The Faculty of Engineering and Science

Department of Computer Science

TITLE:
Querying Mobile Objects in Road Networks

PROJECT PERIOD:
DAT6,
17-02-2003 - 06-06-2003

AUTHORS:
Jose A. Ruano
Eva Blazquez

PROJECT SUPERVISOR:
Heidi Gregersen

NUMBER OF COPIES: 7

TOTAL PAGE NUMBERS: 43

In this report, we are going more deeply into the problem of the nearest neighbor. We deal with the problem of a object, moving or static in a road network, that wants to find its nearest neighbors, which are also static or mobile objects. The mobile object will get the route that allows them meet in the faster way, according to the direction and the others restrictions of the road. We include a partial solution to the reverse nearest neighbor and a first approximation to the traffic jam problem. Finally, we test the k-nearest neighbors algorithm and we present the results.

Jose A. Ruano

Eva Blazquez

Contents

Chapter 1

Introduction

In the context of mobile objects, a number of solutions have appeared to resolve the location's problem. This problem has several important points that have to be solved: data modelling, architecture, improvement of algorithms, etc. It is a context in expansion, where the technology improves quickly, which allows more sophisticated software, with more functionality on it. It is also a fact that we have a real need of knowing the location of cars and people, and we demand more and more facilities for our daily live.

The importance of the mobile object's location is an interesting business area. There are lots of services that an enterprise could offer. Services as games by sms, the nearest places of interest from my current position, the nearest taxis or solutions for the traffic jam problem. These services should not be too expensive, since people are interesting in cheap and useful services. The costs should be as low as possible, to get a high number of users.

A special work area is mobile objects in road networks. There are a large number of issues that must be solved. In [1] and [2], the authors propose a data model, system architecture, and some algorithms for resolving the problem of k-nearest neighbor. This problem can be described as follows: an object wants to find the k nearest neighbors in a road network. For example, a pedestrian is in a street and he needs an ambulance, he is interested in the nearest one. Another typical problem is to know the reverse nearest neighbor, in this case the query object wants to know the objects that have it as nearest neighbor [4]. Following with the example, an ambulance driver is interested in the patient that has his ambulance as the nearest one.

But it is important to define what the nearest neighbor is. In [1] or [2], the nearest neighbor is the closest point in spatial distance to the query object. But for us, it is more important to find the nearest neighbor considering the time: we think that it is better to find the data objects closer to the query object in time than in spatial distance. We find more important the search of the objects closer in time, as we want the data objects that we can reach faster, no matter if it is reached in a longer way. For example, if you are calling an ambulance, you probably want it fast. Let us suppose that there are several ambulances that can go to the requested place by different ways. We need the one that can get there faster; there might be one nearest to the query object, but in a road with heavy traffic, so we will choose the ambulance that has less travel time to get to the point where it is needed. What we try to do in this paper is finding the fastest routes for this case and others. We deal with the case of static and moving query objects, looking for static and mobile objects. Obviously, the case of a static query object looking for a static object is not possible, they will never reach each other.

In this paper, we try to join the best ideas in [1] and [2] and incorporate new ideas to improve these data models. We try to find the nearest neighbor in time, not in spatial distance, when both query object and the data objects are mobile objects. We center the problem in mobile objects that wants to meet in the fastest way. We want the points to meet fast, so the routes of the objects could be through fast road with low traffic levels, better than others shorter but with less speed allowed.

Another typical problem in this context is the uncertainty management. To know the exact position of an object would be too expensive. Indeed, continuous updates in the database are not possible because it will slow down the system. Then, a mechanism to management this uncertainty is needed. This mechanism will try to reduce the uncertainty in the object position without high cost.

The problem of the traditional database management systems with mobile objects is that they are not well equipped to handle continues changing data: the great number of updates makes it an impossible solution for the problem of knowing the location of moving objects. Furthermore, the traditional query languages such SQL are inadequate for expressing queries about location.

In the next chapter we will present same related works in this area. Then, in chapter 3, we present our problem statement, and then our the data model (chapter 4). We propose the system architecture in chapter 5 and algorithms in chapter 6. In chapter 7 we make a proposal for the problem of the reverse nearest neighbor. In chapter 8 we describe a first approximation to solve the traffic jam problem. In chapter 9 we describe the implementation and the experiments that we have made, and then in chapter 10 we make conclusions and propose future work. Finally, in the appendices, we describe how to install the simulation and the data model of the database.

Chapter 2

Related Work

There are a lot of work in the area of mobile objects and their location in a database. It is not a trivial problem because there are several issues that must be solved. We are going to discuss about some the approaches to the problem of the K-NN, and how it is solved. First of all, we will discuss about the data models, and following about the algorithms.

2.1 Data Models

2.1.1 MOST

The MOST (Moving Objects Spatio-Temporal) data model introduces dynamic attributes [5, 6]. A dynamic attribute is an attribute which value changes with the time, without continuous updates. It has 3 sub-attributes:

1. updatevalue: It is the value of the dynamic attribute at time updatetime.
2. updatetime: Time when the updatevalue was updated.
3. updatefunction: It is a function of time, where F(0) = 0.

The value of the dynamic attribute A is A = A.updatevalue + A.updatefuntion(t_o) where current_time = updatetime + t_o. With these three sub-attributes we can know the location of the mobile object at every moment.

An explicit update may change the value of a dynamic attribute, but it is not as often as a continuos update.

Furthermore, the database implicitly represents future states of the system. You can make queries pertaining to the future. You can estimate where a mobile object is expected to be in the near future.

This model manages the uncertainty that there is on the position of the object. It works with two concepts: deviation and uncertainty. The deviation is the difference between the real position of the object and the database value in time t. MOST models the uncertainty with bounds. If someone asks to the DBMS where is the X mobile object, it will answer with a pair (x,y) coordinate and the bound. When the deviation reaches the bound, the object updates its position.

We find the following advantages:

1. The updatefuntion of the dynamic attribute can model the correct movement of the object. If it is a good approximation, then will have a high precision.

2. The model provide a good uncertainty management. It guarantees the object's location into a bound.
3. You can predict the future position of a mobile object.

But this model has the following drawbacks:

1. The modelling of the update function. Sometimes it is very difficult to know the object's future movements. This data model takes advantage of this function to know the future position of the mobiles, but if the object's future movements are unknown, the update function could be impossible to obtain.
2. If you need to assure high accuracy in the object's position, this model is not valid because it would need to update the position too many times.

2.1.2 AVL

Another solution is AVL (Automatic Vehicle Location), [7]. This data model works with pairs (p_o, t_o). Then, at time t_o the moving object is at location p_o (this position could be a coordinate pair (x,y) or a cell id).

There are several forms to get the object's location:

1. GPS.
2. By transmission towers that get the position by triangulation.
3. Fixed sensors in the environment.
4. Cell-id. Each object is now in a cell.

The point is stored in the database and you can retrieve it using SQL.

The model presents the following advantage: it is a simple model. If you do not need high precision in the position it can be your best choice.

But, we found three drawbacks:

1. The interporlation or extrapolation of the object location is not possible. If you need the position of the moving object at time x, you cannot know unless the object gives the position at this time.
2. It needs too many resources if you want precision. If you need the position of the moving object every second, you must obtain and store it each second and it could be too expensive.
3. This method leads to cumbersome and inefficient software. The reason is that the existing database management systems are not well equipped to handle continuously changing data. Furthermore, SQL is not designed and optimized for queries that manage space and time information [7].

2.1.3 Trajectory Location Management

In [7], the author discuss the Trajectory Location Management. This method first obtains the source and destination of the mobile object. Then, by using an electronic map, it finds the best route for going from the source to the destination.

An electronic map is a relation where each tuple represents a city block. There are some attributes that are used in the translation between an (x,y) coordinate and an address. Some examples of them are 'L_f_add' (Left side from street number), 'L_t_add' (Left side to street number) or Name (street name). Then, we can work with

routes giving the start address, the start time and the final address. There is an external routine available in most of Geographic Information Systems that gives us the shortest path.

The certain-trajectory (or c-trajectory) gives us the best path in the following format:

(x_1,y_1,t_1) , (x_2,y_2,t_2), ..., (x_i,y_i,t_i) ,....., (x_f,y_f,t_f).

At time t_i the object's position is (x_i,y_i). Now, we can know the position of the object in all time: we can interpolate the object's position between (x_i,y_i,t_i) and $(x_{i+1},y_{i+1},t_{i+1})$ due to its constant speed in the straight line.

Furthermore, the object itself can update its position in the database if there is a deviation between its real position (GPS can give it) and the expected localization.

We can identify the trajectories that are affected by traffic incident or we can made predictions with historical information about traffic or weather, too.

And finally, there is a set of new operators that can be incorporated into SQL language. With this, we obtain a more efficient software.

It presents two advantages:

1. We can obtain the best route supported in the routines of the GIS.
2. It presents a good handling of the uncertainty.

The problem is that if the trajectory is known but it is not a straight line, you cannot take advantage of this knowledge.

2.1.4 Road Network Data Models

We have based our data model especially in two representations, both of them based in two representations for the road networks.

First, in [1], the authors try to solve the problem about location of mobile objects in road networks.

They define the active ranked k-NN problem and the active ranked k-NN environment ([1], pag.3). The problem is defined like a three tuple: kNNP = (QE,Q,QR) where

1. QE is an active k-NN query environment
2. Q is an active ranked k-NN query
3. QR is an active ranked k-NN query result

The active k-NN query environment is another three-tuple QE = (rn,DP,$dist_{rn}$):

1. rn is a road network. Furthermore, they associate a road network locations ($Loc_{rn} \subset \Re^2$).
2. DP is a set of moving data points. This data points are divided in different groups and a data point can be in more of one set.
3. $dist_{rn}$ is a distance function : Loc_{rn} x $Loc_{rn} \Rightarrow \Re_+$, that satisfies these requirements
   - $\forall l \in Loc_{rn}$ $dist_{rn}(l,l) = 0$.
   - $\forall (l_i,l_j) \in Loc_{rn}$ x Loc_{rn}, $dist_{rn}(l_i,l_j) = dist_{rn}(l_j,l_i)$.
   - $\forall (l_i,l_j,l_k) \in Loc_{rn}$ x Loc_{rn} x Loc_{rn}, $dist_{rn}(l_i,l_k) \leq dist_{rn}(l_i,l_j) + dist_{rn}(l_j,l_k)$.

Then, the environment of the problem is just a road network, a set of moving data points and a distance function which is just a metric. This metric is used for measuring the distance between two points.

The active ranked k-NN query is a four tuple Q = (qp,k,m,prop). The qp is the query point that issues the query. k ∈N is the number of nearest neighbors that you want to obtain. m ∈ {1,...,m} is a NN category. You must remember that the data points pertain to different categories. Finally, prop: T x DP ⇒ {TRUE,FALSE} is a data point property predicate. It describes additional properties of objects of interest, i.e., you can need the 3 nearest taxi but they must be free. Properties are modeled with this function.

Finally, they define the active ranked k-NN query result like a two-tuple QR = (RL,t_a). t_a ∈ T is the time point when the query result is available. The RL is a sorted list that contents the k nearest neighbors ordered by proximity.

