers in the list.

List of floating-point numbers

- Cosine similarity is used when the property is a list of floating pint numbers.

Syntax – named graph variant

```
CALL gds.beta.knn.stream(
  graphName: String,
  configuration: Map
) YIELD
  node1: Integer,
  node2: Integer,
  similarity: Float
```

Where

Parameters and configuration:

- **graphName** (string) – is the name of the named graph, it is mandatory and is stored in catalog
- **configuration** (Map) is optional, it contains algorithm specific configurations.
- **nodeLabels** (List of String, Optional, includes all labels by default) – is used to filter named graph using labels.
- **relationshipType** (List of String, Optional, includes all labels by default) – is used to filter named graph using relationship type.
- **concurrency** (Integer, Optional) – is used to specify the number of concurrent threads for the execution of the algorithm, and its default value is 4.
- **nodeWeightProperty** (String, Optional) – is the name of the node property that contains weights to be used for similarity computation.
- **topK** (Integer, Optional, default = 10) – is used to specify the limit for the number of neighbor for each node, where the K number of nearest neighbors are returned, and its value must be greater than or equal to 1.
- **sampleRate** (Float, Optional, default =0.5) – is used to limit the number of comparison per node, sampleRate must be between 0 and 1, where 0 (exclusive) and 1 (inclusive).
- **deltaThreshold** (Float, Optional, default =0.001) – is a value in percentage between 0 (exclusive) and 1 (inclusive). The deltaThreshold allows the algorithms to determine when to stop, for example if there are fewer updates than the configured value then it stop execution.
- **maxIterations** (Integer, Optional, default =100) – is used to limit the number of iterations after which the algorithms stops.

- **randomJoins** (Integer, Optional, default =10) – is the number of attempts made to connect new neighbors based on random selection between every iteration.
- **randomSeed** (Integer, Optional) – if you set randomSeed, you also need to set concurrency to 1. The randomSeed parameter is meant to control the randomness of the algorithm.

Results:
- node1, node2, similarity

Delete the graph in catalog(myProjectedGraph)

- CALL gds.graph.drop('myProjectedGraph')
- MATCH(n) DETACH DELETE n

Example: - KNN similarity

Cypher to create a graph (we do not need relationship as we will be calculating similarity based on their properties) for finding out KNN using properties age and salary

```
MERGE (anders:Person {name: 'Anders', salary: 25666, age: 24})
MERGE (nina:Person {name: 'Nina', salary: 57000, age: 59})
MERGE  (marley:Person {name: 'Marley', salary: 45699, age: 73})
MERGE (aron:Person {name: 'Aron', salary: 27000, age: 22})
MERGE  (charlie:Person {name: 'Charlie', salary: 27545, age: 24})
MERGE (andrew:Person {name: 'Andrew', salary: 23000, age: 28})
MERGE  (donald:Person {name: 'Donald', salary: 56555, age: 48})
MERGE  (eva:Person {name: 'Eva', salary: 19000, age: 67});
```

In the example, we will to use the KNN algorithm to compare persons based on their age. We will use named graphs and native projections as in the previous examples.

Create named graph with native projection

```
CALL gds.graph.create(
  'myProjectedGraph',
  {
    Person: {
      label: 'Person',
      properties: ['salary', 'age']
    }
  },
  '*'
);
```

Memory estimation – for similarity using 'salary', you should calculate separately if you want to do it for 'age', because it is not allowed to use array list for *nodeWeightProperty*

```
CALL gds.beta.knn.write.estimate('myProjectedGraph', {
  nodeWeightProperty: 'salary',
  writeRelationshipType: 'SIMILAR',
  writeProperty: 'score',
  topK: 1
})
YIELD nodeCount, bytesMin, bytesMax, requiredMemory
```

