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 experimented with Hidden Markov model based map-matching described in[1]. Last time the aim of the project was to familiarize myself with the HMM (Hidden Markov Model) based map-matching. This time I decided to use Barefoot library[28] which uses hmm-lib which was used previously.

I. 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 mainly 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]. The hidden Markov model based map-matching algorithm should provide accurate results quickly so that it is suitable for real time applications.

II. RELATED WORKS

Several map-matching algorithms are surveyed by Qudus 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)).

III. 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.
Previously several tools were explored for importing the map data to database:

- osm2pgsql
- osm2pgrouting
- osmosis
- osm2po
- shp2pgsql

For all of these tools postgreSQL[5] database with postgis extensions were necessary. Osm2pgsql[6] is a command-line based program that converts OpenStreetMap data to postGIS-enabled PostgreSQL databases. The problem with osm2pgsql was that it had only official Linux support and Cygwin[7] was required to build it on Windows. It required several dependencies which were no easily installed on Windows so it was discarded from selection. Shp2pgsql[8] converts a shape-file into a series of SQL commands that can be loaded into a database—it does not perform the actual loading. The output of this command may be captured in a SQL file, or piped directly to the psql command, which will execute the commands against a target database. Shp2pgsql was not suitable for routing due to incompleteness of osm shape-files so broken roads could affect the routing. Osm2po[9] converter parses OpenStreetMap’s XML-Data and makes it routable. It can convert large sets like the whole map of Europe and runs both on Windows and Linux platform. Osm2po was one of the two better choices for parsing the map data into database. It also had a built in routing service which was faster than PG-routing but slower than osrm (more reasons for choosing osm2pgrouting instead of osm2po was because it created better tables for pgrouting topology. And with osm2po with larger maps some edge connection information was corrupted also Os2m2po is also not open sourced, it is freeware.

In this project Barefoot uses Postgresql database for storing the map data. There are two different possibilities as how the database can be set up:

- a) a docker container and b) a self maintained database.

In my project I used Docker container and also tried to set up the stan-alone Postgresql database. Setting up the Docker container requires Docker installation (on Linux) and Docker-machine (on Windows). Creating thee container consists of three main steps:

1. Building a Docker image using Barefoot Dockerfile.
2. Creating the container.
3. Importing OSM map data to the Postgresql database inside the container.

Barefoot Dockerfile states that the docker image is based on ubuntu14.04 FROM ubuntu:14.04 and the following packets will be installed: Postgresql 9.3, Java sdk 1.7, Python with Numpy, Osmosis. The Following command is from the Dockerfile which shows Docker which packets to install on the Ubuntu image. RUn apt-get update & & apt-get -y install patch postgresql-9.3-postgis-2.1 git openjdk-7-jdk python-psycopg2 python-numpy python-gdal. Creating and starting the container is done with the following command:

```
sudo docker run -it -p 127.0.0.1:5432:5432 --name="<container>" -v $(PWD)/bfmap:/mnt/bfmap -v $(PWD)/docker/osm:/mnt/osm barefoot
```

where the -p option exposes the container port 5432 to host port 5432 -v option specifies the folders which will be mounted to container from host. Importing the data is done by Python scripts that are used for creating the database, extracting data with Osmosis and compiling Barefoot map data. When using container, then it is only required to start the container and run the import shell script provided by Barefoot. With this approach it is easy to set up OSM map database and import the data to it in a suitable format for Barefoot map-matching server. In order to import new map we can either re-run the import script with new OSM map or we can create new containers where each container contains different map. The second approach (self maintained database) requires the installation of Postgresql, PostGIS, Osmosis and Python with Numpy, Psycopg2 and GDAL. Then creation of the database and extensions. Setting up the OSM database schema. Osmosis to import the data. And
finally using Barefoot scripts to export the ways and compiling the map data.

Map-matching server can be used mainly in two different ways:

- Stand-alone mode – Barefoot Jar is assembled with maven and started from terminal/command line
- Barefoot library – can be installed with maven and dependency added to the Java project

In this project I used the standalone approach (only tried an example with Barefoot library). The server can be started with the following command:

```
java -jar target/barefoot-0.0.2.jar-with-dependencies.jar [--slimjson|--debug|--geojson] /path/to/server/properties /path/to/maps/server/properties
```

Where the first parameter decides the type of response the server would output. The server properties file lets us decide the server’s port, maximum request and response times before time-out, maximum connections to the server, number of executor threads, standard deviation of GPS error, maximum and minimum matching distance, minimum time interval between GPS locations and the number of response executor threads. The database configuration file contains the host address of the database, port number, database name, user-name and password and also path to acceptable road types file. When using the Barefoot library, it can be simply added to the project as Maven dependency.