The data model that the authors propose ([1], chapter 3) is a model based in two representations. The reason of this decision is that they need a 2D representation because they use it to work with the interaction with the user. The graph representation is better for working with algorithms.

They realize a transformation between both representations ([1], page 9). There is an analogy between the set of vertices V_{2drn} and the set of lines L_{2drn} in the 2D representation, and the set of nodes N_{grn}^{trans} and the set of edges E_{grn} in the graph representation.

The transformation is realized with 3 functions:

The vertex-to-node transformation is a correspondence function VtoN: $IT_{2drn} \Rightarrow N_{grn}^{trans}$, where IT_{2drn} is a set of all intersections and terminal vertices from road network in 2D representation.

The path-to-edge transformation is a correspondence function PtoE: $Paths_{2drn} \Rightarrow \{Edge(n_1,n_2,id) \mid n_1,n_2 \in N_{grn}^{trans}\}$, where $Paths_{2drn}$ is a set of lines where the first line has a terminal start vertex (n_1), the last line has a terminal vertex (n_2), and the rest of the vertices are not terminal vertices. **id** is the identifier of the edge.

The cycle-to-edges transformation is a correspondence function which transforms the cycles (set of lines which have the same intersection vertex in the start vertex of the first line and the end vertex of the last line; the others vertex are all intermediate vertices) to a two edges Edge(n_1,n_2,id1) and Edge(n_2,n_1,id2) where $n_1 \in N_{grn}^{trans}$ and $n_2 \in N_{grn}^{add}$.

It presents the following advantages:

1. The data model works with 2 representations. This allows you the advantages of the two representations. The graph representation is better to work with algorithms and the 2D representation to communicate with the user.
2. Data points have been modelled like moving points. This makes the model very thorough.
3. The operations to move from 2D to graph representation are very simple.

But it presents two big drawbacks:

1. They do not consider the special characteristics of the road network. For example, there is nothing about traffic jam or directions. We find this information very important, as it affects the time that the mobile object spent in going from one point to another.
2. They do not consider the direction of the moving objects. We have also taken this into account, we think that two objects in the same place but going in different direction are not equals in distance. The reason is that in one direction the road network could have a traffic jam but not in the other direction. Furthermore, a mobile object could be in the same street that the query object but going in the opposite direction. It could be possible that another mobile object can reach the query object faster the query point.

The second data model is defined in [2]. They treat with a problem a little bit different: A *mobile* query point is looking for *static* data points, on the contrary, we look for mobile (and static) objects, but they set a good basis. In [2], they introduce the data model and in [3] they deal with the algorithms and the system architecture.

Like the previous one, this data model has two-dimensional representation and graph representation. Graphs are efficient for path calculations on road networks and are easy to get and handle. On the other hand, the 2D representation is closer to the real world, and is more useful for the communication with the users.

For the 2D representation we have:

1. Road Network. A two-tuple RN^{2d} = *(S,C)*, where S is a set of segments and C is a set of connections. Segments are represented by start and end point, the movement direction on the segment and the properties of the segment that influence the movement. Connections are represented by the location and a matrix with the kind of movements that are allowed.
2. Data point. Is the object of interest for a query. The data point is defined by its properties and its location, including this the segment, the point on the segment and the side of the road.
3. Query point. It is identified by its location, consisting in a segment and a point on the segment.

And on the graph representation:

1. Road Network. A two-tuple, with a graph and a relation between edges.
2. Data point. It is represent by its location and the travel and road distance to the vertex
3. Query point. It is defined like a data point, but including the speed and the time of the last changes.

As we can see, the 2D representation is focused in storing the real position and the management of the communication with users; and the graph representation has the information for the algorithms and to processing the query.

It presents the following advantages:

1. As the previous model, we use 2 representations and this leaves us the advantages of both. Moreover, the conversions between representations are easy.
2. Roads are well represented, they include details about allowed turns, directions and changes of direction.

But it presents a big drawback:

1. It does not support mobile data points, it is, we can look for static things, as a pharmacy, but not for a mobile point, like a free taxi, a police car, etc.

2.2 Algorithms

There are two papers particularly interesting for us. Their data models are described in 2.1.4. In the [1], we have observed that their algorithms do not work with directions of the mobile objects. In the other one, [3], they don not consider mobile objects, they look static points. As we said before, the problem of the direction is important in these works. It is not just important that a object is close in distance, but one could also consider the closest in time. The reason is that the object could be difficult to reach by traffic jams or because the street is closed by works.

2.2.1 Active Ranked K-NN Queries for moving query and data point in road networks

The authors ([1]) use two algorithms to find the k-nearest neighbor.

The first one runs in the server. It searches in the database a reduce number of nearest neighbor candidates. In this way, the client does not have to handle a big amount of data, just a very reduced part of it.

The second one runs in the client. It maintains the *active result*: It recomputes the distance between the NN Candidates and the query point, sort them, and refresh the list to avoid imprecisions.

2.2.2 Computational Data Modeling for Queries in Road Networks

In [3], they divide the operations between the server and the client. First they make a *quick selection* in the server. They set a grid on the two-dimensional representation, and, on incremental steps, they select nearest neighbor candidates from query windows. Then, that candidates are validated using the graph representation.

Their kNN search algorithms use best-first search, they use the graph representation on this algorithm, and with the help of the tree they order the results obtained in the last step, the validation. They look for the nearest neighbors considering travel distance. They have different algorithms for the case of a road where mobile objects can turn around or not. They also have others algorithms to maintain those results while objects are moving.

Chapter 3

Problem Setting

Consider a road network. There are mobile objects on it and their movement and location is constrained by the road. Queries can be made from a mobile object or from a static object, and relating mobile objects or static objects. Going back to our example, we deal with the case of an ambulance looking for the nearest hospitals (mobile query object looking for a static object), a doctor in a car that has to meet with an ambulance (a mobile query object looking for mobile objects) and a pedestrian has called for an ambulance (the pedestrian is a static object looking for mobile objects, because he wants the mobile objects in the place where he is). The objects of interest and the query objects are considered in the same way and any of them can be either the mobile object or the query object.

A mobile object on the road network is defined by its location, its properties, its speed and its direction. It is defined with detail in chapter 4.

With respect to the representation, we use a two dimensional representation for the location and communication with the mobile objects and a equivalent graph representation for the computation. Our representation is based in the representation set in [1, 2]. It is described in chapter 4.

We can finally resume the problem in:

1. *RN* is a road network. Locations are associated to road networks ($\mathrm{Loc}_{rn} \subset R^2$). The mobile object's locations are points in Loc_{rn}.
2. *MO* is a set of mobile objects. The mobile objects are divided in different groups where a mobile object can be in more than one set. With this, we divide mobile objects into categories according to their characteristics, because we will need to find objects with some concrete properties.
3. *wf* is a weight function : ($\mathrm{Loc}_{rn} \times Loc_{rn} \Rightarrow \Re_+$) We use this function to measure the travel time between two mobile objects. As we discussed before, we are not looking for objects closer in distance but in time. Because of this, our weight function must satisfy:

   - $\forall l \in Loc_{rn}$ wf(l,l) = 0.

   But unlike the usual distances, it does not always satisfy:

   - $\forall(l_i,l_j) \in Loc_{rn} \times Loc_{rn}$, $\mathrm{wf}(l_i,l_j) = \mathrm{wf}(l_j,l_i)$.
   - $\forall(l_i,l_j,l_k) \in Loc_{rn} \times Loc_{rn} \times Loc_{rn}$, $\mathrm{wf}(l_i,l_k) \leq wf(l_i,l_j) + \mathrm{wf}(l_j,l_k)$.

The weight is not the same for going in one or another direction. It is easy to see if we think in a road with one lane cut. It only affects one direction, so that direction affected is slower than the other.

And concerning the second, we can see it in the next example. As we have pondered the weight of the road better than the Euclidean distance, we would often prefer taking a way longer but more faster.

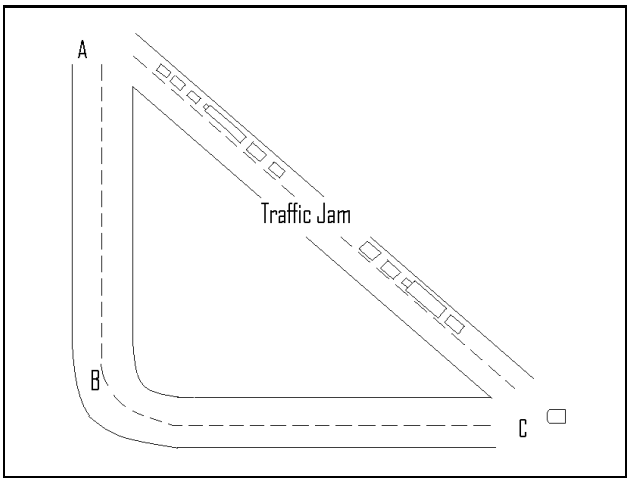

Figure 3.1: Road Network with traffic jam

Example 3.0.1 The ambulance with a serious patient in position C has to go to hospital in position A. There is a 1 km street that goes directly from C to A, but there is a traffic jam in this street. If the ambulance goes through another longer route, it passes through two streets (3 kms: C-B-A), without traffic jam. As probably the second choice is better, that route's weight is less than the first one's.

When making a query, the mobile object (from now, query object) looks for another mobile objects (from now, data objects). These mobile objects pertain to one or more categories. Remember that the mobile object pertain to different categories. Furthermore, these mobile objects must satisfy another requiriments. We define a mobile object predicate that describes additional properties of objects of interest, i.e., you could need the 3 nearest taxi but they also have to be free:

$$\text{prop: T x DP} \Rightarrow \{\text{TRUE,FALSE}\}$$

The query result is a list of data objects, which contains the nearest neighbors requested, it is, the data objects closer in time that also satisfy a series of conditions. The data objects must pertain to the category that the query object requires and must fulfill the conditions that the query object proposed. For example, if the query object needs the 3 nearest free ambulances, then this list contains only free ambulances although there could be others closer.

Chapter 4

Data Model

In this chapter, we join and improve two data models. Our model is based on the models presented in [1, 2]. First, we take the data model proposed in [1], which is a model based on two representations (two-dimensional and graph representation, described in 2.1.4). The reason for this is that a 2D representation is better suited for the interaction with the user. On the other hand, the graph representation is better for working with algorithms. And second, in [2], the authors also define two representations with the same reasons. The main differences between the two approaches are: in [1], the query objective are mobile objects but the definition of the road network is not as deep as we would like: they do not consider the road's direction, the average speed of the road and the problems that it could have; in [2] there is a good defined road network, and we have based ours especially on it, but we have added details about the properties and the weight of the edges. In our case, the edges are defined with the travel weight from both vertices, because we consider that it is the important fact: This allows us to know how much time it will take to each edge every moment.

4.1 Two-Dimensional Representation

The road network in 2D representation is a four-tuple 2drn = (Vx,Ln,Vc,Cd) $\in$ 2DRN where 2DRN is the set of all possible road networks in 2D representation.