To calculate the KNN similarity of each Person based on 'salary' (you can do it for 'age' as well):

```
CALL gds.beta.knn.stream('myProjectedGraph', {
   topK: 1,
   nodeWeightProperty: 'salary',
   // The following parameters are set to produce a deterministic result
   randomSeed: 1337,
   concurrency: 1,
   sampleRate: 1.0,
   deltaThreshold: 0.0
})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS
Person2, similarity
ORDER BY similarity DESCENDING, Person1, Person2
```

"Person1"	"Person2"	"similarity"
"Donald"	"Nina"	0.002242152466367713
"Nina"	"Donald"	0.002242152466367713
"Aron"	"Charlie"	0.0018315018315018315
"Charlie"	"Aron"	0.0018315018315018315
"Anders"	"Aron"	0.000749063670411985
"Andrew"	"Anders"	0.0003749531308586427
"Eva"	"Andrew"	0.00024993751562109475
"Marley"	"Donald"	0.00009210647508519849

KNN for age:

```
CALL gds.beta.knn.stream('myProjectedGraph', {
    topK: 1,
    nodeWeightProperty: 'age',
    // The following parameters are set to produce a deterministic result
    randomSeed: 1337,
    concurrency: 1,
    sampleRate: 1.0,
    deltaThreshold: 0.0
})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS
Person2, similarity
ORDER BY similarity DESCENDING, Person1, Person2
```

"Person1"	"Person2"	"similarity"
"Anders"	"Charlie"	1.0
"Charlie"	"Anders"	1.0
"Aron"	"Anders"	0.3333333333333333
"Andrew"	"Charlie"	0.2
"Eva"	"Marley"	0.14285714285714285
"Marley"	"Eva"	0.14285714285714285
"Nina"	"Eva"	0.1111111111111111
"Donald"	"Nina"	0.08333333333333333

In this example, we used default values for most parameters. However, we set randomSeed and concurrency 1 (concurrency should be 1 to control the randomness), to obtain the same result every time you invoke the procedure. Also, if topK parameter is set to 1, then the result will display a single nearest neighbor for each node.

7. Machine Learning Workflow

Machine learning procedures are designed for machine learning models, and training supervised machine learning models. The Model catalog provides access to machine learning models used to predict graphs. Machine learning models require preprocessing and hyperparameter tuning. Some of the Neo4j algorithms have trained models.

1. **Preprocessing**

   Preprocessing tasks of data are required in most machine learning scenarios. This preprocessing is done to extract useful features for the machine learning algorithm. Some of the feature extraction techniques and options include centrality algorithms, node embedding, auxiliary algorithms.

2. **Tuning parameters for training**

   The link prediction algorithm discussed in Section 7.1 has training parameters for tuning automatically. The parameters give you the options to limit the computational budget, and these parameters include maxEpochs, tolerance, and patience to decide how long the training should last. For example, higher patience and maxEpochs with lower tolerance will lead higher-quality models, but it takes a longer training time. On the other hand, limiting the computational expense can serve to mitigate overfitting and regularization.

7.1. Link Prediction

Link prediction can be carried out in supervised learning or unsupervised learning (using the measures directly).[5,6] In this tutorial, we will use the link prediction using the supervised machine learning technique, which predicts relationships between nodes. The prediction demands having a training model for learning existing pairs of nodes relationships. The implementation of link prediction in Neo4j trains a supervised machine learning model based on node properties and the relationships between nodes for the prediction of the future possibilities of potential links. A simplified workflow of link prediction contains:

1. Creating training and testing graphs
2. Train, evaluate and select models
3. Apply a model for prediction

Supports: undirected graphs

[5] https://neo4j.com/developer/graph-data-science/link-prediction/
[6] https://hackernoon.com/link-prediction-in-large-scale-networks-f836fcb05c88?gi=b86a42e1c8d4

7.1.1. Train, evaluate, and select models

The training mode of link prediction, ***gds.alpha.ml.linkPrediction.train***, do the training, models evaluation, select the best model, and stores it model catalog. The inputs to the training mode of the link prediction take relationship types with 0 or 1 values, where 0 represents a negative example (where you have pairs that are not connected). In contrast, 1 represents a positive example (where you have connected pairs). Also, the inputs to the training mode take two inputs:

