Report on Neo4j Libraries

Prepared by

Aslı Umay Öztürk

Middle East Technical University (METU)
Computer Engineering Department

This report is prepared for Summer Practice held within the project 117E566 supported by TUBITAK.

Ankara, 2019

3. Used Technologies and Methods

Before diving into the detailed flow of the internship, first I want to talk about the new technologies and systems I have learned about during my 6-week internship.

3.1. Graph Databases

Since the group was working with a huge graph data, they have been experimenting with graph database services. Never being heard of such thing, first I did some research on graph databases.

As definition, *a graph database is a database that uses graph structures for semantic queries with nodes, edges, and properties to represent and store data.* [4]

Different from a traditional database, in a graph database the data is stored as graphs instead of tables, which makes storing and querying on a real world graph data easier. Also, it is easier for our human brains to represent everything as graphs, instead of tables, which makes the abstraction, representation and planning phase easier for us.

3.2. Neo4j & Cypher

When we talk about graph databases, *Neo4j* [8] is an outstanding graph database platform that shows up when you look for graph databases online. It uses the query language *Cypher* [1], which is designed specially for graph databases.

3.2.1. Neo4j

Neo4j is one of the leading graph database platforms out there. It's implemented with Java and its creators also created a query language, Cypher. It also has several good libraries for graph algorithms and commonly used methods.

Neo4j also comes with great user interface/visualization tools. Seeing your graph data is something that you'll appreciate after working with tables and tables full of data.

3.2.2. Cypher

Cypher is a query language that can be described as "SQL for a Graph Database". It is designed for graph type data, which makes querying on a graph database easier and straightforward. Also, the resulting queries are significantly shorter.

Find all direct reports and how many people they manage, up to three levels down

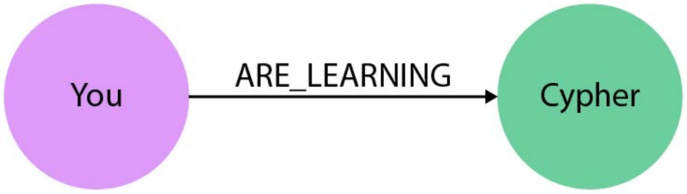
Cypher Query

```
MATCH (boss)-[:MANAGES*0..3]->(sub),
      (sub)-[:MANAGES*1..3]->(report)
WHERE boss.name = "John Doe"
RETURN sub.name AS Subordinate,
  count(report) AS Total
```

SQL Query

Figure 1: Same query in SQL and Cypher

Pattern matches in Cypher code is basically designed as ASCII Art, which makes it even easier to learn and master.




```
(You)-[:ARE_LEARNING]->(Cypher)
```

Figure 2: AsciiArt pattern of a relationship between two nodes

3.2.3. Neo4j Desktop and Neo4j Browser

Neo4j Desktop[7] can be set up to your system and the user interface *Neo4j Browser*[6] can be used via your favourite browser by connecting to `localhost`. You can run your Cypher queries from Neo4j Browser.

