# Hands-On: Path Analytics in Neo4j With Cypher

In this activity, we're going to go perform Path Analytics using Cypher. Before starting, complete the following steps:

1. Start your Neo4j container and open it in https://localhost:7474/browser/.

2. If you started this activity just after finishing the *Hands-On: Basic Queries in Neo4j* activity, make sure to "clean the slate" in Neo4j, as you probably have the terrorist data currently loaded, and for this activity, we will start with a smaller dataset:

First, run this code:

```
MATCH (a)-[r]->() delete a,r;
MATCH (a) delete a
```

3. Load the test.csv data

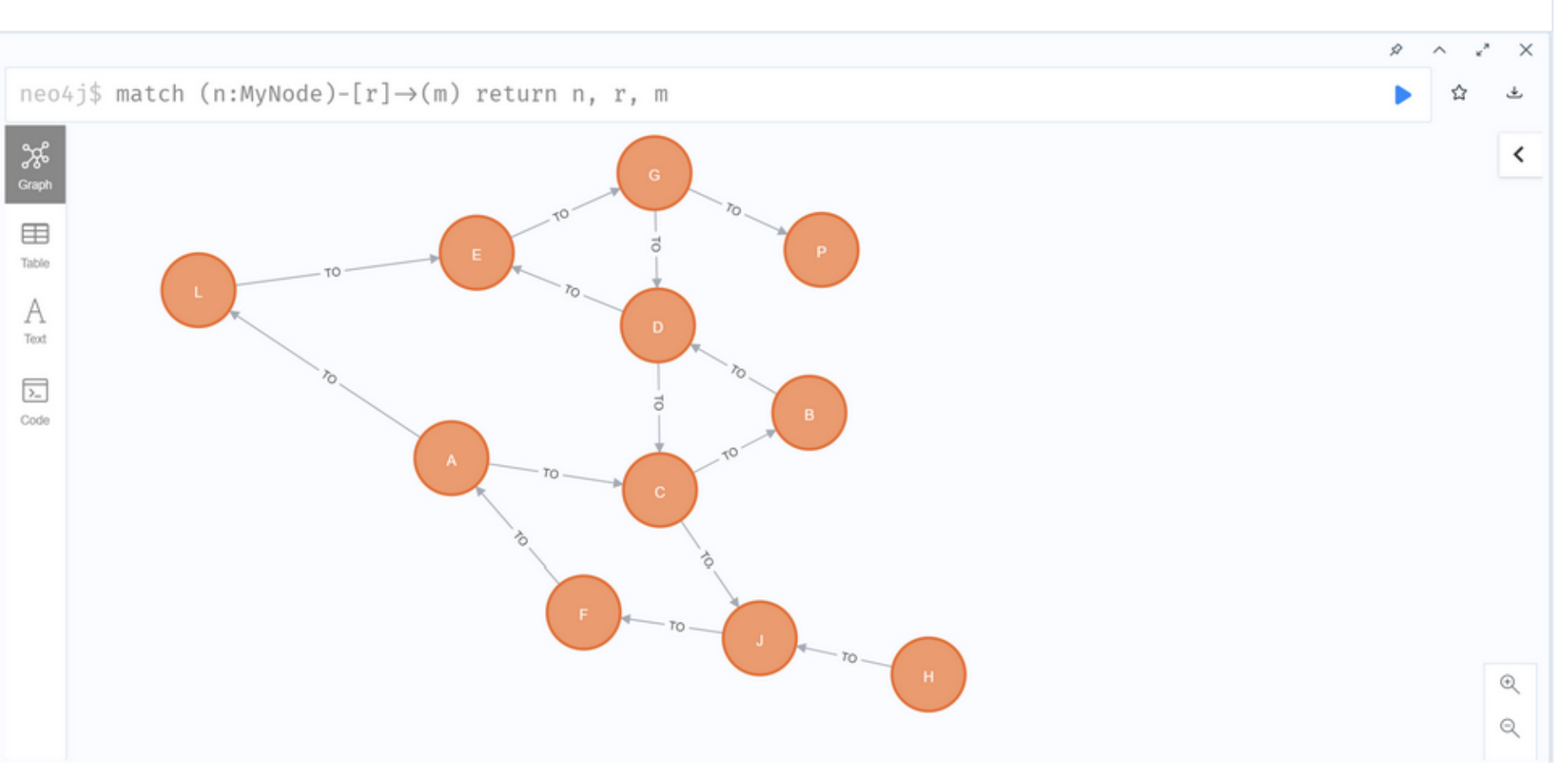
```
LOAD CSV WITH HEADERS FROM "file:///datasets/test.csv" AS line
MERGE (n:MyNode {Name: line.Source})
MERGE (m:MyNode {Name: line.Target})
MERGE (n) -[:TO {dist: toInteger(line.distance)}]-> (m)
```

**Path Analytics with CYPHER**

It's important to keep in mind that because we're working with paths, which are an official structure in graph networks, each one of the examples that we're going to show includes a new variable. In this case, we're using the letter *p* to represent for the actual path objects that we're going to be returning.

**Query 1. Viewing the graph.** Let's start by simply viewing our graph:

```
MATCH (n:MyNode)-[r]->(m)
RETURN n, r, m
```