- Vx is a finite set of vertices.
- Vc is a vertex coordinate function that relates vertices with R^2.
- Ln is a set of lines.
- Cd, a set of 2D road network coordinates, where coordinates are points pertaining to a line in the road network:

$$\text{Cd} = \{\text{coor} \mid \text{coor} \in \text{R}^2 \wedge \exists\ \text{Line}(\text{v}_i,\text{v}_j) \in \text{Ln}\ (\text{coor} \in \text{Line}(\text{v}_i,\text{v}_j))\}$$

The line from start vertex v_i to end vertex v_j is denoted by **Line(v_i,v_j,direction,properties)**. We distinguish between start vertex and end vertex for pinpointing the mobile object's direction. In a line pertaining to the road network, the movement of the object is constrained by the direction's properties, according to the real world situation.

There are four possibilities in direction:

- -1 - The mobile object can only go from the end vertex to the start vertex.

- 0 - Both directions are allowed in the line but it is not allowed to change your direction.
- 1 - The mobile object can only go from the start vertex to the end vertex.
- 2 - Both directions are allowed and also it is allowed to change the direction.

The properties [**properties = {prop**$_{i=1}^{n}$ **|prop(name,direction,value)}**] are a set that defines the line's state. With these properties, we try to model the cost of going through a road. As any problem on the road possibly does not affect both directions, with the values {both,se,es} for the direction, we model in which way the road is affected by the property. The integer value models how much is the road affected by the property described.

There are seven fixed properties:

1. prop(length,both,X) with X $\in$ [0.∞].
2. prop(max_velocity,se,X) with X $\in$ [0,∞].
3. prop(max_velocity,es,X) with X $\in$ [0,∞].
4. prop(traffic_jam,se,X) with X $\in$ [0,∞].
5. prop(traffic_jam,es,X) with X $\in$ [0,∞].
6. prop(traffic_signals,se,X) with X $\in$ [0,∞].
7. prop(traffic_signals,es,X) with X $\in$ [0,∞].

We define the possible traffic jam, traffic signals and the max velocity in each direction because they could be different. These properties are set to measure the real cost of going through a street, where the velocity, the numbers of signals and of course the length affects the time that a mobile object takes in going through it. The traffic signals property is easy to get and measures the cost of crossing the line due to the traffic signals: number of stops, traffic lights, etc. Another properties, in order to improve the model, can be added, such as quality of the pavement or road works.

We classify vertex in three groups:

- **terminal vertex** Vx^{term}, that corresponds to a location $\in$ Cd where the curve is finished or when the next line has a different value for direction.
- **intersection vertex** Vx^{inter}, that corresponds to nodes where a mobile object can continue by more than one line.
- **intermediate vertex** Vx^{interm}, vertices that are not a Vx^{term} or a Vx^{inter}.

The mobile objects have the following characteristics in 2D representation

- The mobile object reference time function (2DTime:MO $\Rightarrow$ T), that provides a time stamp for each data point.
- The mobile object coordinate function (2DCoordinate:MO $\Rightarrow$ Cd), which gives the object's position in the line.
- The mobile object speed function (2DVel:MO $\Rightarrow$ R), which gives the object's speed.

4.2 Graph Representation

The road network in graph representation is a two-tuple grn = (Nd,Eg) ∈ GRN where GRN is the set of all possible road networks in the graph representation. The elements of a road network are defined as follows:

1. Nd is a finite set of nodes.
2. Ed is a finite set of directed edges. The edge from a start node n_i to an end node n_j is denoted by Edge(n_i,n_j,weight$_{SE}$, weight$_{ES}$,turn) where weight$_{SE}$ and weight$_{ES}$ are the weights going for source to end and end to source, respectively. Its value is infinite if it is not possible going in that direction. The weight function is described below. The flag **turn** indicates if the vehicle can turn around in this section of the road or not.

The edge direction is the same than the line direction in the previous representation. The edge weight in each direction is obtained from the properties in the 2D representation, dividing all properties but max velocity concerning that direction between max velocity:

$$Weight_{SE}(edge) = \sum_{l \in p} \frac{\sum x | prop(a,b,x) \in properties \wedge b = \{se, both\}}{y | prop(max\_velocity, b, y) \in properties \wedge b = \{se, both\}}$$

$$Weight_{ES}(edge) = \sum_{l \in p} \frac{\sum x | prop(a,b,x) \in properties \wedge b = \{es, both\}}{y | prop(max\_velocity, b, y) \in properties \wedge b = \{es, both\}}$$

As introduced in [1], to avoid loops (edges starting and ending in the same node), we transform cycles with two edges, introducing an additional node, both with the same weight, which is:

$$Weight_{dir}(edge_1) = Weight_{dir}(edge_2) = 0.5 \times \sum_{l \in p} \frac{\sum x | prop(a,b,x) \in properties \wedge b = \{dir, both\}}{y | prop(max\_velocity, b, y) \in properties \wedge b = \{dir, both\}}$$

Example 4.2.1 In the figure 4.1 have the previous example but measuring the traffic jam factor. Then, we calculate the weight with this values. For example,

$$\text{Weight}_{es}(\text{AC}) = \frac{prop(length, both, 1) + prop(traffic\_jam, es, 40) + prop(traffic\_signal, es, 0.2)}{prop(max\_velocity, es, 60)} = \frac{41.2}{60}$$

$$\text{Weight}_{es}(\text{AB}) = \frac{prop(length, both, 1.5) + prop(traffic\_jam, es, 0) + prop(traffic\_signal, es, 0.5)}{prop(max\_velocity, es, 40)} = \frac{1}{20}$$

$$\text{Weight}_{es}(\text{BC}) = \frac{prop(length, both, 1.5) + prop(traffic\_jam, es, 0) + prop(traffic\_signal, es, 0.7)}{prop(max\_velocity, es, 50)} = \frac{2.2}{50}$$

The mobile objects have the following characteristics in the graph representation:

- GTime:DT ⇒ T, which gives the object's time stamp.

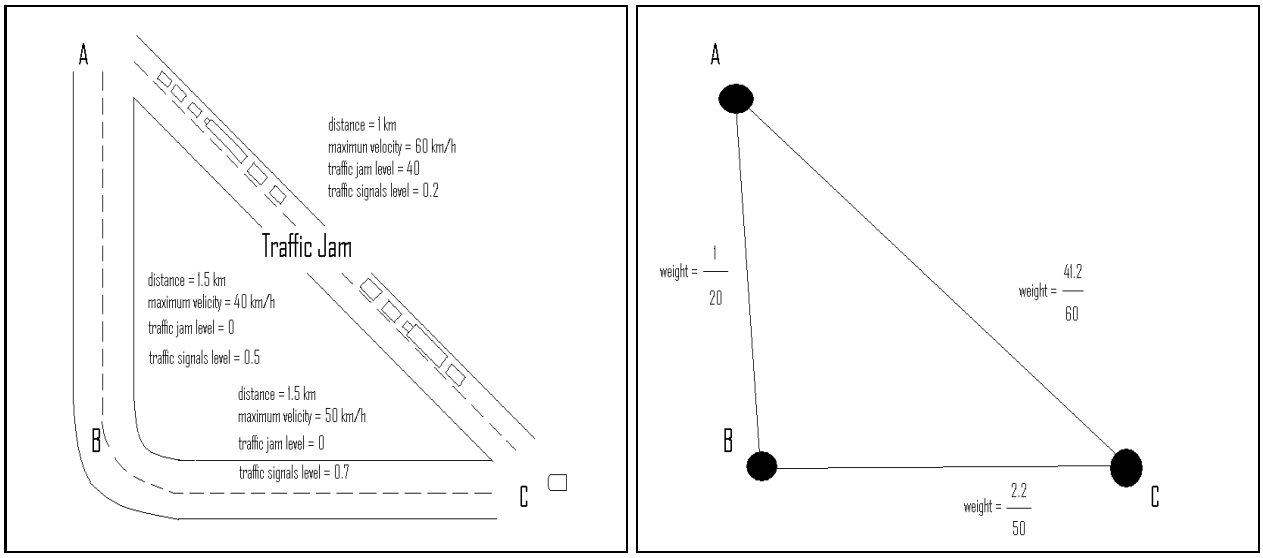

Figure 4.1: Transformation from 2D representation to graph representation.

- GEdge ⇒ Ed, which gives the current edge of the object.
- GPosition$_{SE}$:DT ⇒ X, where X ∈ [0,weight(current edge)], which gives the current position in the edge, in the **SE** direction .
- GPosition$_{ES}$:DT ⇒ X, where X ∈ [0,weight(current edge)], which gives the current position in the edge, in the **ES** direction. This parameter complements the previous one, as the relative position depends on the weight and it is different for each direction.
- GVelocity:DT ⇒ R_+, which gives the speed of the object. It must be transformed from 2D representation.
- GDirection:DT ⇒ {SE, ES}, which gives the direction of the object: Source-End or End-Source. If the velocity in the 2D representation is positive, then the direction will be **SE**, otherwise, the direction will be **ES**.
- GFactors:DT ⇒ {FALSE,TRUE}. For modeling properties that the data point must satisfy.
- GRoute:DT ⇒ {edges}. This function gives a list of edges, which is the route that the objects follow to go to its destination point.

The function **GRoute:DT ⇒ {edges}** is the innovation of this data model. Having the route of all the objects, we could know the traffic jams in the future. Furthermore, we can inform on-line mobile objects of the traffic jam on their routes. We discuss about this problem in chapter 8.

We realize a transformation between both representations similar to [1]. There is an analogy between the set of vertices Vx and the set of lines Ln in the 2D representation, and the set of nodes Nd and the set of edges Eg in the graph representation.

The transformation is realized with 3 functions:

- *The vertex-to-node transformation* is a correspondence function VtoN: {Vx^{term},Vx^{inter}} ⇒ Nd.
- *The path-to-edge transformation* is a correspondence function

$$PtoE : Paths \Rightarrow \{Edge(n_1, n_2, properties) | n_1, n_2 \in Nd,$$

where Paths is a set of lines where the first line has a terminal o intersection start vertex, the last line has a terminal or intersection vertex, and the rest of the vertices are not terminal vertices. Furthermore, the direction of the vertices is equal in all of them. The properties of the edge are the sum of all lines in 2D representation.

- *The cycle-to-edges transformation* is a function that transforms the cycles (set of lines with the same value for direction, with the same intersection or terminal vertex in the first and the last line; the others vertices are all intermediate vertices) to a two edges Edge(n_1,n_2) and Edge(n_2,n_1) where $n_1 \in$ Nd and n_2 is add to Nd now. It is not a cycle when one of the intersection or terminal vertices has a direction different from the others, so in the next figure, there is not a cycle.

