Community Detection Algorithms in Neo4j
Arsal Munawar

The Louvain Algorithm

This algorithm detects communities within a large network. Its objective function is to maximize a modulatory score for each community, where modularity is the density of links within a region. This algorithm has two phases. First, the algorithm dedicates a community to each node, so each node is its own community and there are as many communities as nodes. Second, the algorithm removes a particular node from its community and places it in the communities of one of its neighbors, one by one, and each time the change in molarity is measured. When moving a node to a neighboring community, the community with the greatest increase in modularity will retain that node. If there are multiple communities who have the same increase in modularity, then the tie breaker is decided arbitrarily. This second step of assigning nodes to neighboring communities occurs until no possible increase in modularity can occur. The links between the communities have weights which are equal to the sum of the links between the original nodes of the two communities, This concludes the first phase. During the second phase, the reassigned nodes in their newly formed communities are now combined to create one node per community. If after phase one there were 10 communities in the network, then each of the 10 communities are now nodes, so 10 nodes in the network. Now, the phase one process is applied to the 10 nodes, and then phase two is applied to the newly formed communities from those 10 nodes. The repetition of phases one and two occur until no node reassignments or no new communities can form in phase 1.

An example of this algorithm in action is an investor using it to determine which areas of the market to invest in. If an individual has invested in 15 entities, this algorithm will create communities out of those entities and help visualize commonalities and differences between them. It will help him make decisions such as where he should invest further to diversify his risk, or which areas of his current investments such as technology, pharmacy, etc, are successful.

One of the main issues with this process is what is called the resolution limit. Two distinct communities may be grouped together simply because there is one node which is related to both. The algorithm does not stop at what seems like obvious and unique communities, rather it merges them with other larger communities.

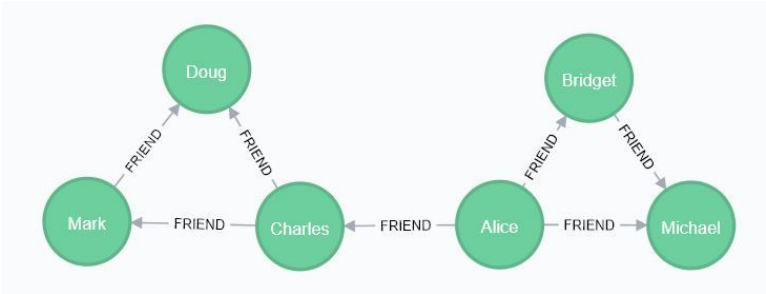
Below is the script to create the nodes and relationships, to call the algorithm, and to return the results of the computation. In the ‘CALL’ statement, it can be seen that whichever community a node is assigned to will be written as a property of that node under the name “community”. The algorithm can also take into account the weight of each relation, and in this instance all relations are given a default value of 1.

```
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);

CALL algo.louvain.stream('User', 'FRIEND', {write:true,
writeProperty:'community', weightProperty:'weight', defaultValue:1.0})
YIELD nodeId, community
MATCH (user:User) WHERE id(user) = nodeId
RETURN user.id AS user, community
ORDER BY community;
```

Below is the graph of the nodes along with the results of the community assignments. The algorithm has placed doug, mark, and charles in one community, and alice, bridget, and michael in the other community. If the algorithm was run again, no movements of nodes from one community to the other would result in an increase in modularity.



user	community
"Charles"	4
"Doug"	4
"Mark"	4
"Alice"	5
"Bridget"	5
"Michael"	5

The Label Propagation Algorithm

This is another algorithm for detecting communities in a network. It requires only the network structure to operate and no prior knowledge of the nodes or any objective function. The algorithm functions by first assigning unique labels to all nodes in the network. Each nodes label is then compared to the labels of its neighbors, and the node will adapt the most common label

from amongst its neighbors. The algorithm iterates until all nodes in the network have the same labels as their neighbors. The program allows labels to propagate throughout a network, but not so easily. Densely connected areas will quickly adapt one label, but any label will have trouble crossing a sparsely connected area. Thus, a particular label will be prevalent in a dense area, but will not be in another. The adapted labels will identify various communities in a network.

