Path-finding Algorithms in Neo4j
Arsal Munawar

The minimum weight spanning tree algorithm

This algorithm starts at a particular node, finds every possible reachable node in the network, and then determines the route that holds the least weight to each reachable node. The algorithm operates as follows. An initial node is selected and added to the path. This initial node is the only node on the tree thus far. All of the relations going out of that node are evaluated, and the relation with the lowest weight is then connected to the initial tree, which now has two nodes. The process of evaluating and selecting the minimal-weight relation from the most recent node continues until there are no more nodes left. While doing this, the algorithm also ensures that any new node that is added to the tree does not form a cycle in the tree. In other words, if there is a relation between two nodes and those two nodes are already in the tree in other relations, then that relation will not be added to the tree or it will form a cycle in the tree.

The major constraint of this algorithm is that it is only useful if the relations have assigned and varying weights. Otherwise, every possible route would be a minimum spanning tree.

An example of this algorithm in action is in sea cruises routes. Typically a cruise ship will visit several places while reaching its destination. If a ship is traveling from Maine to Florida, it will stop along many cities on the east coast. This algorithm would help determine which cities along the path are closest, that way the ship would spend the least amount of time on the water and more time in coastal cities. The occupants would get their money's worth by seeing visiting many cities while en route to their destination, as opposed to going straight from Maine to Florida.

The following script will create a graph with weighted links and run the algorithm. The algorithm will find the minimum spanning tree starting from node A. The 'WITH' term will take the output of the MATCH statement and use it as input for the 'AS' statement. The 'UNWIND' clause will take the contents of the list 'rels', create a new column called 'rel', and create a new row in 'rel' for each entry of 'rels'. The code will return the minimum spanning tree in a table and graph format, with each row of the table showing the starting node, ending node, and cost to traverse between the nodes.

```
MERGE (a:node {id:"A"})
MERGE (b:node {id:"B"})
MERGE (c:node {id:"C"})
MERGE (d:node {id:"D"})
MERGE (e:node {id:"E"})
MERGE (a)-[:LINK {cost:1}]->(b)
MERGE (a)-[:LINK {cost:10}]->(c)
MERGE (b)-[:LINK {cost:1}]->(d)
MERGE (c)-[:LINK {cost:2}]->(d)
```

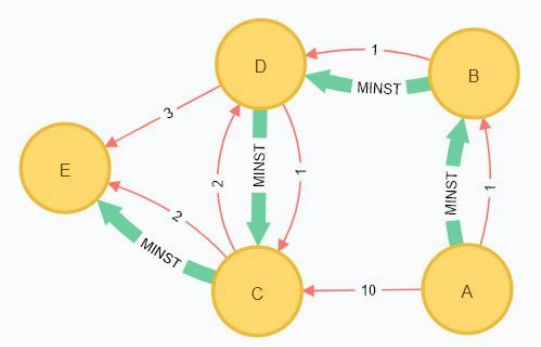
```
MERGE (d)-[:LINK {cost:1}]->(c)
MERGE (c)-[:LINK {cost:2}]->(e)
MERGE (d)-[:LINK {cost:3}]->(e)

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

MATCH path = (n:node {id:"A"})-[:MINST*]-()
WITH relationships(path) AS rels
UNWIND rels AS rel
WITH DISTINCT rel AS rel
RETURN startNode(rel).id AS source, endNode(rel).id AS destination, rel.cost
AS cost

match (n)
return n
```

Below are the results of the query. The graph shows both the cost between nodes (red lines) and the minimum spanning tree (green lines). It can be seen that the path starts at A and travels to B at a cost of 1, as opposed to C at a cost of 10. From B the treen spans to D at a cost of 1, then from D to C at a cost of 1, then from C to E at a cost of 2. The same procedure can also be applied to find the maximum spanning tree.



