WASHINGTON STATE UNIVERSITY
CPTS 415: BIG DATA

Project Group: ACE December 9, 2023

Road Network Optimization: Neo4j and Spark for Scalable Graph Processing

by

Alex Hunt Ethan McPhee Brendan Crebs

Abstract:

This project explores the use of Neo4j and Apache Spark for performing scalable and distributed processing on road network graphs. Utilizing data from OpenStreetMap (OSM), our research aims to optimize the performance of shortest-path calculations and explore the effectiveness of routing over long distances. The road network data is visualized and interacted with through a user interface developed using Leaflet and Node.js. Various methods were employed to increase performance including caching results to MongoDB and utilizing Kafka for low-latency communication between the UI and Spark. The study presents results in terms of processing times and scalability, showing the effectiveness of these integrations and how they apply to real-world applications of road networks.

Keywords: OpenStreetMap, Neo4j, Apache Spark, Graph Algorithms, Nodejs, Leaflet, MongoDB, Apache Kafka

1 Introduction

Road networks play an important role in our daily lives, impacting transportation, urban planning, and various other domains. With the increase in the availability of geographic data, such as OSM which is maintained by a worldwide community of volunteers, new opportunities for extracting insights and analyzing large networks have become possible. However, storing and processing data of this scale has become an increasingly significant challenge.

To address some of these problems, our project utilized industry-leading tools such as Neo4j and Apache Spark to develop a fast and efficient shortest-path algorithm. The primary focus was developing an algorithm that can seamlessly scale up to larger networks while integrating into an interactive application, ensuring a user-friendly and responsive experience.

Our research applies to issues relating to transportation, logistics, and traffic management. One popular example is Google Maps, an application that provides the fastest route to a given location. Our application attempts to replicate this functionality while also providing additional features such as visualizing centrality. The algorithms used in this project provide insights relevant to real-world use cases such as logistics. For example, efficient routing can provide significant economic benefits as 72.6% of freight is moved by trucks providing $940.8 billion in gross freight revenues[1]. Also, road network centrality can help urban planners understand how decisions (construction, road changes, etc.) will affect road usage and can optimize road maintenance.

2 Related Work

The paper ”Can Betweenness Centrality Explain Traffic Flow?” by Kazerani & Winter [2] provides a study on the effectiveness of betweenness centrality in explaining traffic flow. Due to the dynamic nature of road networks, such as closures, traffic volume, and dynamic lane changes, they hypothesized that betweenness centrality alone cannot explain traffic flow. The researchers explored adapting betweenness centrality to capture the dynamics of a road network and how it compares to the static centrality measure. They took into account the pattern of peak travel demands and modified the edge weights to capture this. The result of their study was that traditional betweenness centrality does not accurately capture the measures as they become time-dependent.

This study is relevant to our project as we also incorporated betweenness centrality on road networks. Although we are using a static dataset, the results of the study show that there is potential for betweenness centrality to provide insights on traffic flow.

In another relevant study, ”The EBS-A* algorithm: An improved A* algorithm for path planning” by Wang et al.[3], improvements to the A* algorithm were explored for faster pathfinding. The researchers explored three methods, expansion distance, bidirectional search, and smoothing to create a faster algorithm. Expansion distance refers to adding a margin around obstacles called a collision buffer to prevent a robot from hitting an obstacle. Smoothing is meant to reduce sharp turns to avoid unnecessary movements and create a smoother path between two points. While these two methods may not help much with a road network, the third method could have promising results. In their study, when using bidirectional searching, they found the pathfinding to be twice as fast. In our case, this could be implemented by processing two paths in parallel, one starting from the start node and another from the target node. It can be expected that the two paths will meet up somewhere in the middle to form the complete shortest path.

3 Architecture

Due to the scalability requirements, we utilized a microservices architecture, allowing for isolation between components and being able to provision more resources as needed, such as additional Spark workers. Docker, a containerization service, was used to run each component on an isolated machine. The primary benefit of this was to package all the dependencies so our application could be easily moved to a distributed cluster of machines, potentially taking advantage of resources on a cloud computing platform.