For example, If John mapped out all of his friends, this algorithm would assemble them into groups. John has friends through family, college, high school, work, and the gym. All of these friends would be mapped out along with all relations between them. The algorithm does not need to know which category the friends come from or the relations between them. It will simply assign a label to each node and the label will propagate. Those friends who are from college will naturally be grouped with the other college friends by virtue of the algorithm. It will also help determine which one of his friends from a particular group is actually more closely linked to a different group.

This algorithm has two weaknesses. First, it will cause many specific or descriptive labels to be lost, and the nodes will only adapt more general labels. For example, there could be a community of people who watch movies, but more specifically they may watch english, indian, or anime movies. These specific genres would be lost in this process and they will be referred to as only movie watchers. Second, this algorithm will result in different community structures to appear if run multiple times on the same graph because a label could propagate slightly differently with each trial.

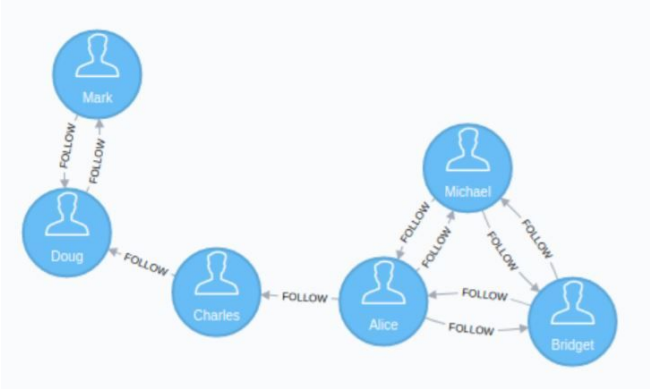
Below is the script to create the nodes and relationships, to call the algorithm, and to return the results of the computation. In the ‘CALL’ statement, the algorithm can take into account the direction of each relation. After creating each node, a ‘SET’ statement is placed in order to assign a label to each node.

```
MERGE (nAlice:User {id:'Alice'}) SET nAlice.seed_label=1
MERGE (nBridget:User {id:'Bridget'}) SET nBridget.seed_label=2
MERGE (nCharles:User {id:'Charles'}) SET nCharles.seed_label=3
MERGE (nDoug:User {id:'Doug'}) SET nDoug.seed_label=4
MERGE (nMark:User {id:'Mark'}) SET nMark.seed_label=5
MERGE (nMichael:User {id:'Michael'}) SET nMichael.seed_label=6
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);

CALL algo.labelPropagation.stream("User", "FOLLOW",
  {direction: "BOTH", iterations: 10})
YIELD nodeId, label
MATCH (user:User) WHERE id(user) = nodeId
RETURN user.id AS user, label
ORDER BY label;
```

Below is the graph of the nodes along with the results of the community assignments. The algorithm has given Doug and Mark the label 4 which is one community, and the other nodes the label 5 which is a different community. In theory, Michael, Bridget, and Alice will all end up sharing the same label. Alice will consult her neighbors and see that Michael and Bridget have adopted one label and will side with them. However when Charles must decide, it will be between Alice and Doug and this will be decided arbitrarily. The same goes for Doug. This demonstrates that a label which is popular in one area (Michael, Bridget, Alice) will most likely not propagate through a sparse area.



user	label
"Doug"	4
"Mark"	4
"Alice"	5
"Bridget"	5
"Charles"	5
"Michael"	5

The Connected Components Algorithm

This algorithm find sets of connected nodes in a network, and is also called ‘Union Find’. All nodes are treated as directionless, so simply a path between two nodes is considered connected. If there is a pair of nodes which are not directly connected but have many nodes in between them, then as long as there is some path to travel from one node to the other then they are considered connected and a part of the same set. The algorithm finds these connected sets by two different search methods, the breadth-first method and the depth-first method. The

