# Android app for dynamic traffic routing

Project Report

Lakesh Kansakar and Yi Jia and Vladimir Coric

**Abstract**

In today's world time has been the most valuable asset for every person. As it is rightly said, every second counts, it is very much important to keep oneself efficient and utilize as much time as possible. People normally travel by cars and lose several hours every day because of traffic and other incidents like accidents, inefficient paths etc. In this situation, if one could know the best path to follow to reach his/her destination then it could save a lot of time. So our main motive for this project is to create an application that could provide dynamic traffic routing information. Based upon the most recent traffic information, it could provide the quickest path to the user given the source and target place and the time of the day. Mobile phones are easily accessible nowadays. Moreover, many individuals have smart phones. So, a mobile application that could actually provide this useful feature to help them find the quickest path to reach their destination will be wonderful. Our another goal is reduction of the bandwidth consumption. There are lots of services out there like google maps which provide this service. One of the problems with these services is that you need to download the map data from their server. This will definitely consume some bandwidth which might not be desirable for people specially if you are running on low bandwidth services. This is where the concept of offline maps fits in. If we have all the map data in the phone itself, then we just need to get the information about the path. This will definitely save some bandwidth. So our overall target is to create a dynamic and bandwidth friendly routing application.

## 1 Introduction

It's always ones utmost priority to reach to his/her destination on time. However, because of traffic issues like accidents, high traffic volume, people lose several hours every day. It is possible to reach a certain destination following several paths. But it might not be feasible for a person to guess which one would be the best, which one would have less traffic and crowd. If only he/she has an application that could actually show the best path then it would be wonderful. Further consider the scenario, lets say we have an accident at a particular place. Then it is obvious that if the person moves out at that period, then he will have to lose some time because of the accident. So, if a system could predict what the traffic condition will be like in say half an hour and tell if it would be better for this guy to move out half an hour later rather than now because of lets say traffic will be less dense, then it will be really useful.

The world is shifting from dekstop based applications to mobile and web based applications. Nowadays mobile phones are easily accessible. With the advent of 3G

and 4G networks and performance of smart phones, it is possible to access high definition content in mobile phones. However, high volume of content requires higher bandwidth which will incur more cost to the consumer. In this situation, it would be beneficial for the person to use an application that could consume as much less bandwidth as possible and provide him all the services that he needs which is routing in our case.

We are actually, proposing a mobile based application that can provide dynamic routing information to the users. Based upon the recent traffic information, the service can actually provide the user with the path that will take the shortest of time to reach the destination. Based upon the speed information available at the stations or sensors, our server can actually update the cost information at regular interval of time. This makes sure that the user is provided the quickest path based upon the updated information rather than the information one year ago. We are actually planning to make the system work in real time.

### 1.1 Related Work

There are lots of sources out there that actually provide this feature in some sense; Google being the most widely used among them. Further, there are popular services like Cloudmade which produces APIs, rendered maps and geographic-related services like shortest path, navigation etc. However, our target is to focus on offline map data, which is provided by openstreetmaps and develop a routing service based upon the recent traffic information.

## 2 Methodology

### 2.1 Architecture

Basically as in every client-server application, our system consists of two components.

- Server side
- Client side

The client side is reponsible for providing the user an interface to view the map, allow him/her to select the locations(source,destination), pick date time of the year and show the route. On the other hand the server side is reponsible for serving the routing requests from the clients. It does the bulk of processing of updating the cost of the different routes/edges based upon the current traffic information update and provide the user with the quickest path to reach his/her destination.