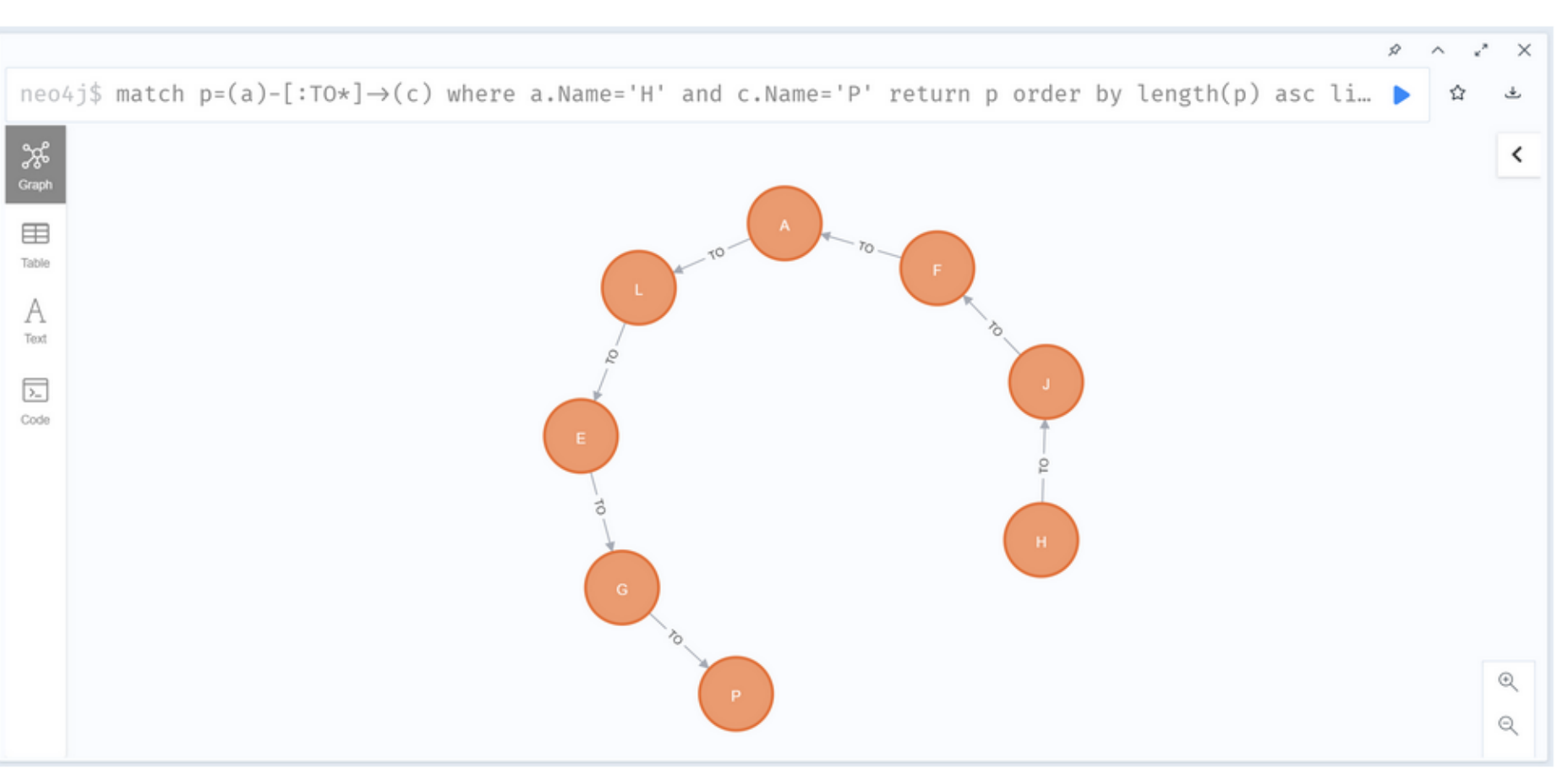
```
neo4j$ match (n:MyNode)-[r]→(m) return n, r, m
```

We're already familiar with this road network.

**Query 2. Finding paths between specific nodes.**

```
MATCH p=(a)-[:TO*]->(c)
WHERE a.Name='H' and c.Name='P'
RETURN p order by length(p) asc limit 1
```

We're going to find a path between the node named *H* and the node named *P.* To do this, we'll use the match command and we'll say match *p,* which is a variable we're using to represent our path, equals node *a* going through an edge to node *c*. We're using a star to represent an arbitrary number of edges in sequence between *a* and *c*, and we'll be returning all of those edges that are necessary to complete the path. Finally, we sort the different *p* by their length (the number of nodes involved) and we return a single path.



```
neo4j$ match p=(a)-[:TO*]→(c) where a.Name='H' and c.Name='P' return p order by length(p) asc li…
```

**Query 3. Finding the length between specific nodes.**

```
MATCH p=(a)-[:TO*]->(c)
WHERE a.Name='H' AND c.Name='P'
RETURN length(p) ORDER BY length(p) limit 1
```

```
neo4j$ match p=(a)-[:TO*]→(c) where a.Name='H' and c.Name='P' return length(p) order by length(p…
```

```
length(p)
7
```

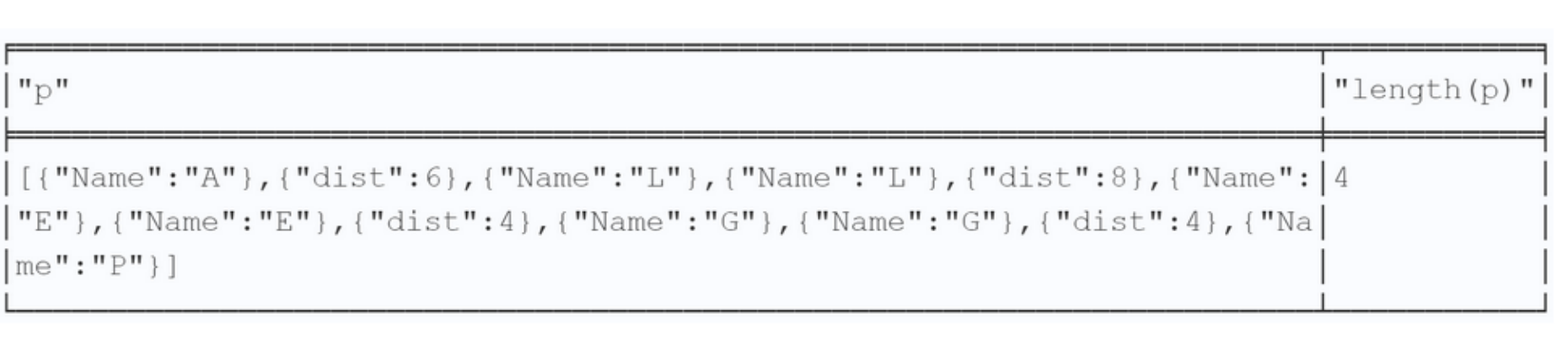
The length between those nodes is 7 as we visually saw in previous graph.

**Query 4. Finding a shortest path between specific nodes.**

```
MATCH p=shortestPath((a)-[:TO*]->(c))
WHERE a.Name='A' AND c.Name='P'
RETURN p, length(p) limit 1
```

Now we introduce a new function *shortestPath* to guarantee that we return only the shortest paths, this time between *A* and *P.* Again, we only return one result in case we have multiple.

