Project: Mapping Ideal Traffic Routes Using The Open Street Maps Dataset.

Mark Gardner, Mingtian Gong, Malachy Swonger, Anteneh Zeleke

Background:

There are currently many options in place for road mapping and navigation programs, but many of the available solutions lack simple features that may be desired by users such as the shortest distance between destinations rather than time. The goal of this project is to provide some of these metrics in a mapping program through the use of big data, in particular, the two main things we hope to achieve are the shortest path between two places by distance, as well as offering a radial search of places that a user can visit from a starting point with a given amount of time/maximum range from the start. This is an important project because our program will potentially be able to minimize gas used by travelers, saving them both money and emissions cost. Additionally, providing travelers with all the places that they are able to reach in a given period allows them to better make decisions on where to go for vacations and other activities, potentially saving them time and providing convenience.

This is an interesting problem for big data because of the way that it represents real problems that people might have. There are thousands of roads in the United States, and navigation has always been a tough task that has required big data to tackle. Most of the current solutions made by companies such as Google and Waze have optimized commuters routes by time, but not necessarily distance or gas consumption. Additionally, there are very few if any services that allow users to radially search an area for reachable destinations based on criteria. By solving this problem, we can potentially increase the accuracy of things like shipping times from distribution centers by better estimating where a package can get within a specific time period.

Project Details

The project uses the Planet OpenStreetMap dataset, a 110GB compressed file that contains the place and road information for the entire globe (*OpenStreetMap)*. Our project uses subsets of this data preformatted by a Python program to make two files; a list of nodes and locations in latitude and longitude format, and a list of edges weighted by the distance between the two nodes in kilometers. This is accomplished through the use of OSMnx, a python package that contains tools to pull subsets of the OpenStreetMap dataset. This data is then processed in Spark using a Single Shortest Path algorithm.

The first step of our project is to use the OSMnx python package (*OSMNX 1.1.2*) to extract the needed sub-data from the PlanetOSM dataset. The original dataset contains many nodes that are unnecessary to process driving directions. Using the OSMnx program we can specify a specific area of the dataset to use, as well as filter out any data that is not associated with driving. We then load this data into a networkx Graph using the nodes as vertices and adding distance between nodes as a weight for edges. The program then writes a file that contains all the node identification numbers and coordinates, as well as a file containing a list of all the edges and their associated weights. During this pre-processing step, we can also get the nearest point to our origin and target coordinates and convert them into their associated node identification numbers using OSMnx's built-in get_nearest_node() function. Finally, the program plots a chart of the target area graph and displays it for the user to visualize.

In the next step of our project, we process our data and perform calculations to determine the shortest distance between our origin and target points. All of our processing takes place in a Spark program written in Scala. The vertice and weighted edge files from the preprocessing step are used as the input to the spark program along with the node identification

numbers of the origin and target point. The program then reads the files into a Graphx graph, a Spark component that is designed to allow parallel processing on directed graphs. Because the edges from the preprocessing step are unweighted, we need to add each edge in reverse as well as normally for our algorithm to properly function. After our street map graph has been loaded into Graphx, we are able to begin processing the shortest path between the origin and target nodes.

This is done with Dijkstra's shortest path algorithm, an algorithm similar to a breadth-first search modified to account for edge weight. Dijkstra's algorithm iteratively works through the entire graph starting at the origin node and calculates the shortest distance to each node in the graph while marking nodes visited as it works. The algorithm keeps track of the shortest distances to every node as it works, and updates these values as shorter paths are discovered. After all the nodes have been marked as visited, the algorithm will be left with the shortest distance to each node in the graph (*Tyagi*).

