Implementation of Vector based map-matching algorithm

Supervisor Amnir Hadachi

Joosep Kibal
Institute of Computer science
university of Tartu
truesigh@ut.ee

Abstract – Map matching is the process of aligning a sequence of observed user positions with the road network on a digital map. It is used in many applications such as moving object management, traffic flow analysis and driving directions. A number of map-matching algorithms have been developed by researchers around the world using different techniques such as topological analysis of spatial road network data, probabilistic theory, Kalman filter, fuzzy logic, and belief theory. In this project I implemented the Heuristic Map-Matching Algorithm by Using Vector-Based Recognition described in[1]. The aim of the project was to implement the algorithm and see if I could achieve similar results as reported in [1]. The source code can be found in Bitbucket repository [7].

1. INTRODUCTION

In Intelligent Transportation Systems (ITS) "Floating car" or "probe" data collection is a set of relatively low-cost methods for obtaining travel time and speed data for vehicles traveling along streets, highways, motorways (freeways), and other transport routes there are manly three different methods used to gather raw data [2]:

- Triangulation method. In developed countries high proportion of cars contain mobile phones. The phones transmit their presence to mobile phone networks. As the car moves so does signal of the phone and by using triangulation, pattern matching or cell-sector statistics the data is converted to traffic flow information.
- Vehicle re-identification. Vehicle re-identification methods require sets of detectors mounted along the road. A unique serial number for a device in the vehicle is detected at one location and then detected again (re-identified) further down the road. Travel times and speed are calculated by comparing the time at which a specific device is detected by pairs of sensors.
- GPS based methods. Vehicles are equipped with GPS systems that have two-way communication with a traffic data provider. Positioning readings are used to calculate vehicle speeds. Modern solutions may use GPS equipped smart-phones.

The main advantages of Floating Car Data (FCD) are that it is less expensive than sensors or cameras, it has more coverage, is faster to set up and works well in all weather conditions.[2]

The map-matching is the procedure of comparing the vehicle tracking data and the digital road map, with the purpose of matching the vehicular positions to the road on which the vehicle actually had driven. When using FDC the GPS data is not precise so it is possible that one GPS location can be matched to several road segments. Also there can be a sampling error caused by the sampling rate [1]. So the Heuristic Map-Matching Algorithm would not only produce good matching results, but also would produce results fast compared to the traditional map-matching algorithms.

2. RELATED WORKS

Several map-matching algorithms are surveyed by Quddus in [3]. Also a great table for some of the different algorithms can be seen from [4] Some of the examples include: Point-to-Curve matching with heading (White et al. 2000), Curve-to-Curve matching (Bernstein and Kornhauser (1996) White et al. (2000) Taylor et al. (2001)), Similarity criteria by weighting system (Greenfeld, J.S. (2002)).

3. ROAD NETWORK DATA AND STRUCTURE

- Node

In digital road map, the road is kept as a line object which is essentially a series of points. If the points are connected then a road network can be shown. Connectivity nodes are intersections of roads, the beginning and the end of a road and the points where vehicles can turn. Other points which does not belong to connectivity node are called common nodes.

- Road segment

If there is a directional pathway between two adjacent nodes then this path is defined as a road segment. Each segment has two nodes which are beginning node and ending node of the segment.

- Link

If there is a directional path between two adjacent connectivity nodes, then this path is defined as a link

and the connectivity nodes are respectively the beginning node and ending node of link. Each link can be connected to one or many road segments. From figure 1 we can see how the road network is formed. We can see that nodes are connected to each other by directional links and if two nodes are connected only by one link then the road is one-directional road but if they are connected by two links which have the opposite direction then the road would be two-directional. From figure 1 we can see that in this example all the nodes are connected by only one link to each other so the road is one-directional.

