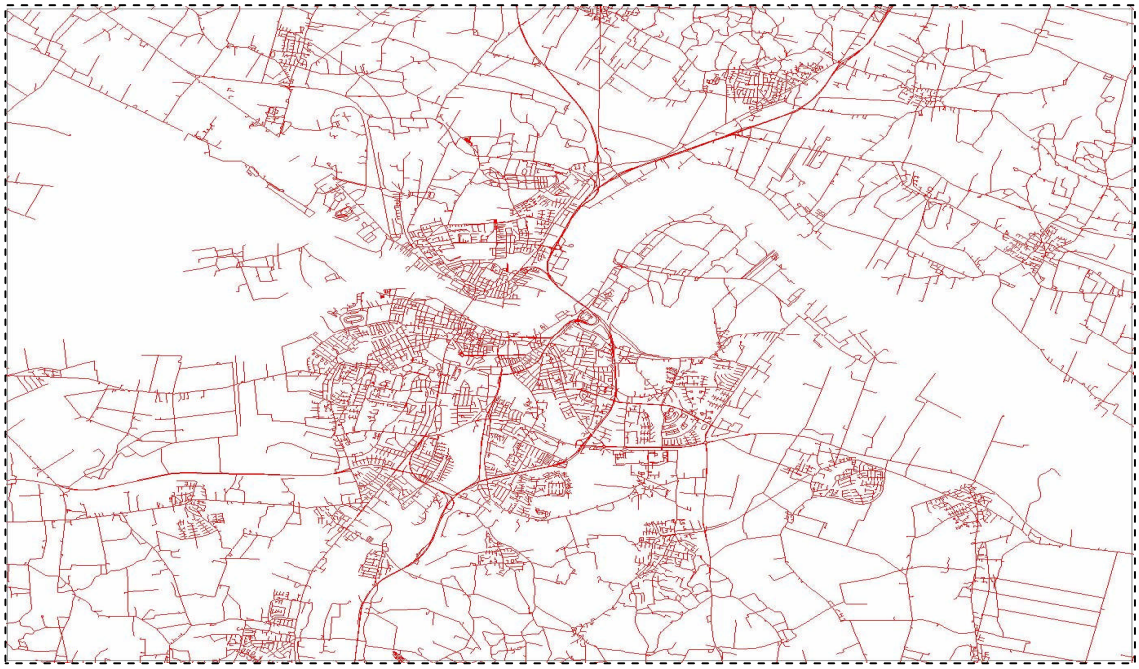
Road network travel-time calculation using GPS data



Department of Computer Science
Aalborg University

PREFACE

This article has foundation on previous work by the same author [21]. Section 1 has been rewritten, and new information is added. Section 2 has been rewritten, and new information is added. Section 3 has been completely rewritten. Section 4 has minor changes, added subsection 4.2.3, figures in subsection 4.1.1 are changed, but the rest of the section contains the same information. Section 5 and 6 are completely new. Section 7 has been completely rewritten.

I would like to thank my supervisor, Kristian Torp, for his support, guidance and useful comments during this and previous project work.

_____

Nermin Mudzelet

Road network travel-time calculation using GPS data

Nermin Mudzelet
Aalborg University Dept. of Computer Science,
Aalborg Ø. 9220, Denmark
nerminm@cs.aau.dk, nermin_m@yahoo.com
KDE4
Group d622b
Jun 2007

ABSTRACT

The number of vehicles on the roads worldwide is increasing each year. The consequence is that the travel-time of the trips is changing each year. It is very frustrating when you discover during the trip that the travel-time gotten from the available services on the internet or other vendors is very unrealistic. In this paper, we propose solution where we calculate travel-times that are more reliable. In order to achieve this we are using a data that is recorded with a GPS (Global Position System) equipped vehicles. We are using this data to discover historical speed patterns, and then these patterns are used to determine the travel-times. In addition, we propose several fallback methods, which handle the cases where we do not have enough data for some parts of a road network. Our solution can be used in domain of the traffic analyses and planning. We can provide the reports with the speed patterns for the parts of the road network. We use in our implementation the shortest travel-time path to verify that our solution can be also used in the navigation and the route planning.

1. INTRODUCTION

In recent years, we are witnesses of very rapid development of the navigation technology, and especially in area of the satellite navigation.

The satellite navigation GPS (Global Position System) has become very popular, and many people are using this navigation to plan trips and routes. With standard GPS device, we can determine the current speed of the movement and the geographical location. This information among other can be stored in logs. We can later use these logs for processing. The GPS navigation inherits some inaccuracy when determining geographical location. This is because of the obstacles that can block the satellite signal that the GPS device is receiving. In the past, the inaccuracy was about 100 meters, but in nowadays this inaccuracy is about 10-20 meters.

Main advantage of the GPS navigation is it´s accessibility and low cost of implementation. In the past, if we want to collect information about the traffic condition we need to install and maintain expensive equipment into roads such as loop detectors [1].

When using a GPS device for the trip or route, the travel-time that is obtained from the GPS device in the most cases is calculated based solely on the speed limits of the roads. This is also case in the most of web sites [9, 13] that are offering services for the route and trip planning. This leads to unrealistic travel-times, because many real traffic conditions are not taken in consideration, such as stops at the traffic lights and traffic jams.

In this work, we propose a method to supply travel-time weights for the segments in the road

network. These weights are based on the historical travel-times gotten from a GPS logs. We store these weights in the data warehouse. In addition, we are proposing fallback methods. These methods are introduced in situations when we do not have enough a GPS data.

Moreover, we implemented application for finding shortest travel-time path in the road network. Our application is based on A* [12] algorithm. The other path finding algorithms can be used, but we choose this algorithm because it can be optimized to yield good performance results.

The paper is organized as follows. In Section 2 we discuss about related work that is done in areas considering our work. Section 3 gives overview of our system architecture. In section 4 we introduce data model that is used in our system. Here we describe the input data and the data in the data warehouse. Section 5 describes methods for the travel-time calculation and the fallback methods. In section 6, we evaluate our approach to others in terms of the route average travel-time. In addition, we provide report about the speed fluctuation during day for given route. At the end, in Section 7 we conclude and give the direction for the future work.

2. RELATED WORK

In this section, we review previous work related to our project. We will focus on the topics like data collection, data storage and retrieval, travel time calculation, and shortest path algorithms. Each of these topics will be considered in context of the traffic domain.

In order to get into any kind of the traffic research, the very important thing is to have a data about the traffic. The most used methods to collect a data about the traffic are induction loops [1], observations from air and space [2] and floating car data [3]. Each of these methods has its advantages and disadvantages. In our project, we are using the floating car data method, where cars are equipped with the GPS devices. The main advantage of this method is its accessibility and low cost of the implementation.

The papers [4, 5, 6 and 7] described how the collected data and the road network data could be stored and retrieved in a data warehouse. These approaches take in consideration the complexity of the data. It means that the data can have spatial-temporal context together with simple data types. Here by the simple data types we are thinking of the data types like numbers, characters and boolean. We also use the data warehouse model to store our data.

In [8] Pfoser *et al*. focus their work to derive dynamic weights from historical data that is collect with the floating car data method. With the dynamic-weight approach, they take into consideration spatial and temporal aspect of the traffic. This is a different approach then some vendors [9, 13] that provide route-planning use. They use a static weights approach, where the road segment speed is associated with the road speed limit.

The dynamic weights that are calculated for every road segment in the network are stored in spatial-temporal data warehouse.