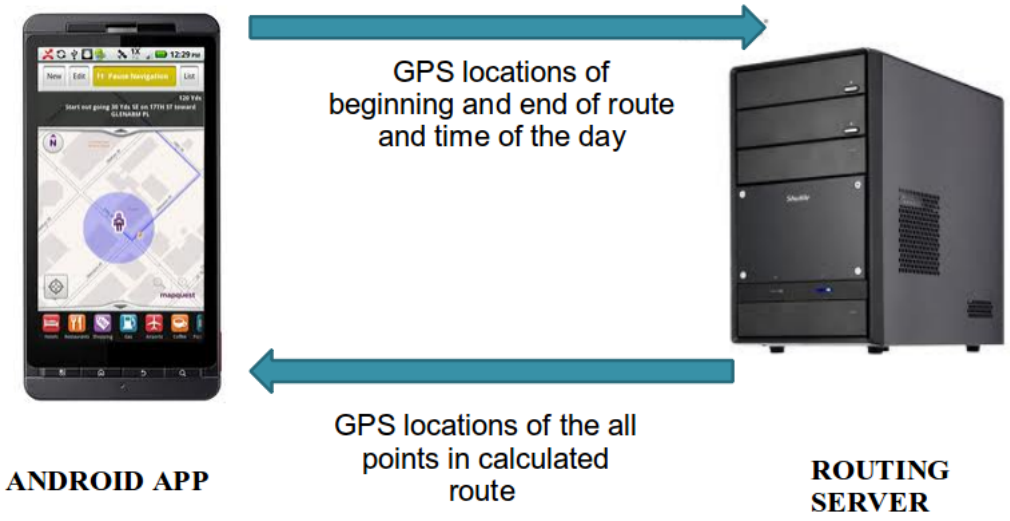



Figure 1: Basic architecture of the system

## 2.2 Client side

### 2.2.1 Map selection

We have chosen openstreetmaps[1] for our offline map. OSM is open and there are lots of contributors out there who are actually adding/modifying/rectifying map information. The best part about OpenStreetMap is that it is free and general people can actually provide map information. Our other option was to use google maps. But since google doesn't allow you to have offline maps we have chosen OSM over google maps.

### 2.2.2 Platform

Out of the available platforms like iOS, Android, Symbian etc, we have chosen Android[2] for our project because it is open source and there are lots of developers, contributors working on this project. Further, it's free and we don't have to pay for it. So, we have chosen this platform instead of others. Morever, Android is constantly evolving bringing in new features. Therefore, we believe that it is the best match for our objective.

### 2.2.3 Application

Our application is a derivative of an open source application called Osmand [3]. As it is rightly said, it is not a good thing to reinvent the wheel, we already have this open source application osmand that actually provides most of the features that we want like offline maps, map rendering feature, interface to choose the start end destination etc. Further, it has already integrated routing services like CloudMade, Yours, OpenRouteService etc. In this situation, we found it efficient to actually add our routing service Temple on top of what it already had and use its exiting code to display the map as well as the route provided by our Temple server.


Figure 2: Osmand application

### 2.2.4 Navigation Feature


Figure 3: Temple routing service in osmand

As you can see in figure 3, Osmand has already integrated with the routing services provided by CloudMade, YOURS, OpenRouteService as well as offline routing (which it recommends when the source and destination are far apart). So we added our own service called Temple on top of that. So, when the user chooses Temple

as a navigation service, our server gets the request for the shortest route and then returns the corresponding path in the form of gps coordinates.