Our program uses an adapted version of this algorithm that implements a way to store the actual paths between nodes rather than just distances. Using this algorithm directly on a very large dataset is not possible due to the massive memory overhead required to iterate through the entire graph, but Spark offers an incredibly useful tool to both speed up and lower the memory usage of this search called Pregels, shown in *Figure 1*. Pregels are a system designed by Google to assist in large-scale graph processing. Pregels work by creating an interface that allows for messages to be passed along the edges of graphs (*Karniol-Tambour*). This means that the state for any node in a massive graph can be dependent on its neighboring nodes, something that conceptually already happens in Dijkstra's algorithm.

```
class GraphOps[VD, ED] {
  def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
    // Receive the initial message at each vertex
    var g = mapVertices( (vid, vdata) => vprog(vid, vdata, initialMsg) ).cache()


    // compute the messages
    var messages = GraphXUtils.mapReduceTriplets(g, sendMsg, mergeMsg)
    var activeMessages = messages.count()
    // Loop until no messages remain or maxIterations is achieved
    var i = 0
    while (activeMessages > 0 && i < maxIterations) {
      // Receive the messages and update the vertices.
      g = g.joinVertices(messages)(vprog).cache()
      val oldMessages = messages
      // Send new messages, skipping edges where neither side received a message. We must cache
      // messages so it can be materialized on the next line, allowing us to uncache the previous
      // iteration.
      messages = GraphXUtils.mapReduceTriplets(
        g, sendMsg, mergeMsg, Some((oldMessages, activeDirection))).cache()
      activeMessages = messages.count()
      i += 1
    }
    g
  }
```

(Figure 1) Example pregal implementation copied from Graphx documentation. (*GRAPHX*)

To convert Dijkstra's algorithm into one that uses pregels, all we have to do is pass messages through the graph with the instructions to bring back the shortest distance to a given point, along with a list of all the nodes visited along the way. This message passing and processing allows the graph to be traversed much more quickly than other implementations and brings down the memory overhead significantly, allowing us to process the shortest paths for any point in a city in under a minute. To do this, we adapted an algorithm from a program called *"Graph analysis with Pregel and PageRank algorithms"* (*artem0)*, based on an example pregal problem in the Graphx Documentation (*GRAPHX*).

The output from the processing step is then converted to a string and saved to the disk. The program saves two files, one containing the path of the origin to the target node along with the computed total distance in meters, and another containing the shortest paths for all other nodes in the graph from the origin point. This file is then taken to the post-processing step to be converted to a more readable format that can be accepted into a plotting program to view the route.

The post-processing step takes place in python, again making use of the OSMnx package to get a latitude and longitude value from each node identification number in the path results file from the processing step. The post-processing program then returns a CSV file containing the node identification number, latitude, and longitude for every point in the path, allowing plotting programs like Google Maps to plot a route to compare with other Mapping programs.

The second part of our project is implementing a radial search from a specific point on a map, displaying every node that is reachable from the origin in a specific amount of time driving. This was accomplished by modifying the weights of our preprocessed input data to represent

the amount of time it would take to traverse an edge, rather than the distance. We were able to calculate this by using a property of the OSM dataset called "speed_kph", that represents the maximum speed for any given edge. We then can use the same algorithm from the first part of the project to calculate the shortest paths to every point in the graph from the origin node, weighted by time rather than distance. Then we filter out the values that do not meet the time requirements specified by the user and print the resulting nodes and paths to a results file.

Dataset:

The dataset which we used for this project is the OpenStreetMaps (OSM) PlanetOSM provided by OpenStreetMaps. We used OSMnx (*Boeing*) to pull subsets of this data and filter the results into subsets that represent individual cities.

Planet OSM. The dataset contains the data of all road networks and its size is about 110 GB. We made a program to break up the dataset into sub-dataset of different cities. We can set the name of the city and state to get road data and a sketch graph of road networks of the city. Then we use the sub-dataset as the input file in our spark program to get the shortest path between two nodes. Of note, the data is in an XML format originally, but the OSMnx program gets that data from the website and converts it to a custom file with a weighted edges list and the nodes. Each city dataset contains approximately 25,000 lines once pre-processed to the minimal values. While this file might seem small compared to other datasets, we feel that the computational power required to compute the entire set of permutations of every possible path

would be far too big to be processed on a single node. Spark was uniquely suited to handle this problem.