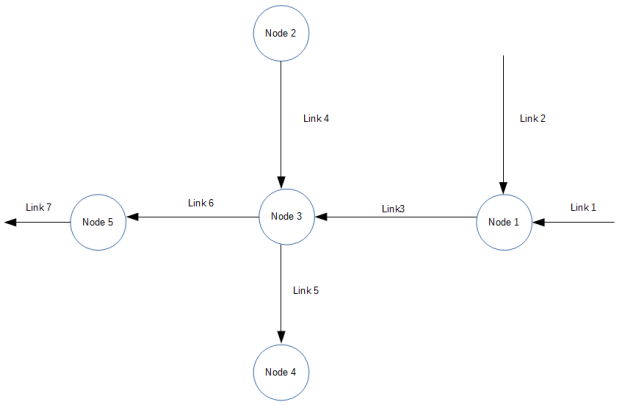


Figure 1 Topology of the graph

The map data is read in from an Open Street Map map file (osm) The osm file is in XML format and is parsed through to get the intersections. Between intersections road segments are made where one intersection is the beginning node and the other intersection is the ending node. Each segment has a list which consists of the segments that this segment is connected to. For example if we take a look at figure 1 then Link 3 (segment 3 is connected to Link 5 (segment 5 and Link 6 so segment 3 has segment 5 & 6 in its list of connected segments (links). After intersection nodes are parsed in the map file is again parsed over. This time the map segments (links) are created by getting all the ways marked in the Street Map file. Next only relevant roads are read in (currently only one or two-directional highways are considered to be relevant). Then for each highway all the road segments are created that are on that highway. In figure 2 we can see how the road segments are created. The painted dots mark the start and end node of the road. The different colors mark different roads. The the places where Links connect to the next link are intersections (connectivity nodes). In figure 2 we see that when parsing the blue road which has 6 nodes in it we start from the starting node and sequentially create all road segments on this road. For example Link1 → Link2 → Link3 → Link4 → Link5. If the road is two-way then reverse directional segments are created at the same time. When we are finished with one road we move on to another one until all the roads are parsed through.

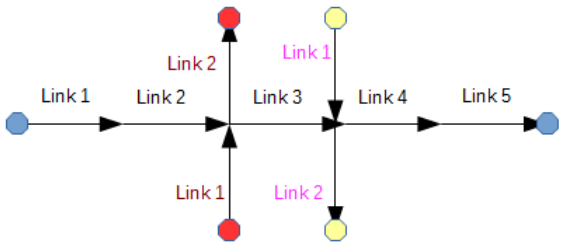


Figure 2 Example of road segment creating

By iterating over the map file and parsing through all the nodes and creating the segments, the road graph is created. After the creation of road segments we connect all the segments together to create the road graph. This approach takes less space because if we take the road network as a graph of road segments then it would be a sparsely populated graph since any randomly picked road segment is connected to small amount of other segments due to real world conditions. Also we do not to store the nodes since after creating the segments we do not need to use the nodes again.

The downside of the approach is that for any larger map the parsing of the Open Street Map map file takes a lot of time. (with Intel i3-4130 & 8 GB of RAM it took 31 minutes to parse in the map of Tartu). The main reason for this is that the map.osm file contains a lot of nodes that are irrelevant for creating a road map as they depict railroad, pedestrian roads or other nodes. The reason for this is that since I used Java's documentBuilderFactory which produces DOM object trees from XML documents. In order to parse through the whole map file it has to go over each of the nodes and while each individual node takes between 20000-30000 nanoseconds to parse even for a small map there are over 13000 nodes so it adds up to ~31 seconds. Currently I'm trying to use the Osmosis tool for faster extraction but have to verify some things before I can use it.

3. MAP-MATCHING ALGORITHM

The input is a series of GPS tracking data on some vehicle in some time. Lets mark it with G = {g[1], g[2], …,g[n]} The GPS data consists of latitude and longitude coordinate, vehicle traveling direction and the time-stamp of GPS sampling. So g[i] = (X_i ,Y_i ,V_i, T_i). Where X is longitude, Y is latitude, V is traveling direction and T is time-stamp.
I used the Haversine formula to calculate the distance between two GPS coordinates.

- Haversine formula:

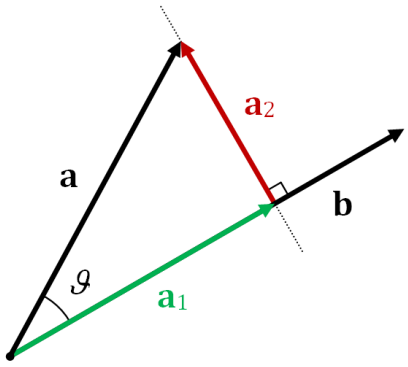
$a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$ then $c = 2\,\mathrm{atan2}(\sqrt{a},\sqrt{(1-a)})$ and $d = R \cdot c$. Where φ is latitude, λ is longitude, R is earth's radius (mean radius = 6372.8km);
Since I used the Haversine formula only in the final mapping steps when I had to determine if the projected GPS point was on the suitable road segment then for most of the distance calculation I used simple distance between two points formula.

- Distance between two points

The distance between two points P1 and P2 is Distance(P1, P2) = $\sqrt{(x1-x2)^2+(y1-y2)^2}$
where P1(x1,y1), P2(x2,y2)
For finding map-matched points I used vector projection where I created two vectors between the suitable road

segment and between the segments start point and GPS point.