1. Relationship type representing training graph
2. Relationship type representing the testing graph

The algorithm forms feature vectors for both source, $s = \{s_1, s_2, \ldots s_n\}$, and target nodes, $t = \{t_1, t_2, \ldots t_n\}$ then to obtain feature vector it uses three types of link feature combiners. The three supported feature combiners are:

- L2 – build feature vector f as: $f = \{(s_1 - t_1)^2, (s_2 - t_2)^2, \ldots, (s_n - t_n)^2\}$
- HADAMARD – build feature vector f as: $f = \{s_1 * t_1, s_2 * t_2, \ldots, s_n * t_n\}$
- COSINE – build feature vector f as: f = Cosine similarity between s and t vectors.

Use ***gds.alpha.ml.splitRelationsips()*** procedure to produce training and testing graphs.

7.1.2. Apply link prediction model

The prediction model uses a previously trained model as an input to the ***gds.alpha.ml.linkPrediction.predict.mutate***. This is done by retrieving the trained model from the model catalog and generating the probability of pairs of relationships that are not connected. There are two mandatory parameters to limit the size of the output, namely:

1. topN – to retain the topmost probable link predictions.
2. threshold – to retain predictions whose probability value is above the threshold

To train link prediction models, we need both training and testing graphs, and to obtain these graphs, we will use ***gds.alpha.ml.splitRelationships()*** by invoking it two times. The splitRelationship procedure creates a test graph and a 'remaining' graph when you run it for the first time. Then, when we invoke the procedure again to creates the training graph, this time it will create smaller 'remaining' graph.

Evaluation

The Area-Under-the-Precision-Recall-Curve (AUCPR) is currently the only supported evaluation metric for the link prediction model in the Neo4j. If you work with sparsely connected graphs, you will encounter the class imbalance problem. Unlike the Area-Under-the-Receiver-Operating-Character (AUROC), the AUCPR is suitable for the class imbalance problem. Assuming we have positive and negative classes, the parameter **negativeSamplingRatio** controls the ratio of sampling between these classes, and the relative weight of classes is controlled by **negativeClassWeight**. Tunning **negativeClassWeight** is done by weighting false positives up or down when computing precision.

It is recommended to set the value of **negativeSamplingRatio**, which is the true class ratio of the graph data. Yet, higher the **negativeSamplingRatio** entails a larger testing dataset and longer time to evaluate. The ratio of the total probability mass of positive vs. negative example test is approximately the multiplication of **negativeSamplingRatio** and **negativeClassWeight.**

The true class ratio of graph data is calculated as:

Total class ration = (q-r)/r,

- where q = n (n-1)/2 which is the total count of undirected relationships
- r is the number of actual undirected relationships.

To calculate r, use the following cypher:

Given, R1 and R2 as two relationship types, to calculate all relationships that are either R1 or R2:

```
MATCH (n1)-[rel :R1 | R2]-(n2)
WHERE n1 < n2
RETURN count(rel) AS r
```

It is recommended to choose a value for **negativeClassWeight** based on the following two factors:

- What is the desired total probability mass of both positive and negative ones in the test set?
- What is the ratio of sampled negative and positive examples?

It is traditionally recommended to obtain a value of 1 for **negativeSamplingRatio** * **negativeClassWeight**. Thus, in other words, the true class ratio should be close to 1.0. So when the recommended class ratio is close to 1.0, the AUCPR reflects the expected precision on unseen extremely imbalanced data.

Syntax – named graph variant

```
CALL gds.alpha.ml.linkPrediction.train(
        graphName: String,
        configuration: Map
) YIELD
        trainMillis: Integer,
        modelInfo: Map,
        configuration: Map
```

Where

Parameters and configuration:

- **graphName** (string) – is the name of the named graph, it is mandatory and is stored in catalog
- **configuration** (Map) – is optional, it contains algorithm specific configurations.

