Query Processing for Moving Objects with Space-Time Grid Storage Model

Hae Don Chon, Divyakant Agrawal, Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{hdchon,agrawal,amr}@cs.ucsb.edu

Abstract

With growing popularity of mobile computing devices and wireless communications, managing dynamically changing information about moving objects is becoming feasible. In this paper, we implement a system that manages such information and propose two new algorithms. One is an efficient range query algorithm with a filtering step which efficiently determines if a polyline corresponding to the trajectory of a moving object intersects with a given range. The other is a nearest neighbor query algorithm. We study the performance of the system, which shows that despite the filtering step, for moderately large ranges, the range query algorithm we propose outperforms the algorithm without filtering.

1 Introduction

According to the recent report, The 2001 Urban Mobility Report, from the Texas Transportation Institute [12], the total congestion cost for the 68 major U.S. urban areas in 1999 came to $78 billion, which was the value of 4.5 billion hours of delay and 6.8 billion gallons (or 25.74 billion liters) of excess fuel consumed. Considering the skyrocketing gas price in the year 2001, the congestion cost would be now well over $80 billion a year. Los Angeles, California was ranked first($12.5 billion) in the congestion cost followed by New York, NY and Northeastern, NJ($9.7 billion). Road expansions slow the growth in congestion, however, they do not seem to be the answer. The solution would be really a diverse set of options, and more efficient operations can derive benefits from existing systems [12].

Recently, efficient storage and retrieval of moving objects in Database Management Systems have received significant interest [13, 14, 6, 11, 8, 3]. With the advances in GPS technology and wireless communications, the management of dynamically changing information about moving objects is becoming feasible and with growing popularity of mobile devices, a large number of vehicles will be equipped with such devices. In addition, the information about what vehicles(moving objects) are in some specific section of a route in some period of time would be useful to divert some vehicles to alternate routes to alleviate congestion, therefore reducing the congestion cost, which is astronomical. Another example of applications which could benefit from using this dynamically changing information is a fleet management application, where accurately estimated arrival time and a shortest path to a certain location are very crucial for just-in-time manufacturing.

However, such information is dynamically changing and keeping track of each and every vehicle's current location by continuously polling is hard to achieve [3]. In general, the future location of a moving object can be modeled as a linear function of time so that the trajectory of a moving object becomes a line segment in a space-time domain. Hence, the future locations of a moving object can be determined without polling its location at regular intervals. Another aspect of this problem is that the trajectories of moving objects are affected by other moving objects. We observed that the trajectory of a moving object is the result of interactions among all the moving objects, which leads us to model the trajectory as a polyline instead of a single line segment. We then develop a model that manages such dynamically changing information. In this paper, we not only summarize the model in Section 2, but also implement the system along with new algorithms to process range queries and k nearest neighbor queries and report the performance of these algorithms.

Query processing of line segments is not new to the database community. Especially, Hoel and Samet [5] use PMR Quadtrees [7] to store and index line segments as opposed to the partitioning approach we take in this paper. Another difference from our work is that the line segments data used in [5] is static since it is used for spatial objects. However, Tayeb et al. [14] model the trajectory of a moving object as a line in a domain space and use also the PMR quadtree to dynamically index the corresponding line. Pfoser et al. [8] model the trajectory of a moving object as a polyline as we do. The polyline in their work, however, is derived from the location information recorded in the past by a moving object whereas it is the result of interactions among all moving objects in the system in our work. In addition, Pfoser et al. focus on querying the past rather than the future, which is in the focus of our work.

The paper is organized as follows: we summarize the system in Section 2 and propose an efficient range query algorithm and a k nearest neighbor query algorithm in Section 3. Section 4 reports the results of the performance study and Section 5 concludes the paper.

2 Background

In the previous work [2], we develop a model that manages dynamically changing information about moving objects. In this section, we give readers the summary of the system which is the implementation of the model. We first explain assumptions we made, data structures we used, and definitions.