In their paper, they also introduce methods to calculate dynamic weights for places on the map where the data coverage is not good. The main difference between our and their approach is in these methods. They use spatial neighboring queries in other to compensate a missing data. These queries can be very costly in terms of CPU time and I/O, and they depend a lot on the size of the network. In our approach, we use knowledge about how the road segments are organized in the map to handle the case where we have no data coverage. It means that each road segment in the road network belongs to some street. We use this information among other to help us where we do not have enough data.

In [10] Zou *et al*. used GPS equipped taxi vehicles to collect data for arterial speed studies. They showed that the accuracy of travel-time increases with the sample size. We differ from them in that way that they do not consider the road segments where the data is missing.

Kanoulas *et al.* [11] propose a solution for calculating a path travel-time from source node to end node in the road network based on the A*[12] algorithm. In their solution, they lay emphasis on the importance of the user's arriving or leaving trip time. The consequence is that the path travel-time is function of leaving or arriving time. They used term fastest-path to describe the path that is generated as a result of their solution. The fastest-path is a generalization of shortest-path in the sense that the cost measure travel-time varies over time. In our approach, we are using an updated version of the A*[12] algorithm, but our cost measure travel-time do not vary over time.

3. ARCHITECTURE

In this section, we will introduce overview of our system architecture. We will describe basic concepts behind our system implementation.

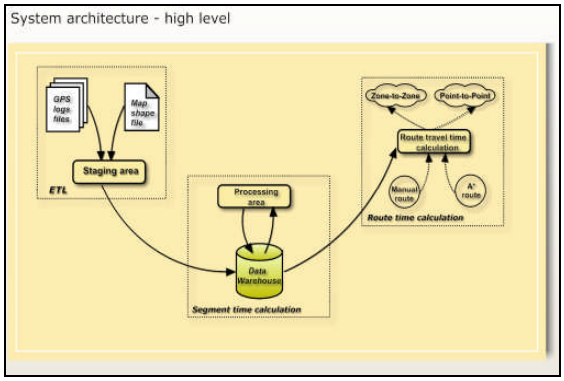


Figure 1. System architecture

Our system architecture consists of three parts. These parts are connected and they interact with each other. This is shown on Figure 1. A data flow is denoted with solid arrows, while with doted arrows we denote choices from a user.

We called this part ETL (Extract Transform Load) process. Main activity in the first part is to prepare data for later processing. This part is important, because if the data contains errors, these errors can have a big impact on a whole system. In addition, we have data from multiple sources. In order to prepare data, we had to develop for each source a different method. Moreover, each method consists of part where we extract data from the source. We have two types of input data. First type is a GPS log data, and second type is a road network data. More details about the data model are given in Section 4.

Afterwards we have part of the method where we transform data to a common format that we use in our system. In addition, here we use a map-matching algorithm to match entries from the GPS logs to road segments in the road network map.

At the end, we have part of the method where we load data into the data warehouse. Before the data is loaded in the data warehouse, it is kept in a staging area.

In the second part of the system architecture, the data that is stored in the data warehouse is processed. We use different algorithms to process the data. This is described in Section 5. After the data is processed, we get an average travel time for each road segment in the road network. Then the data is again stored in the data warehouse in another table. This part of the system architecture is called a segment time calculation.

The next part of the architecture is called a route time calculation. In this part, we are using the segment travel time to calculate the travel time from point A to point B (GPS coordinate points). In our approach, we use the A* path finding algorithm. The result of using the path-finding algorithm is a route or path from point A to point B with shortest travel-time cost. In addition, we can use this approach to calculate the travel-time from one zone to another zone in a road network map. Another possibility is to enter manually route

from point A to point B, and then calculate the travel-time.

4. DATA MODEL

In this section, the data model that we use in our system will be introduced. We will see how the data model is organized and used.

4.1. Input Data

In our system, we have two types of input data: road network data and GPS log data.

The road network data represents a digital map of Nordjyllands Amt. It consists of a 121.463 line elements that describe roads in the network map. This map initially was in shape files [17].

GPS data is data that we use as input for our system that is collected from a GPS device equipped vehicles. The data that is collected from the vehicles is then stored in a GPS log files. These files are flat files.

4.1.1. Road Network Data

In this paper, the road network is defined in terms of polylines (corresponding to a road parts between two junctions) and connections (corresponding to junctions) between road segments.

The domain of polylines is defined as

$$Polyline = (id,\ streetcode,\ speedlimit,\ geom);$$
$$id \in int,\ streetcode \in int,\ speedlimit \in int,\ geom \in spatialtype.$$

Hence, a polyline description consists of an identifier *id*, *streetcode* that tells us to which street the polyline belongs, *speedlimit* indicating allowed speed on that road segment, *geom* describes the polyline geometry. In practice, we can have different types that describe the geometry. For example, we could have the types like point, line and rectangle. In this case, the geometry is type of line. Each polyline consists of points that are connected. The minimal number of the points is two. A point is defined like (*x, y, m*) where $x, y \in number$, and this represent the coordinates of a point, $m \in number$, this indicates measure of the point, or distance of the point from beginning of the polyline. From Figure 2, we can see that S is start point with coordinates (10, 10) and measure equal to 0, because this is beginning of the polyline. A point E is end of the polyline, and it has coordinates (25.4, 18.39) and measure is 15.6. The number '102' defines polyline *id*.

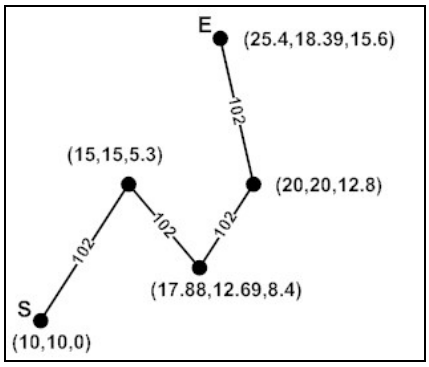


Figure 2. Example of polyline

The domain of connections is defined as follow:

$$Connection = (conn\_id,\ pol\_id,\ pol\_from)$$
$$conn\_id \in int,\ pol\_id \in int,\ pol\_from \in number$$

Conn_id is the identifier, *pol_id* is identifier of the polylines, and *pol_from* is the distance from a connection (junction) to a beginning of the polyline. On following Figure 3, we can see an example how the polylines are connected between each other.

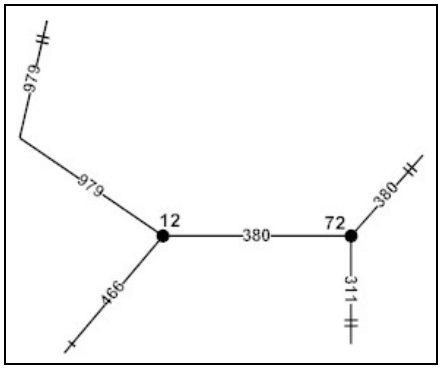


Figure 3. Example of connection

Each polyline has a unique identifier. This identifier corresponds to the *pol_id*. On the Figure 3, it is shown in the middle of the polyline. The connection between two or more polylines is denoted with the black dot. Moreover, each connection has its own identifier. This identifier corresponds to the *conn_id*. In our example we have two connections, and there are labeled with the numbers '12' and '72'.