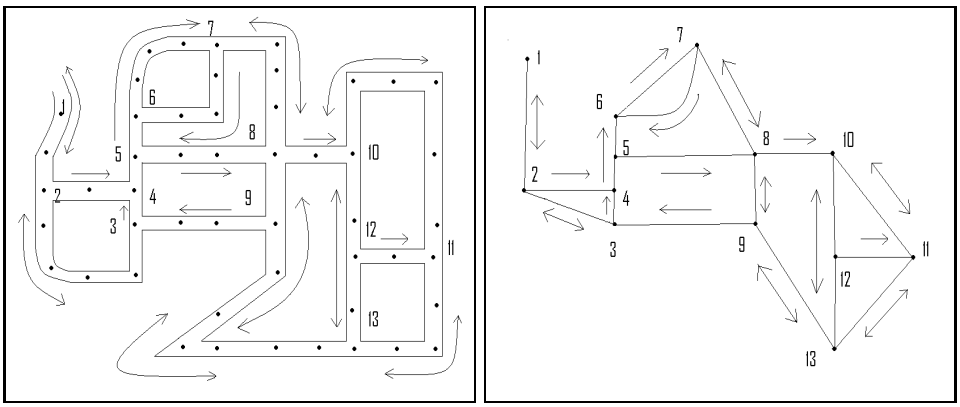

Figure 4.2: Transformation from 2D representation to graph representation.

Example 4.2.2 In the figure 4.2, we can see the transformation from 2D to graph representation. In the 2D representation, the vertices that are labeled are the future nodes of the graph representation.

The edges [11-12], [12-13], [11-13] could form a cycle, but the direction of [11-12] removes this possibility. With both directions in this edge, it would be a cycle and we would have to introduce a new node.

It could be necessary to know the mobile object's position at an arbitrary time, not only at a reference time. This position is calculated as follows:

$$GCurrentPos(t, dp) = GPosition(dp) + GVelocity(dp) \times (t - GTime(dp))$$

This function returns the object's position in its current edge at time t. Finally, we need to transform the data point's position from the 2D representation to the graph representation. We need the GPosition of the point, which is very easy to calculate. We have to obtain fist the values:

$$Length(before) = \sum_{i=0}^{k-1} x_i \, | \, prop(length, both, x_i) \in properties_{linei}$$

where k is the current line and 0 is the first line that we are transforming and 1, (k-1) are the lines between line 0 and line k.

deuc is the Euclidean distance between *vs*, the start vertex in the current line and the current 2D coordinate **Lengh(Path)** is the length of all lines that we are transforming to the edge.

1. If we use a path-to-edge transformation:
$$\text{GPosition} = \frac{Length(before) + deuc(Vc(vs), 2DCoordinate(dp))}{Length(Path)} \times \text{Weight(GEdge(dp))}$$

2. If we use a cycle-to-edge transformation:
$$\text{GPosition} = \frac{2 \times (Length(before) + deuc(Vc(vs), 2DCoordinate(dp))}{Length(Path)} \times \text{Weight(GEdge(dp))}.$$
Furthermore, the current edge will be the first one. Remember that two edges are obtained when there is a cycle.
But if the GPosition is bigger than the weight of the edge:
$$\text{GPosition} = \frac{2 \times (Length(before) + deuc(Vc(vs), 2DCoordinate(dp))}{Length(Path)} \times \text{Weight(GEdge(dp))} - \text{Weight(GEdge(dp))}.$$
and the current edge will be the second one.

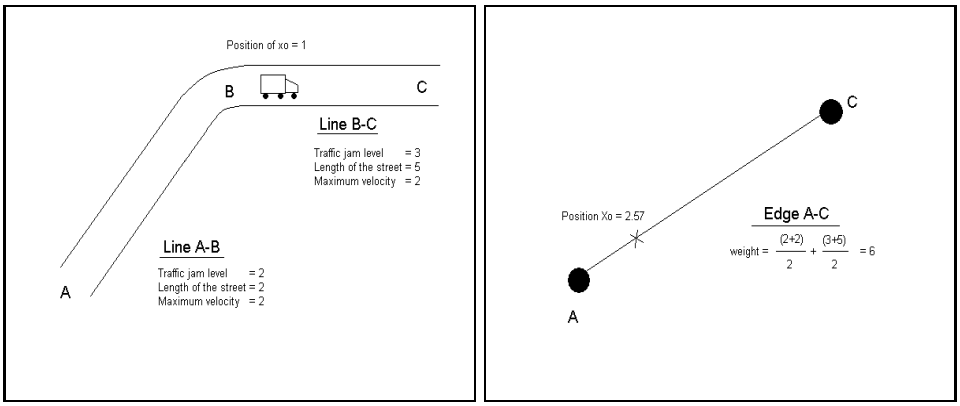

Figure 4.3: Transformation of the position of a mobile object from 2D representation to graph representation.

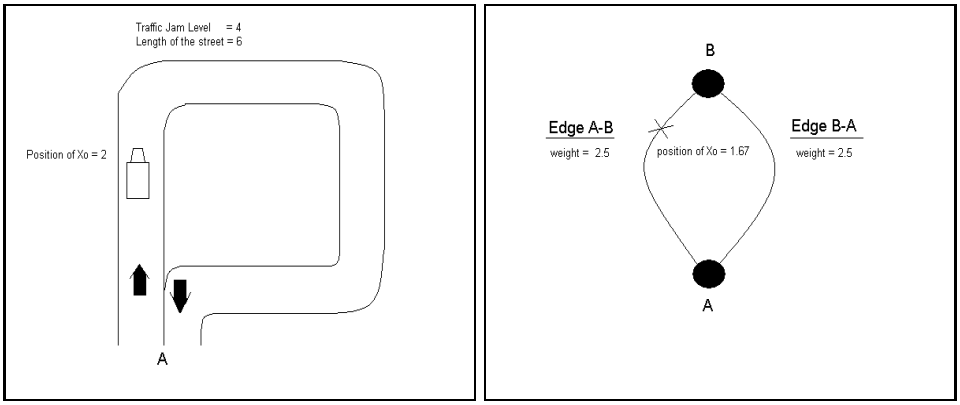

Figure 4.4: Transformation of the position of a mobile object from 2D representation to graph representation.

Example 4.2.3 In this example, we describe the transformation of the mobile object's position in both cases. In the figure 4.3, the mobile object is in a normal line and in figure 4.4, the mobile object is in a cycle.

In the first case, the operation of the transformation is:

$$\text{Position}(x_0) = \frac{2+1}{7} \times 6 = 2.57.$$

It seems a logical result because the position of the mobile object in the 2D representation is in the second street (B-C) , that is longer than the first one and the traffic jam level is higher. If the mobile object was in the end vertex of the second line, the transformation would be:

$$\text{Position}(x_0) = \frac{2+5}{7} \times 6 = 6.$$

The position of x_0 would be the final node in the edge. This result is logical.

And if the mobile object was in the start vertex in the first line, the operation would be:

$$\text{Position}(x_0) = \frac{0+0}{7} \times 6 = 0.$$

In this case, the position of x_0 would be the start node in the edge. These two results demonstrate the coherence of the calculations.

In the second case, the operation of the transformation is:

$$\text{Position}(x_0) = 2 \times \frac{0+2}{6} \times 2.5 = 1.67.$$

Once again, if the mobile object was in the vertex after it cross the line, the operation would be:

$$\text{Position}(x_0) = 2 \times \frac{0+6}{6} \times 2.5 = 5.$$

As the value obtained of the Position(x_0) is higher than the weight of the edge, the current edge will be the second one and the position would be:

$$\text{Position}(x_0) = 2 \times \frac{0+6}{6} \times 2.5 - 2.5 = 2.5.$$

If the mobile object was in the middle of the line, the operation would be:

$$\text{Position}(x_0) = 2 \times \frac{0+3}{6} \times 2.5 = 2.5.$$

And if the mobile object was in the vertex, before it starts the route, the operation would be:

$$\text{Position}(x_0) = 2 \times \frac{0+0}{6} \times 2.5 = 0.$$

Again these three results demonstrate the coherence of the calculations.

This representation has mobile objects categories. A category is a n-tuple $T_{name} = \{A_1,..,A_n\}$, where **name** is the name of the category and $\{A_1,..,A_n\}$ is a set of attributes of the category.

Chapter 5

System Architecture

In this chapter we presents our software architecture. It seems clear that a client-server architecture is required, where clients, which are mobile devices like PDA's, laptops or mobile phones, connect to a server, which makes a part of the processing and manages the database.

Another possibility could be to use a peer-to-peer architecture. In this architecture, all the elements have the same information. They are all clients and servers. This architecture implies new algorithms for looking the nearest neighbors but it presents a big drawback: who and how manages the road network? This drawback makes us choose the client-server option.

The client-server architecture is the one that fits the environment of the NN problem the best. The user works with the client, running in a mobile device, and it communicates with the server.

There are two possibilities within the client-server architecture. The first one is a thick server and a thin client. In this architecture, the client just presents the results to the user and advises the server about its current location. This architecture presents the advantage that devices in client side could be simple, as PDA's or mobile phones. This architecture's drawback is that the server has a lot of work because it must manage the road network's representations, it must calculate the nearest neighbors and it must manage the mobile objects.

The other possibility is a thin server and a thick client. Thus, the client participates in the road network representation and in the k-nearest neighbor algorithm. The server manages the information of all mobile objects and can participate in the first steps of the algorithm. The main advantage is that the server has less amouth of work. But now, devices as mobile phones or PDA's cannot be used because the client device needs a minimum of computational power. Furthermore, the cost of the communication is bigger in this case because more information must be transferred.

We have chosen the first possibility. The main reason is that devices as mobile phones or PDA's can be used. Nowadays, the price of these devices is low and more and more people have a mobile phone or, sometimes, a PDA. With more powerful devices, like laptops, some of the tasks described below can be passed to the client side and thus release the server of some work. Another reason for this choice is that, although laptops could be the most suitable devices for this type of application, pedestrians have also to be considered, and they use small devices instead of a laptop. Furthermore, the problem of our election, an excessive work charge, can be solved with several servers that communicate among them using the database.

In the following sections we will describe the task performed in the server and in the clients.

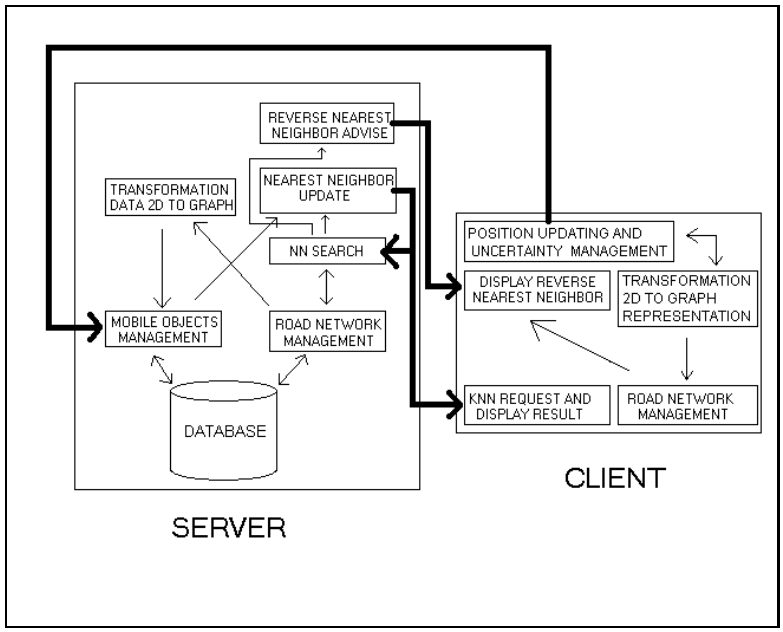

Figure 5.1: System Architecture

5.1 Server Side

The server manages the largest part of the application. It handles the database, and most of the computation. We have considered it this way because of the computational limitations of the devices that can be used as a client.