Algorithm specific configuration

- **modelName** (String, Mandatory) – it is the name of the model to be trained, but name must be absent from the Model catalog, i.e., it must be defined by you.
- **featureProperties** (String, Mandatory) – is the names of properties to be use as an input to the algorithm, all properties must be of type Float or List of Float.
- **nodeLabels** (List of String, Optional, includes all labels by default) – is used to filter named graph using labels.
- **relationshipType** (List of String, Optional, includes all labels by default) – is used to filter named graph using relationship type.
- **concurrency** (Integer, Optional) – is used to specify the number of concurrent threads for the execution of the algorithm, and its default value is 4.
- **trainRelationshipType** (String, Mandatory) – is relationship type for model training.
- **testRelationshipType** (String, Mandatory) – is relationship type for model evaluation.
- **validationFoldes** (String, Mandatory) – is the number of divisions of the graph data for training for model selection.
- **negativeClassWeight** (Float, Mandatory) – is the weight of negative examples in model selection, here positive examples have 1 as its weight.
- **params** (List of Map, Mandatory) – is list of model configuration for training and comparison purposes.
- **randomSeed** (Integer, Optional) –The randomSeed parameter is meant to control the randomness of the algorithm.

Model configuration

- **penality** (Float, Optional, Defult = 0.0) – is by default not applied but you can set it to apply logistic regression.
- **linkFeatureCombiner** (String, Optional, Default = "L2") – is the link feature combiner for combining two node features vectors for the training, here the values are L2, HADAMARD, and COSINE.
- **batchSize** (Integer, Optional, default =100) – is the number of nodes per batch
- **minEpochs** (Integer, Optional, default =1) – is the minimum number of training epocks.
- **maxEpochs** (Integer, Optional, default =100) – is the maximum number of training epocks.
- **patience** (Integer, Optional, default =1) – is the maximum number of acceptable unproductive consecutive epochs.
- **tolerance** (Integer, Optional, default =0.001) – is the acceptable minimal improvement loss to be cosnisdered productive.
- **nodeWeightProperty** (String, Optional) – is the name of the node property that contains weights to be used for similarity computation.
- **topK** (Integer, Optional, default = 10) – is used to specify the limit for the number of neighbor for each node, where the K number of nearest neighbors are returned, and its value must be greater than or equal to 1.
- **sampleRate** (Float, Optional, default =0.5) – is used to limit the number of comparison per node, sampleRate must be between 0 and 1, where 0 (exclusive) and 1 (inclusive).
- **deltaThreshold** (Float, Optional, default =0.001) – is a value in percentage between 0 (exclusive) and 1 (inclusive). The deltaThreshold allows the algorithms to determine when to stop, for example if there are fewer updates than the configured value then it stop execution.

- **maxIterations** (Integer, Optional, default =100) – is used to limit the number of iterations after which the algorithms stops.
- **randomJoins** (Integer, Optional, default =10) – is the number of attempts made to connect new neighbors based on random selection between every iteration.

Results:

- **trainMills** (Integer) – is the milliseconds used for training
- **modelInfo** (Map) – is the information about the wining and the training
- **configuration** (Map) – is the configuration used for the training procedure

Model info fields

- **bestParameters** (Map) – is the model parameters on average validation folds that performed best
- **metrics** (Map) – this maps metric description to evaluated metrics.

Illustration of modelInfo

```
{
    bestParameters: Map,   ← is the best model candidate configuration
    metrics: {       ← metrics map for AUCPR in this example
        AUCPR: {
            test: Float, ← is a value of the best model on the test set
            outerTrain: Float, ← is a value of the best model on the outer train set
            train: [{   ← the train entry lists scores for all candidate models
                avg: Float,
                max: Float,
                min: Float,
                params: Map
            },
            {
                avg: Float,
                max: Float,
                min: Float,
                params: Map
            },
            ...
            ],
            validation: [{← the validation entry lists the scores for all candidate models
                avg: Float,
                max: Float,
                min: Float,
                params: Map
            },
            {
                avg: Float,
                max: Float,
                min: Float,
                params: Map
            },
            ...
            ]
        }
    }
}
```