The symbols ' —|— 'and ' —‖— ' denote start and end of the polyline, respectively. As we said earlier the *pol_from* attribute denotes distance between the connection point and the beginning of the polyline. For example on Figure 3, we can see the connection point with *id* equal to 72. The polylines with *ids* 311 and 380 are connected in this point. The beginning of the polyline 311 corresponds to position of the connection point 72. Therefore, the value of *pol_from* is equal to zero. However, for the polyline 380 *pol_from* has another value. The beginning of the polyline 380 corresponds to the position of the connection point. Therefore, the value *pol_form* in this case is equal to the distance between the connections points 12 and 72.

4.1.2. GPS Log Data

The GPS data that we are using in this project is defined as follows:

GpsData = (*vehicles_id, driver_id, date, time, latitude, longitude, course, speed*)

vehicles_id ∈ *int date* ∈ *date, time* ∈ *time, latitude* ∈ *number, longitude* ∈ *number, course* ∈ *int, speed* ∈ *int*.

Vehicle_id is an identifier of vehicle from which GPS entry is collected; driver_id is identifier of a person that has driven the vehicle; date and time values; longitude and latitude are parts of the GPS coordinate; course indicates direction of a movement of the vehicle, and it is represented in degrees; speed is the speed of the vehicle recorded for particular date and time value.

4.2. Data Warehouse Design

The data warehouse design will be introduced in this section. We will see how the data warehouse is organized and we will see description of the dimension tables and the fact table. On Figure 4, we have the data warehouse model. 'PK' and 'FK' abbreviations stand for primary key and foreign key, respectively. In addition, the abbreviation 'int' is the integer data type.

4.2.1. Dimensions

The time dimension is structured as follows. Attribute *time_id* uniquely defines the time dimension. Attributes *hour* (from 0 to 23), *minute* and *second* define the time granularity. Important thing that we can observe for the time dimension is after we create it, then it is never changed after that.

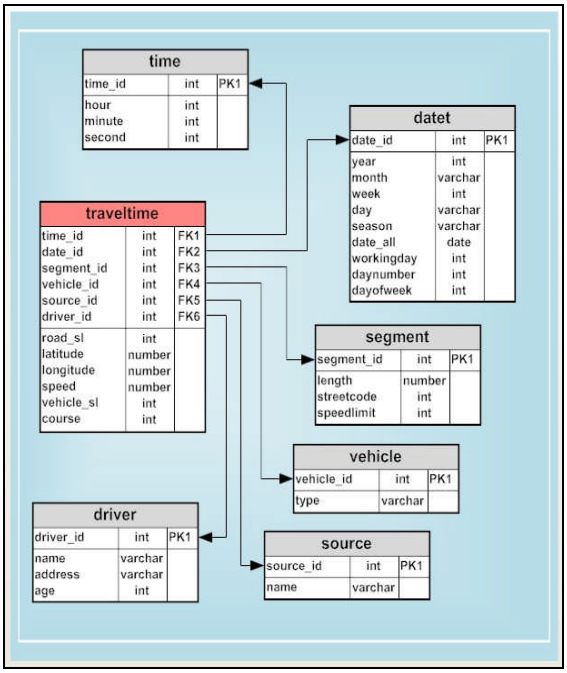


Figure 4. Data warehouse model

The date dimension structure is formed like as follows. A *date_id* uniquely defines this dimension. A *year* attribute determines the year when the GPS data is recorded. A *month* attribute describes months (January - December) in the year. A *week* attribute describes a week number in the year. Attribute *day* describes day names in the week. A *season* attribute describes a year seasons (spring – winter). A *date_all* attribute describes date in the database format. Attribute *working day* determines choice between working days (Monday-Friday) and weekend days (Saturday-Sunday). A *daynumber* attribute shows a day number in the month. A *dayofweek* attribute shows a day number in the week.

The time and date dimensions add temporary context to the data in the data warehouse. With these dimension we can choose different types of the temporary granularity.

The segment dimension relates to a single road link in the road network. The segment dimension is map depended. Each segment has a length and a speed limitation. There is also an attribute *streetcode* in the segment dimension. This attribute denotes to which street belongs given segment.

The vehicle dimension has two attributes, a *vehicle_id* and *type*. The first one uniquely defines the vehicle dimension. The second one relates to what kind of vehicle is used. In practice, buses or trucks have limitation of maximal speed that they can be driven. Moreover, it is important in some analyses to have this kind of information. In this project, we are using only one value for the vehicles type. The reason is that we only have the GPS data with one type of a vehicle. The type of vehicles that we use is a car.

The driver dimension describes driver that is involved in collecting the GPS data. We have attributes driver *name*, *address* and *age*. In addition, there is attribute *driver_id* that uniquely identifies a driver. Information about driver can help in situations when we want to determine why we have some deviation in the speed patterns. In this project, when we planed design of the data warehouse, we had included the driver dimension. However, in the GPS data or in any additional data we did not receive information about drivers. For this reason, the values of vehicles ids from the vehicle dimension correspond to the values of driver ids from the driver dimension.

The source dimension relates to the source of the GPS data that is used in the data warehouse. Therefore, we could have data that is not from the same source. In our system, we are using the data from the Bektra company and the project called “Spar på farten”.

4.2.2. Facts

A fact table ‘traveltime’ is the primary table in the data warehouse model. In this table we store a GPS observations from the GPS data logs. The GPS observation corresponds to an entry in the GPS data log that is recorded with a GPS device at a particular time and a date.

Information that we can get from a single row in the fact table should be as follow. For some time

with *time_id* and date with *date_id* values, there is a vehicle with *vehicle_id* and driver with *driver_id* that was travelled with speed value *speed* on the road segment with *segment_id*.

The current position of the vehicle is recorded in *latitude* and *longitude*, and the data is supplied from source with a *source_id.* In addition, we have a two speed limits, the road speed limit *road_sl* and the vehicle speed limit *vehicle_sl.* The road speed limit is put in the fact table because sometimes the speed limit can be change during the day or week. Then it is easy to change that attribute value for each entry in the fact table. The vehicle speed limit is put in the fact table because we could have different types of the vehicles where the GPS data is recorded. For example, the speed limit for the busses and trucks can be different from the road speed limit.

A *course* attribute is providing us with a direction of the vehicle movement. In addition, the street segment can be one or two way directional. In our project, we are using un-direction approach. However, this approach can be rather easily extended to support the bi-direction heading.

In order to design and implement the data warehouse we used guidance from [14, 15 and 16]. We used books [14] and [15] to get familiarized with the data warehouse concepts. These concepts include ETL stage, fact and dimension tables, and granularity. In Section 3 we discus about ETL, and in this section we discuss the fact and the dimension tables. The granularity determines the level of the details of the data in the data warehouse. When we specify more details then we can say that the level of granularity is lower. The guidance from books [14] and [15] was very useful in the phase where we design the data warehouse. In the phase of implementation, we used guidance from book [16]. In our approach, we are using a star schema data warehouse design. Moreover, the data warehouse is implemented in Oracle [18]. In particular, we use guidance in the index organization of the tables in the data warehouse and query tuning.

4.2.3. Calculation travel time table

In the data warehouse, we have a table where we store the data from the segment travel-time calculation process. This table is shown in Figure 5. The primary key of this table is composed of attributes *segment_id* and *dayperiod.* The first attribute describes for which segment we calculated the average time. The second attribute is a composite attribute. It contains name of the day and time period of the day. Example of one possible value of this attribute could be 'Monday0730'. It means that we calculate the average time for the Monday, and for the time between 07^{30} and 08^{00}.

avg_time		
segment_id	int	PK1
dayperiod	varchar	PK1
average_time	number	

Figure 5. Average time data warehouse table

Attribute *average_time* provides information about the average time for the road segment. The average time is measured in seconds.