source	destination	cost
"A"	"B"	1.0
"B"	"D"	1.0
"D"	"C"	1.0
"C"	"E"	2.0

The Shortest Path Algorithm

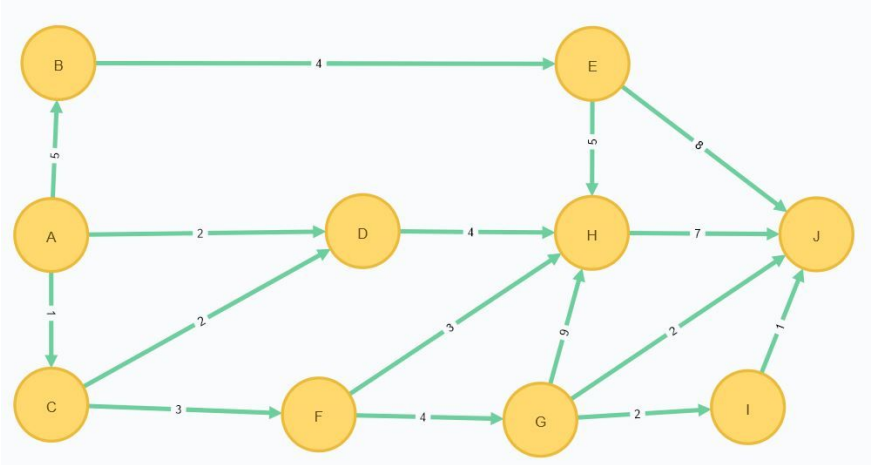
This is a very straightforward algorithm which calculates the shortest weighted path between a source node and every other node in the network. This is also called Dijkstra's algorithm. The algorithm works as follows. An initial node is selected and all other nodes are marked as unvisited. The distances from the initial node to all of its neighbors are calculated, and the node with the shortest path is selected. Just with the initial node, the distances to the neighbors of this node are also calculated, and the overall distance from the initial node to the neighbors are the resulting distance. The algorithm then considers the shortest path in the network thus far, and selects the node which ends at the shortest path. It will then consider the nodes' neighbors and calculate distances. If a node previously had a distance from the initial node, but a shorter distance was found from a different path, then the distance of that node will be updated to the smaller value. This process is done until all nodes have been visited and all distances calculated, and in the end only the shortest path to each node in the network is displayed.

A very obvious use of this algorithm is in determining the shortest distance to travel somewhere. There are many possible paths to take to travel from Baltimore to DC. At different times of the day and days of the week, one path may be quicker than another. The various roads and highways can be mapped out, with the "weight" of the path being the distance in miles and level of traffic on a particular road. If a path is short and has no traffic, it will have the least weight. If a path is long and has more traffic, it will have the greatest weight. If all roads and highways between Baltimore and DC are mapped in such a way, the algorithm can determine which route will have the lowest weight. A constraint of the algorithm is that negative weights are not supported. Adding a relationship to a path will not make a path shorter.

The script below shows use of this algorithm. A network of nodes with weighted paths are created, shown below.

```
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 (a)-[:PATH {cost:5}]->(b)
MERGE (a)-[:PATH {cost:1}]->(c)
MERGE (b)-[:PATH {cost:4}]->(e)
MERGE (e)-[:PATH {cost:8}]->(j)
MERGE (e)-[:PATH {cost:5}]->(h)
```

```
MERGE (a)-[:PATH {cost:2}]->(d)
MERGE (d)-[:PATH {cost:4}]->(h)
MERGE (h)-[:PATH {cost:7}]->(j)
MERGE (c)-[:PATH {cost:2}]->(d)
MERGE (c)-[:PATH {cost:3}]->(f)
MERGE (f)-[:PATH {cost:4}]->(g)
MERGE (f)-[:PATH {cost:3}]->(h)
MERGE (g)-[:PATH {cost:9}]->(h)
MERGE (g)-[:PATH {cost:2}]->(j)
MERGE (g)-[:PATH {cost:2}]->(i)
MERGE (i)-[:PATH {cost:1}]->(j)
```



The shortest distance from node A to node J is calculating using the query below. First, the node A is found and given the name “start”, and the node J is found and given the name “end”. These names are then used in the algorithm input.