3.1 Data Storage

The primary database used to store the OSM road network was Neo4j, a graph database representing data as nodes and the relationships between them. Using a graph database made the most sense as our data can be represented as a graph, with intersections (nodes) being connected with a network of roads

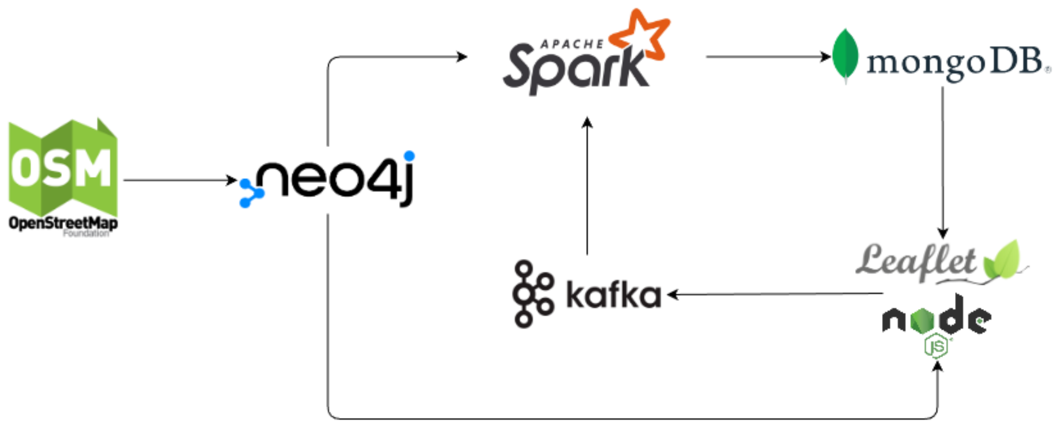

Figure 1: Project Architecture

(edges).

MongoDB, a NoSQL document database, was used to cache the results of the shortest path calculations. This database was chosen due to its fast read/write performance, scalability, and flexible schema. We have two collections, one containing the list of nodes that make up the shortest path and another containing the geographic line segments for that path. Having the second collection allows us to keep the data processing in Spark as the UI can directly pull the pre-computed line segments, improving the performance of the application.

Because the run-time of our Spark jobs was very short, occasionally under a second, we needed a low latency method to pass the parameters from the UI to our Spark cluster. We chose to create a streaming pipeline with the help of Apache Kafka. Kafka is a distributed event-streaming platform used for real-time processing. In our case, we had a long-running Spark job continuously reading from our Kafka topic to process the paths sent by the UI.

3.2 Data Flow

Starting from the user interface, requests for the shortest path between two nodes get published to a Kafka topic. The running Spark job instantly consumes the message and passes it to the shortest path algorithm. The algorithm interacts with Neo4j to return a list of nodes that make up the path. The results get sent to a MongoDB collection for storage. Spark also queries Neo4j for the list of coordinates that connects those nodes and passes that to another MongoDB collection. While this data is being processed, the UI app is querying MongoDB every second for those coordinates. Leaflet is then used to plot the line on an interactive OSM map.

4 Data Processing

OpenStreetMap provides data exports in either XML or protobuf formats. There are also Python packages such as OSMnx and Pyrosm that can make this data available as Geopandas GeoDataFrames.

To get a good understanding of the data, we decided to manually parse the XML export for the city of Pullman, WA into a Pandas data frame. Processing steps included calculating the distance between nodes, creating line segments, and building a mapping function to fill in missing speed limit values. For the shortest path calculation, we needed each road to have some sort of weight so the algorithm could know which routes to prioritize. We calculated this by creating a weighted distance, the distance divided by the speed limit for each road.