5. TRAVEL-TIME CALCULATION

The goal of this paper is to produce a travel times from point A to B. In order to achieve this we are using the GPS data and road network map. From the GPS data, we can obtain information like speed, course, position, time and date of the vehicles movement in the road network. Each entry in the GPS log data is a single GPS observation. In our approach, we do not consider relation among GPS points, just how each GPS point influence the average speed of the road segment. In that context, we can call our approach

the point-based approach. In this section, we will go in more details about our approach.

5.1. Introduction to five-step approach

We have a five-step approach in order to calculate the average speed for every segment in the road network. We use this approach because it is very hard to get coverage of all segments in the road network. By coverage, we mean that for some segments we do not have any GPS observations. If we do not have values of the average speed for some segment, we cannot calculate an average time. The consequence of this is that, we cannot calculate an average time from place A to place B, if that segment is included in a route from A to B.

The average time for a single segment is calculated with a simple formula: the length of the segment divided by the segment average speed.

In our approach, we are assuming that the segment is un-directional. It means that the average speed is the same if you consider traveling from the beginning of the segment to the end, and vice versa. In addition, for the GPS observations that are above a speed limit for that segment, they are lowered to the speed limit. This is done because we want to avoid that the calculated average speed is bigger then the road segment speed limit. We also want to lower the impact of speedy drivers on the road segment average speed.

Figure 6 shows a road network. On the Figure 6, we can see the road segments and the connections between the road segments. Each road segment has a unique id. This id corresponds to the numbers on the figure. The blue line denotes a street that we will call the blue street. In addition, the red line denotes a street that we will call the red street. The segments that belong to the blue street are 1, 2, 3 and 7. Moreover, the segments that belong to the red street are 9, 10 and 11. The other segments 4, 5, 6 and 8 are single-segment streets. These streets are denoted with the black line.

The green dots denote the GPS observations that are map-matched to a segment.

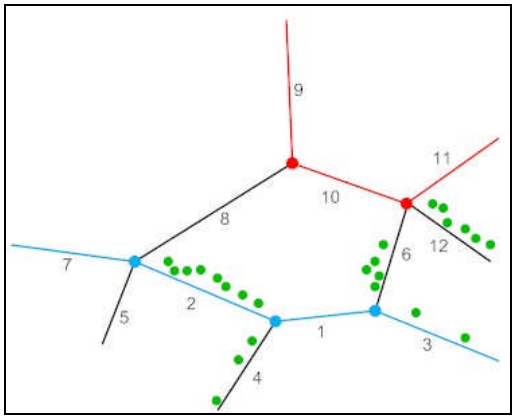


Figure 6. Part of the road network

Our goal is to calculate the average speed for all segments in Figure 6. The reason is that we want to be able to answer the question how much time do we need to travel between any two points in the road network. From Figure 6 we can notice that some of the segments in the road network do not have any GPS observations that could be map-matched to them. These segments are 1, 5, 7, 8, 9, 10 and 11. This is problem where we do not have the GPS data for the segments. The reasons can be that a segment is not passed enough or the length of the segment is small. Another problem that can appear is that we do not have enough GPS observations. This problem is illustrated in Figure 6, where segments 3 and 4 do not have enough GPS observations. This problem rises when we have less then five observations on some segment. It means that the calculated average speed may be biased. The number of five observations is based on the informal studies. In our implementation, this number can be easily changed with another number.

In other to calculate the average speed for all segments in the road network and to handle the problems, we mention above, we introduce a five-step approach.

1. In our first step, we are calculating the average speed based on the speed from the GPS observations. For example in Figure 6, we have segment 2. We will take all GPS observations that are map-matched to this segment. Then we will calculate the average speed for this segment. In addition, if we have GPS observations where its speed is greater then the road speed limit, the speed will be lowered to the speed limit. The same process can be applied to segments 3, 4, 6 and 12. In Table 1 we can see the values that are calculated after the first step.

2. In the second step, we calculate the average speed for segments that do not have enough GPS observations. In Figure 6, we can see that segments 3 and 4 have two and three GPS observations, consequently. When we experience this situation, we will use a simple formula to calculate new average speed. More details about it is given in Section 5.2. From the column step 2 of the Table 1 we can see which values are computed after the second step.

3. In the third step, we group the segments into streets. The idea behind this step is to calculate the average speed for a street, based on the average speed of all street segments. Then we use the street average speed to update values of the segments average speed.

From Figure 6, we will select two streets, the blue street and the red street. We can see that segments 1 and 7 that belong to the blue street. These segments do not have any GPS observations, and the consequence of that is they do not have the average speed. This can be seen in Table 1, columns step 1 and step 2 have no values for these segments. However, the segments 2 and 3 have the average speed values. We can use these values and compute the average speed for the blue street.

In addition, we assume that all segments in the blue street have the same road speed limit. After we compute the average speed for the blue street, then we update values of the segments average speed with the value of the street average speed. This is shown in column step 3 of Table 1.

If we take in the consideration the red street, we can see there are no GPS observations on any segments that belong to the street. The consequence is that we cannot calculate the average speed for this street in this step.

4. We still have segments where we do not have the average speed. These segments are 5, 8, 9, 10 and 11. We can see that in Table 1, these segments do not have values after the three steps. In the fourth step, we use the average speed of the neighboring segments. In addition, the neighboring segments need to have the same road speed limit. For example, we will select the segment 10, and this segment does not have the average speed. In order to calculate the missing value for this segment, we first find the neighboring segments. These segments are 6, 8, 9, 11 and 12. From these segments, we choose only the segments 6 and 12, because only these segments have the average speed. We assumed that segments 10, 6 and 12 have the same speed limit. The average speed of the segment 10 then becomes the average speed of the average speeds from the segments 6 and 12.

This step can be also applied on the segments 5 and 11. On the segments 8 and 9, we cannot apply this step because the road speed limit on these segments is different from the road speed limit of theirs neighboring segments. The result of this step can be seen in column step 4 from Table 1.

5. In the step five, we are using the road speed limit to calculate the average speed. We can see from Table 1 that segments 8 and 9 do not have the average speed after we use the four steps. In order to calculate their average speed, we multiply the road speed limit value with some threshold. This means that we can use different threshold for the segment in the downtown and the peripheral parts of the city. Here, we will use the same

threshold for all segments, and value is 0.8. The threshold is introduced because we want to include situations like the traffic stops and turns.

In the third and fourth step, we can experience situations where some group segments do not have same speed limit. If we for example have situation in the third step where the segments in a street do not have the same speed limit, we will then proceed to the fourth step, to calculate the average speed. If this situation appears in the fourth step, we will proceed to the step five.

In Table 1, we have shown how the average speed is calculated throw the five-step approach. A columns step 1 - step 5 denote steps in our approach. The segments in Table 1 correspond to the segments in Figure 6. The speed limit and the average speed are denoted in kilometers per hour (KM/h). The value of the average speed that is calculated in a particular step is denoted with the bolded and increased font.

segment	speed limit	step1	step2	step3	step4	step5
1	50	-	-	**48,25**	48,25	48,25
2	50	**50**	50	50	50	50
3	50	45	**46,5**	46,5	46,5	46,5
4	50	30	**34**	34	34	34
5	50	-	-	-	**50**	50
6	60	**40**	40	40	40	40
7	50	-	-	**48,25**	48,25	48,25
8	70	-	-	-	-	**56**
9	70	-	-	-	-	**56**
10	60	-	-	-	**35**	35
11	60	-	-	-	**35**	35
12	60	**30**	30	30	30	30

Table 1. Steps average speed

5.2. Five step algorithm approach

In describing an algorithm, we use a notation as follows. Arrays, lists and sets of records are denoted like a '*GpsLog'* or '*GpsAvgSpeed'*. A single instance of a record of the array or of the list is denoted like '*gpsEntry'* or '*segmentId'.* Variables where we store intermediate results in the algorithm are denoted like 'counter' or 'sum'.