Clicking on *Text*, will display the table representation of the path *p*, and the length which is equal to 4 (the number of relationships).



```
|"p"                                                                    |"length(p)"|
|[{"Name":"A"},{"dist":6},{"Name":"L"},{"Name":"L"},{"dist":8},{"Name":|4          |
|"E"},{"Name":"E"},{"dist":4},{"Name":"G"},{"Name":"G"},{"dist":4},{"Na|           |
|me":"P"}]                                                              |           |
```

**Query 5. All Shortest Paths.** We introduce a new command to return all of the potential shortest paths. We are also making an adjustment in line 3 to return all the potential paths in the form of an array.

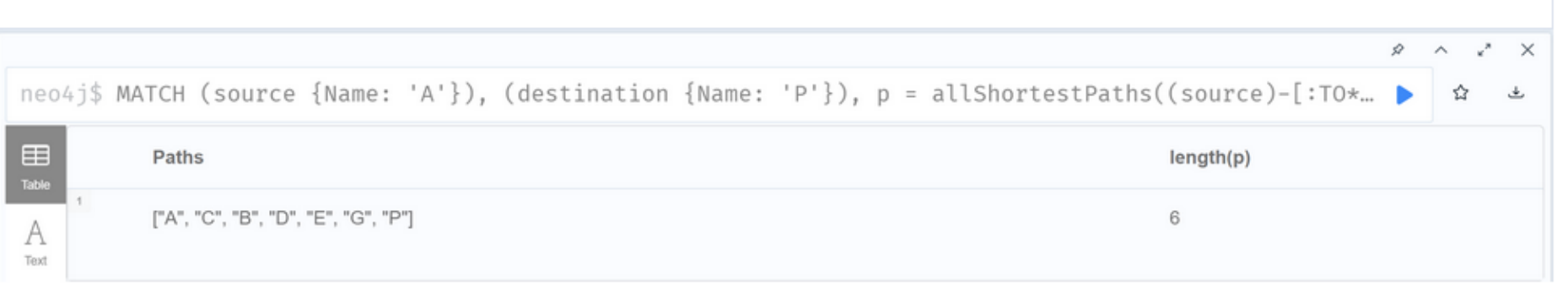
```
MATCH (source {Name: 'A'}), (destination {Name: 'P'}),
    path = allShortestPaths((source)-[:TO*]->(destination))
RETURN [node IN NODES(path) | node.Name] AS Paths
```

```
neo4j$ MATCH (source {Name: 'A'}), (destination {Name: 'P'}), path = allShortestPaths((source)-[:…
```

```
|"Paths"                  |
|["A","L","E","G","P"]|
```

**Query 6. All Shortest Paths with Path Conditions.** Now, in the third line we add a condition in which we ask to return the shortest paths where the number of relationships is at least higher than 5.

```
MATCH (source {Name: 'A'}), (destination {Name: 'P'}),
    p = allShortestPaths((source)-[:TO*]->(destination))
WHERE length(p) > 5
RETURN [n IN nodes(p) | n.Name] AS Paths, length(p)
```



```
neo4j$ MATCH (source {Name: 'A'}), (destination {Name: 'P'}), p = allShortestPaths((source)-[:TO*…
```

```
Paths                                         length(p)
["A", "C", "B", "D", "E", "G", "P"]           6
```

**Query 7. Diameter of the graph.** The definition of the diameter of the graph is the longest continuous path between two nodes in the graph. By using the shortest path command, but returning all possible shortest paths, we're actually going to get the longest path included in those results returned.

```
match (n:MyNode), (m:MyNode)
where n <> m
with n, m
match p=shortestPath((n)-[*]->(m))
return n.Name, m.Name, length(p)
order by length(p) desc limit 1
```

In this case our match command is matching all nodes of type *MyNode*. We'll assign those to the variable *n*. We're also matching the all nodes of type *MyNode* and assigning that to variable *m,* so these matches are the same. However, we want to place a constraint such that the nodes in *n* are not the same as the nodes in *m*, and then we want to find all of the shortest paths between unique nodes in *n* and *m,* to return the names of those nodes as well as the length of that resulting path.

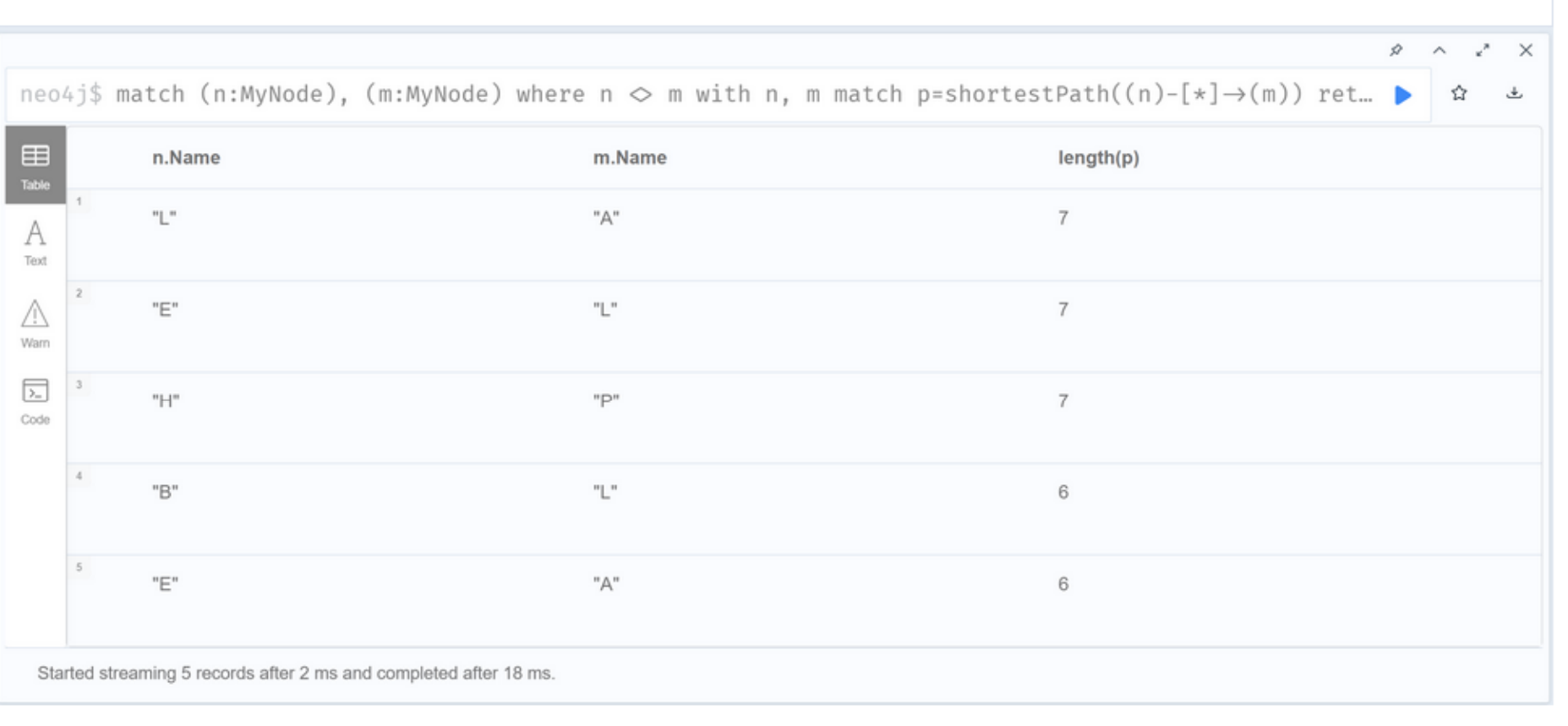
The trick is to use the command order by. If we order the resulting paths by their length in descending order, and only return 1, that path should actually be the longest path. And that's equal to the diameter of the graph.

```
neo4j$ match (n:MyNode), (m:MyNode) where n <> m with n, m match p=shortestPath((n)-[*]→(m)) ret…
```

```
n.Name            m.Name            length(p)
"E"               "L"               7
```