- Vector projection



Drawing 1: Vector projection

From drawing 1 we can see how vector projection works. In my solution the vector b is the segment vector and vector a is the vector from the starting point of the segment till the GPS point.

$$proj_v u = \frac{\vec{u}\cdot\vec{v}}{|\vec{v}|}\frac{\vec{v}}{|\vec{v}|} \qquad proj_v u = \frac{\vec{u}\cdot\vec{v}}{|\vec{v}|^2}\vec{v}$$

Formula for vector projection [5]

The vector projection is the unit vector of $\vec{v}$ by the scalar projection of u on v.

- Vector $\overrightarrow{P1P2}$

To make a vector we need to take two points P1(x1,y1), P2(x2,y2) where x is latitude and y is longitude and make a line segment between them which has a direction and length. P1 is the beginning of the vector and P2 is the ending.

- Angle between two vectors:

$$\cos\alpha = \frac{u_1\cdot v_1 + u_2\cdot v_2}{\sqrt{u_1^2+u_2^2}\cdot\sqrt{v_1^2+v_2^2}}$$

Formula for angle between vectors[5]

Where u and v are vectors and to get the angle in degrees had to convert the cosine a to degrees.

The algorithm matches the first GPS point in G separately. It searches for nearby links to the GPS point and if the distance between the projected GPS point on that link and the actual point is less than the map-matching error (15-30m) then it adds the points into an array of possible starting map segments. Next taking the map-matching point which we found previously as the starting point and the next GPS point as the ending point of the GPS Vector. Then the length of the link (or map segment) on which the map-matched GPS point was on is compared to the length of the Vector created between the map-matched point and the next GPS point. Then there are two cases:

- Case 1- the length of the vector is less than of the map segment. In this case it is easy, we have found that the next GPS point is on the same segment and proceed by finding the map-matched point on the segment (projecting the GPS onto the link).
- Case 2 – the length of the vector is more than the length of the segment. Then there are two choices: vehicle going straight or vehicle turning.
  First due to the theorem of trilateral relationship of triangle: the sum of length of any both sides of the triangle is longer than the third one.

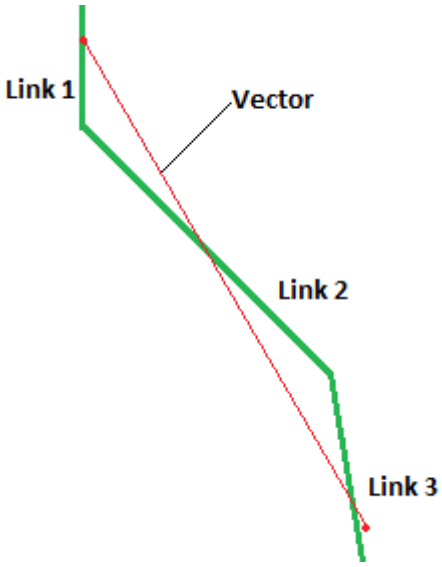


Figure 2: Example of trilateral relationships

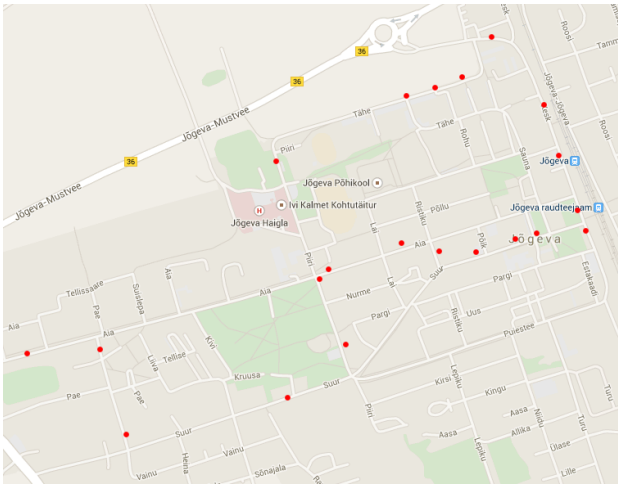
- Next we will check that the angle between the chosen link and the vector is less than 90 degrees.
- Next the length from map-matched point to the GPS point via map segments (links) is less than two times the length of the vector between map-matched point and GPS point.