2.1 Assumptions, Data Structures, and Definitions

The following assumptions are made about the domain space and moving objects: Since any path in a network of one dimensional lines in a two dimensional space can be divided into subpaths, each of which belongs to a one dimensional line, without loss of generality, we assume that a moving object moves on a one dimensional line (can be considered as a road, especially a highway). The one dimensional line on which moving objects move has physical limitations. For example, as the number of vehicles in a section increases, the vehicles may not be able to maintain the intended velocity. For this reason, we assume that each section of a route is associated with a predefined function of the maximum velocity at which a moving object can move. This maximum velocity function associated with a section will return the maximum velocity a moving object can move in the section depending on the number of moving objects currently in the section. We derive the function from statistical data(See Section 4). We partition the domain space $T \times Y$, where T corresponds to the time domain and Y corresponds to the space domain, into an array of cells such that $Y = [y_0, y_1) \cup ... \cup [y_{p-2}, y_{p-1}) \cup [y_{p-1}, y_{max}]$ and $T = [t_0, t_1) \cup [t_1, t_2) \cup ... \cup [t_{q-1}, t_q)$, where $t_q = t_0 + \Delta T$. A cell $g[i, j]$ is defined as a rectangle $[t_i, t_{i+1}) \times [y_j, y_{j+1})$, for some $t_i \in T, y_j \in Y$. Note that although time is infinitely increasing, we only maintain the information about some predefined ΔT period of time. As time evolves, we shift the time domain forward. We also assume that on entry to the system, a moving object m is responsible to provide initial information such as the starting location(s), the starting time(t_s), the destination(e), and the initial velocity(v), which is the maximum velocity at which it intends to move. When a moving object m appears in the system providing its initial information, it is inserted into the system as a polyline and the system returns the polyline information $\langle p_0, ..., p_e \rangle$ to m.

We use the following data structures in the implementation: A singly linked list with an extra pointer to the last point is used to represent the polyline corresponding to a moving object's trajectory. The entire grid is implemented as an array of cells, which efficiently supports random accesses to individual cells. An individual cell stores only the identifiers of intersected moving objects along with the velocity of the moving objects in the cell using a skip list [9]. Since we do not store the entire information about a moving object in each cell, we need to store it somewhere else, which we call an *Object Store*. A hashtable is used for the Object Store, which needs to support very frequent random accesses.

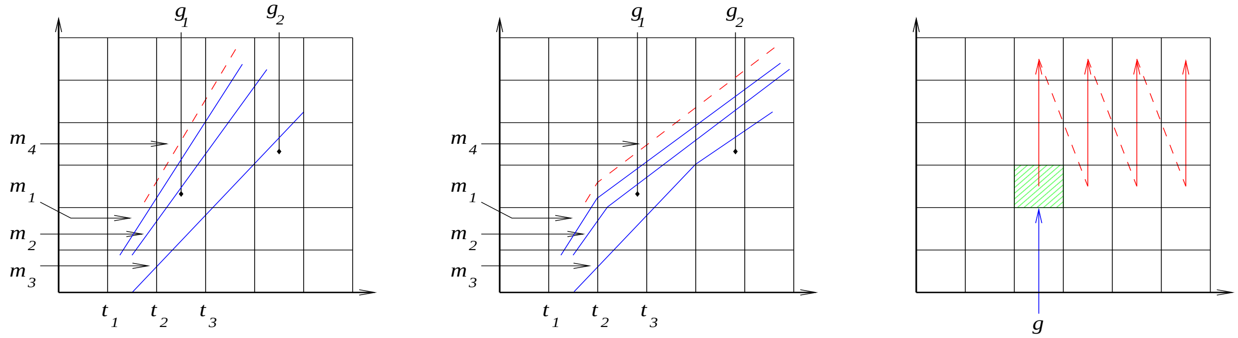
Before we explain the insertion algorithm in Section 2.2, we define a notation $\triangleright$ and a term *entry point* as follows: If the corresponding polyline of m is intersected with a cell g, m is said to intersect with g and denoted by $m \triangleright g$. An entry point of m to a cell g is a point where m's corresponding polyline and g intersect for the first time.

2.2 Insertion

We now explain how to insert m into the system. With the initial information $m = (s, t_s, e, v)$, we find the first cell g such that $m \triangleright g$ and repeat the following steps: we compute the maximum velocity allowed in g which depends on the number of moving objects in the cell, and adjust m's velocity accordingly. When there is a velocity change, we add the corresponding entry point to the linked list associated with m which will be the polyline. We then find next cell to intersect computing the entry point to the cell, and insert its identifier along with the adjusted velocity into g(skip list). We repeat these steps until we process the cell where the destination e of m belongs. In that case, we complete the insertion process by inserting m into the Object Store, and return the polyline information to m.