### 2.2.5 Offline Maps


Figure 4: Downloading offline maps

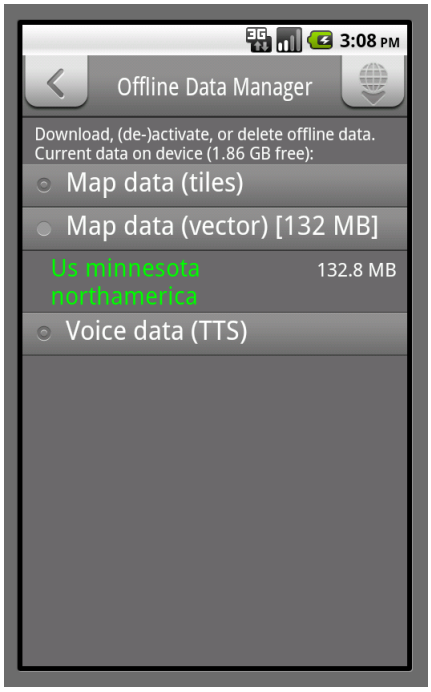



Figure 5: Installed offline maps

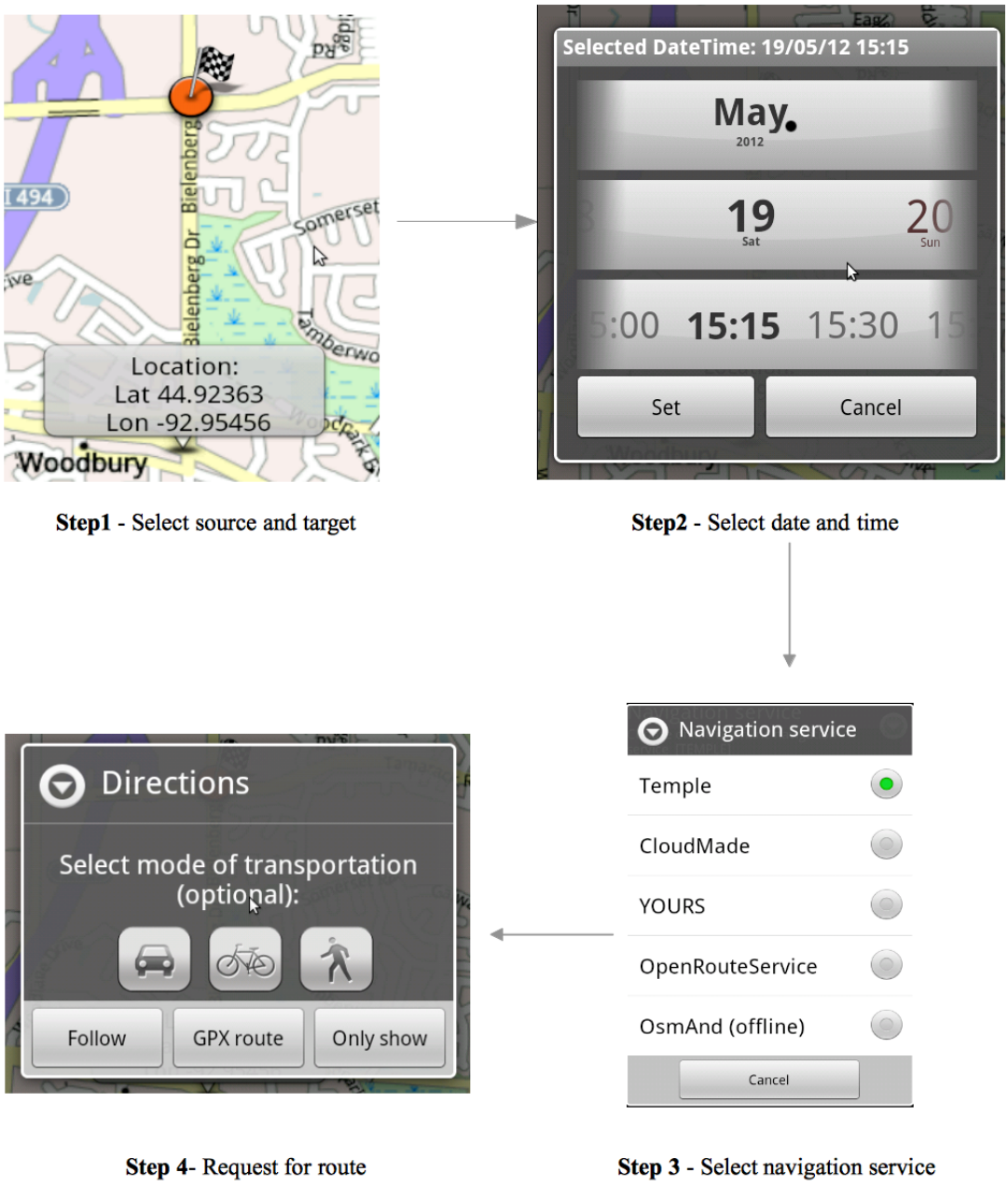
As we had mentioned before Osmand already has this feature of offline data. It can actually fetch the map data online or use the offline data that can be downloaded once and stored in the mobile phone itself. Osmand actually has a repository from where the offline maps can be downloaded and stored in the phone. Furthermore, they have special tools that can compress the default Openstreetmap in .osm format and convert them into their own format called obf format. This format of data has comparatively less size than that of the original osm format. So, this will definitely save some bandwidth. Osmand has this option of downloading both the vector and raster form of maps.