TODO:

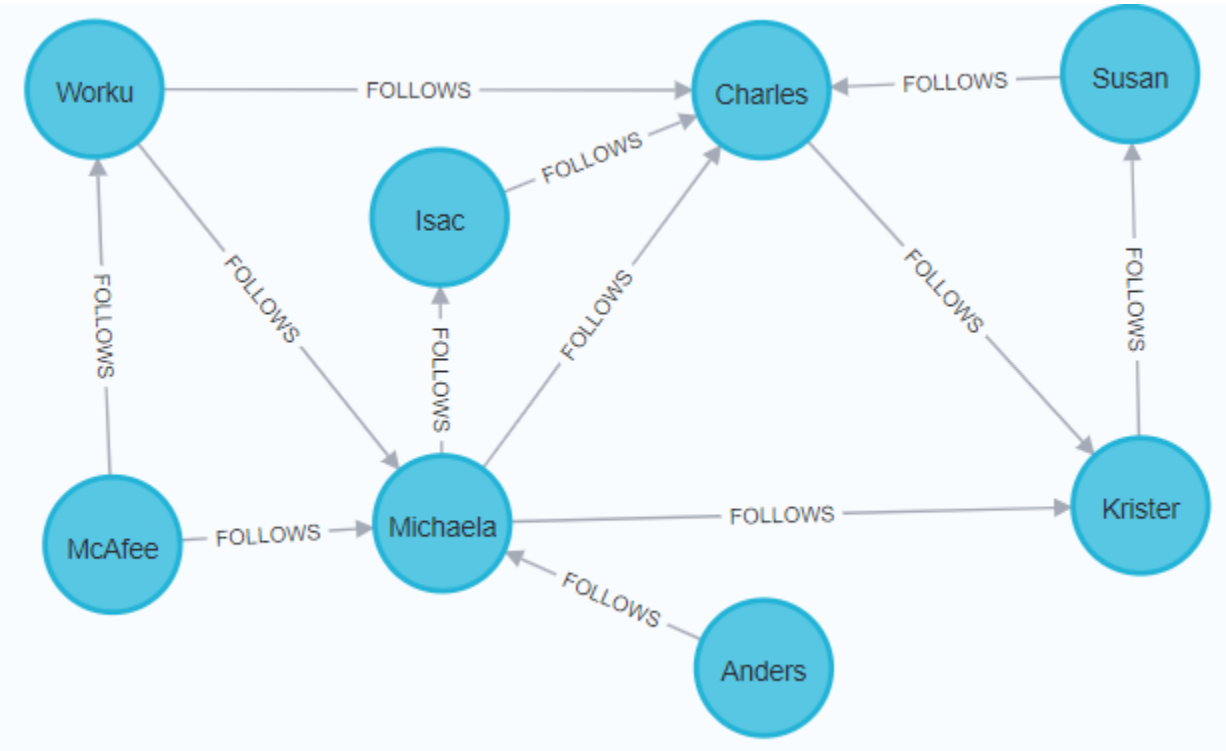
Create a new project with the name LinkPrediction, a database file **socialNetwork,** set the password to graph1234, install graph data science library

If you want to try this exercise on the same database without creating a new project that you have been working on before you begin, delete previously created nodes and graphs.