The field inside record is denoted like '*gpsEntry .timeofday*' or '*GpsAvgSpeed.observations*'.

In cases where we have input variables to the algorithm, we use a notation like '*parameternumber*'. Another case where we need to assign to variable a value of no value, we use 'Null'.

The purpose of *Segment Average Speed* algorithm is to calculate the average speed for some segment for a different day period. The first step is to calculate the average speed from an existing data in the data warehouse. We divide every day of the week in a 48 time segments. The length of the time segment is 30 minutes. This length of 30 minutes can be easily changed to accommodate our needs. We used this length because it provides for us optimal results. If we used for example 15 minutes time segment, then we get in a situation where data coverage is getting worse, especially on the smaller road segments. We can use 15 minutes time interval if we have a very good coverage with the GPS data.

The input to the algorithm is following parameters: *GpsLog, Segment, Day and Time Period.* The *GpsLog* is array of records with fields: segmentid, timeofday, day, speed and speed limit. Single instance of the *GpsLog* is shown below. Array *Segments* is an array of numbers, and here we have segmentid from all segments in the road network. The *Days* is set of string, and here we have names of the days of the week. The *TimePeriods* is a set of time intervals of 30 minutes during one day.

gpsEntry {
segmentid : int,
day : string,
timeofday : Time,
speed : int,
speed limit : int}

Example of instance of record from array *GpsLog*

The output for the algorithm is a list *GpsAvgSpeed.* That is list of records with fields: segmentid, time interval, day, averagespeed and observations.

Algorithm 1: Segment Average Speed (*GpsLog*, *Segments*, *Days*, *TimePeriods*)

```
1  Input GpsLog: array of records,
2        Segments: array of numbers,
3        Days: set of strings,
4        TimePeriods: set of time intervals
5  Output GpsAvgSpeed: list of records
6  for each segmentId in Segments
7    do for each dayName in Days
8       do for each timeInterval in TimePeriods
9          do counter ← 0
10           sum ← 0
11           for each gpsEntry in GpsLog
12             do if segmentId = gpsEntry.segmentid
13                   and dayName = gpsEntry.day
14                   and gpsEntry.timeofday in timeInterval
15                 then if gpsEntry.speed > gpsEntry.speedlimit - - if speed
16                        then speed ← gpsEntry.speedlimit - - greather
17                      else speed ← gpsEntry.speed  - - use speed limit
18                      sum ← sum + speed
19                      counter ← counter + 1
20           end for
21           if  sum = 0
22            then GpsAvgSpeed.averagespeed ← Null - - output preparation
23           else GpsAvgSpeed.averagespeed  ← sum / counter  - - adding
                                                              - - new
24           GpsAvgSpeed.observations      ← counter   - - element
25           GpsAvgSpeed.day               ← dayName
26           GpsAvgSpeed.timeinterval      ← timeInterval
27           GpsAvgSpeed.segmentid         ← segmentId
28         end for
29     end for
30 end for
31 return GpsAvgSpeed
```

The main loop in algorithm starts from line 6 and ends on line 30. Here we are looping over *segmentId* from array *Segments*. We are doing this because we want to calculate the average speed for all segments. From line 7 to line 29, we have another loop. This loop is looping over *dayName* in the set *Days*. The purpose of this loop is to get all days in a week for calculation of the average speed. On line 8, a loop loops over *timeInterval* in set *TimePeriods*. This *timeInterval* represents different parts of the day, but each *timeInterval* has the same granularity represented in time. This loop ends on a line 28.

On lines 9 and 10, we have initialization of variables counter and sum. These variables are used in computation of the average speed.

From a line 11 to 20, we loop over *gpsEntry* in the array *GpsLog.* On lines 12-14, we are checking which *gpsEntry* should be involved in a calculation. Line 12 checks that correct segment are used. Line 13 checks for correct day, and line 14 checks for correct time interval. If all parts of the condition from lines 12-14 are true, then we are proceeding to the line 15. Here we are checking is a speed value of the *gpsEntry* from the *GpsLog* more than speed limit value of the *gpsEntry.* If this is true then on line 16, we assign to variable speed value of speed limit value of *gpsEntry.* Variable speed just store temporarily value of field speed value of *gpsEntry*. Otherwise, if the condition resolves false, then on line 17, we assign to variable speed value of speed value from the *gpsEntry*.

On line 18, we are adding speed value to sum value. In addition, on line 19 we increase counter value with value 1.

On line 21, we are checking to see, did we use any speed value from the array *GpsLog.*

If we did not find any speed value for some *segmentId* then we assign Null to the field averagespeed in the list *GpsAvgSpeed.* This is done on a line 22. Else, if we have value then on line 23, we compute the average speed.

In addition, on lines 24-27 we are assigning corresponding values in the list *GpsAvgSpeed.*

For every corresponding value of *segmentId*, *dayName* and *timeInterval*, we are adding new element to the list *GpsAvgSpeed.*

The output of the algorithm is the list *GpsAvgSpeed* and the list is returned on line 31.

The main goal of Algorithm 2 *Less Observations* is to search for segments where we have less GPS observations then some number. After such value is detected, then it uses a simple formula to compute an average speed.

The inputs to the algorithm are the following parameters: *GpsAvgSpeed, SpeedWeights* and *parameternumber*. The *GpsAvgSpeed* is array of records with the fields: segmentid, timeInterval, day, averagespeed and observations.

The *SpeedWeights* is array of records with fields: paramavg and paramspeedlimit. In this array, we store values that we will use when updating average speed.

The *parameternumber* is number where we saying what is lower bound of the number of the GPS observations when we consider to update the average speed.

Output is a modified *GpsAvgSpeed* array of records.

Algorithm 2: Less Observations (*GpsAvgSpeed, SpeedWeights, parameternumber*)

```
1   Input GpsAvgSpeed      : array of records,
2         SpeedWeights     : array of records,
3         parameternumber  : number
4   Output GpsAvgSpeed     : array of records
5   for each gpsAvgSpeedRecord in GpsAvgSpeed
6     do if gpsAvgSpeedRecord.observations < parameternumber
7          then gpsAvgSpeedRecord.averagespeed← SpeedWeights.paramavg *
8                                  gpsAvgSpeedRecord.averagespeed
9               + SpeedWeights.paramspeedlimit* gpsAvgSpeedRecord.speedlimit
10  end for
11  return GpsAvgSpeed    - - output of algorithm
```

Line 5-10 loops over *gpsAvgSpeedRecord* in array *GpsAvgSpeed*. On line 6, we have a condition to check if the number of the GPS observations is less then a value of *parameter number*. If the value of filed observations from the *gpsAvgSpeedRecord* is less, then we go to line 7. Here we update value of the average speed in the *gpsAvgSpeedRecord*. In order to update this value we use a formula. This formula is shown on lines 7-9. With value of *SpeedWeights.paramavg,* we are multiplying a value *gpsAvgSpeedRecord .averagespeed*. In addition, with value of *SpeedWeights.paramspeedlimit,* we multiplied value of *GpsAvgSpeedRecord.speedlimit.* The result from the first multiplied action is added to result of the second multiplied action. Then with this new computed value, we update the average speed value of *GpsAvgSpeed Record.*

Conceptually the formula calculates the average speed based on the different value of the number of GPS observations. It means that more GPS observations we have on the road segment, the more the average speed of these observations has impact on the new value of *gpsAvgSpeedRecord .averagespeed*. This influence of the segment average speed and the segment speed limit on the *gpsAvgSpeedRecord.averagespeed* new value is determined by values of parameters *SpeedWeights. paramspeedlimit* and *SpeedWeights.paramavg*. The sum of these parameters should always return value one.