**Raster maps(Tiles)**
These types of maps are actually images which are made up of pixels. For a particular zoom level, a set of images also called tiles display the map area. If the user needs to zoom in more then new sets of images cover the display area providing more details. Tiles actually consume more space and bandwidth

**Vector maps**
They consume comparatively less space. The map is actually represented in the form of xml. The building blocks of these type of maps basically contain nodes, ways, relations, tags etc[4]. The vector maps are further compressed in binary format.

Osmand has used its own binary format called osmand binary format(.obf)[5] to compress the vector maps and distribute them.

### 2.2.6 Workflow




**Step1** - Select source and target

**Step2** - Select date and time

**Step 3** - Select navigation service

**Step 4**- Request for route

Figure 6: Workflow in the client side

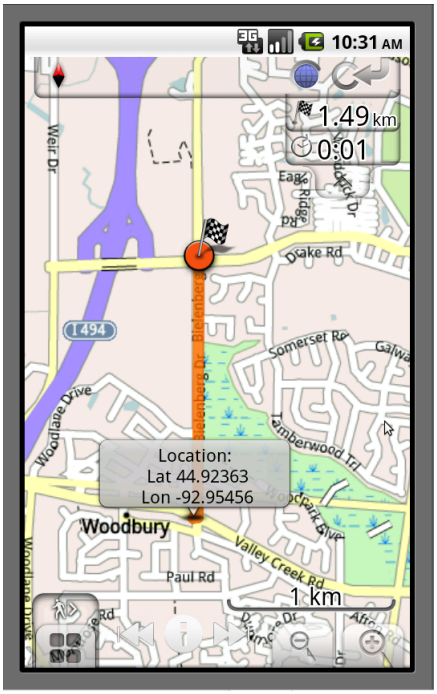



Figure 7: Route displayed in the client app

The figure 7 shows the path displayed by our application. Given the start and end coordinates as well as the datetime, the client requests our Temple server to get the quickest path. The server then returns the path in the form of gps locations in a special file format called gpx format. Our application then plots the route given in the gpx file. Further, as we have mentioned osmand already had this feature to plot the path from a gpx file. So, we used that feature to display our path as well. Currently Osmand has lots more features like voice navigation, choosing routes based upon walking, car or bicycle. It's good to have all these features already built in. However, for our basic purpose we are only using navigation by car. In future, we can actually exploit all these other features that osmand provides.

### 2.2.7 Data Exchange

Our service currently providing the routing services resides in a web server powered by Apache at Temple's internal server accessible at data.ist.temple.edu. To actually pass information from our client side to the server, we make a standard http GET request.

So lets say our start location's latitude and longitude information is 43.0298,-93.1286. Similarly let the end location's latitude and longitude be given by 43.1598,-93.2349. The datetime be 2005-04-12. Then the corresponding **GET** request will be.

**GET** http://data.ist.temple.edu/routes.php?start_latitude=43.0298&start_longitude=-93.1286&end_latitude=43.1598&end_longitude=-93.2349&date_time=2005-04-12

