CrowdPath: Inferring Shortest, Fastest, and Safest Path from Volunteered Geographic Information

Abdeltawab M. Hendawi  
Eugen Sturm

hendawi@cs.umn.edu  
sturm049@umn.edu

Objective

For our project, we would like to obtain GPS tracks from volunteers. From these tracks, we want to extract valid tracks or routes that could be used in finding faster routes than the conventional Google, Bing, and MapQuest maps would provide. From those volunteered tracks we can build our own road network graph with real time and distance estimation for each edge at each time slot to the day. After that we plan to add a risk value for each edge in our road network graph such that it can be used to infer the safest path which is the one with the least risk.

Current status

- We obtained some GPS tracks from OpenStreetMap and parse, and store them as set of text files. Each track has set of GPS readings with lat, long, and time, e.g. (44.9779772, -93.2455653, 2012-09-30T19:42:19Z, 70939).
- We compared the total travel time and distance for the trips in these tracks against what we get from Google maps.
- We did query Google to get the corresponding routes for what we have in the valid GPS tracks.
- We found that a good percentage of the tracks we have beat Google either in travel time cost, total distance cost, or both.
- We designed the CrowdPath system architecture and started the implementation of its main modules.
• We investigated the existing data structures that qualified to be employed in our system, and adapted the most suitable one (time aggregated graph) to be able to fulfill the needs of CrowdPath.

• Currently, we are investigating the consideration of other sources of data and build our own road network graph with the new travel time cost and distance obtained from the GPS tracks.

• Currently, we still are working to integrate the data about car accidents and crimes to our system, so CrowdPath being able to answer all safest, shortest and fastest path queries.

• Currently, according the advice by the professor and the TA, we intend to complete this work for publication.

Challenges

• Some challenges would be if the location we choose to analyze doesn’t have many GPS traces available. For example, we downloaded GPS traces for the Minneapolis area from OpenStreetMap using a program calls JOSM and were only able to obtain 115 GPS tracks for all of Minneapolis. So we either need a method to collect more data or find an area that has significantly more GPS trace in a somewhat concentrated area.

• Some more issues related to the data itself. When collecting these GPS tracks, they will not be map matched so that is a task we will have to do ourselves.

• Additionally, when exploring the gpx files representing the GPS traces, some of them do not provide additional information other than each GPS point’s latitude and longitude. Since we are also using the time metric to compare routes, this means that traces like these will have to be thrown out.

• We have limited number of queries to send to Google Map per day. So we need to have a facility to increase this limit such that we can do comparison with all start-end trips combinations we extracted from the GPS tracks.
System Architecture

As shown in Figure (1), the CrowdPath system consists of two main modules namely, the data analysis and maintenance module, and the query processing module, and a shared data structure that holds the road network graph with the extracted weights for different attributes, travel time, distance, and risk. The data analysis module is responsible for extracting the GPS tracks from the volunteered data source. Then it compares each GPS track to the equivalent one obtained from the mapping and routing providers including Google maps, Bing Maps, MapQuest. The result of comparison is either to find that the volunteered track has less travel time, travel distance, or both, then we cash this track for further queries with the same source and destination. At the same time we update our road network graph by correcting the weights of the participating edges with the correct travel cost at the corresponding time slot.

Figure (1), The System Architecture for CrowdPath
Data Structure

We studied the existing techniques for storing the extracted road network graph and being able to support a wide variety of queries such as shortest path, fastest path, and/or safest path. It is important that each of the extracted travel cost elements such as distance, time, and risk can have different weights according to the time of the day. We found that there are two common qualified alternatives, namely, time series graph Figure (2), and time aggregated graph Figure (3). Although the later saves a lot of the storage cost and computation time consumed be the former, however it still suffers from redundancy which can be removed by introducing a new data structure called attributed time aggregated graph (ATAG) Figure (4). ATAG is an adapted version of time aggregated graph where each edge can have many attributes each of them can have many weights according the time slot of the day. Obviously, ATAG is the best in reducing the storage cost and accordingly computation time. Moreover, it is the only data structure that can support queries with multiple criteria (safest and fastest path) through a single graph traversing.

![Figure (2a), Time Series Graph for Distance Attribute](image)
Figure (2b), Time Series Graph for Travel Time Attribute

Figure (2c), Time Series Graph for Risk Attribute
Figure (3), Time Aggregated Graph for Each Attribute

Figure (4), Attributed Time Aggregated Graph (ATAG)
**Implementation**

For implementation, we decided to try and show that VGI data has the capability to beat optimal route given by Google and other mapping services. Our implementation consisted of 3 steps. First, we had to find and obtain a large enough VGI dataset that we would be able to easily find routes that were more optimal than routes provided from online mapping services. We then had to collect a large amount of data on our VGI dataset from the online mapping services and store it in data structures that would allow for easy analysis and comparison between the VGI data and the online data. Finally, using the data structures we stored the VGI and online mapping services data in, we then had extract routes where the VGI data was more optimal and draw the routes on a map to visualize it.

**Data Collection**

To collect a large amount of VGI data, we used OpenStreetMap (OSM). Using a client application called JOSM, Figure (5), we were able to query OSM to return tracks for areas specified by a bounding box.

Our chosen target area was Minneapolis, which we were able to extract a dataset of over 150 tracks. The tracks we extracted were provided to us in a gpx file format where each track was broken up into track segments of gps points that gave us a latitude, longitude and timestamp. Some tracks in the dataset did not have time stamps associated with their gps points, so these tracks were useless to us and thrown out. This was only the case for about 10% of the tracks we extracted from OSM.

