Improving the implemented Vector based map-matching solution

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 improve the implemented algorithm which was based on[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 IN THE SOLUTION

- 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.

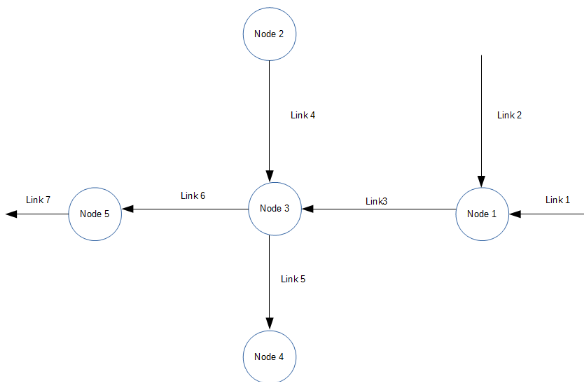


Figure 1 Topology of the graph

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. 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.

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.

4. IMPROVING INPUT READING

The downside of the previously mentioned approach was 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 and create an object of then even for a small map there are over 13000 nodes so it adds up to ~31 seconds.

How I decided to improve this was by trying out different third party .osm file parsers and integrating them into my project. The first parser tried was Osmosis[8] which is a command line Java application for processing OSM data. Osmosis consists of pluggable components that can be chained to perform a larger operation. For my project it had two important features: extraction of data inside bounding boxes and extracting data based on certain types of tags. In this project I used the Osmosis tag-filter parameter to extract all highways using the following command: *osmosis --read-xml map.osm --tf accept-ways highway=* --used-node --write-xml highways.osm,* where *--read-xml* defines that the input file is a xml file, *map.osm* is the input map file downloaded via Overpass api. The *–tf accept-ways* parameter filters which types of ways are to be read from the input file and *–used-node* parameter is used to extract only those ways specified by the filter parameter. The *–write-xml* parameter specifies the type of the output file and *highways.osm* is the output file name. In case of small towns the time taken by extraction was between 200 – 600 millisecond. In case of medium sized town it took between 3000 and 7000 milliseconds (Tartu, Tallinn). Extracting highways from large cities took about 32000 milliseconds.

Extracting the bounding box is done similarly. The idea was that in case of large map files e.g map of Europe or map of whole world we can use Osmosis to extract necessary bounding box for our input data. We only need to add *–bounding-box* parameter to Osmosis command and then specify top, left, bottom, right boundary coordinates for example *top=49.5138 left=10.9351* etc. This allows Osmosis to extract necessary data from large map files about regions that are interesting to us. Combining it with node filtering speeds up the process of reading in the data considerably. In previous version where we checked manually through and selected suitable tags it took ~30 minutes to read in the file which now after using Osmosis to process the file first takes less than a minute. The current bottleneck is the creation of the road segments, since reading in the data takes only seconds now (reading in Tartu took ~600 milliseconds but connecting road segments took 20-30 seconds). The main reason for that is because the road segments are connected after creating the segments and not during the

creation of the segments the program will loop over the road segments twice to connect all the roads to the segments next to them. Some ideas to improve this has been considered but have not yet been implemented.

5. IMPLEMENTED 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 = sin²(Δφ/2) + cos φ_1 ⋅ cos φ_2 ⋅ sin²(Δλ/2) then c = 2 atan2(√a,√(1−a)) and d = R 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 $\overrightarrow{P1P2}$

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.

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.

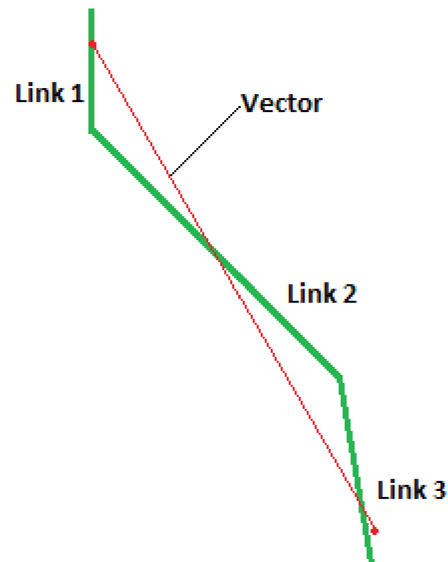


Figure 2: Example of trilateral relationships

- 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

6. DISPLAYING THE MATCHED POINTS

Several different approaches for displaying the matched points on a map were considered. I experimented with JmapViewer[9], JXmapViewer2[10], JOSM[11] program and also looked at MapPanel[12]. I finally decided to use JmapViewer since it was easy to implement and had the required functionality. The map is created using JmapViewer which is a java component

which allows to easily integrate an OSM map view into a Java application. The only downside of using JmapViewer is that it requires network connection to display the map. There was an implementation with offline support but I decided that it was not important to have offline map capability at this point.