On line 11, we return a modified version of *GpsAvgSpeed* array if we made some changes. Else, returning the array will be same as the input array *GpsAvgSpeed*.

Algorithm 3 *Street Average Speed* is needed when we do not have enough data from the first two algorithms to calculate average speed for every segment. In this algorithm, we are trying to compute the average speed for segments that we did not compute in first two steps. We assume that segment in the same street and the same speed limit should experience similar behavior. Because we calculated the average speed for some segments in first two steps, then we can use these values to calculate average speed for a whole street, even if we have segments that do not have any GPS observations.

The input to the algorithm is two arrays. The first GpsAvgSpeed is an array of records with the fields segmentid, streetcode, day, time interval, observations and averagespeed. The second array *GpsStreetAvgSpeed* has the same field attributes as array *GpsAvgSpeed*.

The output of algorithm is a modified array of records *GpsAvgSpeed.*

Algorithm 3: Street Average Speed (*GpsAvgSpeed, GpsStreetAvgSpeed*)

```
1  Input   GpsAvgSpeed        : array of records,
2          GpsStreetAvgSpeed  : array of records
3  Output  GpsAvgSpeed        : array of records
4  for each avgSpeedEntry in GpsAvgSpeed
5    do if avgSpeedEntry.averagespeed = Null
6       then sumobservation ← 0
7            sum ← 0
8            counter ← 0
9            for each gpsStreetAvgEntry in GpsStreetAvgSpeed
10             do if avgSpeedEntry.streetcode = gpsStreetAvgEntry.streetcode
11                  and gpsStreetAvgEntry.averagespeed ≠ Null
12                  and avgSpeedEntry.day = gpsStreetAvgEntry.day
13                  and avgSpeedEntry.timeinterval = gpsStreetAvgEntry.timeinterval
14                  and avgSpeedEntry.speedlimit = gpsStreetAvgEntry.speedlimit
15                then sum ← sum + gpsStreetAvgEntry.averagespeed
16                     counter ← counter + 1
17                     sumobservation ← sumobservation +
18                                      gpsStreetAvgEntry.observations
19           end for
20           if sum ≠ 0
21             then avgSpeedEntry.averagespeed ← sum / counter
22                  avgSpeedEntry.observations  ← sumobservation
23 end for
24 return GpsAvgSpeed
```

From line 4 to 23, we have the main loop. We loop over the elements in the array *GpsAvgSpeed*. When we find a record which average speed value is missing we go to line 6, else we go to the next record from array. A condition that decides this is on line 5. On lines 6-8, we initialize variables sumobservation, sum and counter.

From line 9 to line 19, we are looping over elements in the array *GpsStreetAvgSpeed*. Furthermore, from line 10 to line 14, we have the conditions that decide which segments from the array *GpsStreetAvgSpeed* will enroll in computation of the average speed. These conditions say that value of the street code from an *avgSpeedEntry* needs to be same as streetcode value from a *gpsStreetAvgEntry*. Moreover, the average speed value of the *gpsStreetAvgEntry* should not be a Null value. In addition, the day and timeInterval should match from both *avgSpeedEntry* and *gpsStreetAvgEntry*. In addition, both entries should have the same speed limit.

When all the conditions are true, then we add the value from *gpsStreetAvgEntry.averagespeed* to the variable sum on line 15. Furthermore, on line 16, we increase value of counter variable by value 1. The counter variable is used to remember how many segments are used to calculate the average speed. Sumobservation variable on line 17 is used to remember the total number of observations for the segment.

Line 20 is reserved for a condition that is checking is sum variable different from zero. It means when the sum is equal to zero, then the average speed for that segment cannot be calculated. In addition, the values of the fields' averagespeed and observations of *avgSpeedEntry* are not updated.

Line 25 is the last line of the algorithm, where we return a modified version of the array *GpsAvgSpeed.*

In the Algorithm 4 *Neigbours Avg Speed*, we are trying to fill missing values that we did not compute in the first three steps. Here we are using a neighbor approach. It means that for example, we have segment that do not have an average speed, and we want to compute that value with observing segments neighbors. We assume that with less accuracy then third step, that neighboring segments with same speed limit will experience similar behavior.

The input to the algorithm is three arrays. The first *GpsAvgSpeed* is an array of records with the fields segmentid, streetcode, part of day and averagespeed. The second array *TempAvgSpeed* has the same field attributes as the array

GpsAvgSpeed. The third array *Neighbors* is array of record with the fields segmentid and segmentneighbourid. This is an array where each segment and its neighboring segments are stored. In addition, all neighboring segments have the same speed limit value.

The output of the algorithm is a modified array of records *GpsAvgSpeed*.

Algorithm 4: Neigbours Avg Speed (*GpsAvgSpeed*, *TempAvgSpeed*, *Neighbors*)

```
1  Input   GpsAvgSpeed    : array of records,
2          TempAvgSpeed   : array of records,
3          Neighbors      : array of records
4  Output  GpsAvgSpeed    : array of records - - this array is modified
5  for each avgSpeedEntry in GpsAvgSpeed
6    do if avgSpeedEntry.averagespeed = Null
7       then sumobservation ← 0
8            sum ← 0
9            counter ← 0
10           for each neighborsEntry in Neighbors
11             do if avgSpeedEntry.segmentid = neighborsEntry.segmentid
12                  then for each tempAvgSpeedEntry in TempAvgSpeed
13                         do if neighborsEntry.segmentneighbourid =
14                                  tempAvgSpeedEntry.segmentid
15                            and tempAvgSpeedEntry.averagespeed ≠ Null
16                            and avgSpeedEntry.day = tempAvgSpeedEntry.day
17                            and avgSpeedEntry.timeinterval=
18                                      tempAvgSpeedEntry.timeinterval
19                            then sum ← sum + tempAvgSpeedEntry.averagespeed
20                                 counter ← counter + 1
21                                 sumobservation ← sumobservation +
22                                                  tempAvgSpeedEntry.observations
23                                 break  - - when record is find, then leave loop
24                       end for
25           end for
26           if sum ≠ 0
27            then avgSpeedEntry.averagespeed ← sum / counter - - output preparation
28                 avgSpeedEntry.observation    ← sumobservation
29 end for
30 return GpsAvgSpeed
```

From line 5 to line 29, we have the main loop. We are looping over the elements of the array *GpsAvgSpeed.* We can see that on line 6 we have a condition that is used to find records in the array *GpsAvgSpeed* who is missing a value for the average speed field. When this condition is satisfied we go to line 7, otherwise we go to next record from the array *GpsAvgSpeed*. On lines 7-9, we initialize variables sumobservation, sum and counter.