As you can see all the parameters like start and end locations are passed as standard **GET** parameters. The server can then read these parameters and return the quickest path to the client. We had discussed that the server exchanges the quickest path information in the form of gpx[6] file.

```
<gpx version="1.1" creator="OsmandRouter" xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GI
 - <trk>
   - <trkseg>
       <trkpt lat="44.998707" lon="-93.260239"/>
       <trkpt lat="44.998707" lon="-93.258881"/>
       <trkpt lat="44.998703" lon="-93.261551"/>
       <trkpt lat="44.998707" lon="-93.260239"/>
       <trkpt lat="44.9987023" lon="-93.2631767"/>
       <trkpt lat="44.998703" lon="-93.261551"/>
       <trkpt lat="44.998703" lon="-93.2633298"/>
       <trkpt lat="44.9987023" lon="-93.2631767"/>
       <trkpt lat="44.9987052" lon="-93.263772"/>
       <trkpt lat="44.998703" lon="-93.2633298"/>
       <trkpt lat="44.998703" lon="-93.264618"/>
       <trkpt lat="44.9987052" lon="-93.263772"/>
       <trkpt lat="44.998702" lon="-93.2659381"/>
       <trkpt lat="44.998703" lon="-93.264618"/>
       <trkpt lat="44.998705" lon="-93.2674053"/>
       <trkpt lat="44.998702" lon="-93.2659381"/>
       <trkpt lat="44.998701" lon="-93.26911"/>
       <trkpt lat="44.9987508" lon="-93.3063494"/>
       <trkpt lat="44.99876" lon="-93.306938"/>
       <trkpt lat="44.99876" lon="-93.306938"/>
       <trkpt lat="44.998764" lon="-93.308197"/>
       <trkpt lat="44.998764" lon="-93.308197"/>
       <trkpt lat="44.998775" lon="-93.309471"/>
       <trkpt lat="44.998775" lon="-93.309471"/>
       <trkpt lat="44.998784" lon="-93.3107714"/>
       <trkpt lat="44.998784" lon="-93.3107714"/>
       <trkpt lat="44.998795" lon="-93.31205"/>
       <trkpt lat="44.998795" lon="-93.31205"/>
       <trkpt lat="44.998806" lon="-93.313332"/>
       <trkpt lat="44.998806" lon="-93.313332"/>
       <trkpt lat="44.999081" lon="-93.340225"/>
       <trkpt lat="44.99894" lon="-93.340622"/>
     </trkseg>
   </trk>
</gpx>
```

Figure 8: GPX file returned by the server

Figure 8 is an example of a gpx file returned by the server. As it can be seen, the quickest path is represented by a set of coordinates. Our application then simply joins these points by straight lines which gives our quickest path.

## 2.3 Server Side

We used PostgreSQL[7] for storing the spatial data related to maps. Currently we are focusing on the Minneapolis region of Minnesota USA. So, initially we got the openstreetmap for the Minneapolis region. This data was in osm format which was xml. We used a tool called Osm2psql[8] to convert the OpenStreetMap(.OSM) data into a format that can be loaded into PostgreSQL. For the support for geographic objects to the PostgreSQL object-relational database we used PostGIS[9]. Actually PostGIS spatially enables the PostgreSQL server, allowing it to be used as a backend spatial database for geographic information systems.

For calculating the shortest path we used another open source application called pgRouting[10]. It actually extends the PostGIS/PostgreSQL geospatial database to provide geospatial routing functionality. PgRouting provides functions for

- Shortest Path Dijkstra
- Shortest Path A-Star
- Shortest Path Shooting-Star
- Traveling Salesperson Problem(TSP)
- Driving Distance calculation

In our current implementation, we are using Dijkstra algorithm for calculating the shortest route.

### 2.3.1 Finding the corresponding edges the start and end points belong to


Figure 9: Finding the corresponding edges of the start and end point

The first step in the server process is to actually find out the edge that the given start and end points belong to. For example in the figure given above the red circles denote the start and end points. However, they are off the edge. So the first thing to do is to find the edge that the points belong to. So what we do is create a bounding box around the circle and find the nearest edge for the points. Again pgRouting already provides this feature [11]