On the map the original GPS coordinates are displayed as yellow circles. The matched coordinates are displayed as red circles. The application allows to quickly zone in to the region on the map by clicking on the “SetDisplayToFitmarkers” button. It also displays the level of zoom which shows how many meters one pixel is on the map. There is a check box for showing the tiles grid overlay. There is also a slider and two buttons for zooming in and out. Zooming can also be done with the middle mouse button. Moving the map can be done by holding down the right mouse button and then moving the mouse.

7. MAP DATA STRUCTURE IMPROVMENT

The following data structures were considered when trying to improve the finding candidate map segments for first GPS point matching: Kd-tree[13], 2d-interval tree/ segment tree[14] and R-tree[15].

Kd-tree was not suitable in my implementation since I was using map segments as my data. Each map segment has a start coordinates and end coordinates. Since Kd-tree allows for fast nearest-neighbor searches for points then it would allow us to find the nearest intersections to the GPS point. But if we imagine a case where the GPS point is near map segment but the starting and ending points of the segment are further away than some other intersection point which is not suitable in our case then we either do not find any suitable map segments to start mapping or start mapping from the wrong map segment which gives us false information.

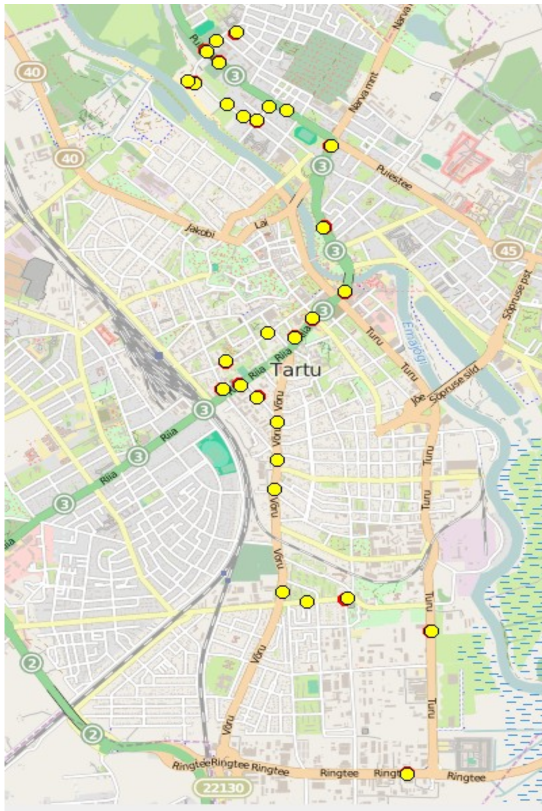
When looking into interval/ segment trees I found that 2 dimensional tree would be too time consuming to implement and was not too certain if it would be a good choice.

R-tree was considered since R-tree provides spatial access methods and is used in indexing multidimensional information. The main idea behind R-tree is to group together and represent nearby objects in within a minimum bounding rectangle in the higher level of the tree. Since Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. R-tree seemed o be suited for storing map segments and for speeding up finding the candidate segments for the first GPS coordinate. The obstacle in implementing R-tree was that most of the existing implementations in Java were used to store simple data structures namely arrays of integers depicting either lines or rectangles. Then I found an implementation which could be quickly integrated with my project and did not require any external libraries. Currently my project is not using R-tree but I have implemented it and am trying to adapt it for my needs. There are three issues remaining: dividing map into rectangles and assigning a limit to the maximum number of elements in one rectangle. Dividing the map into rectangles can be done by simply taking the bounding box of the map and dividing it into rectangles of equal size. For assigning the rectangle capacity limit I'm currently just trying to get the maximum number of map segments that fall into one rectangle. The last issue is to decide what to do with map segments that fall into several rectangles. Meaning that the start of a segment is in one rectangle and the end is in another one. Currently the idea is to put the segment into both of the rectangles. This should enable me to use nearest-neighbor search that gives n nearest map segments to the first GPS point in reasonably fast time. The complexity of searching a R-tree is $O(M \log_M n)$ where M is the maximum number of elements in a page(rectangle) in the R-tree. As we can see it is a significant improvement over O(n) which I am using currently.