It is possible that more than one path qualifies as the diameter of the graph. Change the limit to 5 to inspect this:

```
MATCH (n:MyNode), (m:MyNode)
WHERE n <> m
WITH n, m
MATCH p=shortestPath((n)-[*]->(m))
RETURN n.Name, m.Name, length(p)
ORDER BY length(p) desc limit 5
```



```
neo4j$ match (n:MyNode), (m:MyNode) where n <> m with n, m match p=shortestPath((n)-[*]→(m)) ret…
```

```
n.Name            m.Name            length(p)
"L"               "A"               7
"E"               "L"               7
"H"               "P"               7
"B"               "L"               6
"E"               "A"               6
```

Started streaming 5 records after 2 ms and completed after 18 ms.

As we see, there are indeed 3 paths that qualify as the diameter of the graph.

Until now we've been calculating path length based on the number of hops between our beginning node and our end node. This is roughly equivalent to counting the number of towns between one town and another town. But it doesn't really get at the value that is usually of greatest importance to us, and that is the actual distance between one location and another location, which is found in the values that we've assigned to the edges between the nodes.

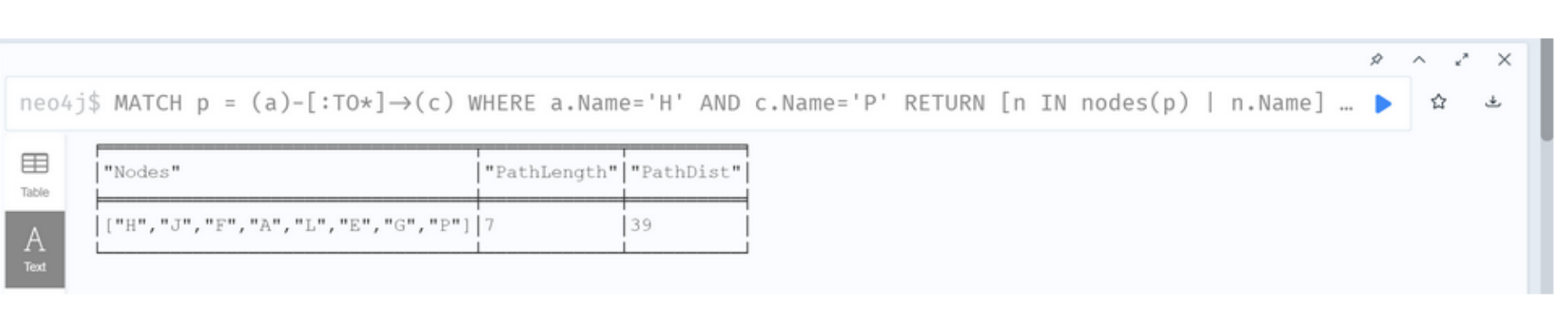
**Query 8. Extracting and computing with node and properties:**

```
MATCH p = (a)-[:TO*]->(c)
WHERE a.Name='H' AND c.Name='P'
RETURN [n IN nodes(p) | n.Name] AS Nodes, length(p) AS PathLength,
REDUCE(s = 0, e IN relationships(p) | s + toInteger(e.dist)) AS PathDist
ORDER BY PathDist asc LIMIT 1
```

We match a path between node *a* and node *c*, where the first node is *H*, and the second node is *P.* Then, we extract the names of the nodes and the path that is been returned, creating a listing of those names as well as a length of the path, all in a variable named *pathLength*.

We've added a third element to our *return* statement, and that is using the *reduce* statement. The purpose of the *reduce* statement is to take a set of values and reduce them down to a single value. In fourth line of code, we begin by setting a variable *s* equal to 0. Then, we define a variable *e*, which represents the set of relationships in a path that's returned, or in other words, the edges. We pass that into this variable *s*, and add to it the value of the distance that we've assigned to that edge. In conclusion, we're performing an aggregate calculation. and returning the final results to a variable called *pathDist*.

Finally, we limit our results to a single value, which is the shortest distance.