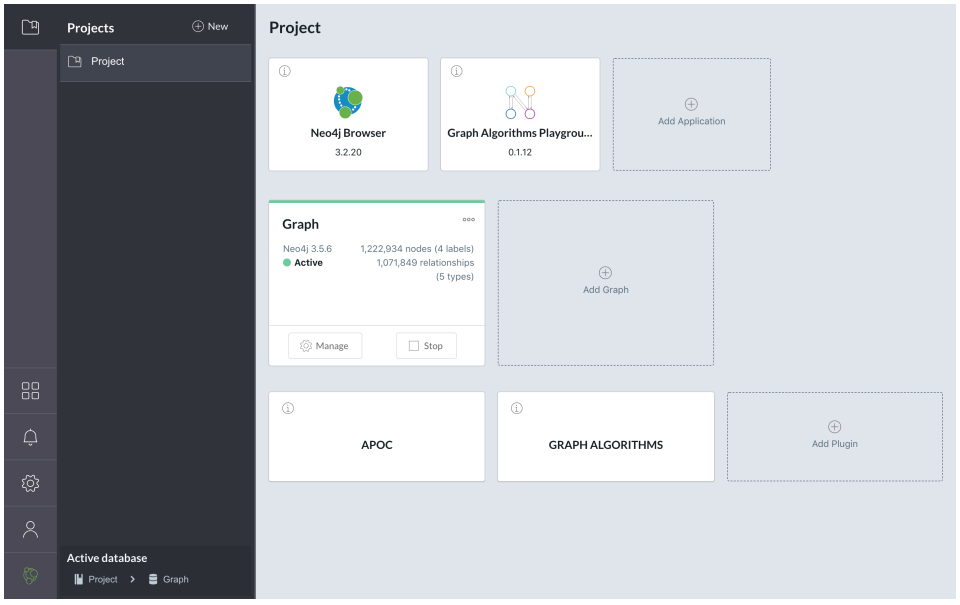


Figure 3: UI of Neo4j Desktop

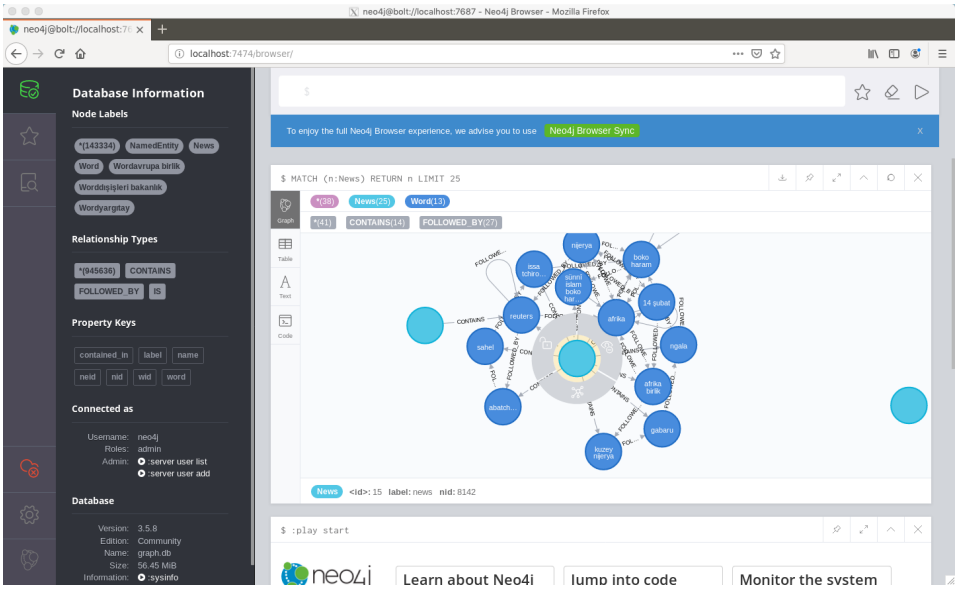


Figure 4: UI of Neo4j Browser

3.2.4. Cypher Shell

Also, *Cypher Shell* [2] can be used to run queries from terminal, without any user interface.

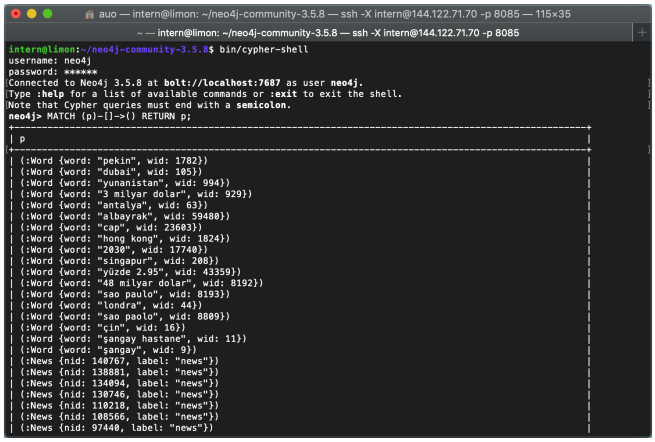


Figure 5: Running a query on Cypher Shell

3.2.5. Neo4j Libraries

There are many official and unofficial libraries and language drivers for Neo4j. I have mainly used the unofficial language driver *Py2neo* [10] and official library for several graph algorithm implementations, *Graph Algorithms* [3].