breadth-first method searches the network by level. It starts off by selecting a single node and exploring all of its direct neighbors, then exploring its neighbors' neighbors while flagging every visited node along the way so as to not visit the same node twice. If the initial node is level 1 and its neighbors are level 2, then it searches for all nodes in level 2, then moves on to level 3 and searches for all nodes in level 3 before moving on to the next level. The depth-first method performs in an opposite way. It attempts to reach the highest possible level from the initial node before backtracking. It will start at the initial node level 1, then move on to level 2, and will then proceed to level 3 even if there are more unvisited nodes in level 2. It will continue to progress to newer levels until it cannot progress any further. level n, It will then backtrack to level n-1 and search all neighbors, then it will backtrack to level n-2 and so on until it reaches level 1. This algorithm is commonly applied on graphs before any other algorithms, as it determines if a graph is fully connected. Otherwise, errors and anomalies could be present in algorithm results and the user may not know that a disconnected graph is the cause.

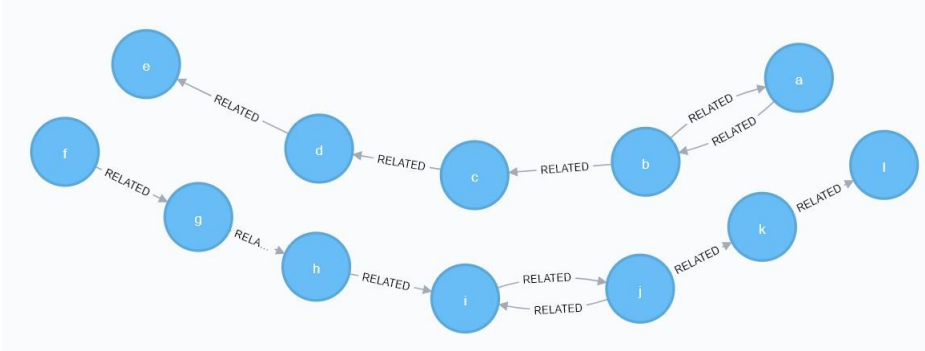
Below is the script and query to call the algorithm.

```
MERGE (a:node {name:"a"})
MERGE (b:node {name:"b"})
MERGE (c:node {name:"c"})
MERGE (d:node {name:"d"})
MERGE (e:node {name:"e"})
MERGE (f:node {name:"f"})
MERGE (g:node {name:"g"})
MERGE (h:node {name:"h"})
MERGE (i:node {name:"i"})
MERGE (j:node {name:"j"})
MERGE (k:node {name:"k"})
MERGE (l:node {name:"l"})
MERGE (a)-[:RELATED]->(b)
MERGE (b)-[:RELATED]->(c)
MERGE (c)-[:RELATED]->(d)
MERGE (d)-[:RELATED]->(e)
MERGE (f)-[:RELATED]->(g)
MERGE (g)-[:RELATED]->(h)
MERGE (h)-[:RELATED]->(i)
MERGE (i)-[:RELATED]->(j)
MERGE (j)-[:RELATED]->(k)
MERGE (k)-[:RELATED]->(l)
MERGE (a)<-[:RELATED]-(b)
MERGE (i)<-[:RELATED]-(j)

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