Now that we have a decent VGI dataset, we had to decide how we would best query online mapping services to retrieve their data for the routes that could be represented by the VGI tracks. There were many possible methods to do this. We first tried to partition each track into sub-tracks of a specified interval, such as 5, 10, or 15 minute, and run queries on these sub-tracks.
Although this method did provide us with results, it was far from optimal because it didn’t take into account all possible routes that could be represented by a track. For example, it would not return data about the route from \( P_1 \) to \( P_4 \).

![Figure (5), Extracting VGI data from OpenStreetMap using JOSM](image1)

Although this method did provide us with results, it was far from optimal because it didn’t take into account all possible routes that could be represented by a track. For example, it would not return data about the route from \( P_1 \) to \( P_4 \).

![Figure (6a), Partitioning a track into 4 points and obtain distance and duration data for each sub-track (\( P_1 – P_2 \), \( P_2 – P_3 \), \( P_3 – P_4 \))](image2)

![Figure (6b) Extracting all possible sub-tracks from a track of four points](image3)
To overcome this issue, we next chose to again partition our tracks in the same manner, break the track up into sub-tracks by extracting points from the whole track using a specific interval. Next, instead of querying online mapping services as we did before, we chose to query every possible sub-track by selecting each point as a possible start location for a route and every subsequent point in the track as a possible destination for a route starting from the previously mentioned start point. Although this would create a lot of overlap within our dataset, it was necessary to ensure that we all of the VGI data was being used to its fullest potential, Figure (6).

**Building CrowdPath Measurements Matrix**

Using the partitioning method previously mentioned, we then had to obtain the route information from online mapping services so that we could compare it with the VGI data. For the purpose of our project, we only targeted Google Maps because of its prominence. To obtain distance and duration data for routes from Google, we had to use their distance matrix web API. To use this API, you must provide it with a set of origin points and destination points. For this project, we represented these points as latitude and longitude values, but in the API, the points can be represented in other ways. Using these origin and destination points, the API will build a matrix where the each row corresponds to an origin point, each column corresponds to a destination point and each element in the matrix is a distance and time pair from an origin point to a destination point.

Based off of the distance matrix returned by the Google Web API, we built a modified distance matrix to store the distance and duration values returned by the Google Web API and the durations extracted from the VGI track GPS point timestamps for all possible routes in a track. For our modified matrix, we used all the points we extracted from a track as the origin points and destination points in our matrix while still storing the data returned from the Google Distance Matrix as the elements in the matrix. Creating a matrix such as this to represent a single track in
the VGI data gave us an easy to use data structure to compare the times between Google and the VGI dataset for all possible routes in a track.

Figure (7a) Matrix returned by a distance matrix API request with N origin points and N destination points.

Figure (7b) Modified distance matrix using extracted points from a track as origin and destination points to store Google route data alongside the VGI partitioned data.
**Visualization**

With our modified distance matrix compiled, we can easily search through all of the tracks a find routes that exist in our VGI data that have faster times between points that what Google Maps would provide. The next step of our implementation was to draw the routes that beat Google’s time on a map. Now, the VGI data that we extracted from OpenStreetMap was just a list of gps points that had not been map matched. So, in order to draw the route of the VGI data, we would have to map match the points. To do this, we used another Google Web API, the directions API. When using the directions API, you provide a single origin and destination point in the query and the API will return a set of direction for the route between those two points. Included in the response for the query is an encoded polyline. This polyline can be decoded into a list of map matched gps points.

Using the directions API, we built a polyline to represent routes from our VGI data that had faster times than what Google would provide. To build the polyline, we had to send directions API requests between each point in the VGI track, extract the polyline for each sub-route from the web response, decode each encoded polyline into a list of points, combine each sub-routes decoded polyline points into a single list of points, the re-encode the polyline. Through the process of conducting these directions requests, we can accumulate the distance of each sub route being return to gather an estimate for the total distance of the VGI data route. Since, the distance of the tracks is not provided in the gps files when we initially collected the data, this was the only method we could use to estimate the distance of the routes.

This process can successfully build a decent looking polyline, but there are a couple inherent problems with this method. The main issue is since the gps points from the VGI data are not map matched to begin with, the sub route returned by the directions requests may not be the correct sub route for the route as a whole. This causes noise in our polylines which, although not too severe, made them seem to veer off course temporarily. Additionally, if you recall how we estimate the distance of our routes, this noise added to the polylines also inflates the estimated distance of our routes. Because of this, it was more difficult to find routes that beat Google in time and distance.
Results & Statistics

To show the results, we extracted 5 example route which we all beat Google in time from origin to destination. Only one route beat Google by both distance and time, which as previously stated was more difficult to do. In the 115 tracks of VGI data we built distance matrices for, we searched them for sub-tracks that were a minimum of five minutes long and beat Google’s optimal route by at least 100 seconds. Searching our data, we found 130 routes that matched these constraints. Of these 130 routes, only 13 routes beat Google’s optimal route in both distance and time. The average time saved for each route was 170 seconds. In all of the following images, the blue route is Google’s optimal route and the red route is the route we generated to represent the VGI data.

Figure (8), Examples of noise created by the polyline generation process.
Figure (9a), Google’s Time: 19 min  
CrowdPath Time: 14 min

Figure (9b), Google’s Time: 16 min | Google’s Distance: 13.8 mi  
CrowdPath Time: 14 min | CrowdPath Distance: 11.3 mi

Figure (9c), Google’s Time: 6 min  
CrowdPath Time: 3 min

Figure (9d), Google’s Time: 7 min  
CrowdPath Time: 3 min
Figure (9e), Google’s Time: 6 min
CrowdPath Time: 4 min