From line 10 to line 25, we are looping over elements of array *Neighbors.* A condition on line 11 is telling us that we only need records from array *Neighbors* whose value of filed segmentid is equal to filed segmentid value from array *GpsAvgSpeed.*

When the condition is satisfied, we continue to line 12. From line 12 to line 24, we have another loop. In this loop, we are looping over elements of the array *TempAvgSpeed.* The meaning of the conditions placed from line 13 to 18 is that only records from the array *TempAvgSpeed* who have the same segmentid filed value equal to segmentneigbourid filed value from array *Neighbors* will be selected. Moreover, day and timeinterval values of *avgSpeedEntry* should be equal to the corresponding day and timeinterval values in *tempAvgSpeedEntry.*

When all the conditions are satisfied, then we add values to variables sum, counter and sumobservation.

When we find satisfied record from array *TempAvgSpeed* then we break from the loop. This is done on line 23.

Line 26 is reserved for the condition that is checking is a sum variable different from zero. It means when sum is equal to zero, then the average speed for that segment is not calculated. In addition, the values of the fields averagespeed and observations of *avgSpeedEntry* are not updated.

On line 30, we are returning a modified array *GpsAvgSpeed*

6. IMPLEMENTATION

In this section, we will describe details about our project that are more technical. In addition, we will perform several experiments to verify our system implementation.

6.1. Technical details

In other to implement our system, we used Java programming language [19] and PL/SQL [20].

Java is used in ETL, map-matching and A* implementation. The number of the code lines is around 3100.

PL/SQL is used in the map-matching, A* and in implementation of the algorithms described in Section 5.2. The number of the code lines in PL/SQL is around 1550.

In Table 2, we have summarized description of the tables in the data warehouse. In this table, we described in a column size, the size of the tables in terms of the storage. The size is denoted in MB. In addition, the number of rows for the each table is given in column nb. of rows. With 'M', we denote million of rows.

table name	size (MB)	nb. of rows
traveltime	632	8.1M
time	2	86400
datet	0.5	7305
segment	11	121463
vehicle	0.06	401
source	0.06	2
driver	0.06	401
avg_time	310	5.84M

Table 2. Description of tables in the data warehouse

6.2. Experiment

In other to verify our system, we conducted an experiment. In our system, we have two ways of selecting the route for the travel-time calculation. We can either manually select (user defined route) the route or we can enter the start and the stop point of the route, and then used A* (automatically defined route) to determine the route. For the experimental purposes, we will use both ways.

In this experiment, we take one route in the road network. The route corresponds to the Østre Alle street in Aalborg. We could use any route between two points in the road network map. We choose this route because it is very frequent and very important route in the traffic of the Aalborg city. We used this route to calculate the average travel-time for day period 07:30-08:00. The route is shown on Figure 8.

The results that we get from the experiment we compare with several sources. Our sources of comparison are web sites that provide services for the travel and route planning.

The first source is the krak.dk [9]. This web site is specialized to provide travel planning from address to address in Denmark. The second source is the Google maps [13]. This web site provide among other, service for travel planning from a point to point. These points correspond to geographical points. They are given in form of latitude and longitude.

Another way of comparing is to drive our experimental route and measure travel-time, but we did not have equipment to complete that task.

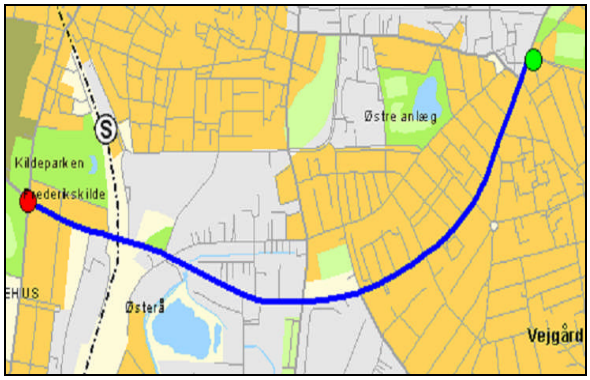


Figure 7. Krak.dk route

On Figure 7, we can see route that is taken from the krak.dk web site. The green dot denotes the start of the route. The red dot denotes the end of the route. Resulting route distance is 2.8 KM. Estimated travel-time is about three minutes.

On Figure 8, we have route that is taken from the Google maps web site. The route is denoted

with the blue line. Start of the route is denoted with the green dot. End of the route is donated with the red dot. Resulting route distance is 2.8 KM. Estimated travel-time is about four minutes.

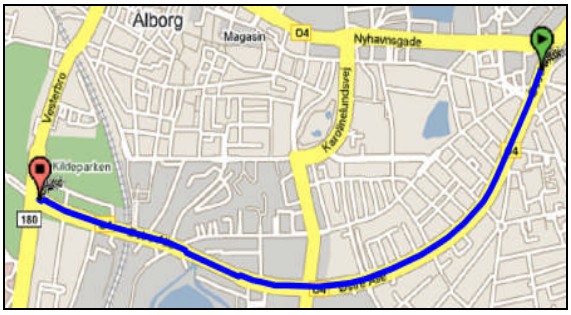


Figure 8. Google maps route

Input parameters for our manual approach are the road segments of the given route. This route corresponds to route shown on Figure 8. The route has a 39 road segments. The least length of the road segments in this route is three meters. The largest length of the road segment is 290 meters. The average length of the road segments is 74 meters. The total number of the GPS observations for the route is 599. The length of the route is 2.8 KM. Calculated travel-time is seven minutes and 49 seconds.

We also conducted our A* approach experiment. On Figure 9, it is shown the route of the A* approach. The route is denoted with the blue line. The green dot denotes the start of the route. The red dot denotes the end of the route. Input parameters for this approach are the same like in the Google map approach. The resulting route has a 49 road segments. The least length of the road segments in this route is five meters. The largest length is 270 meters. The average length of the road segments is 67 meters. The total number of the GPS observations for the route is 544. Result distance is 3.4 KM. Calculated travel-time is five minute and 54 seconds. Execution of A* approach for this route is 22 seconds.

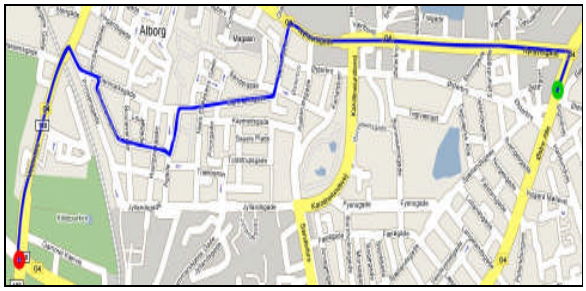


Figure 9. A* route

In Table 3, we have a summarized comparison of different approaches.

approach	distance(KM)	travel-time(min:sec)	average speed(KM/h)
Krak.dk	2.8	03:00	56
Google maps	2.8	04:00	42
Our manual	2.8	07:49	21.3
Our A*	3.4	05:54	34.6

Table 3. Summarized results

We can see in Table 3 that the krak.dk and the Google maps have an optimistic solution. Their estimation is mainly focused on the road speed limits. Their approaches do not consider condition in the traffic like the traffic light stops and rush hours. In our approach, we use knowledge from the GPS observations to take these traffic conditions in consideration.

The A* approach cannot be directly compared to rest of the approaches, because it is using a different route. It is putted in Table 3 because we want show that another route is possible with the less travel-time. However, we can compare routes from manual and A* approach in terms of the traffic conditions. The A* distance is greater, but the travel-time is lower. This is due to nature of the traffic in that part of the city in particular time of the day. In particular time of the day, we have a lot of vehicle that are moving from the west part of the city to the east part of the city. In addition, we have many traffic lights and junctions. We can see that the route on Figure 8 is congested in that part of the day. With A* approach we can choose another route even it is longer and has more turns, but it is less congested, and the travel-time is lower.