### 2.3.2 Maintaining information about the cost of edges and stations

| id [P | osm_id bigint | osm_name character | osm_sour bigint | osm_targe bigint | cl in | fl ir | source integer | target integer | km double pre | kmh integer | cost double pre | reverse_c double pre | x1 double pre | y1 double pre | x2 double pre | y2 double pre | geom_way geometry |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5994201 | E 103rd St | 33419270 | 34504644 | 32 | 1 | 1287 | 1288 | 0.10285643 | 50 | 0.00514282 | 0.00205712 | -93.277206 | 44.817142 | -93.275902 | 44.817139 | 0105000020 |
| 2 | 5994202 | SE Delaware | 33315439 | 33315517 | 32 | 1 | 1289 | 1290 | 0.10852637 | 50 | 0.00542631 | 1000000 | -93.225673 | 44.9724981 | -93.224293 | 44.972481 | 0105000020 |
| 3 | 5994203 | Lake Shore | 33729587 | 33727392 | 32 | 1 | 1291 | 17933 | 0.19652507 | 50 | 0.00982625 | 0.00393050 | -93.493160 | 44.9479259 | -93.491318 | 44.9469492 | 0105000020 |
| 4 | 5994203 | Lake Shore | 33727392 | 33378974 | 32 | 1 | 17933 | 4170 | 0.07043764 | 50 | 0.00352188 | 0.00140875 | -93.491318 | 44.9469492 | -93.490767 | 44.9464499 | 0105000020 |
| 5 | 5994203 | Lake Shore | 33378974 | 33315195 | 32 | 1 | 4170 | 18770 | 0.08475122 | 50 | 0.00423756 | 0.00169502 | -93.490767 | 44.9464499 | -93.490212 | 44.9458073 | 0105000020 |

Figure 10: Database table containing edge details

As you can see, the information about the edges is maintained in the database. Each edge has a unique osm id as well as the id given by the database. It has the information about the source and end vertex, their corresponding latitude and longitudes. Further, it has information about the speed limit in that edge given by the column kmh. The distance of the edge is given by the column km. Based upon these information we can calculate the cost of the edge given by the cost column.

$$speed = \frac{distance}{time}$$

$$or, cost = time = \frac{distance}{speed}$$

Hence the cost in our case is the time it requires to actually cover the length of the edge. The information about the speed of traffic at a particular location is given by speed sensors called stations. Since Minnesota Department of Transportation provides the station location (http://www.dot.state.mn.us/tmc/trafficinfo/developers.html), we use the same tools 3D box to find and manually connect the station id and edge id, which indicated as following image. Here we only used the main road station which contains 769 stations. An edge can even have more than one sensors associated with it. In this situation the speed of traffic at the edge will be the average of the sensors associated.

| station_id [PK] integ | edge_id integer | dist double pre | geom geometry | station_la double pre | station_lo double pre |
|---|---|---|---|---|---|
| 4 | 78388 | 1.20812483 | 0101000020 | 44.95512 | -93.27083 |
| 5 | 78388 | 7.87485988 | 0101000020 | 44.95169 | -93.27377 |
| 6 | 66212 | 8.10483457 | 0101000020 | 44.94636 | -93.27494 |
| 7 | 78389 | 0.00016359 | 0101000020 | 44.93984 | -93.27498 |
| 8 | 78389 | 5.83364500 | 0101000020 | 44.93464 | -93.27495 |
| 9 | 78390 | 0.00010478 | 0101000020 | 44.92727 | -93.27492 |
| 10 | 78395 | 5.66982537 | 0101000020 | 44.92003 | -93.27488 |

Figure 11: Stations associated with edges

As you can see in figure 11 a station is associated with an edge and multiple stations could be associated with an edge. The information given by the stations helps us to update the information regarding the traffic/speeds at different segments or edges. So, the idea is to regularly get the updates from the sensors, lets say a period of half an hour. From this information we can maintain records about the costs of edges at different interval.

| | | | | | |
|---|---|---|---|---|---|
| !5276 | 138311 | 106682 | 106683 | 0.00640023 | 2003-03-01 |
| !5276 | 138311 | 106682 | 106683 | 0.00629531 | 2003-03-01 |
| !5276 | 138311 | 106682 | 106683 | 0.00640023 | 2003-03-01 |
| !5276 | 138311 | 106682 | 106683 | 0.00640023 | 2003-03-01 |
| !5276 | 138311 | 106682 | 106683 | 0.00619377 | 2003-03-01 |
| !5276 | 138311 | 106682 | 106683 | 0.00650871 | 2003-03-01 |
| !5276 | 138311 | 106682 | 106683 | 0.00629531 | 2003-03-01 |
| !5276 | 138311 | 106682 | 106683 | 0.00619377 | 2003-03-01 |