- **Mobile Objects Management** The server stores all the mobile objects, with its properties, and its routes in both representation. When the client sends its current position, the server updates it. Furthermore, the server must verify that the clients have modified their positions.
- **NN Search** It is the search of the KNN, described in chapter 6.1. The search of the K-Nearest Neighbors is done in the server. The server receives a request from the client. It starts the search and returns the results to the client.
- **Reverse NN Advise** A mobile object warns its nearest neighbor when the next nearest neighbor is too far. It is described in chapter 7. This task could be taken over by the client. We have placed this functionality on the server side because it is faster if it is the server who warms. It also releases simple clients of work. Furthermore, a single communication is necessary, but if this task is performed by the client, it would need two communications: first, the client would warn the server that its second nearest neighbor is too far; and second, the server would warn the reverse nearest neighbor.
- **Road Network Management** With this task, the server manages the two representations. It manages the edges and the nodes, in the graph representation, and the lines and the vertices in the 2D representation. In cooperation with the following task, this task refreshes the values of the properties of the lines. Therefore, the weight of the edges must be refreshed too. Furthermore, when the client sends its position, the part of the road network that it is implied must be sent to the client. The client cannot store all the road network, so the server sends a part of the road network to the client.
- **Transformation Data 2D to Graph** The server transforms the road network in 2D representation to the graph representation.

- **Nearest Neighbor Update** When a mobile object does not update its position for a long time, this task verifies if this mobile object is in another object's neighbor list. In this case, the server will notify the client that one of its nearest neighbor is off-line.

5.2 Client Side

The clients communicates with the server to make queries and to manage the location, especially to update its position. It is running on the devices that are going to make queries and in the devices that can be searched.

- **k-NN Request and Display Results** The client sends a request with the number of nearest neighbors that it needs. When the server sends the results, this task presents the query result to the user. It can be in graphical or text mode, depending on the kind of device we are using. With a powerful device, the result of the query can be displayed as a city map.
- **Position Updating and Uncertainty Management** The mobile objects have a positioning unit. We suppose that the client receives the location data in 2D representation coordinates (the static objects, obviously, do not have this task and their position is stored in both representations too). Then, the client stores its position and sends it to the server in both representations.

  In the uncertainty management, the client verifies when the deviation of the position with respect to the expected position reaches a bound. For the update of the position, we use the disconnection detection dead reckoning policy, which decreases the bound as time passes. Then, the object will update the position sooner or later, except in the case of disconnection. This policy is very useful to discard the objects that are not on-line. It is described in detail in [5], section 5.4.2. This policy presents a drawback: it increases the number of communications. But for us it is more important to know the correct position of the mobiles. Furthermore, if new tasks about traffic jam detection are added, the number of mobile objects on-line must be known.

  It is very useful for this task knowing the route. In [1], the mobile waits in the edge's final node until the threshold is exceeded. In our system, it knows the following edge and the client can obtain the current position in the following edge.
- **Transformation Data 2D to Graph** This task is like the server's one. The client transforms its own data 2D representation to the graph representation, because it needs it for the previous task, the position updating.
- **Display Reverse NN** In this task, the server warns the client that it is another mobile object's nearest neighbor and this mobile has its second nearest neighbor very far. This task is explained in chapter 7.
- **Road Network Management** In this task, the client stores a small part of the road network in both representations. It needs to store it in order to do the uncertainty management and to display the results of the k-NN algorithm.

Chapter 6

Algorithms

In this chapter, we propose some algorithms to resolve the nearest neighbor problem. The algorithm must consider the directions of the mobile objects and, of course, the weight of the street.

6.1 Algorithm for k-Nearest Neighbor Search

In this algorithm we try to find the list of the k-nearest neighbor, when the objects of interest are in movement and are going to meet the query object.

There are two different parts. In the first one, we will search the future meeting point between the query object and the data objects. If the objects we are looking for are static instead of mobile, the meeting point will be the point where the nearest object is.

The algorithm works in different way if it is looking for static objects instead of a mobile object. A static point is different from a mobile data object with speed zero. The difference is that a mobile data object, with speed zero, can change its velocity and go towards the meeting point. However, a static object cannot move towards this meeting point. For example, a taxi can be stopped in a taxi stop and then starts going towards a pedestrian, but a pharmacy cannot move. For us, the search is different and the algorithm presents a slight difference.

In the second part, we cover the search tree for the k-nearest neighbor. It gets to the mobile objects by checking in each node if there is a data objects going towards that node. We get the k-NN and also the route that the data object has to follow to get faster to the meeting point.

```
(1) procedure Find_knn(qo:query_object; meeting_time: TIME; k,max_weight:integer; prop:properties;
(1)                    search_for_mobile_objects: boolean)
(2) begin
(3)     if (searching_for_mobile_objects = true)
(4)         meeting_point, meeting_edge ← Find_MeetingPoint(qo,meeting_time);
(5)         NN ← Graph.Search_MobileObject(meeting_edge, prop);
(6)     else
(7)         meeting_point ← GPostion(qo);
(8)         NN ← Graph.Search_StaticObject(meeting_edge,prop);
(7)     end_if
(9)     Node1 ← meeting_edge.n_i;
(10)    Node2 ← meeting_edge.n_j;
(11)    Node1_cost ← meeting_edge.weigth_se - meeting_point_se(qo);
(12)    Node2_cost ← meeting_edge.weigth_es - meeting_point_es(qo);
```

```
(13)    Tree.insert(meeting_point,0);
(14)    Tree.insert(Node1,Node1_cost);
(15)    Tree.insert(Node2,Node2_cost);
(16)    previous_point ← meeting_point;
(17)    Repeat
(18)        current_node, current_weight ← Tree.low_cost_leaf(previous_point);
(19)        following_nodes, following_costs ← Graph.explore(current_node,searching_for_mobile_objects);
(20)        if (searching_for_mobile_objects = true)
(21)            NN ← Graph.Search_MobileObject(current_node,prop);
(22)        else NN ← Graph.Search_StaticObject(current_node,prop);
(23)        Tree.insertleaf(current_point,following_nodes,following_costs);
(24)        previous_point ← current_node;
(25)    Until ((NN.card() > k and Tree.low_cost_leaf > NN.max_weight) or current_weight > max_weight)
(26) return NN;
(27) end
```

First, in (4), we obtain the meeting point. We will explain in the following section the way to obtain it. Basically, this point is the previous point in which the query object wants to meet with the nearest neighbors. In order to find it, we consider the velocity, the direction and the position of the query object. If the query object is looking for static points, the algorithm is a bit different. In this case, the algorithm begins looking for points in its current position.

Then, we search the mobile objects in the current edge that are going to the meeting point (5). This algorithm is explained in the section 6.3.

Next, we explore the graph with a search tree to find the nearest neighbors, considering the cost. First, we find out which vertex of the edge is the nearest in cost (9-12). Then, we insert in the search tree both nodes. The following current node will be the node with the lower cost in going to the actual current_node to it. We use this tree to find the following current node (18). When a node is inserted in the tree, its weight is the sum of the all weight from the root in the tree, that is the meeting point. Thus, the weight of the node represents the cost of going form the meeting point to that point by the faster way.

The next step is to look in the graph the following nodes from which you can reach that node (19). The point is going from a node to another one by the shortest way. In this function, when we are looking for mobile objects, the function will give us the nodes from which we can go to the current node. If the query object is looking for static points, we'll get with this function the nodes where we can go from the current node. This is just taking the edges with one direction or another, and allows us using the algorithm for both searchs.

When a node is reached, the data objects that go towards it must be added to the NN list (20-22). We use the tree to help us in the search of new nodes.

The algorithm stops when the k-nearest neighbors are found and the associated minimum weight to any leaf in the tree is bigger than the last nearest neighbor in NN. The reason is that sometimes a mobile object can go towards a node in smaller weight than other in the NN list if the associated node has an lower weight than this nearest neighbor. Another condition for finish the algorithm is when a maximum weight from the meeting point is reached.

We do not consider the current speed of the objects for finding the nearest neighbors, as it can change in another street, and it always depends on the route. That is considered when setting the weights for the streets.

Note that with this algorithm, when we are looking for mobile objects, we also get the route that the objects have to follow for going to the meeting point faster. When looking for static points, we will get the route that the query object has to follow for going to the NN.

6.2 Algorithm for calculate the meeting point

With this algorithm we try to find the predicted meeting point between the query object and the k-nearest neighbor. This meeting point is used in the previous algorithm.

```
(1) procedure Find_MeetingPoint(qo:query_object; m_time:TIME)
(2) begin
(3)      meeteing_point ← GPosition(qo) + GVelocity(qo) × [m_time - GTime(qo)];
(4)      edge ← GEdge(qo);
(5)      n_time ← GTime(qo);
(6)     While (meeting_point > edge.weight)
(7)          previous_edge ← edge;
(8)          old_velocity ← GVelocity(qo);
(9)          edge ← GRoute.Following(qo,edge);
(10)         n_time ← (previous_node.weight - GPosition(qo)) / old_velocity + n_time;
(11)         GPosisiton(qo) ← 0;
(12)         meeting_point ← GVelocity(qo) × [m_time - n_time];
(13)     end_while
(14)     return meeting_point, edge;
(15) end
```

First, we calculate the meeting position and the edge where this meeting point will be (3). If the value of this meeting position is higher than the current edge total weight, it means that the meeting point will be in the following edge (9). Now, the meeting point is calculated advancing in this edge. First, the time stamp in the end node in the previous edge must be calculated (10) and then the meeting point can be calculated using this time stamp (12). The process finish when the value of the meeting position is lower than the current edge total weight, as that location is on that edge.

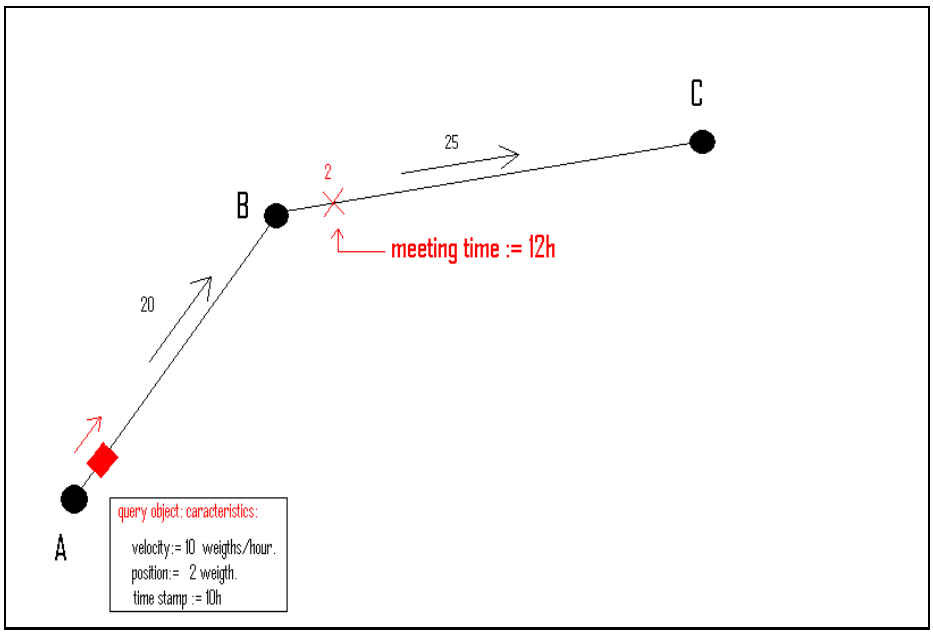

Figure 6.1: An example of the operation of the meeting point algorithm.