Sending the request against stand-alone map-matching server is done via TCP/IP connection. The default request format is a JSON array of JSON objects, each object consists of id, time(stamp) and point, which is the GPS measurement in WKT(well-known-text) format with WGS-84 projection. An example request:

```
{"id":"a1396ab7-7c3a-4c31-9f3c-8982055e3de6","time":1410324847000,"point":{"POI NT (11.564388282625075 48.16350662940509)"}}
```

The response may be in different formats depending on the server argument for format. Slimjson – server outputs road id and the fraction as precise position on the road and routes are sequences of road ids. Debug format additionally returns timestamps and geometries in WKT format. And Geojson outputs the geometry in GeoJSON format e.g:

```
gerometry":
{"type":"MultiLineString","coordinates":
[[26.67383664570538,58.35893596376536],[26.6737922,58.3588961],[26.6736453,58.3587644]]}
```

Where type shows geometry type and has a list of matched coordinates and routes between them (Each sub-list is a route from one matched point to the next one).

### IV. ROUTING

Two different algorithms with four different tool were tested in last project to see which one was the most suitable for real-time routing approach: A* (or Dijkstra) and contradicting hierarchies. Osm2po, Routino[12] and Pg-routing used some version of A* (or Dijkstra) and OSRM uses contradicting hierarchies. PG-routing extends the PostGIS / PostgreSQL geospatial database to provide geospatial routing functionality. The advantages of PG-routing also include that the data is easily accessible via many clients including: QGIS and uDig through JDBC, ODBC, or directly using PL/pgSQL. The clients can either be PCs or mobile devices. And Data changes can be reflected instantaneously through the routing engine. This was the first approach tried and the routing functionality worked well but the routing queries took around a second per query. This is definitely not suitable for real time applications since for each GPS point we need to perform around 3-7 routing queries. Later I managed to make it faster including a bounding box for the routing query and this scaled the timings down to couple of hundred milliseconds but I had already switched to OSRM routing by then. Pg routing uses Dijkstra and A* for finding shortest path. Routino uses modified A* algorithm and it discards most of the nodes while keeping only interesting nodes (e.g Nodes which have at least a junction of 3 ways or a junction of two ways with different properties.) it gave similar results to PG-routing in regards of time used without optimization. Since Routino did not improve the computation time I decided to try something different. In previous project I decided to try OSRM (Open Source Routing Machine)[13] as the routing component of the application. It computes shortest paths in a graph. It was designed to run well with map data from the OpenStreetMap Project. OSRM does not use an A* variant to compute shortest path, but Contraction Hierarchies. This results in very fast query times, usually below 100ms for data sets like Europe, making OSRM a good candidate for responsive web-based routing applications and websites. Contraction hierarchies[14] is a technique to speed up shortest-path routing by first creating precomputed "contracted" versions of the connection graph. It generates a multi-layered node hierarchy in the preprocessing stage. When preprocessing of the original graph is done, we have a CH graph which consists of the original graph with node ordering added and with shortcut edges introduced. For querying a bi-directional Dijkstra is used – the algorithm searches from both the starting node and the end node. If the shortest path exists, those two searches will meet at some node v. The algorithm finds the shortest path due to preprocessing stage.

OSRM-backend (the name of the OSRM routing component) processing flow composes of several steps. The main flow is as follows: import raw OSM data, compute routes, serve data. Osmr-extract generates .osm and .osrm.restrictions it takes a profile field as an argument to specify the road types used for the routing. I used a default car profile for testing the osrm routing functionality. It defined several restrictions which made the routing use ways only vehicles can access. This helped to reduce routing errors due to the fact that some pedestrian roads would be used which cars cannot travel. The profile file also defined default speed limits for different types of roads. For example motorway limit is 90 km/h residential road is 25 km/h etc. Osmr-prepare reads the extracted data and prepares it for
routing. What it basically does is that it will read the Intermediate OSM format files generated from extraction and creates OSM server data files (edges, nodes etc). And the most important step is osrm-routed. It loads the prepared data and starts a routing server. As I used local server for routing then default configuration was suitable. After running the command osrm server will respond to routing requests received on a specific IP and port, and return computed routes. The service needed for routing is called viaroute. It provides shortest path queries with multiple via locations. Example query as below:

http://[server]/viaroute
loc={lat,lon}&loc={lat,lon}&<loc={lat,lon} ...>

where loc is the previously matched gps candidate coordinates on roads and last loc is the current gps candidate point. This allows to get all shortest paths or fastest paths for markov model.