In addition, if we take the different part of the day, for example the time-period 19:00-19:30, and consider the same start and end point like in Figure 9. Then we conduct another experiment with A* approach. The resulting route is the same as route shown on Figure 8 and includes the same road segments like in the manual approach. Calculated travel time is 3 minutes and 58 seconds. In this part of the day there is less vehicles that are driving in this route. With this, we argumented the correctness of our approach.

6.3. Traffic analyses

One of the additional purposes of our approach can be used in the traffic analyses and planning. We can provide reports about traffic conditions in aspect of average speed or average time. With these reports, we can determine rush hours or frequency of the road segments.

On Figure 10, we have a diagram where we show how the route speed depends on the different day periods. In this example, a day is Monday. The route corresponds to the route on Figure 8. The solid line denotes the route speed. The number of the GPS observations for whole day period is 8889. The average speed of the whole day sample is 36.5 KM/h. Moreover, the average speed is denoted with the dashed line.

From Figure 10, we can see that from 0:00 to 5:00, we have that the speed is very close to the speed limit. This is because in this period there are a small number of vehicles that are driving on this route. From 5:00 to 8:00, we can see that the speed is decreasing, and around 8:00, it reaches the lowest value during the day. This corresponds to the traffic conditions in this part of the city, because this route is one the major roads in the city. During the 8:00-15:30 period, we have fluctuations in the speed. In the period close to 17:00, we have another decrease in the speed. This is also corresponding to the fact that many people are driving in this route. In the period close to 20:00, we have decrease in the speed. The reason for this is unknown to us. Nevertheless, we the information that is provided with our approach this period of the day can be then closely examine.

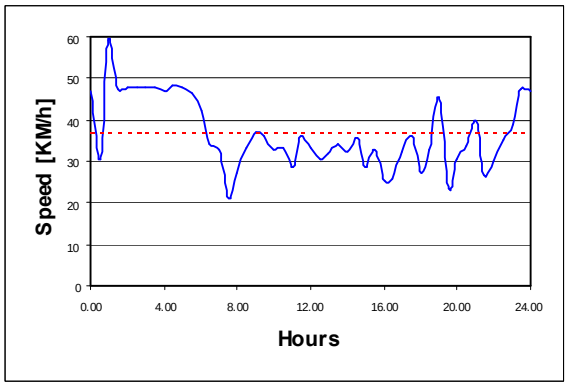


Figure 10. Day period speed report

7. CONCLUSION AND FUTURE WORK

In this work, we introduced methods for calculating travel-times for the road segments in the road network and fallback methods in cases where we do not have the GPS data coverage. These fallback methods are important because we want to be able to calculate the travel-time between any two points in the road network. The methods are described in means of algorithms and they are practically implemented.

We used the data warehouse as basis for all our computations. It is organized in terms of spatial-temporal context. It means that we store the data in the data warehouse that have spatial characteristics, such as road network map, and temporal characteristics, such as date and time attributes of the GPS observations.

In the experiment that we conduct, we compare our solution to other that are available and in use. We show that the route travel-time in our approach is slower than in others approaches. This is expected, because their approaches do not consider the traffic conditions. In the experiment, we also showed an implementation of the shortest travel-time path. The implementation is based the on A* algorithm. With this implementation, we showed that our approach could be used in the

navigation and the planning applications. We can conclude that the sampling interval of the GPS devices in our approach does not play crucial role. We can say that distribution of the GPS data on the road network is more important.

Focus in the future work will be on optimizing aspect of our solution. This can be done in execution of queries in the data warehouse. The data warehouse will grow and the queries should be more optimized in other to get acceptable execution times. This is important in aspect of the web services.

Moreover, optimization of the shortest travel-time path implementation is needed for bigger road networks. One of approaches could be to use partition of the network to the smaller parts, and then to apply the implementation on these parts. Another approach could be in optimizing memory resources that this implementation is using. In practice now we have implementations of double-sided memory bounded A* algorithm. We could use similar approach in our solution.

We will take in to the consideration the street segment heading. This could improve the correctness of our approach.

ACKNOWLEDGMENTS

We would like use this chance to thank Agne Brilingaite that gives us the guidance for the map-matching algorithm. In addition, we want to thank *Bektra* company and *Spar på farten* project that provided us with the gps data.

REFERENCES

[1] Karl Petty, Hisham Noeimi, Kumud Sanwal, Dan Rydzewski, Alexander Skabardonis and Pravin Varaiya. *The Freeway Service Patrol Evaluation Project: Database, Support Programs, and Accessibility*.

[2] DLR projects: LUMOS, Eye in the Sky. http://www.dlr.de/

[3] Ralf-Peter Schäfer, Kai-Uwe Thiessenhusen, and Peter Wagner. A Traffic Information System by Means of Real-TimeFloating-car Data. In *Proc. ITS World Congress*, Chicago USA, 2002.

[4] Sotiris Brakatsoulas, Dieter Pfoser and Nectaria Tryfona. Modelling, storing and mining moving objects databases. In *IDEAS '04*: *Proceedings of the International Database Engineering and Applications Symposium (IDEAS'04)*, pages 68-77. IEEE Computer Society, 2004.

[5] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, Yufei Tao. Query Processing in Spatial Network Databases. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003

[6]Elzbieta Malinowski and Esteban Zimányi. Representing Spatiality in a Conceptual Multidimensional Model. In: *Proceedings of the 12th annual ACM international workshop on Geographic information systems*, November12-13, 2004 Washington DC, USA

[7] Christian S. Jensen, Augustas Kligys, Torben BachPedersen, and Igor Timko. Multidimensional data modeling for location-based services. *The VLDB Journal,* 2004.

[8] Dieter Pfoser, Nectaria Tryfona and Agnes Voisard. Dynamic Travel Time Maps – Enabling Efficient Navigation. In *SSDBM'06* : *Proceedings of the 18th International Conference on Scientific and Statistical Database Management*.

[9] http://www.krak.dk/

[10] Liang Zou, Jian-Min Xu, and Ling-Xiang Zhu. Arterial Speed Studies with Taxi Equipped with Global Positioning Receivers As Probe Vehicle. IEEE, 2005.

[11] Evangelos Kanoulas, Yang Du, Tian Xia and Donghui Zhang. Finding Fastest Paths on A Road Network with Speed Patterns. In *ICDE'06*: *Proceedings of the 18th International Conference on Scientific and Statistical Database Management*. 2006.

[12] S. Russell and P. Norvig. *Artificial Intelligence : A Modern Approach. Prentice Hall*, Englewood Cliffs ,NJ ,2nd edition, 2003.

[13] http://www.maps.google.com

[14] Ralph Kimball, Margy Ross. *The Data Warehouse Toolkit*, Second Edition, 2002.

[15] W.H. Inmon. *Building the Data Warehouse*, Third Edition, 2002.

[16] Bert Scalzo. *Oracle DBA Guide to Data Warehousing and Star Schemas*, 2003.

[17] Shape files technical documentation. http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

[18] http://www.oracle.com/

[19] http://www.sun.com/java/

[20] http://www.oracle.com/technology/tech/pl_sql/ index.html

[21] Nermin Mudzelet. *Road network travel-time estimation using gps data*, 2007.