Example 6.2.1 We try to explain with an example the way to obtain the meeting point. An ambulance goes to

a city by the highway and it needs some police motorists in order to escort it in two hours. When the ambulance makes the call, it is in the position 2 of the edge [A,B] and it has a velocity of 10 weight units/hour. Then, the Find_MeetingPoint function calculate the future meeting point in order to warn the police motorists.

In the calculus in (3), the result is:

$$\text{meeting\_point} = 2 + 10 \times [12 - 10] = 22.$$

This value is bigger than the edge's weight (6). Then, the following edge is [BC] and we calculate the time in the node B (10):

$$\text{n\_time} = (20 - 2) \text{ weight} / 10 \text{ weight/h} + 10\text{h} = 11.8$$

And finally we calculate the meeting point:

$$\text{n\_time} = 10 \text{ weight/h} \times [12\text{h} - 11.8] = 2 \text{ weight}.$$

The algorithm returns 2 in the edge [BC].

6.3 Algorithm for search the mobile object in a edge

With this algorithm we try to find the mobile objects that go towards a node in an edge. We use this procedure in the algorithm described in 5.1.

```
(1) procedure Search_MobileObject(current_node:node; NN:mobile_objects; prop:properties)
(2) begin
(3)     all_edges ← Graph.followingEdge(node);
(4)     for each X ∈ all_edges
(5)         for each mobile_object ∈ X
(6)             if towards(mobile_object.direction,node) = true AND mobile_object.fulfil(prop)
(7)                 if (NN.busy() = true) and (NN.max_weight > mobile_object.weight)
(8)                     NN.delete_last();
(9)                 end_if
(10)                NN.insert_point(mobile_object);
(11)                NN.sort();
(12)            end_if
(13)        end_for
(14)    end_for
(15) end
```

The way to finding the mobile object is that it must go towards the new current node. Then, if there are k nearest neighbors already found, the mobile object is added if it has a weight lower than the last nearest neighbor.

As the process finishes when the minimum weight in the tree is higher than the weight of the last nearest neighbor, we can assure that the k nearest neighbor are found with this three algorithms.

When the algorithm is looking for static points, the function **towards(mobile_object.direction,node)** in line (6) works in a different way. With mobile data objects, the function checks that the mobile object can go towards this node. With the static data objects, the function checks that the query object can go towards the static point from this node.

6.4 An example of the operation of the knn algorithm.

Now we are going to explain with an example the operation of the algorithm and it will explain the several conditions in our algorithms.

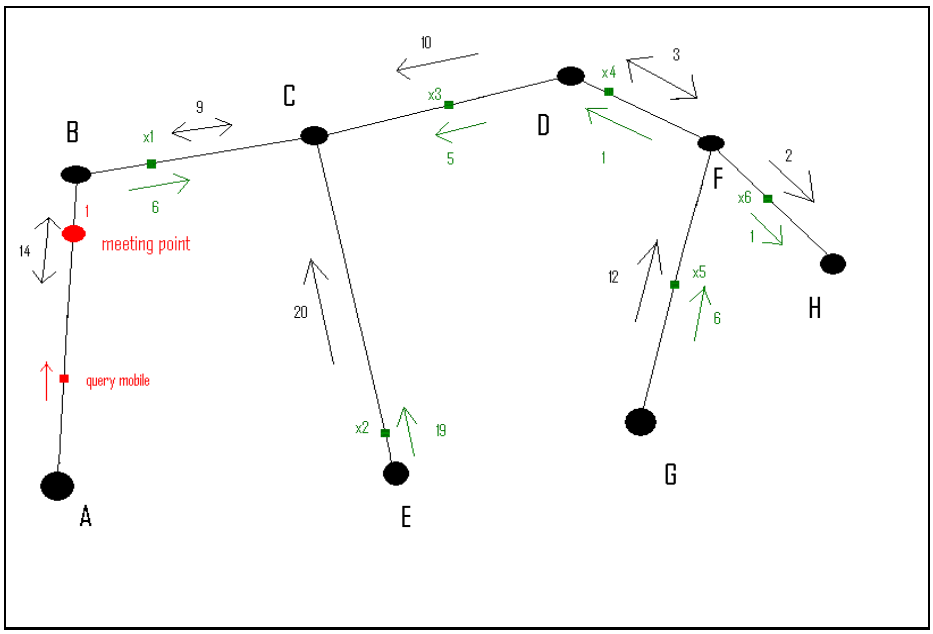

Figure 6.2: An example of the operation of the knn algorithm.

Example 6.4.1 This example is simple but large enough to illustrate the operation of the algorithm. An ambulance is looking for the 3 nearest police motorist. The green colour represents the motorists, its direction and the distance in weight units for reaching the node. The black arrows represent the direction of the edges and the black numbers represent the weight of the edges.

The meeting point is in the edge [A,B]. Then the following node is B and there is no motorist going towards it. Then, the following node is C, and it has a total weight of 10. The police motorists that go towards it are x_1, x_2 and x_3 and their weights are 16, 29 and 15 respectly. There are 3 nearest neighbors but the following node is D, its weight is 20 so the algorithm continues. Then, the police motorist x_4 has a lower weight than x_2, so this one is eliminated from the NN list, x_4 is incorporated and its weight is 21. Following with the example, the next node is F but its weight is 23, the third nearest neighbor, which the last of the list, has a weight of 21 so the algorithm must finish.

Finally, if we could continue with the example, the following node would be G and not H because the direction in this edge is from F to H and x_6 cannot go towards the meeting point by F.

Chapter 7

A partial solution for the reverse nearest neighbor problem

We try to explain with an example a partial solution for the reverse nearest neighbor problem. In this problem, a query object wants to know the mobile objects that have it as their nearest neighbor. In [4], the authors say that this problem is an unexplored one. Furthermore, they say that a straightforward solution for computing reverse nearest neighbor queries is to check for each point whether it has a given query object as its query nearest neighbor. The problem of this solution, as they say in ([4], page 1), is when the number of points is large.

But sometimes the problem must be considered and therefore it can be interesting to solve it.

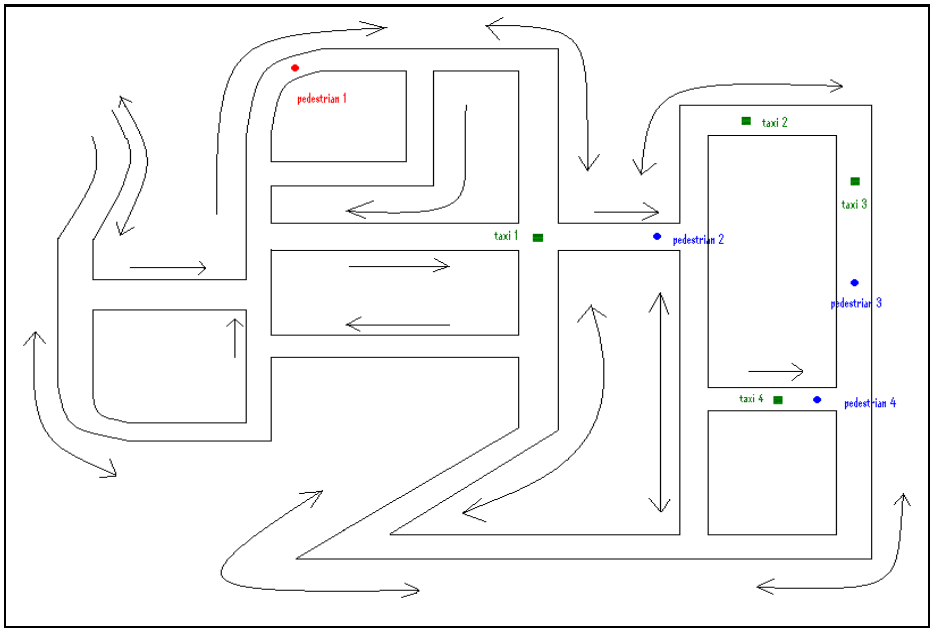

Figure 7.1: A solution for the reverse nearest neighbor problem.

Example 7.0.2 In this case, the pedestrians are looking for a taxi. Taxis are represented by green squares and pedestrians are represented by the blue circles except our pedestrian of interest, who is the red circle.

The pedestrians 1 and 2 have the taxi 1 as their nearest neighbor. But for the pedestrian 2 is not a problem taking taxi 2 and perhaps it is better if taxi 1 goes to get pedestrian 1, and pedestrian 2 takes taxi 2 or taxi 4. Pedestrian 2 would have to wait more time, but just a little bit, and pedestrian 1 would wait less than half the time. In this case, the pedestrian 1 could warn taxi 1 and say to him: " you are my nearest taxi and the second nearest one it is too far from me". The pedestrian 2 does not warn taxi 1 because he has another taxi that is not too far from it.

The problem now is when does the mobile object warns its nearest neighbor and tell him that it is its reverse nearest neighbor. There are at least two solutions:

1. The mobile object warns its nearest neighbor when the difference between its first nearest neighbor and its second nearest neighbor reaches a bound.
2. The mobile object warns its nearest neighbor when the difference between its two nearest neighbors reaches a proportion, as the double or the triple.

The two options have problems. In the first case, the difference does not have the same meaning when the first nearest neighbor is to a weight of 20 and the second one is to 40 that the first nearest neighbor is to 200 and the second one is to 220.

The second option has the problem that it can be possible to have the first nearest neighbor to a weight of 1 and the second one is to a 2. It is the double but it is not significant.

We propose a mixed solution, which combines both of them. A mobile object will warn its nearest neighbor it is its reverse nearest neighbor when:

$$\frac{weight_d(2-NN)}{weight_d(1-NN)} > ratio$$

and

$$weight_d(2-NN) - weight_d(1-NN) > bound$$

With these conditions, we assure a proportional difference and we eliminate the problem in the short distances.

Chapter 8

First approximation to solve the traffic jam problem

In this chapter we are going to describe an algorithm for trying to reduce traffic jam problems. This problem is very important problem in some big cities. Sometimes, people spend a lot of time in traffic jams and it reduces the quality of life in the cities. But sometimes the traffic jam problem does not have solution. In effect, sometimes a lot of mobile objects want to go towards the same place and they cannot be turned aside. In this case, the problem is a capacity problem: there are too many cars for the capacity of the highway.