If we select the link that satisfies restriction 1 and is subsequent to the link that the map- matching point for the previous GPS point is on. Then if the map-matching result can be successfully gotten, we can say that the car may have driven on this link (road segment). If we cannot get the map-matching point and the link can satisfy restrictions 1 and 2 then we will continue following along the links until we can get a map-matching point or run out of links that satisfy restriction 1 and 2

4. TESTING THE ALGORITHM

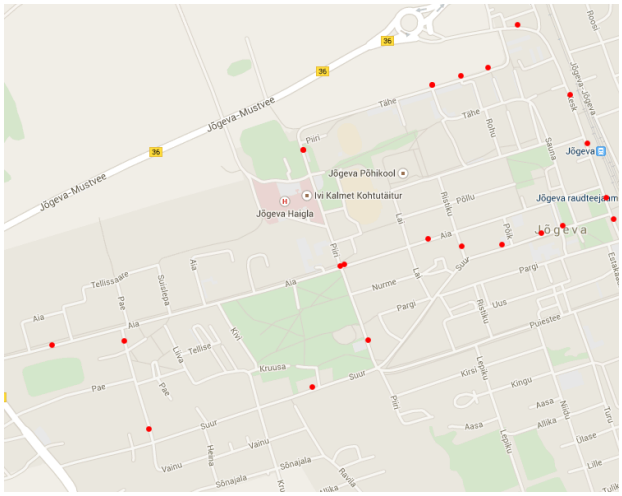
Due to time restraints I have not had the time to test on real world collected data. Currently I have tested by

creating a sequence of semi-randomly picked points from the map as to simulate the GPS points. By semi-random I mean that the route is created by selecting random road segments (links) in some logical order. Meaning that the simulated car is moving coherently not jumping around all over the town (For example the next GPS point is one or several segments away not at the other side of town). So far I can see that the algorithm manages to match the points to road segments quite well if GPS point does not position near an intersection. If the matching gives multiple possible points for one GPS then for each point we assign a weight to each matched point and after matching the most likely route is calculated.
With small maps and small number of points the algorithm seems to work quickly – taking 130 ms on average to match the points.



Drawing 2: GPS coordinates before matching

From drawing 2 we can see that compared to drawing 3 some points are 5-15 meters off the road. Drawing 3 shows how the GPS points are mapped suitable segments.
Time it takes to match 30 GPS points is around 100 – 200 milliseconds. Increasing the number of points does not increase the computation time so much because during the matching process it takes the most time to match the first point(s) as we need to iterate through all the links to find the first suitable link(s). Where as during the matching process for each GPS point we need to iterate through only a small number of links until we reach the next GPS point.



Drawing 3: Map-matched GPS coordinates

5 FUTURE DEVELOPMENTS

There were several features that were considered but due to time constraints were not implemented or are not yet finished.

- Graphical user interface and self generated map.
- Improvement of the input file parsing. Currently the most time is used to read in the map. If this functionality could be improved the speedup should be significant already with small maps we can see that parsing in the map takes ~95% of the total execution time.
- Making the algorithm or parts of it parallel. Because map can be divided into bounding areas then it should be possible to divide up the work also GPS coordinates could be per-processed to see if they could be divided up for parallel matching.
- Improving the current code. Since during the implementation there were several larger changes then there might be some unnecessary data that could be removed.
- Adding a database to hold road graph data for faster processing.

6 CONCLUSION

I managed to implement the vector based map-matching algorithm. As I had no time to test it against real-world data I cannot say if my implementation achieves the 90% accuracy rate given in [1]. Currently in my implementation if it finds a suitable segment for matching the GPS point then it also checks if some other links that were connected to the previous segment before the matched segment are suitable candidates for matching and if not the it proceeds to next GPS point. The missing feature that I'm trying to add but ran into some difficulties is the correct route calculation. I'm using a tree like structure to store the candidate car moving paths and am also assigning a weight for each mapped point and each segment is containing the GPS

points matched to that segment but parsing the tree and calculating the most likely path is not operational yet. I am using hamstermap[6] for displaying the multiple coordinates on one map. The problem with hamstermap is that it is not accurate to OpenStreetMap map. So actually the coordinates are matched correctly on the road but Hamstermap may display them slightly to the side of the road (have verified this with OpenStreetMap and they are indeed matched on the road not beside the road.). At the current state it seems that the different paths are created correctly (at least when I check it against the map) but the output processing is not working yet.

6 REFERENCES

1. Dongdong Wu, Tongyu Zhu, Weifeng Lv, Xin Gao. A Heuristic Map-Matching Algorithm by Using Vector-Based Recognition
2. http://en.wikipedia.org/wiki/Intelligent_transportation_system#Floating_car_data.2Ffloating_cellular_data
3. Mohammed A. Quddus, Washington Y. Ochieng , Robert B. Noland. Current map-matching algorithms for transport applications: State-of-the art and future research directions
4. Quddus, Mohammed A. Ochieng, Washington Y. Zhao, Lin Noland, Robert B. A general map matching algorithm for transport telematics applications
5. http://www.vitutor.com/geometry/vec/angle_vectors.html
6. http://www.hamstermap.com/quickmap.php
7. https://truesight@bitbucket.org/truesight/ds-mapmatching.git