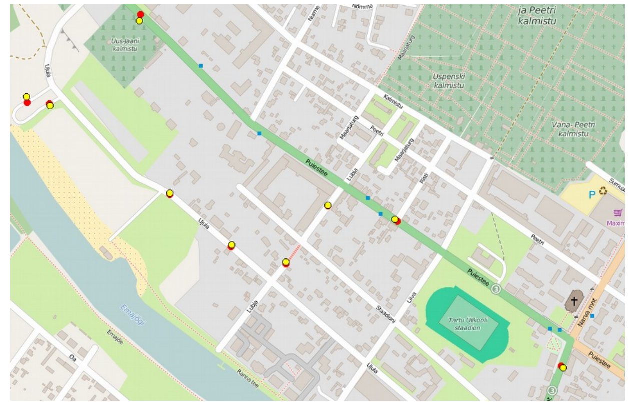
8. TESTING THE ALGORITHM

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 130-200 ms on average to match the points. On the map the size of Tartu it took ~4.6 seconds to map 30 GPS points across the whole Town.



Drawing 1: GPS coordinates as seen on the map



Drawing 2: Map-matched GPS coordinates are red dots, original are yellow

Drawing 2 shows the original GPS points and map match points in the same window which is zoomed out to show all of the points in one JmapViewer window. In drawing 3 I have zoomed in on a section of the matched points where we can see that the yellow points are original GPS coordinates which in some cases can be off the road a bit. The red coordinates are the matched points which have been mapped to the corresponding map segments.

Increasing the number of points does not increase the computation time when the points are close to each other so much because the smaller the distance between two points the less routes are considered by the algorithm. But the situation is different when we have a sparse GPS data, that means the points can be a lot further away from each other. Then the algorithm has to traverse several different routes which in turn can also split into several new routes at each intersection. In this case time taken can grow very quickly. In case of dense data where 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 and time taken grows slowly.

9. FUTURE DEVELOPMENTS

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

- Improve the speed of connecting road segments
- Deploy the program as a web service
- 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.

10. CONCLUSION

I managed to improve the vector based map-matching algorithm implemented last time. As I had no chance 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. I tried a simple backtracking from the last matched GPS points for calculating candidate routes and it worked well in cases where we had several possible route endings but failed to take into account different routes in the middle. Then I switched over to depth first search based route searching, where I still started from the end of the route and backtracked to the beginning. When there were forks in the path, meaning several possible routes then I went through all of them and calculated the sum of the matched points weights and compared the sums. The route with the largest sum was assumed to be the best candidate route and all other points were to be discarded. But when I started testing it it seemed that it did not work correctly. I found that there is something wrong with my backtracking solution

and currently I am looking in to it.

The second feature that I am still trying to improve is connecting road segments (or graph edges) to each other. The current solution is not optimal and I am trying to change it to another solution. Currently the program loops over edges twice to connect all of them correctly but I am trying to change it to a form where I just loop over all of the intersections and create edges and connect them at the same time (they are done separately at the moment).

In conclusion I can say that there is definitely a lot more to do to make it more efficient but even with current implementation testing the algorithm on small to medium scale towns/ cities is a possibility.

11. 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.
7. https://truesight@bitbucket.org/truesight/ds-mapmatching.git
8. http://wiki.openstreetmap.org/wiki/Osmosis
9. http://wiki.openstreetmap.org/wiki/JMapViewer
10. http://wiki.openstreetmap.org/wiki/JXMapViewer2
11. http://wiki.openstreetmap.org/wiki/JOSM
12. http://mappanel.sourceforge.net/
13. http://en.wikipedia.org/wiki/K-d_tree
14. http://en.wikipedia.org/wiki/Segment_tree
15. http://en.wikipedia.org/wiki/R-tree