The Data format of the original OSM file is as shown in *Figure 2*. Each OSMelement contains the version of API, a block of nodes, a block of ways, and a block of relations. Relations are designed for large areas; ways are small area paths; and, nodes are single points.

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
 <bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900" maxlon="12.2524800"/>
 <node id="298884269" lat="54.0901746" lon="12.2482632" user="SvenHRO" uid="46882" visible="true"
version="1" changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
 <node id="261728686" lat="54.0906309" lon="12.2441924" user="PikoWinter" uid="36744" visible="true"
version="1" changeset="323878" timestamp="2008-05-03T13:39:23Z"/>
 <node id="1831881213" version="1" changeset="12370172" lat="54.0900666" lon="12.2539381" user="lafkor"
uid="75625" visible="true" timestamp="2012-07-20T09:43:19Z">
  <tag k="name" v="Neu Broderstorf"/>
  <tag k="traffic_sign" v="city_limit"/>
 </node>
 ...
 <node id="298884272" lat="54.0901447" lon="12.2516513" user="SvenHRO" uid="46882" visible="true"
version="1" changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
 <way id="26659127" user="Masch" uid="55988" visible="true" version="5" changeset="4142606"
timestamp="2010-03-16T11:47:08Z">
  <nd ref="292403538"/>
  <nd ref="298884289"/>
  ...
  <nd ref="261728686"/>
  <tag k="highway" v="unclassified"/>
  <tag k="name" v="Pastower Straße"/>
 </way>
 <relation id="56688" user="kmvar" uid="56190" visible="true" version="28" changeset="6947637"
timestamp="2011-01-12T14:23:49Z">
  <member type="node" ref="294942404" role=""/>
  ...
  <member type="node" ref="364933006" role=""/>
  <member type="way" ref="4579143" role=""/>
  ...
  <member type="node" ref="249673494" role=""/>
  <tag k="name" v="Küstenbus Linie 123"/>
  <tag k="network" v="VVW"/>
  <tag k="operator" v="Regionalverkehr Küste"/>
  <tag k="ref" v="123"/>
  <tag k="route" v="bus"/>
  <tag k="type" v="route"/>
 </relation>
 ...
</osm>
```

(Figure 2) Example XML data format from PlanetOSM dataset. (*OpenStreetMap*)

Planned Timeline

Week	Task	Team Member
Week 1	Finalize project proposal	Malachy
Week 2	Initial formatting setup	All
Week 3	Initial formatting setup	All
Week 4	Task 1: Shortest Distance -Setup	Mingtian Mark
Week 5	Task 1: Shortest Distance -Finished	Malachy
Week 6	Task 2: Radial Search -Setup Report Writing -Outline	Anteneh Malachy
Week 7	Task 2: Radial Search -Implementation Report Writing -Some data	Anteneh Mingtian Mark
Week 8	Report Writing	Anteneh

	-All data -Finished Slideshow	Mingtian Malachy Mark

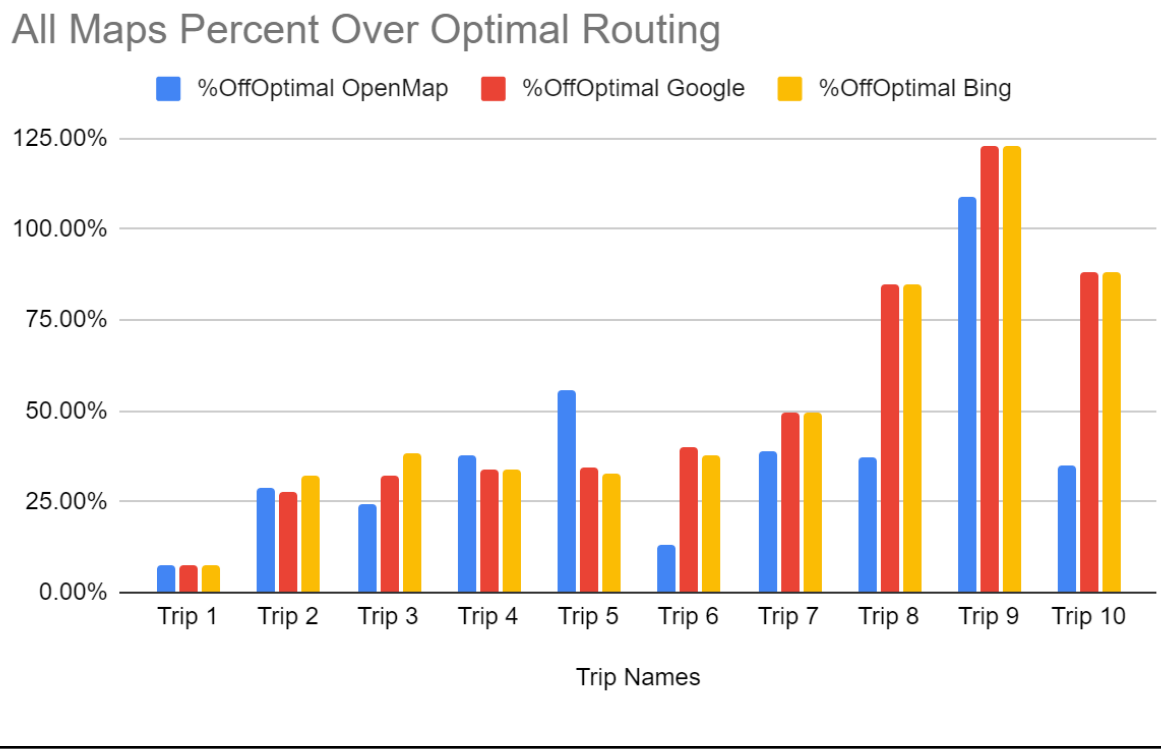
Analytics

As mentioned earlier, the shortest path algorithm was used to calculate the distance between two locations. Our shortest path program takes longer to execute because it is calculating the shortest path of one node to every other node in the dataset. The average time it takes for our program to calculate the distance between one node to another is 5 milliseconds, and a total of 40 seconds for the program to run a job. Whereas companies like Google have a more refined way of approaching the shortest paths through means of using a storage mechanism that isn't accessible for this project. This causes time latency in our program compared to our program's results.