The solution that we propose does not try to be a final solution because we think that this problem is not trivial and it requires more time of investigation. Furthermore, we consider that a fine solution for this problem requires lot of resources in machines, with big servers.

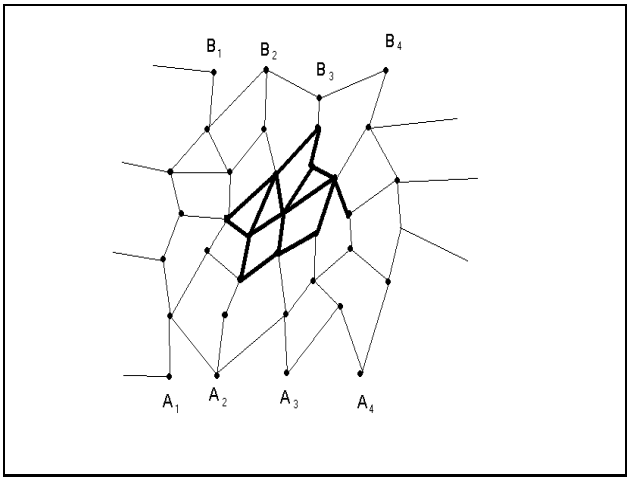

Figure 8.1: A traffic jam in the road network

The way that we have considered the data model and the algorithm to solve the nearest neighbor problem can help us in the task of solving the traffic jam problem. Indeed, we take advantage of our algorithms and data model and they are used in our solution. The reason is in the **GRoute:DT** $\Rightarrow$ **{edges}** function. We use it in this solution.

The first part of our algorithm is to detect the zone where a traffic jam will appear. This part of the algorithm is expensive in computation time. The idea is simple. In the data model, the servers could obtain the route

and the velocity of the objects and speed limits of the streets. With this information we can obtain the future position of the object, for example, 10 minutes later. The **Find_MeetingPoint()** function can be used for this task. Furthermore, the experience can be taken advantage of. In effect, traffic jams appear frequently in the same zones and in the same hours.

In the figure 8.1, the heavy lines indicate where the traffic jam is. Of course, the traffic jam could affect several edges (remember that an edge could represent several streets from 2D representation).

The following step should be to turn aside some cars of the zone when there is the traffic jam. The best way to describe the idea is with the figure 8.1. The algorithm fixes some points outside the zone where the traffic jam is. Then, these points will be meeting points (points A) in the algorithm that looks for the nearest neighbors, and the others will be mobile objects that want to reach these meeting points (points B). What is the reason? Because our algorithm for looking for the nearest neighbor finds the less heavy routes. Although there is just a mobile object in the road network that goes towards the meeting point, the algorithm always takes the route that goes towards the meeting point in the most efficient way. Then, the mobile objects never cross the traffic jam zone because it would have a big cost. The mobile objects would go around this zone.

Finally, the algorithm notifies the route to the mobile objects, that entry from the points A and exit by the points B. The point is that the mobile objects that need to cross the zone where there is a traffic jam, could be turned aside. The mobile objects that have as final destination the zone with the traffic jam should not be turned aside. The problem without solution will appear when there is a point where lots of mobile objects want to go. None of the mobile objects should be turned aside.

Sometimes, the zone of the road network where the algorithm is working must be refreshed. This is because the weight of the edges can change. Our algorithm for finding the nearest neighbor will work with a meeting point and a mobile object (A_i,B_i). When this algorithm finds a route, a number of mobile objects could be turn aside for this route. Then the road network in this zone must be refreshed and the algorithm is repeated to find the nearest neighbor once again.

The algorithm to find the nearest neighbor must be modified and the nearest neighbor must include its route. This modification is minimum: when a nearest neighbor is reached, the algorithm only must examine the tree in reverse order by the father of the nodes until to reach the meeting point. Furthermore, now the meeting point should not be calculated and it must be introduced.

The algorithm to resolve the traffic jam problem finishes when the edges in the zone have a similar weight.

```
(1) procedure Solve_traffic_jam(future_time:Time)
(2) begin
(3)     traffic_jam_detected ← false;
(4)     while (traffic_jam_detected = false or current_time > future_time)
(5)         qo ← DATABASE;
(6)         meeting_point, meeting_edge ← Find_MeetingPoint(qo,future_time);
(7)         FUTURE_DATABASE ← meeting_edge;
(8)         traffic_jam_detected ← Find_traffic_jams();
(9)     end_while
(10)    while (traffic_jam = true)
(11)        set_pointsA ← Fix_PointsA(FUTURE_DATABASE);
(12)        set_pointsB ← Fix_PointsB(FUTURE_DATABASE);
(13)        for (i=1;i<=Num_Elements(set_pointsA);i++)
(14)            nearest_neighbor ← Find_knn(set_pointA(i),set_pointB(i),1,∞,null,true);
(15)            FUTURE_DATEBASE ← Turn_aside_mobiles(FUTURE_DATABASE,nearest_neighbor);
(16)        end_for
(17)        traffic_jam_detected ← Find_traffic_jams();
(18)    end_if
(19) end
```

In the steps 3-9, the algorithm tries to detect future traffic jams. In this part, several computers should work in the detection because there could be a lot of mobile objects in the system. Furthermore, if the experience says that in certain hours and places there use to be traffic jams, the servers could look for mobile objects that are near these zones. The function **Find_traffic_jams()** is not very difficult. The point is fixing the relation between the length and the traffic jam value of the street, information that the system can obtain from the lines in the 2D representation.

The functions **Fix_PointsA** (11) and **Fix_PointsB** (12) could look for the points in depending on the number of edges that there are in the traffic jam. It should be an empirical work to find the best relation between distance from these points to the traffic jam zone and the number of edges in the traffic jam.

In (14), the algorithm looks for the best route to go from the meeting point A_i to the fictitious mobile object B_i. Afterwards the function **Turn_aside_mobiles** (15) will choice some mobile objects that go towards the point A_i. The server could advise these objects for updating their routes towards the point B_i with the new obtained route.

Chapter 9

Implementation and Experiments

First, we are going to describe the implementation of the algorithm for k-Nearest Neighbor Search. We have implemented the algorithm in a java program. With JDBC, we have connected this program to an Oracle database. We have only stored the graph representation in the database because the algorithm just works with this representation. In appendix A there is an installation guide for doing the experiments and the data model we have used is in appendix B. Finally the program and the sql files are attached in CD. The program executes the algorithm that looks for the nearest neighbor 100 times. We have made the experiments twice.

The main idea is to know the influence in the time of search of the size of the road network, the number of the nearest neighbors that the algorithm is looking for and the number of nodes that the algorithm has to use from the query object to the nearest neighbors.

First, we want to know the influence of the size of the road network. Before we ran the algorithm, we thought that the size of the road network was not important. It could get more importand when the algorithm is looking for mobile objects in the database, but with indexes for the big tables, the problem is solved. The reason was that the algorithm starts to look for nearest neighbors and it finishes when it reach the necessary number of nearest neighbor. We thought that the road network's size was not important in the search because the algorithm finishes when it reaches the nearest neighbors.

Because of this, the number of nearest neighbors is fixed in the tree. Furthermore, the four nearest neighbor was close in the edges from the query object one. The first experiment is the example 6.2 in the section 6.4. This example was executed 100 times. The results are shown in the following table. Then, the next experiment was with a road network with 12 nodes and 19 edges. The algorithm was executed 100 times and the program calculates the average. The results are shown in the same table. Finally, we introduced a big road network, with 185 nodes and 269 edges in the database. Once again, the program executed 100 times the algorithm and the average was calculated. The result is also in the same table.

Table 1. Results with different road networks (in milliseconds).

	small road network		medium road network		big road network	
Average	2451.13	2450.44	1412.8	1377.6	2658.97	2637.85
Minimum Value	2248	2177	1221	1228	2525	2551
Maximum Value	3838	4727	2833	3457	6053	3206

The results show that the size of the road network does not matter, as we thought. It is because the algorithm finishes when the k nearest neighbor are reached and the following nodes have a bigger weight than the worst nearest neighbor. The algorithm does not process all the road network if there are enought objects.

The following step is to know the influence of the number of nearest neighbor that the algorithm must look for. We thought that the number of nearest neighbor is not important if the objects that the algorithm is looking

for are close. The algorithm finish when it reaches all the nearest neighbor (or when the maximum weight is reached). If all of them are close in weight, the algorithm reaches the same number of edges and nodes, and it does not increase the time.

We used the model of a big city. First, we looked for the first nearest neighbor and then we looked for 3,5,10,14 and 15 nearest neighbors. In this example, all the nearest neighbors are close to the query object too. The results are shown in the table 2.

Table 2. Results with different numbers of nearest neighbors (in milliseconds).

number of nearest	1		3		5	
Average	1006.78	1012.56	2658.97	2637.85	8612.89	8600.69
Minimum Value	957	961	2525	2551	8429	8390
Maximum Value	1114	1586	6053	3206	9048	9582

number of nearest	10		14		15	
Average	12590.73	12600.08	21227.32	21263.58	64694.76	64373.37
Minimum Value	12121	12074	20576	20453	63350	63018
Maximum Value	13537	13189	22166	22404	66737	66195

The number of nearest neighbor candidates is important too. The reason is that the algorithm continues with the search and it implies reaching more nodes. If the algorithm is looking for more nearest neighbors than candidates are in the database, it must continue with the search and it could stop if the weight of the edges reaches the bound. The results with 15 nearest neighbors are worse than 14 because the 15^{th} nearest neighbor is in other area of the networks and the algorithm have to pass through more nodes in order to get to it.

The last step is to know the incidence of the number of nodes that the algorithm must cross in the search of the nearest neighbor. We thought that the most important influence in the time of the search was the number of nodes that the algorithm cross in the search of the nearest neighbors. The reason of it is the way in which our algorithm looks for the nearest neighbor.

We have used the previous example and now we look for the 3 nearest neighbor. But now, we have changed the properties that they must satisfy. We made three groups of nearest neighbors. The first group is close to the query object. The second group is the medium distance one, concerning the number of nodes that they must cross to go to the meeting point. The third group is further away to the query object. The results are shown in the table 3.

Table 3. Results with different distance of the nearest neighbors (in milliseconds).

distance of nearest neighbors	nearer		in the middle		more distant	
Average	11055.48	11058.85	47172.26	47085.71	56211.9	56383.32
Minimum Value	10561	10626	45844	46067	55152	54679
Maximum Value	11485	11592	48043	48275	58541	58016

It shows the importance of this parameter. The distance in nodes implies that the program must manage more nodes. The differences between the experiments are revealing: the results show us that the management of the nodes must be improved.

But the time with 3 or 5 nearest neighbors in the big road network when they are not located near the query object is not bad. It needs between 5 or 10 seconds. Sometimes, when someone wants to know her or his nearest neighbors, she or he is interested in near mobile objects, not in objects that are far from the query object.

But if it is needed to improve the time of the algorithms, the tree that manages the nodes can be changed for another structure. The algorithm can work in the same way because the management of the nodes is not important in the way that the algorithm looks for the nearest neighbors.