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

The results are shown below. The shortest path from A to J is through C, F, and J. The cost column shows the total column to reach that particular node. It costs 1 to get to C, 1+3 to get to F, 1+3+4 to get to G, and a total of 1+3+4+2=10 to get to J.

name	cost
"A"	0.0
"C"	1.0
"F"	4.0
"G"	8.0

"J"	10.0

The Single Source Shortest Path Algorithm

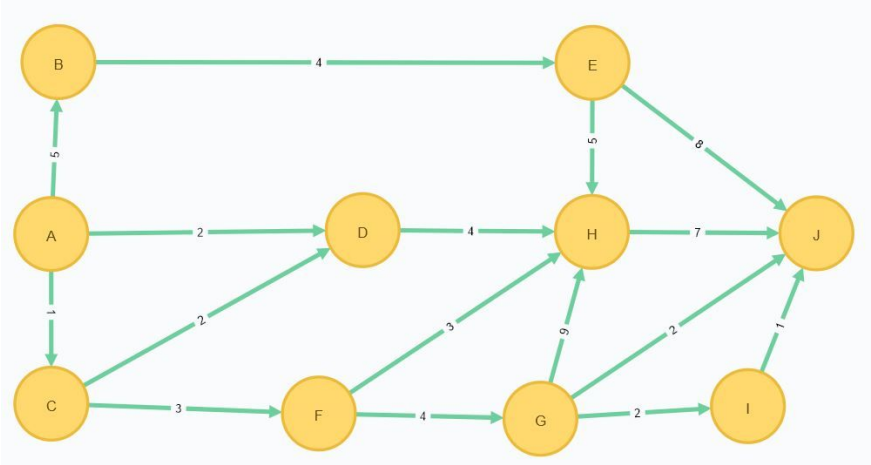
This algorithm calculates the shortest weighted path from a node to all other nodes. This is different from the previous algorithm in that the previous one calculates the shortest weighted path from designated starting and ending nodes, as opposed to all nodes. This algorithm operates the same way as the shortest path algorithm. The steps to determine the shortest path to all other nodes is the same, except in this case the output is the final cost to reach all other nodes as opposed to the cost to travel from initial to ending node.

Once again this algorithm can be extended to determining travel times. If a person needs to stop at a gas station, this algorithm will factor in all the gas stations in the area and calculate the shortest possible distance to each of them. Perhaps this is how the search results works when you search for a particular establishment on Google Maps. The major constraint with this algorithm is that is cannot accept negative weights.

Using the same data as above, the shortest path is calculate from A to all other nodes. The script is below.

```
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 (a)-[:PATH {cost:5}]->(b)
MERGE (a)-[:PATH {cost:1}]->(c)
MERGE (b)-[:PATH {cost:4}]->(e)
MERGE (e)-[:PATH {cost:8}]->(j)
MERGE (e)-[:PATH {cost:5}]->(h)
MERGE (a)-[:PATH {cost:2}]->(d)
MERGE (d)-[:PATH {cost:4}]->(h)
MERGE (h)-[:PATH {cost:7}]->(j)
MERGE (c)-[:PATH {cost:2}]->(d)
MERGE (c)-[:PATH {cost:3}]->(f)
MERGE (f)-[:PATH {cost:4}]->(g)
MERGE (f)-[:PATH {cost:3}]->(h)
MERGE (g)-[:PATH {cost:9}]->(h)
MERGE (g)-[:PATH {cost:2}]->(j)
MERGE (g)-[:PATH {cost:2}]->(i)
```

```
MERGE (i)-[:PATH {cost:1}]->(j)
```



The algorithm is called, using node A as the starting point.

```
MATCH (n:node {name:'A'})
CALL algo.shortestPath.deltaStepping.stream(n, 'cost', 3.0)
YIELD nodeId, distance
MATCH (destination) WHERE id(destination) = nodeId
RETURN destination.name AS destination, distance
```

The results are below. A is the origin, so the distance is 0. The total distance to J is 10 which was also proved in the last algorithm. The algorithm also supports bidirectional shortest path searches.

destination	distance
"A"	0.0
"B"	5.0
"C"	1.0
"D"	2.0
"E"	9.0
"F"	4.0
"G"	8.0
"H"	6.0
"I"	10.0
"J"	10.0