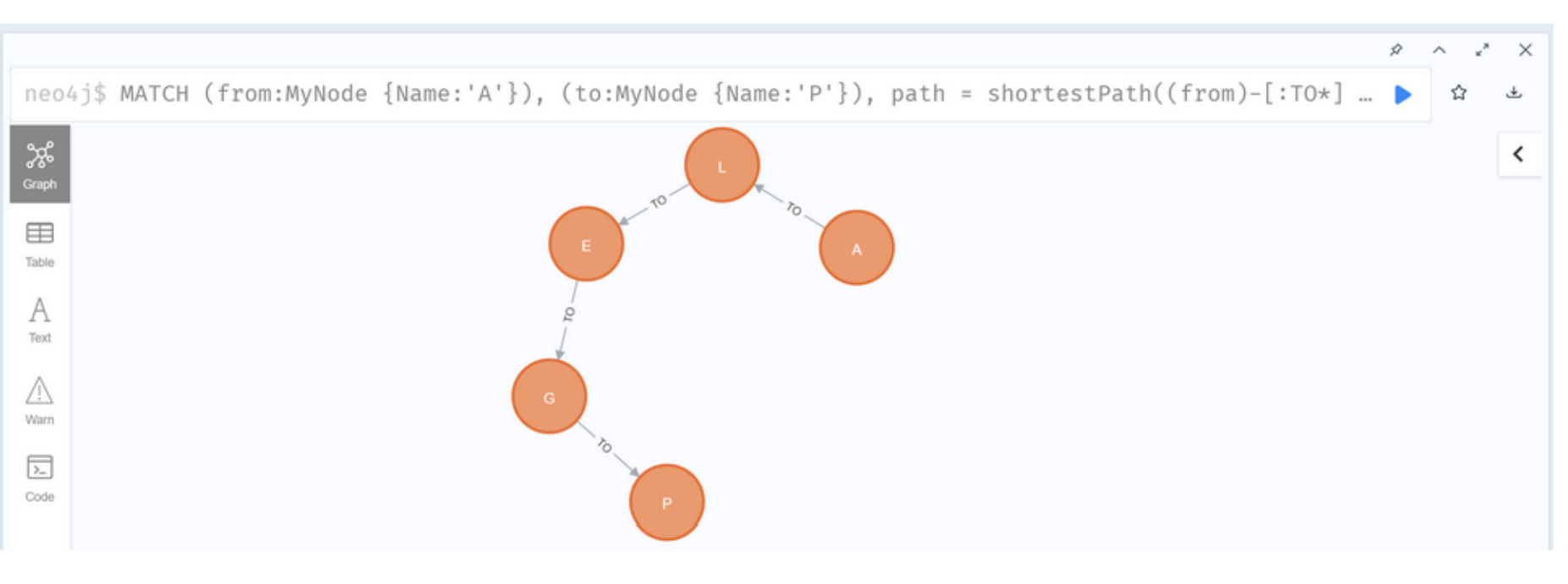
```
neo4j$ MATCH p = (a)-[:TO*]→(c) WHERE a.Name='H' AND c.Name='P' RETURN [n IN nodes(p) | n.Name] …
```

```
|"Nodes"                              |"PathLength"|"PathDist"|
|["H","J","F","A","L","E","G","P"]|7           |39        |
```

**Query 9.Dijkstra's algorithm for a specific target node.**

```
MATCH (from:MyNode {Name:'A'}), (to:MyNode {Name:'P'}),
path = shortestPath((from)-[:TO*]->(to))
WITH REDUCE(dist = 0, rel IN relationships(path) | dist + toInteger(rel.dist)) AS distance,
RETURN path, distance
```

First, we're going to match the node with the name *A,* and the node with the name *P*. and we're going to find the shortest path in terms of hops from *A* to *P,* setting that equal to the variable *path*. Then, we'll perform a *reduce* command, and set the variable *dist = 0.* We'll go through and sum all of the distances of each of the edges in our shortest path. Finally, we return that value as a *distance*, and we also return the path *variable.*

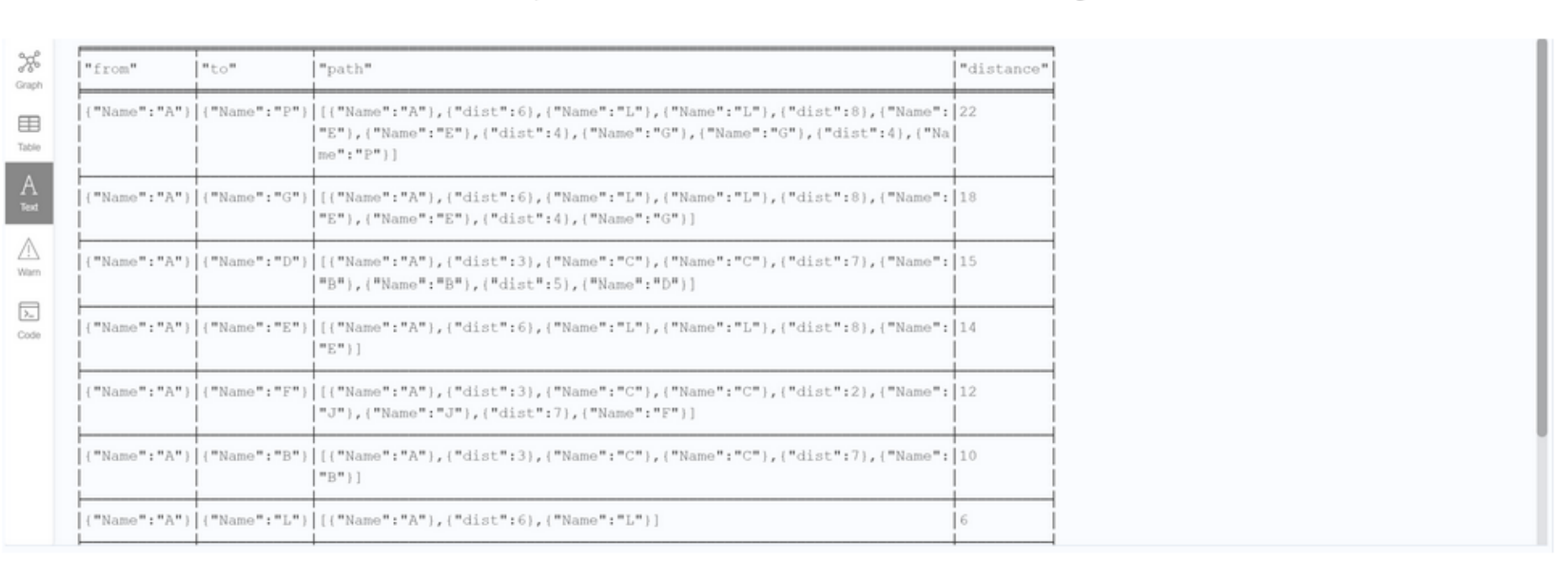


```
neo4j$ MATCH (from:MyNode {Name:'A'}), (to:MyNode {Name:'P'}), path = shortestPath((from)-[:TO*] …
```

**Query 10. Dijkstra's algorithm - Single Source Shortest Path (SSSP).** In our previous query, we specified that we wanted a match for the source node and the destination node. But if we don't specify a destination node, we can apply Dijkstra's single source shortest path algorithm from node A to any other node.

```
MATCH (from: MyNode {Name:'A'}), (to: MyNode)
WHERE from <> to
MATCH path = shortestPath((from)-[:TO*]->(to))
WITH from, to, path, REDUCE(dist = 0, rel in relationships(path) | dist + toInteger(rel.dis
RETURN from, to, path, distance
ORDER BY distance DESC
```

We add a constraint in line two to specify that the same node *A* cannot be the origin and destination.