```
MATCH (u:node) WHERE id(u) = nodeId
RETURN u.name AS node, setId
ORDER BY setId
```

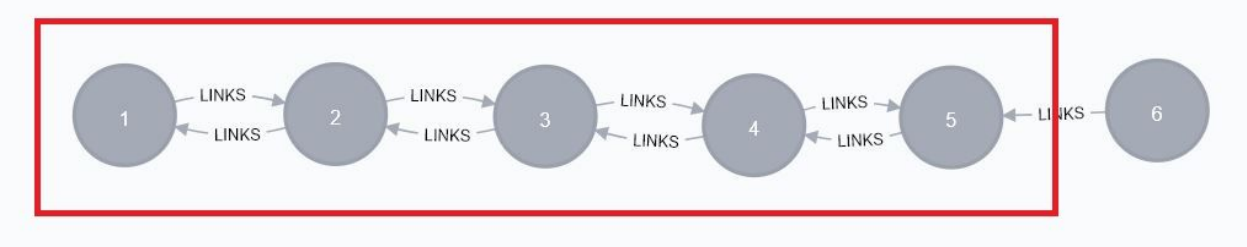
Here is the graph and results of the algorithm. Nodes a through e have been assigned one community, while the other nodes are another community. It can be seen from the results that the direction of the relations do not impact the connectivity of nodes.



node	setId
"a"	4
"b"	4
"c"	4
"d"	4
"e"	4
"i"	9
"j"	9
"k"	9
"l"	9
"f"	9
"g"	9
"h"	9

The Strongly Connected Components Algorithm

This algorithm is used for the same purpose as the Connected Components algorithm, except this algorithm requires that any relationship between nodes be bidirectional. If there are 6 nodes in a row such that the first five are bidirectional, and the relation between node 5 and 6 is unidirectional, then nodes 1 through 5 will be a part of a strongly connected set but node 6 will not be, as illustrated below.



A possible use for either of the connected components algorithms is in illuminating hierarchy within an organization. For example, BP Gas stations have owners, but each owner must make sure their stations have met the company standards in appearance and product quality. If a certain gas station is not inspected or is not visited by any higher employee or manager, then the station will eventually be run based on the standards of the owner. This algorithm can show that the hierarchy ends at the owner level, and that anyone beneath the owner of a station is within a connected set. The algorithm can than be used to determine that the cause of a certain gas station location not doing well could be due to a lack of oversight.

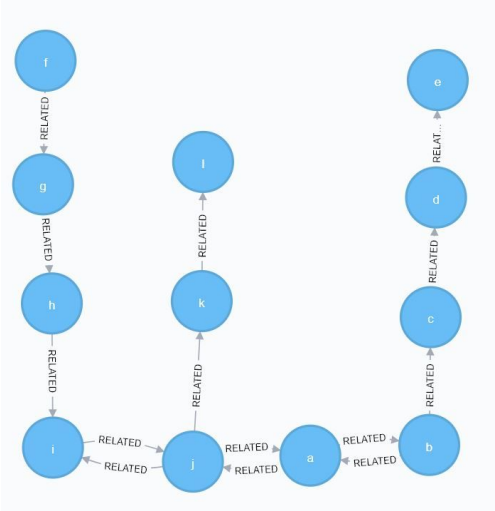
Ultimately both connected components algorithms are useful, but if one is specifically looking for bidirectional relations then the strongly connected algorithm is appropriate. To my knowledge, neither have any constraints as they are used visualize the graph structure.

Below is the script and query to call the SCC algorithm. It is a slightly altered form of the script for the previous algorithm.

```
MERGE (a:node {name:"a"})
MERGE (b:node {name:"b"})
MERGE (c:node {name:"c"})
MERGE (d:node {name:"d"})
MERGE (e:node {name:"e"})
MERGE (f:node {name:"f"})
MERGE (g:node {name:"g"})
MERGE (h:node {name:"h"})
MERGE (i:node {name:"i"})
MERGE (j:node {name:"j"})
MERGE (k:node {name:"k"})
MERGE (l:node {name:"l"})
MERGE (a)-[:RELATED]->(b)
MERGE (b)-[:RELATED]->(c)
MERGE (c)-[:RELATED]->(d)
MERGE (d)-[:RELATED]->(e)
MERGE (f)-[:RELATED]->(g)
MERGE (g)-[:RELATED]->(h)
MERGE (h)-[:RELATED]->(i)
MERGE (i)-[:RELATED]->(j)
MERGE (j)-[:RELATED]->(k)
MERGE (k)-[:RELATED]->(l)
MERGE (a)<-[:RELATED]-(b)
MERGE (i)<-[:RELATED]-(j)
MERGE (a)-[:RELATED]->(j)
MERGE (j)-[:RELATED]->(a)
```