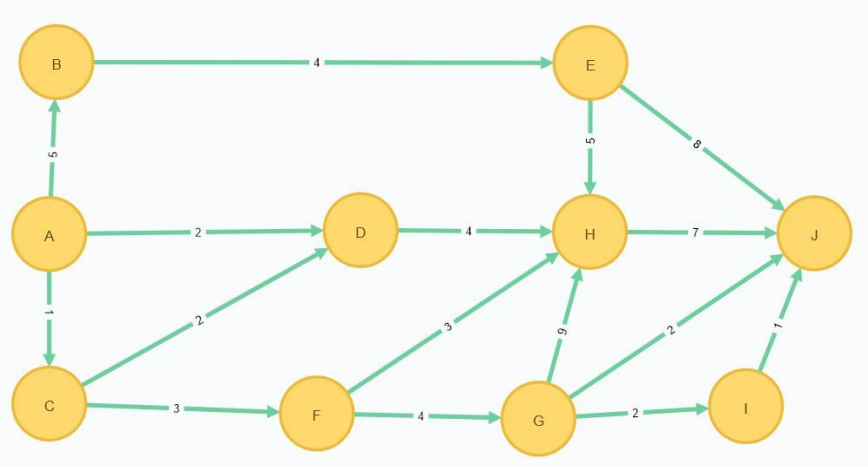
The All Pairs Shortest Path Algorithm

This algorithm calculates the shortest path between all pairs of nodes, not just between the starting node and others. If no path exists between a pair of nodes, the value ‘infinity’ will be returned. This algorithm could be used in services such as Amazon Fulfillment. Amazon has distribution centers all over the United States, so when a customer purchases an item it will be

delivered from one of these centers, or sent to a post office to be delivered. Amazon may use this algorithm to determine the placement of a fulfillment center that would optimize the proximity to both residential areas and post offices. That way, the fulfillment center is closest to all possible delivery hotspots and all post offices. Amazon may also use this algorithm to determine what products will be held in a distribution center. For example, if in a certain area a specific type of product is ordered often, then Amazon will stock more of this product in the closest fulfillment center to reduce shipping times.

Below is the script and graph to demonstrate this algorithm, the same dataset that has been used in the previous two algorithms.

```
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 (a)-[:PATH {cost:5}]->(b)
MERGE (a)-[:PATH {cost:1}]->(c)
MERGE (b)-[:PATH {cost:4}]->(e)
MERGE (e)-[:PATH {cost:8}]->(j)
MERGE (e)-[:PATH {cost:5}]->(h)
MERGE (a)-[:PATH {cost:2}]->(d)
MERGE (d)-[:PATH {cost:4}]->(h)
MERGE (h)-[:PATH {cost:7}]->(j)
MERGE (c)-[:PATH {cost:2}]->(d)
MERGE (c)-[:PATH {cost:3}]->(f)
MERGE (f)-[:PATH {cost:4}]->(g)
MERGE (f)-[:PATH {cost:3}]->(h)
MERGE (g)-[:PATH {cost:9}]->(h)
MERGE (g)-[:PATH {cost:2}]->(j)
MERGE (g)-[:PATH {cost:2}]->(i)
MERGE (i)-[:PATH {cost:1}]->(j)
```



The query and data are shown below. It can be seen that the distribution of distances is fairly even. Some nodes are closer to some but nearer to others. However, three of the longest paths between any two nodes involve J, that is B to J, D to J, and A to J.

```
CALL algo.allShortestPaths.stream('cost',{nodeQuery:'node',defaultValue:1.0})
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE algo.isFinite(distance) = true
MATCH (source:node) WHERE id(source) = sourceNodeId
MATCH (target:node) WHERE id(target) = targetNodeId
WITH source, target, distance WHERE source <> target
RETURN source.name AS source, target.name AS target, distance
ORDER BY distance DESC
```

source	target	distance
"B"	"J"	12.0
"D"	"J"	11.0
"A"	"I"	10.0
"A"	"J"	10.0
"A"	"E"	9.0
"G"	"H"	9.0
"C"	"I"	9.0
"C"	"J"	9.0
"B"	"H"	9.0
"A"	"G"	8.0
"E"	"J"	8.0
"H"	"J"	7.0
"C"	"G"	7.0
"A"	"H"	6.0
"F"	"I"	6.0
"F"	"J"	6.0
"C"	"H"	6.0
"A"	"B"	5.0