Delete the graph in catalog(myProjectedGraph)

- CALL gds.graph.drop('myProjectedGraph')
- MATCH(n) DETACH DELETE n

Example: - Link prediction – on a sample social media network which is illustrated below:



Cypher to create this graph of teachers with relationship – FOLLOWS and node property numberOfCitation

```
Merge (anders:Teacher {name: 'Anders', numberOfCitations: 76})
Merge (michaela:Teacher {name: 'Michaela', numberOfCitations: 136})
Merge (krister:Teacher {name: 'Krister', numberOfCitations: 60})
Merge (charles:Teacher {name: 'Charles', numberOfCitations: 264})
Merge (worku:Teacher {name: 'Worku', numberOfCitations: 12})
Merge (mcAfee:Teacher {name: 'McAfee', numberOfCitations: 64})
Merge (isac:Teacher {name: 'Isac', numberOfCitations: 58})
Merge (susan:Teacher {name: 'Susan', numberOfCitations: 6})

Merge (anders)-[:FOLLOWS]->(michaela)
Merge (michaela)-[:FOLLOWS]->(krister)
Merge (michaela)-[:FOLLOWS]->(charles)
Merge (michaela)-[:FOLLOWS]->(isac)
Merge (worku)-[:FOLLOWS]->(michaela)
Merge (worku)-[:FOLLOWS]->(charles)
Merge (mcAfee)-[:FOLLOWS]->(michaela)
Merge (mcAfee)-[:FOLLOWS]->(worku)
Merge (isac)-[:FOLLOWS]->(charles)
Merge (susan)-[:FOLLOWS]->(charles)
Merge (krister)-[:FOLLOWS]->(susan)
Merge (charles)-[:FOLLOWS]->(krister);
```

You will need to project the graph for the execution of Link Prediction. Here you will use the **Teacher** node and **FOLLOWS** relationship and the **numberOfCitations** property to be used as a model feature, and you also need to set the orientation to UNDIRECTED as link prediction only supports undirected graphs.

Create named graph with native projection

```
CALL gds.graph.create(
   'myProjectedGraph',
   {
      Teacher: {
         properties: 'numberOfCitations'
      }
   },
      {
      FOLLOWS: {
      orientation: 'UNDIRECTED'
      }
  }
);
```

Train the model

- Run the gds.alpha.ml.splitRelationship procedure to split test and train graph data:
  - Compute class radio – q = n (n-1)/2 = 8 *(8-1)/2 = 28
  - r = 12 (is the number of connections or undirected relationships) OR use the following cypher to compute

```
MATCH (n1)-[rel :FOLLOWS]-(n2)
WHERE n1 < n2
RETURN count(rel) AS r
```

  - ➔ which gives (q – r)/q = (28-12)/12 ≈ 1.33 = class ratio (use it for **negativeSampleRaio**)

Here, we will create relationships two, FOLLOWS_REMAINING and FOLLOWS_TESTGRAPH from the projected graph. All relationships.

- Run the following for the first time to create FOLLOWS_TESTGRAPH

```
CALL gds.alpha.ml.splitRelationships.mutate('myProjectedGraph', {
  relationshipTypes: ['FOLLOWS'],
  remainingRelationshipType: 'FOLLOWS_REMAINING',
  holdoutRelationshipType: 'FOLLOWS_TESTGRAPH',
  holdoutFraction: 0.2,
  negativeSamplingRatio: 1.33,
  randomSeed: 1984
}) YIELD relationshipsWritten
```

The result found after running the above cypher was:- relationshipsWritten = 25

Now all FOLLOWS_TESTGRAPH relationship will have a label property, where non existing relationships will have a label of 0.

- Run the following cypher to create FOLLOWS_REMAINING to create training set.

```
CALL gds.alpha.ml.splitRelationships.mutate('myProjectedGraph', {
  relationshipTypes: ['FOLLOWS_REMAINING'],
  remainingRelationshipType: 'FOLLOWS_IGNORED_FOR_TRAINING',
  holdoutRelationshipType: 'FOLLOWS_TRAINGRAPH',
  holdoutFraction: 0.2,
  negativeSamplingRatio: 1.33,
  randomSeed: 1984
}) YIELD relationshipsWritten
```