Barefoot seems to use Dijkstra algorithm for route calculation. It uses the routing table generated by importing the OSM map into database and matcher server reads the roads into memory. It uses in-memory map data structure for increased route calculation. Basically it is a roadmap with directed roads and it has spatial search of roads with a spatial-index. It uses GeographicLib[25] and ESRI’s geometry API for spatial search and operations (WKT import/export, Calculation of distances, intersections, also Quad-tree spatial index and many other operations).

V. IMPLEMENTED MAP-MATCHING ALGORITHM

In the previous project the first approach was to start implementing my own Markov model in java. This was deemed too time consuming since creating an effective and fast Markov model approach would take too much time. The next approach was to see if there were already some open sourced hidden Markov models implemented. Some of the implementations I looked at were JAHMM[15] and hmm-lib[16]. Both of them support the use of Viterbi[17] algorithm for calculating the most likely sequence of hidden states. JAHMM also includes forward – backward algorithm[18]. I decided to try to use hmm-lib and see if I can use it for my implementation. The Viterbi algorithm is a dynamic programming algorithm for finding the most likely sequence of hidden states – called the Viterbi path – that results in a sequence of observed events, especially in the context of Markov information sources and hidden Markov models. It can be used to quickly find the most suitable matched gps points on the road. The complexity of the algorithm is O(T * |S|) where S is the state space and T is the number of outputs. We give the observation space, state space, the sequence of observations, transition matrix, emission matrix and initial probabilities as input and get the most likely hidden state sequence as output. In my approach each position measurement triggers an iteration of the algorithm and updates the position estimates.

The current approach is to first determine map position candidates on road segments that have minimum geodesic distance to the measurement gps point. The road segments must overlap with a square around the GPS point. The emission probability is calculated via the following formula:

\[ p(z_t | s_t) \sim \frac{1}{\sqrt{2\pi \sigma_z^2}} \exp \left\{ -\frac{||z_t - s_t||^2}{2\sigma_z^2} \right\} \]

where theta z is the standard deviation of gps measurements and z_t - s_t is the distance between the gps point and the matched point on the road segment. For finding the transition probabilities calculating the route from previous positioning candidate to current candidate is required. In [1] they found that transition probabilities have been experimentally determined to fit a negative exponential distribution:

\[ p(s_t | s_{t-1}) \sim \lambda \exp \left\{ \lambda (||z_t - z_{t-1}|| - ||s_t - s_{t-1}||) \right\} \]

where it remains to find the best parametrization (estimate of \(\lambda\)) for a specific sampling set, i.e. the sequence of position measurements. Then use the Viterbi algorithm to calculate the most suitable sequence for current states (GPS points) and start the next iteration until all GPS points are matched.

Barefoot uses the above mentioned hmm-lib for its map matching. It is basically the implementation of the approach described in [1]. In my project I set up the map database using Docker approach. Since Docker should be simple to use on both Windows and Linux machines the first try was with Windows installation but I ran into problems while exposing ports to the host machine. It worked inside the Docker virtual machine but I could not access the database from Windows host. Also Docker only supports mounting host file system locations only in User directory. For example if our map container has mounted folders from any other directory, then it will not work. (-v parameter in the docker command). Then I decided to switch to Linux virtual machine and it worked properly. Importing map data seems to heavily depend on the OSM source file, since I could easily import parts of Germany or other European countries but had problems importing the map of Estonia due to some missing data in the input field using GeoFabrik[26] OSM mirror page. Exporting the raw OSM map over Overpass API solved the problem (It might have been a problem with pbf file format). For road types I used all of the roads where a vehicle could travel on (including residential, living, service and unclassified). I tried the map matching server with different parameters to see how the result would differ (no minimum time interval, both small and large distance between points and so on). For sending the data to the matcher I created a small program that allows us to either stream data to matcher server (online matching) and process the result or send the data as one payload (so called offline map-matching). In case of offline matching we can specify an interval parameter which allows us to only select every x-th input GPS position. This allows us to simulate either dense or sparse data. When streaming the GPS positions the application gets an input of one GPS location and it sends the last x locations to matcher server. This allows a significant speed-up over offline matching where all points were sent at once. E.g. it takes about 2-4 seconds for the
executing database queries. The viewparams property is parameters that will be sent to GeoServer. And for variable to the map code which holds the WMS GET Openlayers map and add the WMS GET parameters EPSG:4326 reference had to be in type of the st_makeline is LineString. The coordinates only validated numbers. And in the attributes values the regular expression was used: ^-?\[[\d.]+$.