Mainstream mapping platforms like Google Maps and Bing Maps were used to track the accuracy of our program's output. Accuracy is measured by comparing the path of the program and the output path of one of the mainstream mapping programs. This comparison is interpreted as the percentage offset from the "optimal route", again referring to the output path of Google Maps, for example.

Our algorithm's performance was impacted by the size of the city. For instance, our algorithm was unable to run complete datasets for Denver and Boulder, though we were able to map smaller regions of each of the cities that were unable to run in their entirety. Cities such as Fort Collins and Steamboat were both able to be mapped. Another complication arose from incomplete datasets. In some instances, our OSM dataset didn't have nodes or edges found in both Google and Bing's data, which caused our algorithm to perform relatively poorly. Complicating our analysis even further, given that Google selected different shortest paths between the same nodes over different days, the accuracy of our analysis is complicated by the variability of Google's algorithm on any given day.

One trend is evident from the table: our algorithm performed better on shorter routes. The histograms included below contain each pairing of nodes, representing a route, and the percentage of distance above the optimal path for each of the three algorithms. We attributed this to the naivety of our algorithm, meaning that it ignores all traffic laws like one-way roads and illegal left turns. Additionally, Google appears to deliberately re-route traffic away from neighborhoods, whereas our algorithm will include routes through neighborhoods simply because they are shorter. Also, Google appears to have hubs through which their algorithm routes traffic. Presumably, the viability of the location of these hubs is a product of frequent and high volumes of traffic through those points.



(Figure 3)

The graph in *Figure 2* represents our program's shortest path algorithm in comparison to Google Maps' shortest path algorithm and Bing Maps' shortest path algorithm. The y-axis represents the percentage above the optimal distance between points. Each group of data on the x-axis represents a trip with a starting and endpoint. On average, our program performed approximately 14% better than both Google and Bing maps in regards to distance, albeit taking much more time to process.

Challenges we faced:

Reading in our data was perhaps our greatest challenge. Initially, we intended to use MapReduce to implement a breadth-first-search algorithm, but we quickly realized that this path didn't work, though we didn't find a solution for some time. Ultimately, we decided to use scala and spark to create a Dijkstra's algorithm implementation to find the shortest route. Choosing

this path presented different challenges, as our team had differing levels of familiarity with both scala and spark, with one team member being unable to run spark commands until the final week of the project.