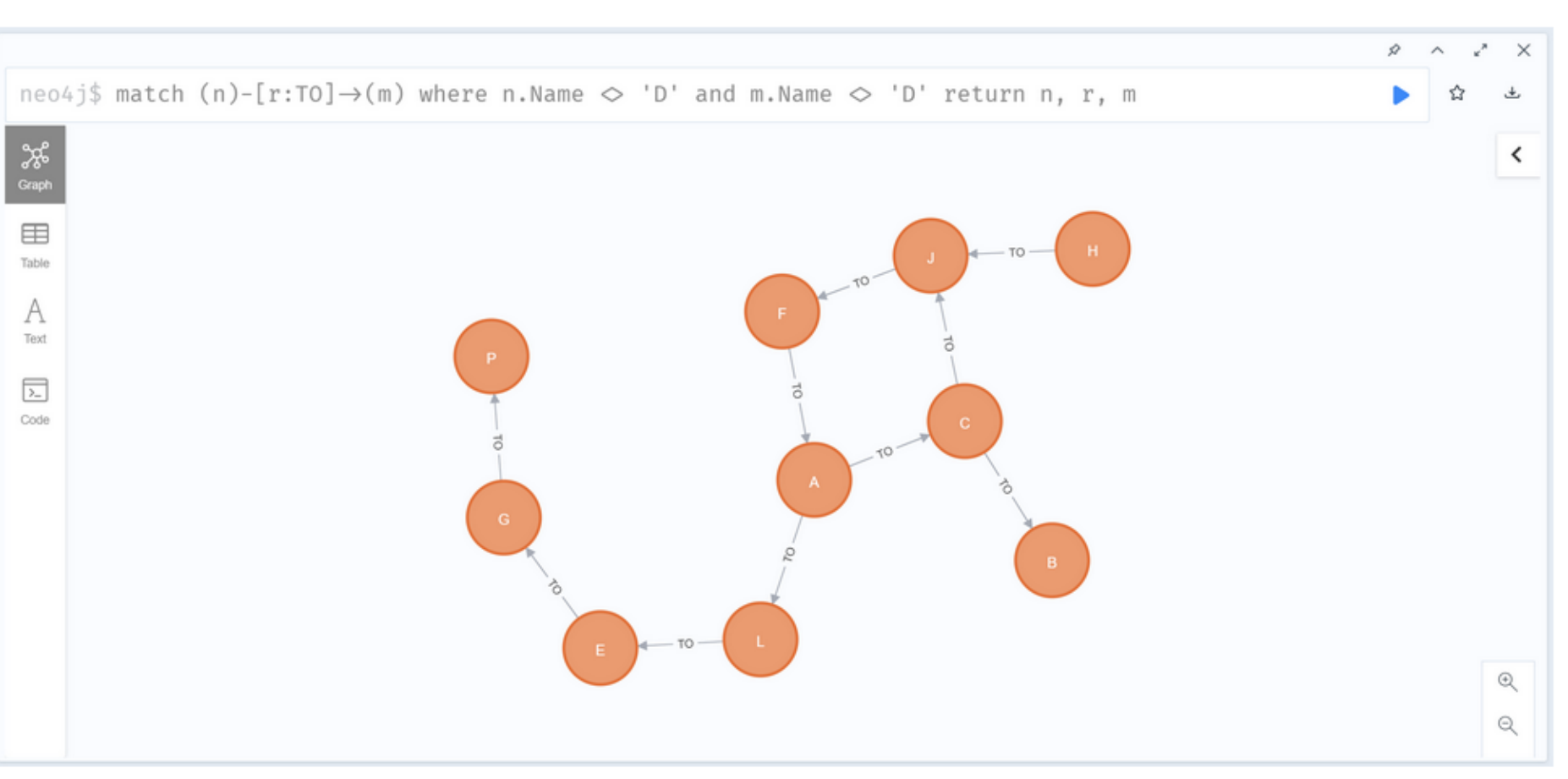
```
|"from"      |"to"        |"path"                                                                 |"distance"|
|{"Name":"A"}|{"Name":"P"}|[{"Name":"A"},{"dist":6},{"Name":"L"},{"Name":"L"},{"dist":8},{"Name":|22        |
|            |            |"E"},{"Name":"E"},{"dist":4},{"Name":"G"},{"Name":"G"},{"dist":4},{"Na|          |
|            |            |me":"P"}]                                                              |          |
|{"Name":"A"}|{"Name":"G"}|[{"Name":"A"},{"dist":6},{"Name":"L"},{"Name":"L"},{"dist":8},{"Name":|18        |
|            |            |"E"},{"Name":"E"},{"dist":4},{"Name":"G"}]                             |          |
|{"Name":"A"}|{"Name":"D"}|[{"Name":"A"},{"dist":3},{"Name":"C"},{"Name":"C"},{"dist":7},{"Name":|15        |
|            |            |"B"},{"Name":"B"},{"dist":5},{"Name":"D"}]                             |          |
|{"Name":"A"}|{"Name":"E"}|[{"Name":"A"},{"dist":6},{"Name":"L"},{"Name":"L"},{"dist":8},{"Name":|14        |
|            |            |"E"}]                                                                  |          |
|{"Name":"A"}|{"Name":"F"}|[{"Name":"A"},{"dist":3},{"Name":"C"},{"Name":"C"},{"dist":2},{"Name":|12        |
|            |            |"J"},{"Name":"J"},{"dist":7},{"Name":"F"}]                             |          |
|{"Name":"A"}|{"Name":"B"}|[{"Name":"A"},{"dist":3},{"Name":"C"},{"Name":"C"},{"dist":7},{"Name":|10        |
|            |            |"B"}]                                                                  |          |
|{"Name":"A"}|{"Name":"L"}|[{"Name":"A"},{"dist":6},{"Name":"L"}]                                 |6         |
```

The results displayed consist of the actual original path from A to P with a distance of 22, along with a display of all of the intermediate paths generated in the process, all the way down to a single edge path between A and C.