In SQL view parameters: Guess parameters from SQL.

```
SELECT ST_H贱eLine(route.geom) FROM (
  SELECT geom FROM pg_st_MakeLine(geom), h=6, k=64, k2=64, k2=64
  ORDER BY agg AS route
) IN SQL view parameters: Guess parameters from SQL must be selected. And for each parameter the default value must be zero. The following validation using regular expression was used: ^-?\[[\d.]+$, which only validated numbers. And in the attributes values the type of the st_makeline is LineString. The coordinates reference had to be in EPSG:3857 instead of EPSG:4326. The last part was to create a simple Openlayers map and add the WMS GET parameters variable to the map code which holds the WMS GET parameters that will be sent to GeoServer. And for executing database queries The viewparams property is then set on WMS GET parameters object. The value of this property has a special meaning: GeoServer will substitute the value before executing the SQL query for the layer. And the response from the GeoServer is then drawn as a new layer. This approach was dropped when PG-ROUTING was deemed too slow for real-time applications. Additionally, before limiting the layer query SQL statement it queried the whole table making it unusable with larger maps (whole Europe for example, with Estonia seemed to be working fast ~100 – 200 ms).

The second platform considered was Mapnik[22]. Mapnik is basically a collection of geographic objects like maps, layers, datasources, features, and geometries. For example OpenStreetMap runs on Mapnik service. A simple python based map was implemented and it seemed to work really well. But Mapnik was discarded for a number of reasons: lack of up-to-date Java bindings and the complexity of setting it up. Since I did not want to convert my whole program to C++ or Python at this stage I decided to not use Mapnik for displaying the map-matched route due to the previously mentioned reasons.

Openlayers 3 and leaflet were the main map services considered for displaying the map. Openlayers has really good integration with PG-routing and Mapnik map server where as leaflet has better integration with OSRM routing service.

Leaflet has several open source implementations that already utilize the OSRM API for example Leaflet Routing Machine or Project OSRM Frontend which is purely Javascript written frontend to enable OSRM based routing. Although they provide OSRM server api support and enable to make routing calls from the frontend, this was not required in the current solution since only one route per vehicle is required to be displayed at the end of the map-matching process. Previously I used Openlayers3 for displaying the results. Openlayers is really similar to Leaflet but it has more features is a bit more complex and has a greater size ~1 mb vs Leaflet ~100 kb.

This time I decided to use Leaflet for displaying the results. Leaflet has really good support for displaying GeoJSON geometries on the map. For real-time map displaying I decided to use leaflet-realtime[27] plugin. In Leaflet it is quite simple to add geojson data to map for displaying I decided to use Leaflet for displaying the results. Openlayers has several open source implementations that already utilize the OSRM API for example Leaflet Routing Machine or Project OSRM Frontend which is purely Javascript written frontend to enable OSRM based routing. Although they provide OSRM server api support and enable to make routing calls from the frontend, this was not required in the current solution since only one route per vehicle is required to be displayed at the end of the map-matching process. Previously I used Openlayers3 for displaying the results. Openlayers is really similar to Leaflet but it has more features is a bit more complex and has a greater size ~1 mb vs Leaflet ~100 kb.

This time I decided to use Leaflet for displaying the results. Leaflet has really good support for displaying GeoJSON geometries on the map. For real-time map displaying I decided to use leaflet-realtime[27] plugin. In Leaflet it is quite simple to add geojson data to map we only need to create a ma variable and add the geojson features to it. Leaflet-realtime allows us to add server url from where the geojson data will be queried. An example of the map variable below:

```
var map = L.map('map'),
realtime = L.realtime([
  url: 'https://localhost:8000/original.json',
  crossOrigin: true,
  type: 'json',
], {
  interval: 3 * 100000,
onEachFeature: onEachFeature, onEachFeature,
  pointToLayer: pointToLayer
}).addTo(map);
```

Leaflet map

The L.realtime shows that it is using the realtime functionality, where url is the server url where data is queried (in my case geojson file), interval is data
querying interval and onEachFeature and PointToLayer are Leaflet functions we can use to manipulate how the data is displayed on the map. Leaflet allowed me to add real time result showing capability to the map without having to create my own custom solution.

In the above example the matcher server received about 30 input coordinates (client application interval was 3 and the total was about 105). We can see the original GPS markers (orange), the matched points (blue) and the path calculated using Markov Model based map-matching (blue line). Below are the same input GPS markers but interval is set to take every tenth:

So we can see that the results are quite similar.

And below we can see how server discarded some points due to properties. It is really dense data so some time constraints and some distance constraints were violated.

Another solution which was tried was uDig [24]. The goal of uDig is to provide a complete Java solution for desktop GIS data access, editing, and viewing. The advantages of uDig are that it is easy to use, runs on different platforms, is internet oriented (supports WMS, WFS, etc formats) and is GIS ready. It is a good tool for debugging the map data since it is possible to only display roads and/or intersections with uDig, making it possible to quickly spot error in the road network in case of unexpected results. The reason it was not considered suitable for this project is that it requires GIS compatible data for displaying the map and is more complex to use than Openlayers or Leaflet.

VII. MAP DATA STRUCTURE

In the previous project following data structures were considered when trying to improve the finding candidate map segments for first GPS point matching: R-tree[19] and Quad-tree[20]. During last project kd-tree and interval-tree were considered as well but found not suitable for the following reasons:

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-neighbour 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
In last project I experimented with R-tree and quad-tree. 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 a minimum bounding rectangle in the higher level of the tree. 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 to 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. 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.

Barefoot uses k-State data structure for holding the states. From Barefoot manual: “A k-State data structure is a state memory for map matching. It organizes sets of matching candidates as candidate vectors, and provides access to a matching estimate which is the most likely matching candidate \( s_t \) for time \( t \) and represents an estimate of the object's current position on the map, and an estimate of the sequence which is the most likely sequence of matching candidates \( (s_0 \ldots s_t) \) and represents the object's most likely path on the map.”

VIII. FUTURE DEVELOPMENTS

Next step would be to integrate all of the solutions to each other that it can be easily set up and run. After that there are many different things that could be done to try to increase the accuracy of the matching e.g. start predicting the results to speed up the process (by relying on the historical data.). Also Barefoot should be scalable to run in Hadoop Spark.

IX. CONCLUSION

In the previous project I saw that using OSRM for routing and R-tree for storing the road segments of map data seems quite efficient. When I implemented the Vector based approach I used simple arrays to hold road data and it took seconds to find the first point and all of the nearby road segments. This time I decided to use uniform solution for matching to see how it compares to my last solution. As I am mostly using the standalone matcher service I cannot compare the routing times but since Barefoot uses in memory map data, I am guessing that it is similar in performance to OSRM approach. Personally I believe that contraction hierarchies would be faster than using Dijkstra for routing but at this time with map files the size of Estonia I do not believe we have issues using Barefoot as real time map matching service. Barefoot is easy to set up (standalone) and get it to working but since it is a quite new development (Autumn 2015) there is not much examples or documentation available about it. The official manual is good for getting started with standalone server but is a bit lacking when using Barefoot lib. I think that this is definitely a good candidate when creating a real-time map matching service with Java.

X. REFERENCES

1. Kazi Khaled Al-Zahid, Birgit Engelmann, Andreas Hildisch, Stefan Holder, Oleksiy Lazarevych, Daniel Mohr, Felix Sedlmeier, Sebastian Mattheis, Richard Zinck. Putting the car on the map: A scalable map matching system for the Open Source Community


4. Quddus, Mohammed A. Ochieng, Washington Y. Zhao, Lin Noland, Robert B. A general map matching algorithm for transport telematics applications

5. PostgreSQL open source object-relational database system: http://www.postgresql.org/

6. osm2pgsql a command-line based program that converts OpenStreetMap data to postGIS-enabled PostgreSQL databases http://wiki.openstreetmap.org/wiki/Osm2pgsql

7. Cygwin - a large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows. https://www.cygwin.com/

8. Tool for importing shapefiles to postgresql database: http://man.cx/shp2pgsql%281%29


10. Osmosis is a command line Java application for processing OSM data: http://wiki.openstreetmap.org/wiki/Osmosis
11. osm2pgrouting is a command line tool that allows to import OpenStreetMap data into a pgRouting database http://pgrouting.org/docs/tools/osm2pgrouting.html

12. Routino is an application for finding a route between two points http://wiki.openstreetmap.org/wiki/Routino


14. contraction hierarchies is a technique to speed up shortest-path routing https://en.wikipedia.org/wiki/Contraction_hierarchies


16. Hmm library in Java https://github.com/bmwcarit/hmm-lib


22. Mapnik is an open source toolkit for developing mapping applications. https://github.com/mapnik/mapnik

23. Leaflet is a JavaScript library for mobile-friendly interactive maps http://leafletjs.com/

24. uDig provides a complete Java solution for desktop GIS data access, editing, and viewing http://udig.refractions.net/


26. OSM map dataset https://www.geofabrik.de/data/download.html

27. Put realtime data on a Leaflet map: live tracking GPS units, sensor data or just about anything. https://github.com/perliedman/leaflet-realtime


APPENDIX A