"E"	"H"	5.0
"A"	"F"	4.0
"D"	"H"	4.0
"F"	"G"	4.0
"B"	"E"	4.0
"F"	"H"	3.0
"C"	"F"	3.0
"A"	"D"	2.0
"G"	"I"	2.0
"G"	"J"	2.0
"C"	"D"	2.0
"A"	"C"	1.0
"I"	"J"	1.0

The A* Algorithm

This algorithm is a variation of the Dijkstra algorithm in that it provides a more informed search and therefore can make improved suggestions. Typically the cost of a path is calculated by simply adding weights of the connections. In this case, the cost is split into two variables, g(n) and h(n). If a path starts at node 1, ends at node 3, but travels through node 2, then g(n) is the cost from node 1 to node 2, and h(n) is an estimated cost from node 2 to node 3. This estimation is done by an intelligent guess, which could be a computation of distances before the algorithm runs, or one of either the Manhattan, Diagonal, or Euclidean distance approximation formulas. This algorithm is often used in video games when an object in the game is traveling from point A to point B. For example in “tower defense” games where the user must create obstacles to defend territory, the computer enemy will use this algorithm to determine the shortest path to reach its goal. This algorithm also can be used when determining the distance between two locations on a map, if GPS coordinates are known. The algorithm will use Euclidean Distance measurement, which is the straight line measurement between two points in space, using the GPs coordinates. One major constraint however is that the A* algorithm will not always produce the shortest path, since it is guessing a path. This algorithm ma take less time to complete, but with a tradeoff in accuracy in some cases. Nonetheless, studies have shown it to be reliably accurate, and hence it is used widespread.