**Query 11. Graph not containing a selected node.** Now let's say that we want to display the graph, but without *D.*

```
MATCH (n)-[r:TO]->(m)
WHERE n.Name <> 'D' and m.Name <> 'D'
RETURN n, r, m
```

In the second line of code, we specify that none of the *n* nodes should have the name *D,* and we use the same condition with the m nodes. We then return all remaining nodes and relationships.

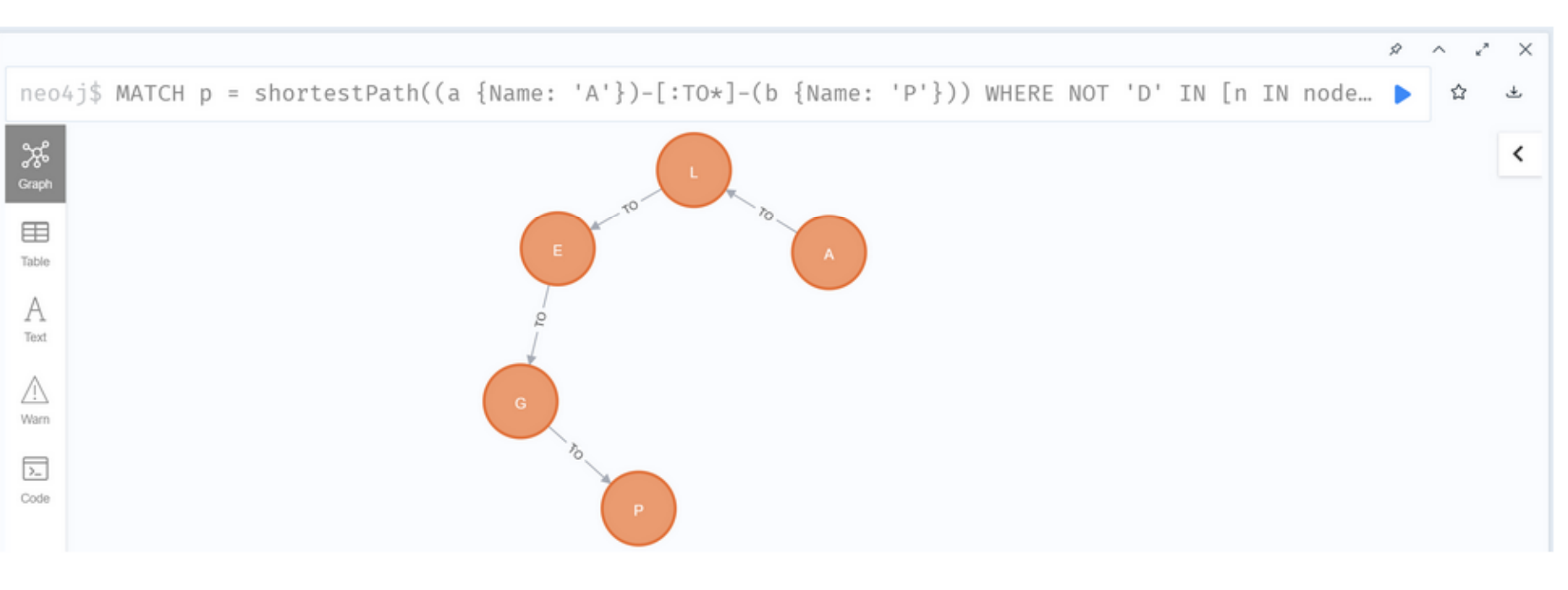


```
neo4j$ match (n)-[r:TO]→(m) where n.Name <> 'D' and m.Name <> 'D' return n, r, m
```

**Query 12. Shortest path over a Graph not containing a selected node.** Now let's say we want to go from town *A* to town *P*. but we don't want to pass through town *D.*

```
MATCH p = shortestPath((a {Name: 'A'})-[:TO*]-(b {Name: 'P'}))
WHERE NOT 'D' IN [n IN nodes(p) | n.Name]
RETURN p, length(p)
```

In the second line, we want to issue sort of a negative statement in which the resulting list of node names that we extract, using the *extract* statement, cannot contain the node *D*.



```
neo4j$ MATCH p = shortestPath((a {Name: 'A'})-[:TO*]-(b {Name: 'P'})) WHERE NOT 'D' IN [n IN node…
```

**Query 13. Graph not containing the immediate neighborhood of a specified node.** We're looking fora graph that doesn't contain the immediate neighborhood of a specific node. This means all of the nearest, or the first neighbors of a specific node.

```
MATCH (d {Name:'D'})-[:TO]-(b)
with collect(distinct b.Name) as neighbors
MATCH (n)-[r:TO]->(m)
WHERE
NOT (n.Name in (neighbors+'D'))
AND
NOT (m.Name in (neighbors+'D'))
RETURN n, r, m

```

We match the node *D*, and all edges between *D*, and any other node. Then, we issue a collect command to collect all of the distinct neighbors of *D*, and we apply a constraint to that in which the returned list of neighbors cannot contain the node with the name *D*. Likewise, the neighbors list for the target nodes, can also not contain the node *D*.

```
neo4j$ MATCH (d {Name:'D'})-[:TO]-(b) with collect(distinct b.Name) as neighbors MATCH (n)-[r:TO]→(m) WHERE NOT (n.Name in (n…
```