3.2.6. Graph Algorithms in Detail

Graph Algorithms is a library for Neo4j that implements centrality, community detection, path finding, similarity and link prediction algorithms. As in version 3.5, most of the methods are now called unofficial and only experimental.

Overview of the methods available:

- **Centrality Algorithms**
  PageRank, ArticleRank (experimental after 3.5), Betweenness Centrality, Closeness Centrality, Harmonic Centrality (experimental after 3.5), Eigenvector Centrality (experimental after 3.5), Degree Centrality
- **Community Detection Algorithms**
  Louvain, Label Propagation, Connected Components, Strongly Connected Components (experimental after 3.5), Triangle Counting/Clustering Coefficient (experimental after 3.5), Balanced Triads (experimental after 3.5)
- **Path Finding Algorithms**
  Minimum Weight Spanning Tree (experimental after 3.5), Shortest Path (experimental after 3.5), Single Source Shortest Path (experimental after 3.5), All Pairs Shortest Path (experimental after 3.5), A* (experimental after 3.5), Yen's K-shortest paths (experimental after 3.5), Random Walk (experimental after 3.5)

- **Similarity Algorithms**
  Jaccard Similarity (experimental after 3.5), Cosine Similarity (experimental after 3.5), Pearson Similarity (experimental after 3.5), Euclidean Distance (experimental after 3.5), Overlap Similarity (experimental after 3.5)

- **Link Prediction Algorithms**
  Adamic Adar (experimental after 3.5), Common Neighbors (experimental after 3.5), Preferential Attachment (experimental after 3.5), Resource Allocation (experimental after 3.5), Same Community (experimental after 3.5), Total Neighbors (experimental after 3.5)

4. Flow of the Intership

Systems, methods and services that I have listed on the previous section were all used by me to evaluate if Neo4j is a service that offers improvements on the group's research or not. Before the detailed report on Neo4j, let me talk a bit about the flow of my internship, more in a weekly schedule manner.

4.1. First Assignment: Tryouts and Hands-On Learning

On the first meeting, I was assigned to experiment with Neo4j to have an idea about the concept of graph databases. I have never heard of the graph database concept before nor have I ever used one. So I started by setting up Neo4j on my desktop, creating my workplace and reading lots of documentation and articles.

There were plenty of Neo4j and Cypher tutorials provided by the Neo4j and Cypher teams themselves, so first I have completed them. Later on, I realized I should have a dataset to experiment with. With the guidance of the group, I have obtained a book-author-year-genre graph dataset, and an advice about checking

the *Graph Algorithms* library.

The dataset I obtained was a *csv* file that can be imported to the Neo4j Desktop by either built-in import methods or a faster Python script written by one of the group members that utilizes Py2neo. I modified the script and imported the csv file, then I started to experiment on the data using Neo4j Browser to run queries and visualize nodes and relationships.

While looking around for different examples of Graph Algorithms library methods, I have run into the add-on tool *Graph Algorithms Playground*, which makes you choose the configurations for each algorithm and later creates and runs the Cypher queries for you.