Chapter 10

Conclusions and future work

In this data model we try to improve two data models, [1] and [2]. We use the best ideas of this data models. In fact,first we improve [1] with the best ideas in [2]. The reason is that the first data model has a good problem definition and the transformation between both models is simple but logical. Another reason is that this data model works with moving data points and in [2] they do not consider it.

Another improvement is to introduce the route of the mobile objects into the data model. The reason is that this way you can anticipate future traffic jams and problems on the route. For example, if you know the route of all mobile objects in the road network and you know that a big number of them are crossing a street at the same time, you can warn some of them and they could change their routes.

We have considered in our data model details like the direction of the roads, the characteristics of the pavement and the traffic jam. These facts affects the speed of the mobile object, so are as important as the legth of the road. The combination of all the facts helps the system to decide on the route to take. We model these characteristics with the properties in the 2D representation. When we transform it to a graph representation, we are working only with weights but taking into consideration these factors.

Someone can say that how you calculate the level of traffic jam or the level of traffic signals. For example, the level of traffic jams can be calculated as the number of mobile objects in the street divided by the length of the street. Or can be estimated by a external agent: for example police cars can notify the server of the incidents in the roads, as they move around in the city; It also can be updated from the information taken from cameras all around the city.

But why we have chose the nearest neighbors in time as our nearest neighbors? Nowadays the time is more important than another questions. And it is a logical idea. People is not interested if a taxi is only one kilometre away. People are interested in knowing that a taxi can go towards him or her in three or five minutes. Sometimes, a mobile object can be closer in distance than another one, but it can need more time to go towards the query object because a traffic jam in the street where he is.

Another idea that we have introduced is the partial solution for the reverse nearest neighbor problem. We think that it is not reasonable that someone must wait for one of its nearest neighbor for a long time. Perhaps the sum of all times that the query objects must wait for their nearest neighbor is bigger in this way, but no one must wait for a long time: the average of the delay can be bigger but no one waits for a long time.

Finally, we propose an algorithm that tries to solve the traffic jam problem. This algorithm take advantage of the route in the mobile objects. It calculates the future position of the mobile objects and it turns aside some of them when a traffic jam is detected.

Concerning the future work, we think that the data model cannot be improved much. In this work we have improved two data model and we have taken the best ideas in each one. We think that our algorithm could be improved and can probably use another data structure that improves its efficiency. Even the algorithm can be

modified, for example considering the velocity of the nearest neighbors.

Another research area can be the traffic jam problem. If the system knows the routes of the all query object, it can detect the streets or zones where a future traffic jam can appear. It is an interesting area because the traffic jam problem is pressing in some big cities. The new routes of the mobile objects should consider the final destination of the mobile objects in this algorithm. Furthermore, it must be careful and take care of not introduce another traffic jam in other areas. Our algorithm is a first approximation and it could be implemented. Several simulations could prove its efficiency and what could be modified.

This new task could be very expensive in computation time. For this reason, another investigation area could be the management of the nodes, edges and mobiles between several servers. A protocol between the servers could be introduced. Each server could manage an area in the city road network, and a mobile object changes its zone as it is moving; The server must consider this possibility. Furthermore, as we have proposed a thin client and a thick server so the client can be implemented in all type of devices, the server could have too much work, and could be very good to divide this work between several servers.

Bibliography

[1] Jan Kolar, Igor Timko. *Active Ranked k-Nearest Neighbor Queries for Moving Query and Data Points in Road Networks*,June 2002

[2] Irina Aleksandrova, Augustas Kiglys, Laurynas Speicys. *Computacional Data Modeling for Queries in Road Networks*, June 13th, 2002

[3] Irina Aleksandrova, Augustas Kiglys, Laurynas Speicys. *Processing of Active Nearest Neighbor Queries in Road Networks*, June 13, 2002

[4] Rimantas Benetis, Cristian S.Jensen, Gytis Karciauskas, Simonas Saltenis. *Nearest Neighbor and Reserve Nearest Neighbor Queries for Moving Objects*, October 17, 2001

[5] Oury Wolfson, Bo Xu, Sam Chamberlain, Liqin Jiang. *Moving Objects Databases: Issues and Solutions*

[6] A.Prasad Sistla, Ouri Wolfson, Sam Chamberlain, Son Dao. *Modeling and Querying Moving Objects*

[7] Oury Wolfson. *Moving Objects Information Management: The Database Challenge*

Appendix A

Installation guide for the simulation.

A.1 Setting up enviroments

In this appendix we are going to describe all the steps to realize the experiments in the machine luke. This machine pertains to Aalborg University. We have realized the experiments on this machine and for running the experiments in another machine some steps could change in the beginning.

First, in your ".cshrc" file you must write:

```
setenv ORACLE_HOME /pack/oracle
setenv ORACLE_SID blob
setenv TWO_TASK $ORACLE_SID
setenv PATH $ORACLE_HOME/bin:$PATH
setenv LD_LIBRARY_PATH $ORACLE_HOME/lib:$ORACLE_HOME/jdbc/lib
setenv JAVA_HOME /pack/jdk-1.3.1
setenv CLASSPATH $JAVA_HOME/lib/tools.jar:/user/joserua/OracleClass/oracle.jar:
/pack/oracle/jdbc/lib/classes111.zip:/pack/oracle/jdbc/lib/nls_charset11.zip:
/pack/oracle/jdbc/lib/classes12.zip:/pack/oracle/jdbc/lib/nls_charset12.zip:.
```

It configures your UNIX environment. Now, you can use the program sqlplus and compile the java programs. For others machines this part could be different, if your machine uses another JDK version.

In the following step, you must save the two java programs in the same directory and update the con_db.java file. In the variables DB_url, DB_user and DB_password in the program, write:

```
private static String DB_url = "jdbc:oracle:oci:@blob";
private static String DB_user = "user";
 // Write your user in the oracle system.
private static String DB_password = "password";
 // Write your password in the oracle system.
```

Then, compile the file simulation.java with "javac simulation.java". It compiles the con_db.java file too. Then, put all the sql files in same directory. Finally, in sqlplus, write "@model". You must start sqlplus in the same directory where you have put the sql files.

This final step creates the tables in the database. There are seven tables, as you can see in Appendix B. There is a table that will store the nodes and another one that will store the edges. The table "mobiles" will store the

data of the mobiles but the current edge is in another one: route. The current edge is stored in this table and the value of the field "step" for the current edge is zero. The factors of the mobiles are store in another table.

There are two additional tables, experiment and conditions, which are used only for your comfort. In this table configure the parameters of the experiment and the conditions that the nearest neighbor must satisfy. These paramaters are the query object, the name of the experiment, the meeting time, the maximum weight until the algorithm stops the search, the type of search (zero implies that the algorithm look for static objects and an one implies that the algorithm looks for mobile objects), the number of conditions that the nearest neighbors must satisfy and the number of experiments that the program executes. The table conditions describes the conditions that the mobiles must satisfy in the system. When the program starts the execution, it uses all this parameters.

A.2 First experiment.

After you have done the previous steps, you can start the experiments.

In sqlplus you must write "@example1". It loads the nodes, the edges and all the information about the mobiles in the database. This data is the nodes, the edges and the mobile objects in the example 6.2 (section 6.4). Furthermore, it loads the parameters of the experiments. Finally, you can write "java simulation" and the program looks for the 3 nearest neighbor (if you want to look for a different number of nearest neighbors, you must write "java simulation (number of nearest neighbors)". The default number is three).

When the program finishes, you must write "@example2" in the sqlplus program and then execute the java program ("java simulation") another time. The last road network is loaded in the database writing "@example4" in the sqlplus program. It just loads the big road network (185 nodes and 269 edges). Write "example4a" for loading the mobile objects and the configuration of the experiment). Then, execute once again the java program.

If you write in the sqlplus program: "@example3", the medium road network is loaded and the program looks for static points.

A.3 Second experiment.

After you have done the previous steps, you can start the second experiment.

In sqlplus, write "@example4" (remember! it only loads the nodes and the edges) and then "@example4a". In the prompt in your UNIX session write:

"java simulation 1", that looks for the first nearest neighbor.
"java simulation 3" (or only "java simulation"), that looks for the three nearest neighbor.
"java simulation 5", that looks for the five nearest neighbor.
"java simulation 10", that looks for the ten nearest neighbor.
"java simulation 15", that looks for the fifteen nearest neighbor.

A.4 Third experiment.

After you have done the previous steps, you can start the third experiment.

In sqlplus, write "@example4" and then "@example4b". Finally, write in your UNIX session "java simulation".

When the program finishes, update the database in the sqlplus program with:

"update conditions set value = 'Taxis B' where name ='2';"

"commit;"

Then, run the program another time.

Finally, update another time the database in sqlplus program with:

"update conditions set value = 'Taxis C' where name ='2';"
"commit;"

and run the program.

Appendix B

The data model in the database.

```
drop table conditions;
drop table experiment;
drop table edges;
drop table factors;
drop table route;
drop table mobiles;
drop table nodes;

create table nodes (
name VARCHAR(4) primary key);

create table edges (
start_node VARCHAR(4),
end_node VARCHAR(4),
weight_se NUMBER(8,4) not null,
weight_es NUMBER(8,4) not null,
change INT not null,
constraint edges_pk primary key (start_node,end_node),
constraint nodes_s1_fk foreign key (start_node) references nodes(name) on delete cascade,
constraint nodes_e1_fk foreign key (end_node) references nodes(name) on delete cascade);

create table mobiles (
name VARCHAR(8),
current_time INT not null,
current_position_se NUMBER(8,4) not null,
current_position_es NUMBER(8,4) not null,
current_velocity NUMBER(8,4) not null,
current_direction VARCHAR(2) not null,
constraint mobiles_pk primary key (name));

create table factors (
num_id INT,
name VARCHAR(8),
name_factor VARCHAR(30),
value VARCHAR(20) not null,
```

```
constraint factors_pk primary key (num_id,name),
constraint name1_fk foreign key (name) references mobiles(name) on delete cascade);

create table route (
name VARCHAR(8),
step INT,
start_node VARCHAR(4) not null,
end_node VARCHAR(4) not null,
direction VARCHAR(2) not null,
constraint route_pk primary key (name,step),
constraint nodes_s2_fk foreign key (start_node) references nodes(name) on delete cascade,
constraint nodes_e2_fk foreign key (end_node) references nodes(name) on delete cascade,
constraint name2_fk foreign key (name) references mobiles(name) on delete cascade);

create table experiment (
name_experiment VARCHAR(12),
name VARCHAR(2) not null,
meeting_time INT not null,
max_weight NUMBER(8,4) not null,
type_search INT not null,
number_of_conditions INT not null,
number_of_experiments INT not null,
constraint experiment_pk primary key (name_experiment),
constraint name3_fk foreign key (name) references mobiles(name) on delete cascade);

create table conditions (
name_experiment VARCHAR(12),
name VARCHAR(2) not null,
value VARCHAR(12) not null,
constraint conditions_pk primary key (name_experiment,name),
constraint name_experiment_fk foreign key (name_experiment) references experiment(name_experiment)
on delete cascade);
```