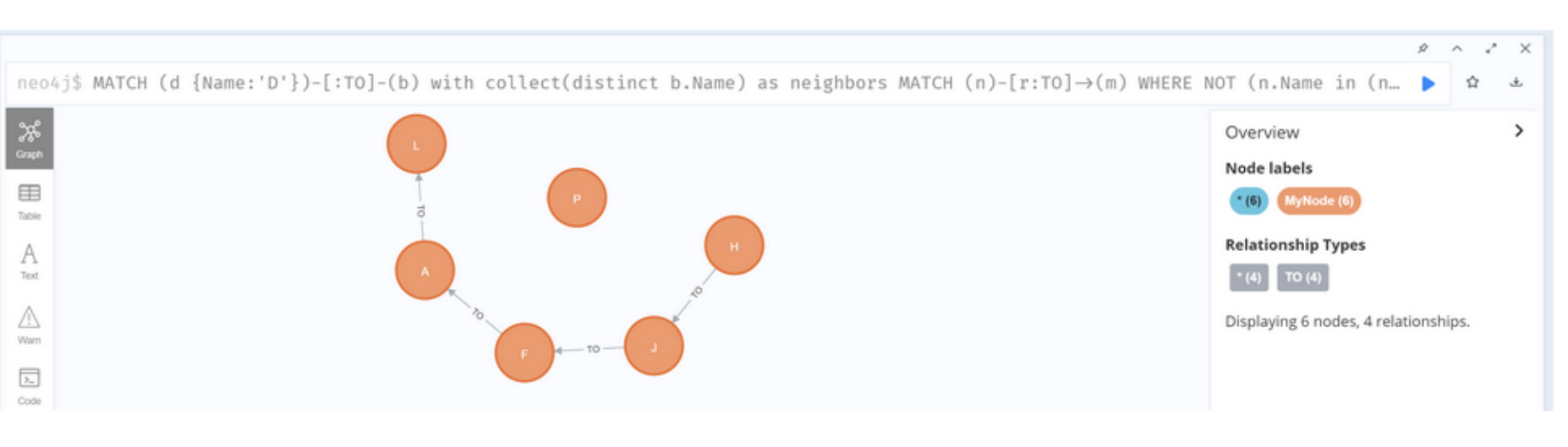
Displaying 5 nodes, 4 relationships.

If you look at the returned graph and compare it to the the original network, you will notice that we indeed removed *D* and it's neighbors, but our code also removed node *P*, which is not a first neighbor of *D* and should be included in our result, even if it doesn't have any relationship with the rest of the returned nodes.

We can do an extended version of the previous query to handle this:

```
MATCH (d {Name:'D'})-[:TO]-(b)
with collect(distinct b.Name) as neighbors
MATCH (n)-[r:TO]->(m)
WHERE
NOT (n.Name in (neighbors+'D'))
AND
NOT (m.Name in (neighbors+'D'))
MATCH (d {Name:'D'})-[:TO]-(b)-[:TO]->(leaf)
WHERE NOT((leaf)-->())
RETURN (leaf), n,r,m
```

In this case, the node *P* was a leaf node, so we want to make sure that not only do we match the nodes that satisfy these constraints above, but we also want to include the node or nodes that are leaf nodes, which may also arguably be part of the results you expect to be returned.



```
neo4j$ MATCH (d {Name:'D'})-[:TO]-(b) with collect(distinct b.Name) as neighbors MATCH (n)-[r:TO]→(m) WHERE NOT (n.Name in (n…
```

Displaying 6 nodes, 4 relationships.

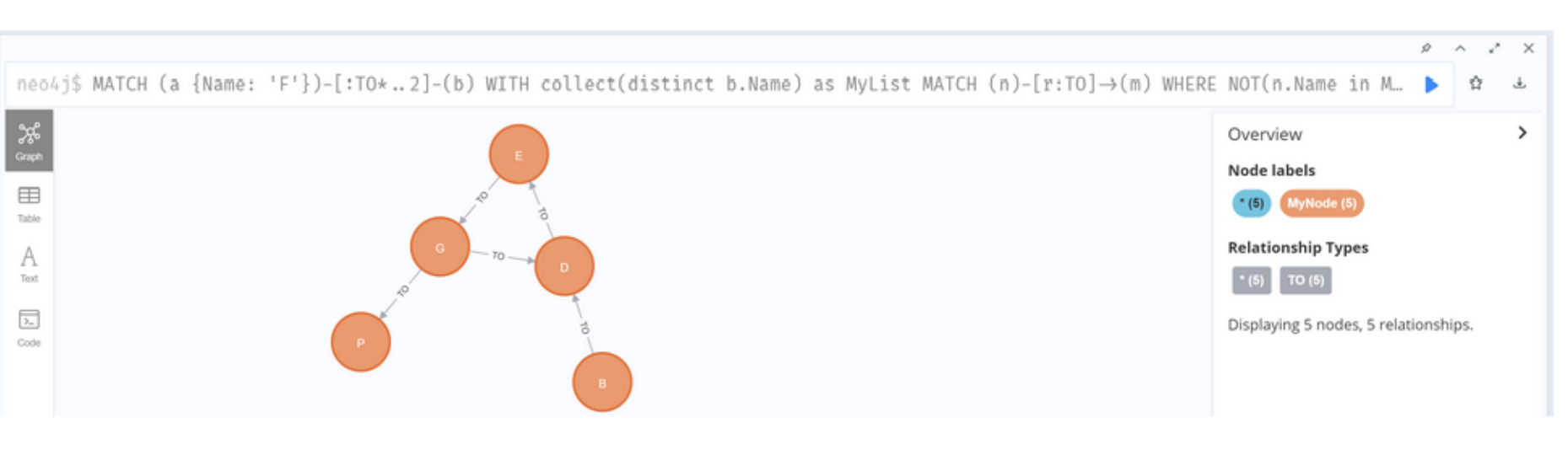
Similarly, we could have root nodes which should be returned as part of our results:

```
MATCH (d {Name:'D'})-[:TO]-(b)
with collect(distinct b.Name) as neighbors
MATCH (n)-[r:TO]->(m)
WHERE
NOT (n.Name in (neighbors+'D'))
AND
NOT (m.Name in (neighbors+'D'))
MATCH (d {Name:'D'})-[:TO]-(b)<-[:TO]-(root)
WHERE NOT((root)<--())
RETURN (root), n,r,m
```

**Query 14. Graph not containing a selected neighborhood.**

```
MATCH (a {Name: 'F'})-[:TO*..2]-(b)
WITH collect(distinct b.Name) as MyList
MATCH (n)-[r:TO]->(m)
WHERE NOT(n.Name in MyList) AND NOT (m.Name in MyList)
RETURN distinct n, r, m
```

We want to eliminate all of the second neighbors of *F*. We match all of those nodes that are second neighbors of *F*, including *F* itself, and we'll place those in a variable called *MyList*. Then, we go back through the network and match all of the nodes and edges, where the source nodes are not part of the nodes in the *MyList* and the target nodes are not contained in *MyList*. Finally, we return those nodes and edges.



```
neo4j$ MATCH (a {Name: 'F'})-[:TO*..2]-(b) WITH collect(distinct b.Name) as MyList MATCH (n)-[r:TO]→(m) WHERE NOT(n.Name in M…
```

Displaying 5 nodes, 5 relationships.

Go to next item   ✓ Completed