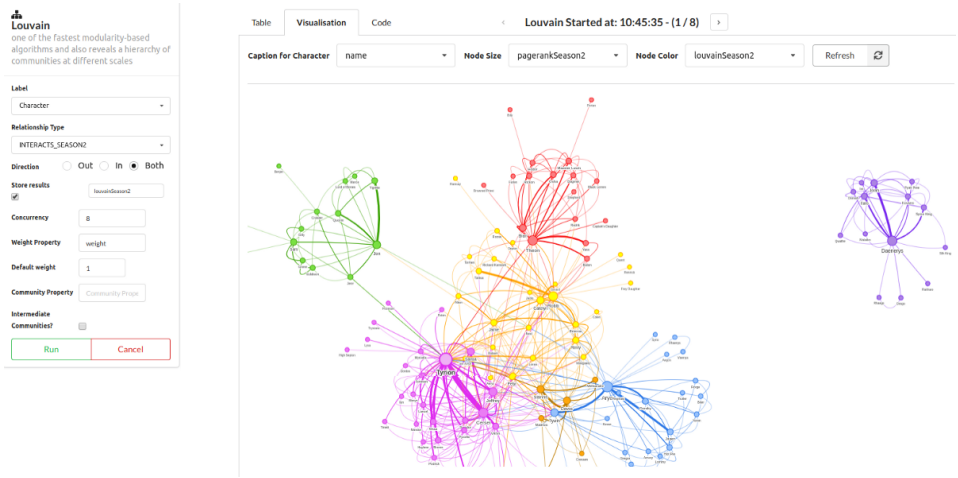


Figure 6: UI of Graph Algorithms Playground

4.2. Realizing Something is Wrong

While using the Graph Algorithms Playground to run queries, I realized that some methods had some problems, I first thought that the book-related dataset is not a fit for them, and I tried to modify the data by creating new nodes and relationships using a C script.

With the edited data I again tried to run some queries, managed to make them work with less problems. Having the confidence of finally being learnt, I asked for the main dataset that the projects use.

4.3. Obtaining Real Data & Server Access

Diving into the methods with some mock or arbitrary data is a good way to learn about Neo4j, and that's what I did first. But to see if Neo4j is something that can improve the performance, I need to experiment with the real data that the team uses. Hence, I was asked for the data soon after I was comfortable with the Neo4j itself.

At this point, since the data is huge, I have also obtained a user in a remote server in our department to run Neo4j on. After setting up the and fixing issues related to `ssh` connection, I was finally ready to run the queries and collect statistics on the nodes, as well as see if the Graph Algorithms methods really work as expected.

4.4. Experimenting with Real Data

The research team uses a news data for their projects. To get a better understanding, I first read the *csv* files of the node and relationship description where the data is imported from. Later, since Neo4j makes it really easy, I have used Neo4j Browser to visualize the data by running some queries to understand the further relationships.

News data contains some main `News` nodes where each node is in a `CONTAINS` relationship with one or more `Word` nodes. Each `Word` has a type like `LOCATION, TIME, DATE...` that is shown by `IS` relationship. Also, each `Word` can be in a `FOLLOWED_BY` relationship between several `Word` nodes (including themselves).

At this point, there were some changes on the Graph Algorithms library since I have started this internship and I realized that most of the methods in this library was now are called *experimental* and *unofficial*. Which made my main goal to be updated as *"try to see how many of them actually work as expected."* rather than *"collect some statistics about the data."*

5. Report of the Graph Algorithms Library

Using a remote machine and the data from the projects the team has been working on, I have tried all the methods to see if they work as they expected. Since most of the methods that Graph Algorithms library offers are experimental, it was crucial to find out and pinpoint the issues each method has. Doing so, I can help the team to understand whether if Neo4j will result in higher performance and better results or not. Here in this section, I will go into the details of the queries for some methods to give examples, share some statistics and comment on the reliability of the methods offered.

While calling the methods, there are certain configurations that I can use to limit the relationships and nodes of the graph. On most of the methods it makes sense to include everything, but sometimes I'll be limiting the graph by node label or relationship type.

5.1. Centrality Algorithms

4 out of 7 of centrality algorithms are called "officially supported" in version 3.5, but still, official support or being experimental does not really say a final thing about the consistency of the methods.