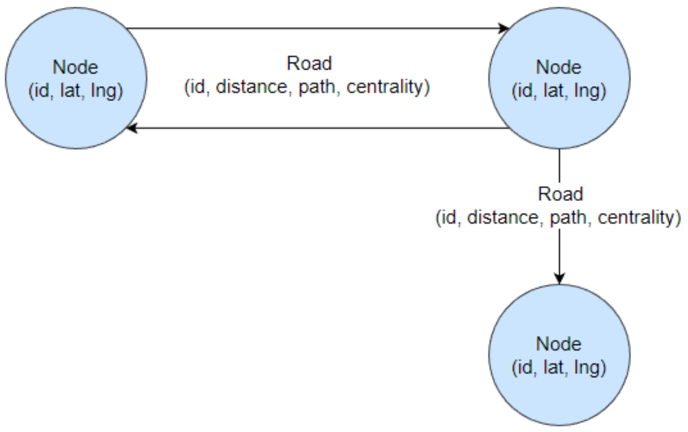
Node
(id, lat, lng)
Road
(id, distance, path, centrality)
Node
(id, lat, lng)
Road
(id, distance, path, centrality)
Node
(id, lat, lng)

Figure 2: Neo4j Data Model

Our data processing resulted in two CSV files, one for the nodes and another for the edges. This translates to the Neo4j data model shown in Figure 2. One thing to notice is that Neo4j relationships are directional, which helps when processing one-way streets.

For the Neo4j upload, we tested multiple methods such as multi-processing, bulk file imports, and regular looping. We found that the following Cypher query was able to upload over 300k roads in a few minutes. Combined with parallel processing, this method could easily handle any amount of data.

```
"UNWIND $batch AS row "
"MATCH (u:Node {id: row.u}), (v:Node {id: row.v}) "
"CREATE (u)-[:Road {distance: row.distance, road: row.road}]->(v)"
```

However, when scaling up, we found that manually processing the XML file would be too inefficient and time-consuming. We decided to use the OSMnx package to generate our data frame. Using this method, we were able to download Washington State's road network in around 40 minutes. Another benefit of using the package is that it removes the manual step of downloading the XML export locally. In our case, we were able to fit all data in memory but if needed, OSMnx could allow us to distribute this processing across multiple sub-regions.

5 Algorithms

5.1 A* Search Algorithm

5.1.1 Algorithm Overview

A* is a modified version of Dijkstra's algorithm, which is a greedy algorithm that utilizes a min-heap and a closed set of nodes to explore each node in the order from cheapest to reach to most expensive to reach. Because the cheapest path is always explored first, the optimal path is always found. The A* algorithm slightly modifies Dijkstra's by adding an estimated cost (heuristic) to the actual cost when deciding which node to explore next. As long as the heuristic is admissible, A* will also always return the optimal path.

5.1.2 Heuristic Calculation

The heuristic we chose is distance divided by the maximum speed limit.

Because our data is geographic, the locations are in latitude and longitude. This means that we have to consider the curvature of the earth with long enough distances. Due to this, we use the Haversine formula to calculate the distance between two nodes.

$$Heuristic(node, goal) = Haversine(node.lat, node.lon, goal.lat, goal.lon)/max\_speed \tag{1}$$

5.1.3 Heuristic Admissibility

In the context of path-finding algorithms such as A*, a heuristic is admissible if and only if the heuristic never overestimates the cost to a goal node. Our edge weights for the roads are calculated from the distance of that road segment divided by the speed limit, so to ensure that there is no overestimation, we assume the maximum speed limit, and assume we travel straight from the current node to the goal node. Because this is the most perfect path possible by definition (straight line at maximum speed), there is no possibility for overestimation, and the heuristic is admissible.

5.2 Betweenness Centrality

5.2.1 What is Betweenness Centrality?

Betweenness Centrality is a measure used to quantify the importance of an edge as part of the shortest paths in the graph as a whole. A score can be generated which represents the frequency in which a single edge shows up in all shortest paths. This score can help analysts visualize which parts of a graph are most crucial in reducing costs of paths in the entire graph.

5.2.2 Calculating Betweenness Centrality

The betweenness centrality score ($C_B(v)$) of an edge v in a graph is defined by the formula:

$$C_B(v) = \sum_{s \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2}$$

where σ_{st} is is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through edge v

This value was calculated in parallel using Spark. All edges were extracted from MongoDB and put into a Spark RDD. We then took the count of each edge and divided it by the total count of paths. Lastly, the roads in Neo4j were updated with this centrality value where it would then be pulled from the user interface.

6 User Interface

The user interface is made with Leaflet, a JavaScript library providing OSM integration for interactive maps. Node.js was used for the server side to support the integration between Neo4j, MongoDB, and Kafka. The two main features, shown in Figure 3, include plotting the shortest path between two markers and displaying the betweenness centrality for the current map view.