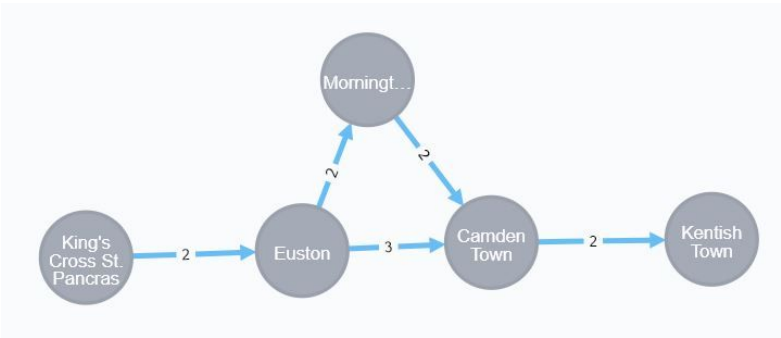
Below is the script and query to call this algorithm. The script is taken from the Neo4j documentation. GPS coordinates are set as properties of each node, but edges are still created between nodes. The algorithm will use the coordinates to determine the straight line distance between nodes, and then suggest that path which matches the shortest straight line distance and calculate that sum.

```
MERGE (a:Station{name:"King's Cross St. Pancras"})
SET a.latitude = 51.5308,a.longitude = -0.1238
MERGE (b:Station{name:"Euston"})
SET b.latitude = 51.5282, b.longitude = -0.1337
MERGE (c:Station{name:"Camden Town"})
```

```
SET c.latitude = 51.5392, c.longitude = -0.1426
MERGE (d:Station{name:"Mornington Crescent"})
SET d.latitude = 51.5342, d.longitude = -0.1387
MERGE (e:Station{name:"Kentish Town"})
SET e.latitude = 51.5507, e.longitude = -0.1402
MERGE (a)-[:CONNECTION{time:2}]->(b)
MERGE (b)-[:CONNECTION{time:3}]->(c)
MERGE (b)-[:CONNECTION{time:2}]->(d)
MERGE (d)-[:CONNECTION{time:2}]->(c)
MERGE (c)-[:CONNECTION{time:2}]->(e)

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

Below is the graph and the results. The algorithm suggests that the shortest path from Kings Cross to Kentish Town is through Euston and Camden Town at a total cost of 7, which indeed is the lowest cost path.



station	cost
"King's Cross St. Pancras"	0.0
"Euston"	2.0
"Camden Town"	5.0
"Kentish Town"	7.0

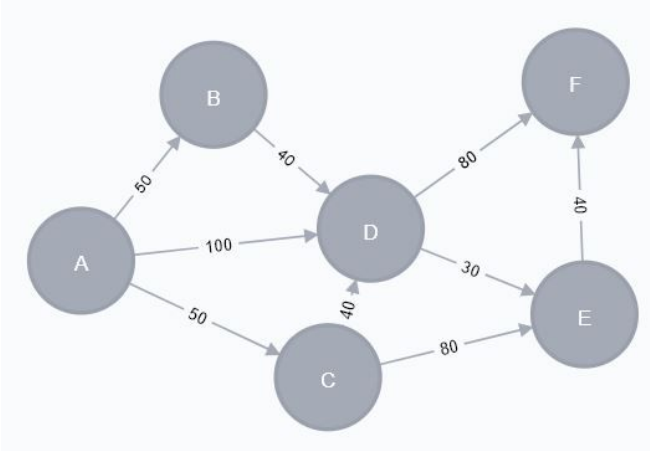
The Yen's k-shortest paths algorithm

This algorithm calculates the single source shortest path to a destination node using Dijkstra's search algorithm, then proceeds to calculate k-1 deviations of the shortest path, where k is a

user defined integer. The algorithm will find the shortest path from A to B, then find alternate routes. This is particularly useful in travel and maps. A software can provide alternate routes which may take a longer path or consume more time, but will allow a traveler to avoid traffic, road work, tolls, etc. This can also be used in a more abstract sense, when attempting to come to a conclusion about an idea. If a person is attempting to prove a point, he can use this algorithm to find alternate thought processes that contain different ideas but come to the same conclusion to prove his point. The drawback of this algorithm is that it does not support negative weights.

Here is sample data taken from the documentation, as well as the resulting network of nodes.

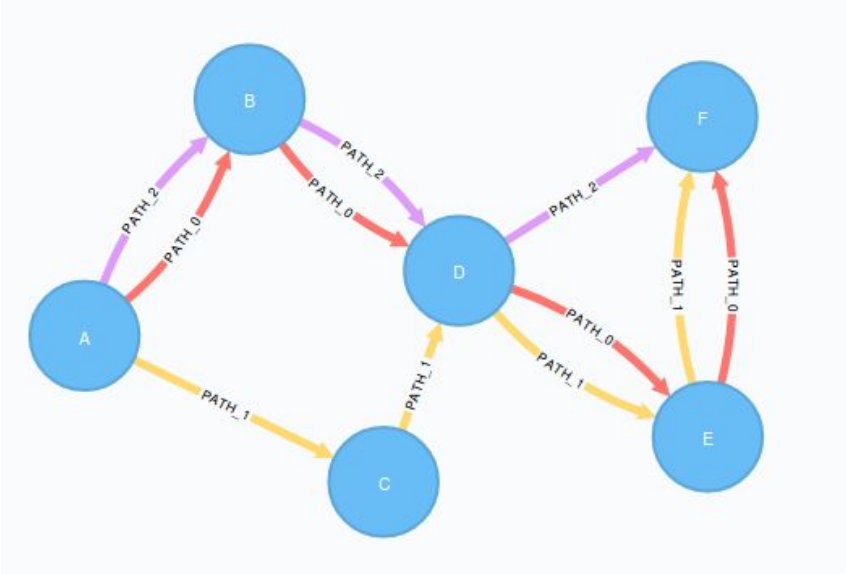
```
MERGE (a:Loc {name:'A'})
MERGE (b:Loc {name:'B'})
MERGE (c:Loc {name:'C'})
MERGE (d:Loc {name:'D'})
MERGE (e:Loc {name:'E'})
MERGE (f:Loc {name:'F'})
MERGE (a)-[:ROAD {cost:50}]->(b)
MERGE (a)-[:ROAD {cost:50}]->(c)
MERGE (a)-[:ROAD {cost:100}]->(d)
MERGE (b)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:80}]->(e)
MERGE (d)-[:ROAD {cost:30}]->(e)
MERGE (d)-[:ROAD {cost:80}]->(f)
MERGE (e)-[:ROAD {cost:40}]->(f);
```



Below is also the query to call the algorithm, as well as the graph displaying the potential paths. The algorithm finds the shortest path from A to F (path_0) to be through D, D, and E at a cost of

160. The alternate paths found are labeled path_1 and path_2 at costs of 160 and 180, respectively

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.kShortestPaths.stream(start, end, 3, 'cost' ,{})
YIELD index, nodeIds, path, costs
RETURN [node in algo.getNodesById(nodeIds) | node.name] AS places,
    costs,
    reduce(acc = 0.0, cost in costs | acc + cost) AS totalCost
```