The inherently complicated nature of this project created a separate set of challenges. We had to write various other programs to get the data from OSM, then to process the data into a format that we could work with, for instance we used a python program to map the nodes and paths of the entire dataset, thereby rendering a map of each city we considered.

Another aspect of this project that was challenging was optimizing our program to run on larger subsets of the Planet OSM dataset. Because of the need for our algorithm to iterate through every vertex of the dataset graph, the computational complexity of the program grows extremely quickly as the search area increases. Our first attempt at solving this problem used a standard Dijkstra's Algorithm approach designed for smaller datasets. This algorithm was unable to perform a search in any mid-sized city and required a significant amount of time to run. By optimizing our algorithm to take advantage of Pregels, we were able to vastly improve the search radius and cut down run times significantly. Using the pregal search, mapping major cities becomes possible with increased memory and processing power, but larger datasets such as states and countries are still beyond this level of optimization. We agree that it may be possible to adapt our approach to continuing expanding our search radius by trimming unlikely edges from each graph at runtime or using drivers' data to find more used routes, but we feel that these solutions are beyond the scope of this project.

Contributions of Each Person

Anteneh - Research on data collection and shortest path algorithm

Mark - Research on data collection and shortest path algorithm; helped with reviewing and debugging algorithm and code; wrote portions of the final paper.

Malachy - Implementation of algorithm, pre/post-processing, and Radial Search in scala and compilation of results

Mingtian - Research on data collection and try to process the data in a framework. Helped with reviewing the shortest path algorithm.

Works Cited

1. Boeing, G. 2017. OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks. Computers, Environment and Urban Systems 65, 126-139. doi:10.1016/j.compenvurbsys.2017.05.004
2. “GRAPHX Programming Guide.” *GraphX - Spark 3.2.0 Documentation*, https://spark.apache.org/docs/latest/graphx-programming-guide.html#pregel-api.
3. Karniol-Tambour, Orren, et al. *Lecture 8, Distributed Algorithms and Optimization*. 22 Apr. 2015, https://stanford.edu/~rezab/classes/cme323/S15/notes/lec8.pdf.
4. “OSMNX 1.1.2.” *OSMnx 1.1.2 - OSMnx 1.1.2 Documentation*, https://osmnx.readthedocs.io/en/stable/.
5. *OpenStreetMap Wiki*, 19 July 2020, https://wiki.openstreetmap.org/wiki/Main_Page.
6. artem0. “Graph Analysis with Pregel and PageRank Algorithms.” *GitHub*, 12 Aug. 2017, https://github.com/artem0/spark-graphx.
7. Tyagi, Neelam. “What Is Dijkstra's Algorithm? Examples and Applications of Dijkstra's Algorithm.” *What Is Dijkstra's Algorithm? Examples and Applications of Dijkstra's Algorithm*, 13 Dec. 2020, https://www.analyticssteps.com/blogs/dijkstras-algorithm-shortest-path-algorithm.