5.1.1. PageRank

PageRank is one of the official methods, and it works quite fast. It returns after around 0.5 seconds when run on all nodes and all relationships. A sample query for PageRank method is as follows:

```
CALL algo.pageRank.stream(
    null,null,{iterations:50, dampingFactor:0.85})
YIELD nodeId, score
RETURN algo.asNode(nodeId).word AS description, score
ORDER BY score DESC;
```

Figure 7: Cypher query for PageRank on all nodes

description	**score**
"PERSON"	4262.147945632489
"ORGANIZATION	3134.20248181316
"MONEY"	1786.049925264471
"LOCATION"	1708.0299525968696
"türkiye"	998.5374834925137
"PERCENT"	827.4932531329265
"DATE"	575.7568497967907
"abd"	435.48279780589576
"istanbul"	298.40404949749194
"rusya"	271.4267402492537

Table 1: results of the PageRank query on all nodes

We can interpret this result to obtain the insight that these nodes have the highest importance in the dataset. It makes sense since most of the news data has news about Turkey, Russia etc. as well as all the type nodes are easy to reach from any node.

We can modify the query to obtain only some type of relationship instead of all graph connections. For example, if we were only to check `CONTAINS` relationship to see which nodes are mentioned the most in the general dataset, we can do the following and obtain the following result:

```
CALL algo.pageRank.stream(
    null,'CONTAINS',{iterations:50, dampingFactor:0.85})
YIELD nodeId, score
RETURN algo.asNode(nodeId).description AS description, score
ORDER BY score DESC;
```

Figure 8: Cypher query for PageRank on `CONTAINS` relationship

description	**score**
"türkiye"	204.13141403198242
"abd"	67.77411727905273
"rusya"	42.538142919540405
"avrupa"	41.96432061195373
"istanbul"	36.6232672214508
"ab"	31.691536378860473

Table 2: results of the PageRank query on `CONTAINS` relationship

5.1.2. ArticleRank (experimental after 3.5)

ArticleRank is an experimental method and it works a bit slow. It returns after around 9 seconds when run on the whole graph. Another problem with the method is that it does not work consistently at all. At every run it returns different scores, causing the ranking to change.

5.1.3. Betweenness Centrality

Betweenness centrality is one of the official methods, but it works very slow even when it's run on only `FOLLOWED_BY` type of relationships, which only includes `Word` nodes. It returns after a wait of around 7 minutes, but at least it works consistently.

5.1.4. Closeness Centrality

Closeness centrality is one of the official methods, and it works fine. Since our data has different sets of connected components, if a word node is in a 3-node connected component isolated from the rest of the graph, it automatically has the highest score, which can be a bit misleading. Query returns after around 1.5 seconds when run on only `FOLLOWED_BY` type of relationships, which only includes `Word` nodes.

5.1.5. Harmonic Centrality (experimental after 3.5)

Harmonic centrality is an experimental method, but it seems to be working consistently. It returns after around 30 seconds when run on only `FOLLOWED_BY` type of relationships, which only includes `Word` nodes.

5.1.6. Eigenvector Centrality (experimental after 3.5)

Eigenvector centrality is an experimental method, it seems to be working fine but slow. It returns after around 2.5 minutes when run on only `FOLLOWED_BY` type of relationships, which only includes `Word` nodes.

5.1.7. Degree Centrality

Degree centrality is also one of the official methods, and it works fast and it's consistent. Also, it says a lot about our data and nodes' connections.

Degree centrality returns the number of connections from a node, being its degree. A sample Cypher code for this query is like the following:

```
CALL algo.degree.stream(
    null,"FOLLOWED_BY",{direction:"both"})
YIELD nodeId, score
RETURN algo.asNode(nodeId).word AS description, score AS scr
ORDER BY score DESC;
```

Figure 9: Degree Centrality query

For readibilty, we can add the line `WHERE algo.asNode(nodeId).content=<type>` before the `RETURN` line and check types seperately.

description	**score**
"donald trump"	342.0
"şimşek"	382.0
"binali yıldırım"	452.0
"vladimir putin"	534.0
"barack obama"	538.0
"yıldırım"	913.0
"recep tayyip erdoğan"	1116.