```
CALL algo.scc.stream('*', '*', {})
YIELD nodeId,partition
MATCH (u:node) WHERE id(u) = nodeId
RETURN u.name AS node, partition
ORDER BY partition
```

Here are the results of the algorithm. It can be see that the only strongly connected nodes are those with bidirectional relations (a,b,i,j), and they are all assigned one community. All other nodes have only one-way relations, and thus are assigned their own communities.



node	partition
"a"	0
"b"	0
"i"	0
"j"	0
"c"	2
"d"	3
"e"	4
"f"	5
"g"	6
"h"	7
"k"	10
"l"	11

The Triangle Counting/Clustering Coefficient Algorithm

This algorithm measures the number of “triangles” that each node is a part of. A triangle is when a node is connected to two other nodes which are also connected to each other, creating a triangle of relations. In this algorithm, directionality does not matter. This algorithm also measures the clustering coefficient, which has two types. The local clustering coefficient is the

probability that the neighbors of a particular node are also connected to each other. The global clustering coefficient is the normalized sum of the local clustering coefficients. Clustering coefficient is the measure of the degree to which nodes tend to cluster together.

The local clustering coefficient is found using the following formula:

$$\frac{2N}{D(D-1)}$$

Where D is the degree (number of neighbors a node has), and N is the number of links between the neighbors of the node. The global clustering coefficient is the average of all the local clustering coefficients of every node in the network.

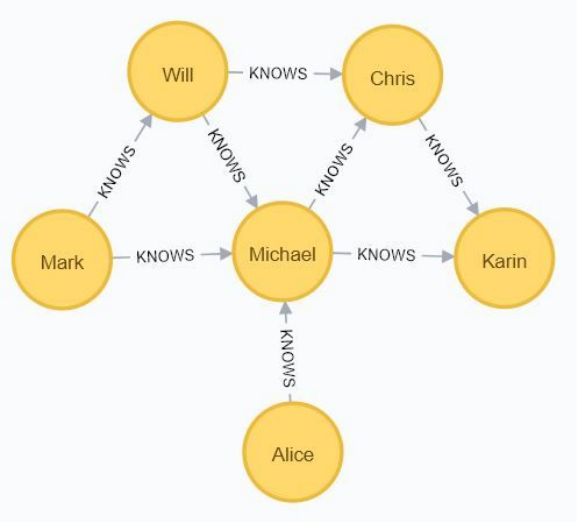
This algorithm is useful in determining how cohesive a network or community is. In soccer, it is very important that the manager builds cohesiveness amongst the players. It is common for players from different parts of the world who speak different languages to be brought onto the same team. A team based in England, may bring a Brazilian player onto their team. If this player who is from a different country and does not speak english has no connection with the rest of the team, then he will be represented as a single node with no relation. However if the Brazilian has a previous connection with another player due to playing on the same team, or if there are other players who speak Portuguese, then this can create a triangle of cohesiveness. In real life this is actually an issue. A manager will not bring a new player who speaks a different language unless there is another player already on the team who speaks that language. This algorithm can be applied to determine if a potential node will make cohesive connections not just with two other nodes, but many more. If the addition of a node results in a good clustering coefficient, then the addition can be deemed worthwhile.

Below is a script to create data and call the algorithm.

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

```
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.name AS nodeA, b.name AS nodeB, c.name AS nodeC
```

Here is the graph ad the results. The script will find all triangles in the network and assign each node in a triangle a designation of nodeA, nodeB, or nodeC. In the results, each node that is given a designation of nodeA is in the first column, and so on. Each row represents a triangle. Notice that alice is not even mentioned in the results since she is not a part of any triangle.



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

The clustering coefficient of each node can also be found using the query below. Michael is a part of more triangles than anyone, but Karin and Mark have the greatest coefficients because all of their neighbors know each other. Will, Chris, Michael, and Alice all have neighbors who do not know each other and hence are not as “clustered”. Mark has 2 neighbors, and there is one connection between the two neighbors. By the above mentioned formula, his clustering coefficient will be $\frac{2*1}{2(2-1)} = 1$.

```
CALL algo.triangleCount.stream('Person', 'KNOWS', {concurrency:4})
YIELD nodeId, triangles, coefficient
MATCH (p:Person) WHERE id(p) = nodeId
RETURN p.name AS name, triangles, coefficient
```

ORDER BY coefficient DESC

name	triangles	coefficient
"Karin"	1	1.0
"Mark"	1	1.0
"Chris"	2	0.6666666666666666
"Will"	2	0.6666666666666666
"Michael"	3	0.3
"Alice"	0	0.0