The following Neo4j query takes the coordinates sent from the UI and returns the closest node in the database. The two nodes are then used to query the MongoDB cache and publish to the Kafka topic where it can be consumed and processed by Spark.

```
MATCH (n)
WHERE n.lon IS NOT NULL AND n.lat IS NOT NULL
RETURN n.id
ORDER BY point.distance(
point({ longitude: n.lon, latitude: n.lat }),
point({ longitude: $lng, latitude: $lat })
)
LIMIT 1
```

When the 'Show Centrality' button is pressed, the bounds of the map are sent to the following query to retrieve all edges and their centrality values. To reduce the noise of the data, especially in larger cities, only the top 25% of centrality values are displayed on the map. This helps make the visualization more effective by clearly identifying roads with the highest importance.

```
MATCH (start)-[edge]->(end)
WHERE start.lat >= $minLat AND start.lat <= $maxLat
```

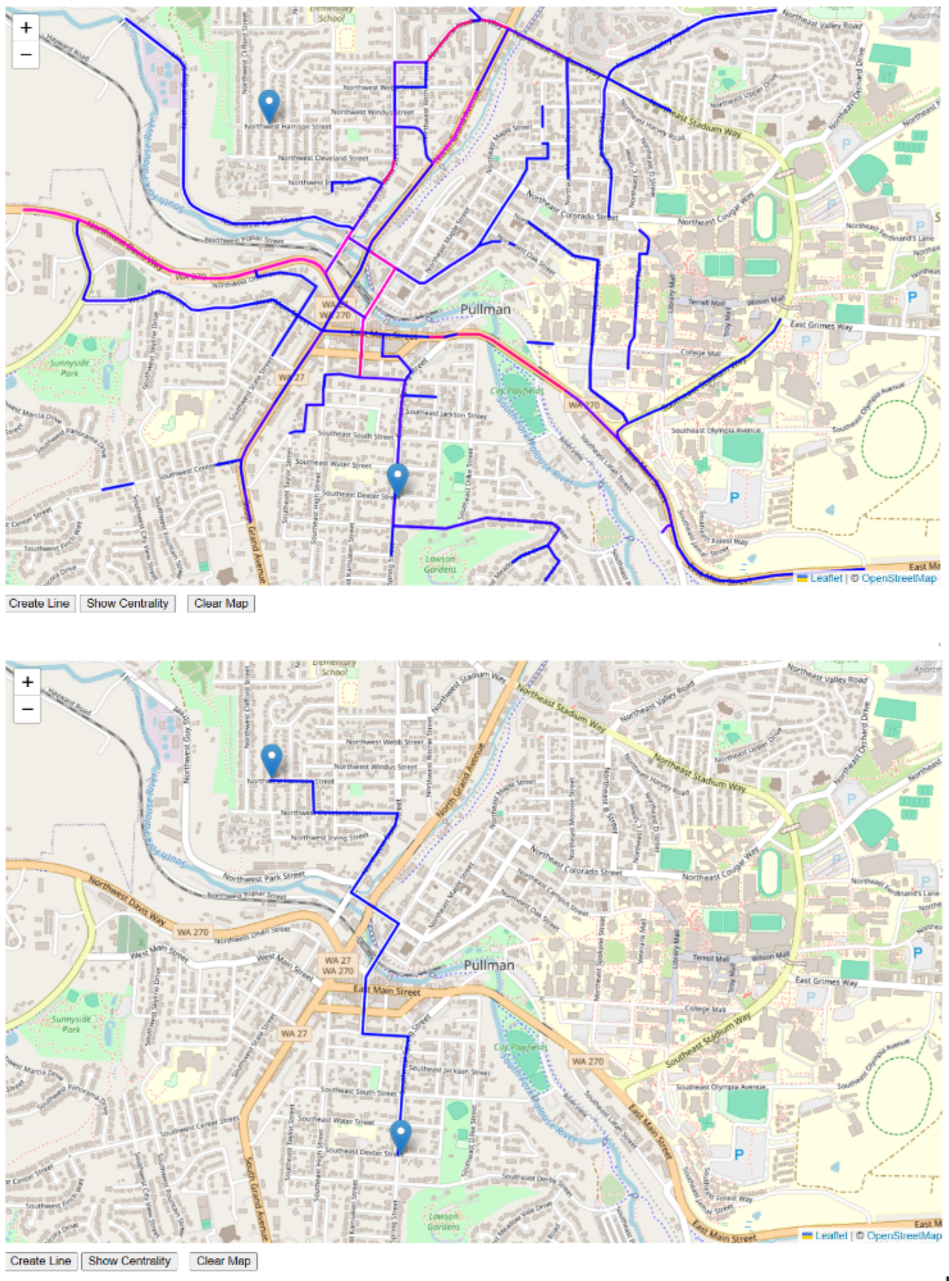

Figure 3: User Interface

```
AND start.lon >= $minLon AND start.lon <= $maxLon
AND end.lat >= $minLat AND end.lat <= $maxLat
```

```
  AND end.lon >= $minLon AND end.lon <= $maxLon
  AND edge.centrality IS NOT NULL
RETURN edge
```

7 Results

Table 1: Data Loading

Location	**Nodes**	**Edges**	**Processing Time**	**Upload Time**
Whitman County	3,491	9,251	28.3s	7.7s
Washington State	375,294	915,112	43m	11.2m

Table 1 shows the results of the Neo4j data loading process and how it scales up to a larger data set. The Neo4j Cypher queries were effective for uploading the nodes and edges for data of any size. However, scaling up to all of Washington became difficult when creating the data frame as this was at the limit of our memory usage. Anything more than that and the data would need to be broken up and processed in chunks. Also, for both cases, the performance comes from single-threaded processing, this data can easily be processed in parallel to handle further scalability.

Table 2: Shortest Path

Algorithm	**Distance**	**Processing Time**
Dijkstra's	5 miles	15.8s
Dijkstra's	10 miles	20.1s
Dijkstra's	40 miles	38.8s
A*	5 miles	1.2s
A*	10 miles	2.3s
A*	40 miles	4.3s