Table 3: results of the Degree Centrality query on `PERSON`

5.2. Community Detection Algorithms

Community detection algorithms are half official, half experimental. But even official ones seems to be having problems.

5.2.1. Louvain

Louvain method is an officially supported one. But it seems like it still has some problems. Sometimes it fails to load all nodes, total node count changes at each run. Hence the communities and community count changes at each run, makes the method even more inconsistent.

Still at each run there is a significant realization that most of the news data we have is connected somehow, making the count of communities real low and one community really huge in node count.

5.2.2. Label Propagation

Label propagation is also an official method, but it has a huge problem. As I have mentioned before, configurations can be made to include or exclude nodes and relationships while running a query. But even though the documentation says the method has a setting as `direction="BOTH"`, it results in a `NullPointerException`. So probably, that means it is not working properly.

5.2.3. Connected Components

Connected components is also an official method, and it seems to be working fine.

5.2.4. Strongly Connected Components (experimental after 3.5)

Strongly connected components is not an official method, and it does not support the `direction="BOTH"` configuration. Hence, our data seems as if it lacks any strongly connected component.

5.2.5. Triangle Counting/Clustering Coefficient (experimental after 3.5)

Triangle counting/clustering coefficient method is an experimental method but it seems to be working fine and fast. Query returns in around 2 seconds on whole data. A sample query and its result can be shown as the following:

```
CALL algo.triangleCount.stream(null, null, {concurrency:4})
YIELD nodeId, triangles, coefficient
RETURN algo.asNode(nodeId).word AS description,
    triangles, coefficient
ORDER BY triangles DESC;
```

Figure 10: Triangle Count/Clustering Coefficient query on all nodes

description	triangles	coefficient
"türkiye"	243454	8.083024971578234E-4
"abd"	132163	0.0018348326321741234
"avrupa"	88581	0.0034437329239155106
"istanbul"	86429	0.0025988924133895476
"rusya"	86268	0.003245233624243717
"almanya"	67418	0.004932324481369936
"ab"	66808	0.005480803773557353
"çin"	66658	0.0042201935927865825

Table 4: results of the Triangle Count/Clustering Coefficient query on all nodes

5.2.6. Balanced Triads (experimental after 3.5)

Balanced triads method is also an experimental one, and it gives a nasty `ArrayIndexOutOfBoundsException` error. I couldn't resolve the issue, and so does the Internet.

5.3. Path Finding Algorithms

All path finding algorithms are called to be experimental.

5.3.1. Shortest Path (experimental after 3.5)

Shortest path method is an experimental one, but it works consistently and fast. A sample query between two nodes and its result can be seen as following:

```
‘MATCH (start:Word{wid:72}), (end:Word{wid:109})
CALL algo.shortestPath.stream(start, end, null)
YIELD nodeId, cost
RETURN algo.asNode(nodeId).word AS name, cost;
```

Figure 11: Shortest Path query between two `Word` nodes

name	**cost**
"türkiye"	0.0
"1 ocak"	1.0
"3 ocak"	2.0

Table 5: result of the Shortest Path query between two `Word` nodes

5.3.2. Single Source Shortest Path (experimental after 3.5)

Single source shortest path method is also an experimental method, and it has some issues. It acts as it can load the relationships on both directions with `direction="BOTH"` configuration, but the results seems otherwise.

5.3.3. All Pairs Shortest Path (experimental after 3.5)

All pairs shortest path is also an experimental method. I cannot make it return any value even though I have increased heap size.