The Random Walk Algorithm

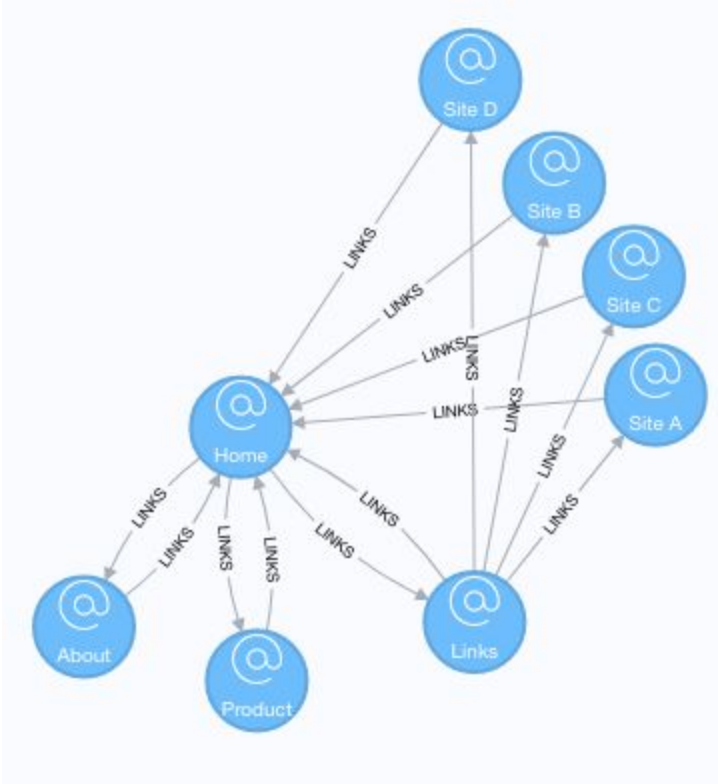
This algorithm starts at an initial node and randomly selects one of its neighbors to travel to, and continues traveling randomly while noting the path. An example of this is on the number line, where the walk starts at zero and moves +1 or -1 with each step. Random Walks can be used to detect communities in a network. If a random walk is iterated many times and it consistently returns the same nodes, then that is an indication of a community structure being present.

Random walks should not be used in graphs that only accept the ‘OUT’ direction because this increases a chance of a dead end, when the path cannot move forward because it has reached the end but also cannot go backwards because of the direction of the arrow. If this occurs, then the algorithm returns only the initial node as being a part of the path.

Below is a script to create data. This is a network of pages which link to each other.

```
MERGE (home:Page {name:'Home'})
MERGE (about:Page {name:'About'})
```

```
MERGE (product:Page {name:'Product'})
MERGE (links:Page {name:'Links'})
MERGE (a:Page {name:'Site A'})
MERGE (b:Page {name:'Site B'})
MERGE (c:Page {name:'Site C'})
MERGE (d:Page {name:'Site D'})
MERGE (home)-[:LINKS]->(about)
MERGE (about)-[:LINKS]->(home)
MERGE (product)-[:LINKS]->(home)
MERGE (home)-[:LINKS]->(product)
MERGE (links)-[:LINKS]->(home)
MERGE (home)-[:LINKS]->(links)
MERGE (links)-[:LINKS]->(a)
MERGE (a)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(b)
MERGE (b)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(c)
MERGE (c)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(d)
MERGE (d)-[:LINKS]->(home)
```



Below is the query to call the algorithm. It can be seen that the starting node is “Home”, the length of the walk will be 3, and only 1 path will be returned. By default, the algorithm will move

across a path regardless of direction. The results show that a random walk starting at Home will travel to Site C, then to Links, then to Site A.

```
MATCH (home:Page {name: "Home"})
CALL algo.randomWalk.stream(id(home), 3, 1)
YIELD nodeIds
UNWIND nodeIds AS nodeId
MATCH (n) WHERE id(n) = nodeId
RETURN n.name AS page
```

page
"Home"
"Site C"
"Links"
"Site A"