To show the efficiency of the A* algorithm, we ran a comparison with Dijkstra's graph search algorithm. We found that A* was on average 9.5 times faster at finding the shortest path between two nodes. The goal of this project was to find out if it would be possible to develop an algorithm that can support a fast and scalable routing application. With calculations under a few seconds, we can conclude that the integration of Neo4j and Spark with the A* algorithm is effective in providing the shortest paths in a responsive user interface.

Table 3: Heuristic Calculation

Data Size	Processing Time
Whitman County	0.4s
Washington State	20.6s

In table 3, we can see the pre-calculated heuristic processing time. Although the paths are the same, using the full data ends up taking much more time to calculate the distances. This suggests an area for improvement as time is spent calculating distances to unnecessary nodes. One solution to test would be limiting the calculation to nodes within a bounding box of the expected path.

Table 4: Betweenness Centrality

Spark Workers	Processing Time
1	2.3 min
2	1.4 min

Table 4 shows the processing time for the centrality on 17,264 shortest paths with varying levels of computing nodes. Using a Spark cluster with two worker nodes, we found that the calculation was completed in nearly half the time than when only using one worker. This shows the effectiveness of Spark's parallel processing and how we used it to scale up our algorithms.

8 Conclusions

This project set out to explore the scalability of graph algorithms by integrating Neo4j and Spark, aiming to handle the complexities of large road networks efficiently. Through the implementation of optimized algorithms like A*, enhanced by parallel processing of heuristics and path caching in MongoDB, we achieved the development of a responsive application with minimal latency, ensuring a smooth user experience. Despite the positive outcomes, there are further opportunities for enhancement, including the implementation of bi-directional searching, node bounding, and additional resource provisioning to improve processing times.

References

[1] American Trucking Associations. `https://www.trucking.org/economics-and-industry-data`. Accessed: 2023-12-10.

[2] A. Kazerani and S. Winter. Can betweenness centrality explain traffic flow?

[3] H. Wang, S. Lou, J. Jing, Y. Wang, W. Liu, and T. Liu. The ebs-a* algorithm: An improved a* algorithm for path planning. *PLoS ONE*, 17(2):e0263841, 2022.