5.3.4. Yen's K-shortest Paths (experimental after 3.5)

Yen's K-shortest paths method is also an experimental one, but it works fast and consistent. It has two settings, to return the nodes or the paths. Sample query on the nodes/costs result can be seen here:

```
MATCH (start:Word{wid:72}), (end:Word{wid:257})
CALL algo.kShortestPaths.stream(start, end, 3, null, {direction:'BOTH'})
YIELD index, nodeIds, costs
RETURN [node in algo.getNodesById(nodeIds) | node] AS description,
    costs, reduce(acc = 0.0, cost in costs | acc + cost) AS totalCost;
```

Figure 12: K-Shortest Paths query between two nodes

description	costs	totalCost
"türkiye", "LOCATION", "boşkan"	[1.0, 1.0]	2.0
"türkiye", "LOCATION", "buşehr", "boşkan"	[1.0, 1.0, 1.0]	3.0
"türkiye", "alp", 32367, "boşkan"	[1.0, 1.0, 1.0]	3.0

Table 6: result of the K-Shortest Paths query between two nodes

5.3.5. Random Walk (experimental after 3.5)

Random walk is also an experimental method, but it works fine. It does not repeat itself and really seems random, returning paths that really exist. For example, some random paths of length 6 can be found as follows:

```
MATCH (start:Word{wid:72})
CALL algo.randomWalk.stream(id(start), 6, 1)
YIELD nodeIds
UNWIND nodeIds AS nodeId
RETURN algo.asNode(nodeId) as node;
```

Figure 13: query for a Random Walk of length 6 starting from the specified node

first run	**second run**
(:Word word: "türkiye", wid: 72)	(:Word word: "türkiye", wid: 72)
(:Word word: "200 milyon dolar", wid: 4673)	(:News nid: 112564, label: "news")
(:News nid: 52183, label: "news")	(:Word word: "suriye", wid: 93)
(:Word word: "suriye", wid: 93)	(:Word word: "zekeriya kars", wid: 56320)
(:News nid: 123910, label: "news")	(:News nid: 56319, label: "news")
(:Word word: "el nusra", wid: 6668)	(:Word word: "azez", wid: 39147)
(:Word word: "daeş", wid: 489)	(:Word word: "rusya", wid: 42)

Table 7: 2 runs of a Random Walk of length 6 starting from the same node

5.4. Similarity Algorithms

All similarity algorithms are said to be experimental.

5.4.1. Jaccard Similarity (experimental after 3.5)

Jaccard similarity is an experimental method, but it seems to be working consistently. It takes around 2 minutes for the query to return.

5.4.2. Overlap Similarity (experimental after 3.5)

Overlap similarity method is an experimental one, and like APSP it never returns in a reasonable time with reasonable space usage, it's probably because they use the same methods internally.

5.5. Link Prediction Algorithms

All link prediction algorithms are said to be experimental. But they all seem to be working fine and fast. They all calculate a score based on the degrees of the nodes, and since the degree method works fine, they all work fine.

References

[1] *Cypher Query Language*. URL: https://neo4j.com/developer/cypher-query-language/.

[2] *Cypher Shell*. URL: https://neo4j.com/docs/operations-manual/current/tools/cypher-shell/.

[3] *Graph Algorithms Library, Neo4j*. URL: https://neo4j.com/docs/graph-algorithms/3.5/.

[4] *Graph Databases, Wikipedia(via WikiWand)*. URL: https://www.wikiwand.com/en/Graph_database.

[5] *METU Computer Engineering Department*. URL: https://ceng.metu.edu.tr/.

[6] *Neo4j Browser*. URL: https://neo4j.com/developer/neo4j-browser/.

[7] *Neo4j Desktop*. URL: https://neo4j.com/developer/neo4j-desktop/.

[8] *Neo4j Graph Platform*. URL: https://neo4j.com/.

[9] *Personal Webpage of Pınar Karagöz*. URL: http://user.ceng.metu.edu.tr/~karagoz/.

[10] *Py2neo, Python Language Driver for Neo4j*. URL: https://py2neo.org/.

[11] *TEGHUB, Textual Event Graph Hub*. URL: https://teghub.ceng.metu.edu.tr/.