Visualizing Wikipedia using a Graph Database

Robin Chowdhury robinch@kth.se
Ludvig Hagberg ludvigha@kth.se
Jacob Sievers jsievers@kth.se
Hanna Nyblom hnyblom@kth.se

May 29, 2015

Abstract

In this paper a way to visualize large amounts of linked data using graph databases and clustering algorithms is presented. The example uses wikipedia as a source of data and Fruchterman-Reingold and ForceAtlas2 as layout algorithms to visually cluster the data provided by a neo4j database. The most computationally critical part of the visualization were the layout algorithms themselves as neo4j proved to give enough performance even for very large graphs.

1 Introduction

Wikipedia [2] is a great source of information but it's hard to get an overview of how that data is interrelated. Wikipedia provides summary articles and lists on large subjects, but those are created and edited by humans, which does not give a true picture of how the data is related and which articles that actually are most important. To see how the articles are actually related, we visualize the data as a graph so that humans easily could get an overview of the relations between wiki articles. This is done using a graph database [1, 4] as a source for the information and layout algorithms for the visualization itself.

2 Method

2.1 The database

Neo4j [5] is used to store the link info. Neo4j is a highly performant, scalable graph database. [4] It provides a query language called Cypher [6] that is used to populate and query the database.

2.2 Populating the Graph Database

To populate the database, a dump [7] from dbpedia [8] was downloaded. The dump contained only the internal links from the Swedish Wikipedia due to it being of smaller size than the English. The dump was parsed through a parser written in node.js [3] and populated the database using seraph [9] (neo4j database bindings for node.js).

Every link from an article to another did two things:

1. Added one or both of the two articles as vertices if they don't already exist.
2. Added an edge between the two vertices if there isn't already one.

This is accomplished with several MERGE queries in cypher, the SQL-like query language used by neo4j. Instead of using the internal IDs provided by neo4j to connect the graph the links themselves was used as IDs since these were unique.

2.3 Visualization

To retrieve data from the database a node.js server is used, using express.js [10] and seraph. Express provides a framework for creating web-servers and seraph is an API for querying graph databases. The server is used to query the database and format the JSON [11] response for easier display in the browser. To get data from the server a simple get request is used with different parameters depending on the queries. The data is sent in JSON format in a long list for easy display.

The front end is built with javascript using the linkurious.js [12] graph visualisation toolkit. The JSON is retrieved with a get call and then it is inputted into the linkurious API. Linkurious is using Sigma.js [13] to draw all the nodes and edges and format them in a visually appealing way.

The way that the nodes were drawn was a product of experimentation, in the beginning the nodes were drawn only with random x and y coordinates but that resulted in a big blob, increasing the interval did not help, this probably have to do with linkurious camera rendering using a form of relative sizes and distances. To get an even spread of the nodes an algorithm was used to draw the edges in a somewhat random grid by using random positioning and after a certain interval increasing x and/or y a fixed amount. An even spread was preferred so that a human easily could see how big the graph was and how it was connected.

The features that were implemented were a visualization of the shortest path between two nodes, a filter that could be applied to only show nodes within a certain interval of degrees, an option to hide edges, an option to resize edges relative to their degree and two clustering algorithms. The shortest path algorithm was constructed with a built in function in neo4j, that was parsed in the server and sent to the front end. The degree filter was constructed with linkurious filtering API, which was also used to hide edges. This was done in the front end and was not using the database. To resize the edges we used a loop that looped through all the edges and set the size of each edge to degree plus one (so no edge would have size zero), this was implemented so that you could easily see how connected a node was.

The two clustering algorithms that were used were ForceAtlas2 [14] and Fruchterman-Reingold [15]. Both of them had the same basic principle that nodes repel other nodes and edges attract the two connected nodes [14]. The main difference was that ForceAtlas2 was feasible to use up to 200'000 nodes while Fruchterman-Reingold only could handle around 200 [14]. Only around 200 nodes were used in the tests so both worked fine but one test of 20'000 nodes was made and ForceAtlas2 could easily handle it, but the code and options in

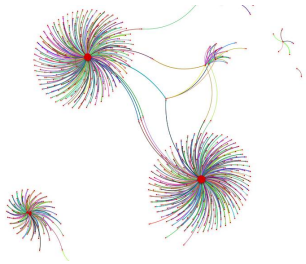

Figure 1: Example of FR.

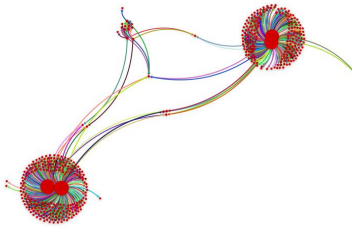

Figure 2: Example of FA.

linkurious were not optimised enough to render the graph without heavy lag. This shows that as long as you got the database populated with the data you want to visualise the frontend can scale well as long as the hardware can handle it. One thing to consider in the case of these clustering algorithms is that the initial positioning of the nodes matter for the end result, so the algorithm that was used for the initial visualization had an impact.

2.4 Results and Discussion

When analyzing the graphs after applying the clustering algorithms it was observable that ForceAtlas2 had a smaller spread of the nodes than Fruchterman-Reingold making it harder to get a good estimation of a single cluster as they were stacked on top of each other. This might have been helped by changing the options of ForceAtlas2 but due to lack of time only options provided by the official example code was used. This was less apparent if the nodes relative to their edges was resized giving less probability to draw over each other as a lot of the nodes where of degree one. Fruchterman-Reingold was the algorithm we thought gave the best visual representation of the graph as it did not have a lot of overlapping nodes and an even spread giving a clear idea of the cluster size and relative size.

The data was connected in such a way that there were a few big nodes with high degree connection to several nodes with only one degree. This we believe was due to the dataset we used, because the first article will create nodes for every link it has without doing a search from that link, so to get a real visualization we would need to use the entire Wikipedia database.

2.5 Conclusion

We believe we were successful in visualizing a part of Wikipedia. After applying the clustering algorithms, especially Fruchterman-Reingold, it was very clear how the sample dataset was structured and you could easily hover over a node to see what article it was referring to and a click would take you to said article. We also think that with some tweaks this project could be scaled to an arbitrarily large wiki with only the limitation of computational power and memory.

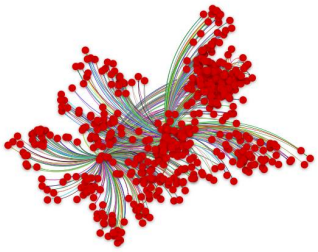

Figure 3: Example with no clustering applied.

References

[1] B.Shao, H.Wang and Y.Xiao "Managing and Mining Large Graphs: Systems and Implementations" Proceeding. SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.Pages 589-592

[2] "Wikipedia" - https://www.wikipedia.org/

[3] "Node.js" - https://nodejs.org/

[4] "Graph Database" - http://neo4j.com/developer/graph-database/

[5] "Neo4j" - http://neo4j.com/

[6] "Cypher" - http://neo4j.com/developer/cypher/

[7] "Swedish wiki dump" - http://downloads.dbpedia.org/3.9/sv/

[8] "dbpedia" - http://wiki.dbpedia.org/

[9] "Seraph.js" - https://github.com/brikteknologier/seraph

[10] "Express.js" - http://expressjs.com/

[11] "JSON" - http://json.org/

[12] "linkurious.js" - https://github.com/Linkurious/linkurious.js

[13] "Sigma.js" - http://sigmajs.org/

[14] ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software by Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann and Mathieu Bastian

[15] Graph Drawing by Force-directed Placement by Thomas M. J. Fruchterman and Edward M. Reingold