Figure 12: Costs associated with edges at different interval

As you can see, we have the different cost information for the same edge at different time interval. One of the issues that we will face with this approach is that we will have lots of data eventually. So lets say if we keep the records for the cost information every minute for an interval of 1 month, then it could easily consume lots of disk space. Furthermore, it would be time consuming to actually query the cost information from this huge table. Horizontal partitioning or indexing might help to resolve this issue. However, we propose that at any time we might not need to maintain all the cost information. Cost information of the recent 20-30 discrete intervals should be enough. At any time a person will be interested in the recent traffic information rather than the traffic information a year ago. So, we can actually archive the traffic information of the past and keep the fresh records in the production. Furthermore, a user may be interested in how the traffic will shape in the near future. This might help to answer his queries like if he needs to go from point A to point B, then will it be good for him to go now or after a period lets say half an hour. So our target will be to actually predict how the traffic will look like in that period. If it's less dense then it would be beneficial for him to move out later.

Table 1: Costs associated with edges

| | |
|---|---|
| Cost of edges with station | Length/speed provided by sensor |
| Cost of edges without station: highway | Length)/speed limit |
| Cost of edges without station: local roads | Length/(20kmh) where 20 is default speed for local roads |

In the Minneapolis data we had the following stats regarding the edges and stations
Edges without stations = 180000
Edges with station = 769

### 2.3.3 Getting the quickest path

We used pgRouting's Dijsktra algorithm to calculate the quickest path. PgRouting actually provides special functions which we can query to get the quickest path. Further it allows us to specify the cost of the path. In our case it will be the time it takes to actually cover the path. As shown in the figure above, we can change the cost column, with that of the cost related to a particular time, lets say 30 minutes ago. So, now when pgRouting does the query to find the shortest path, it is based upon the traffic information 30 minutes ago. Further, we can maintain the cost information regarding the different

edges at a different table and when making the actual query for the quickest path, we can pass the cost information for the edges from this separate table based upon the datetime information provided by the user.

```
SELECT * FROM shortest_path('SELECT gid AS id, source::int4,
            target::int4, length::double precision AS cost,
       FROM dourol',3, 7, false, false);
```

```
 vertex_id | edge_id | cost
-----------+---------+-------------------------
         3 |       2 |    0.000763954363701041
         4 |      21 |    0.00150254971056274
         6 |       5 |    0.000417442425988342
         7 |      -1 |    0
(4 rows)
```

Figure 13: Calculating the quickest path

Figure 13 shows the example of a query for a shortest path from the start node 3 and end node 7. The returned results shows that shortest path which consists of a list of edges and their corresponding cost. This means the person is to follow the path specified by the edges(which is the shortest) to actually reach his destination.

So, the basic algorithm to calculate the quickest path will be

- Based upon source and target point calculate the edge where the points belong to
- Take the source vertex of the calculated starting edge as start node. Take the end vertex of the calculated ending edge as end node.
- Use the shortest_path function provided by pgRouting to get the quickest path passing it the start and end node and the cost information of the edges based upon the date time information
- Shortest_path function returns the edges and their cost. From these edges calculate their corresponding start and end locations
- Append the start and end locations of all the edges belonging to the shortest path into a gpx file
- Return the gpx file to the client

# References


[1] http://www.openstreetmap.org/

[2] http://developer.android.com/

[3] http://osmand.net

[4] http://wiki.openstreetmap.org/wiki/Elements

[5] http://wiki.openstreetmap.org/wiki/Elements

[6] http://www.topografix.com/gpx.asp

[7] http://www.postgresql.org

[8] http://wiki.openstreetmap.org/wiki/Osm2pgsql

[9] http://postgis.refractions.net

[10] http://www.pgrouting.org

[11] http://workshop.pgrouting.org/chapters/php_server.html