- Here, the relationshipsWritten = 20, now we have testing and training graphs so we can proceed with training models. Let us use 5 validation folds. Also, we know that the negativeSamplingRatio = 1.33, we can obtain the negativeClassWeight to 1/1.33 to balance both classes by assigning equal weight to both classes.

```
CALL gds.alpha.ml.linkPrediction.train('myProjectedGraph', {
  trainRelationshipType: 'FOLLOWS_TRAINGRAPH',
  testRelationshipType: 'FOLLOWS_TESTGRAPH',
  modelName: 'lp-numberOfCitations-model',
  featureProperties: ['numberOfCitations'],
  validationFolds: 5,
  negativeClassWeight: 1.0 / 1.33,
  randomSeed: 2,
  concurrency: 1,
  params: [
    {penalty: 0.5, maxEpochs: 1000},
    {penalty: 1.0, maxEpochs: 1000},
    {penalty: 0.0, maxEpochs: 1000}
  ]
}) YIELD modelInfo
RETURN
  { maxEpochs: modelInfo.bestParameters.maxEpochs, penalty: modelInfo.bestParameters.pe
nalty } AS winningModel,
  modelInfo.metrics.AUCPR.outerTrain AS trainGraphScore,
  modelInfo.metrics.AUCPR.test AS testGraphScore
```

"winningModel"	"trainGraphScore"	"testGraphScore"
{"maxEpochs":1000,"penalty":0.5}	0.8243978517319824	0.3424035623606439

- For the interpretation of the AUCPR read developer blog([7])

Run the algorithms on the stream mode.

```
CALL gds.alpha.ml.linkPrediction.predict.stream('myProjectedGraph', {
  relationshipTypes: ['FOLLOWS'],
  modelName: 'lp-numberOfCitations-model',
  topN: 10,
  threshold: 0.45
})
 YIELD node1, node2, probability
 RETURN gds.util.asNode(node1).name AS person1, gds.util.asNode(node2).name AS person
2, probability
 ORDER BY probability DESC, person1
```

The result with a threshold of less than 45% and for the top 10 relationships.

[7] https://neo4j.com/developer-blog/exploring-supervised-entity-resolution-in-neo4j/

"person1"	"person2"	"probability"
"Charles"	"McAfee"	0.9999999996608482
"Anders"	"Charles"	0.9999999957072341
"Michaela"	"Susan"	0.9999001265556566
"Anders"	"Susan"	0.9352120749630541
"Anders"	"Worku"	0.9030296338606805
"McAfee"	"Susan"	0.8620351283734978
"Isac"	"Susan"	0.8134363206212121
"Krister"	"Worku"	0.7780663363586696
"Worku"	"Isac"	0.7598658600681436
"Anders"	"Isac"	0.5436455230845717

There is a 99.999% probability for:

- Charles to follow McAfee
- Anders to follow Charles
- Michaela to follow Susan

There is a 93.521% probability for:

- Anders to follow Susan, etc.

8. References

Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, *25*(2), 163-177.

Brandes, U., & Pich, C. (2007). Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, *17*(07), 2303-2318.

Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, *30*(1-7), 107-117.

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, 35-41.

Freeman, L. C. (1978). Centrality in social networks conceptual clarification. *Social networks*, *1*(3), 215-239.

Gleich, D. F. (2015). PageRank beyond the web. *Siam Review*, *57*(3), 321-363.

Li, J., & Willett, P. (2009, November). ArticleRank: a PageRank-based alternative to numbers of citations for analysing citation networks. In *Aslib Proceedings*. Emerald Group Publishing Limited.

Manaskasemsak, B., & Rungsawang, A. (2005, July). An efficient partition-based parallel PageRank algorithm. In *11th International Conference on Parallel and Distributed Systems (ICPADS'05)* (Vol. 1, pp. 257-263). IEEE.

Needham, M., & Hodler, A. E. (2018). A comprehensive guide to graph algorithms in neo4j. *Neo4j. com*. https://neo4j.com/product/graph-data-science/?ref=pdf-ebook-graph-algo

Raghavan, U. N., Albert, R., & Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, *76*(3), 036106.

Schank, T., & Wagner, D. (2005, May). Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms* (pp. 606-609). Springer, Berlin, Heidelberg.

https://neo4j.com/docs/graph-data-science/current/