There are, however, cases where one insertion may cause a series of updates. Consider the situation shown in Figure 1(a) where three moving objects, $\{m_1, m_2, m_3\}$, are in the system and a new one, m_4, is about to be inserted. Assuming that three moving objects is a threshold for moving objects to reduce their velocities to a certain degree, the cell g_1 is about to have three objects so that existing objects as well as the new one will need to reduce their velocities based on the maximum velocity function associated with g_1. Furthermore, the updates of m_1 and m_2 cause the cell g_2 to have three moving objects. As a result, another update on m_3 has to be executed. Figure 1(b) shows the result of the insertion of m_4. A moving object m_4 in this case is said to cause a ripple effect. For this purpose, we need to check if the number of objects in g reaches a threshold, before inserting a moving object m into a cell g, by checking $f_g(|g|) > f_g(|g|+1)$, where $f_g(n)$ is the maximum velocity function associated with g. If it is true, we invoke an algorithm called *RippleInsert* providing a pair of (row, column) of g. Otherwise continue the insertion of m until we finish it.

The RippleInsert algorithm works as follows: starting with the first cell g whose maximum velocity needs to be changed because of the insertion of a moving object, we adjust information about all moving objects in g according to the new velocity allowed in g and continue this process with the cells whose time and location intervals are greater than or equal to those of g. We do not have to process any other cells since an insertion would not cause any update to the past and to the previous locations. The cells on the left are processed first and among those with same time interval, lower cells (smaller location value) first. See Figure 1 (c).



(a) before (b) after (c) ripple order

Figure 1: Ripple Effect

3 Query Processing

In this section, we describe query processing algorithms in the context of space-time grid storage model. We first consider the class of queries that are referred to as the range queries. A range query retrieves the identifiers of moving objects that are in the specified space and time domain. The other class of queries we consider is nearest neighbor queries which are of significant interest in designing solutions for location-based applications.

3.1 Range Query over Space-Time Grid Model

Range queries are crucial in many applications. A typical range query Q is *Retrieve all moving objects that would be between locations* y_a *and* y_b *during time period* $[t_a, t_b]$. In other words, find m such that $m \triangleright R = [t_a, t_b] \times [y_a, y_b]$. To answer this query, we need to first find the cells which intersect with R. There are two ways for a cell g to intersect with R. One is to be contained entirely, $g \subset R$. The other is that g partially intersects with R, $g \not\subseteq R$ and $g \cap R \neq \phi$. If $g \subset R$, all $m \triangleright g$ are part of the answer. Otherwise, we need to filter out those m such that $m \triangleright g$ but $m \not\triangleright R$. However, this involves reading the Object Store since g contains only the pointer to m. Thus, we can make this filtering step optional so that we can have an answer quickly in time critical applications. Without filtering, however, we can only give an approximate answer as follows: We find the cell $g_s[r_s, c_s]$ such that $(t_a, y_a) \in g_s$ and the cell $g_e[r_e, c_e]$ such that $(t_b, y_b) \in g_e$. Then cells $g[r, c]$ satisfying $r_s \leq r \leq r_e$ and $c_s \leq c \leq c_e$ intersect completely or partially with R. Note that since a moving object is most likely inserted into multiple cells, the cells intersected with R may have multiple instances of a moving object. Since cells store moving objects' identifiers in a skip list ordered by the identifiers, we can apply the merge sort algorithm to merge the skip lists of those cells intersected with R, which will be the approximate answer to the range query. We call this an *approximate* algorithm.

We now develop an algorithm that retrieves only the moving objects that are actually intersected with R and we call this a *basic* algorithm. As an example, see Figure 2 where a range R intersects with 9 cells which contain 5 moving objects all together. The basic algorithm we develop here should retrieve only m_2, m_3, m_4, whereas the approximate algorithm retrieves all five objects. The basic algorithm first merges the cells g which are completely contained in $R(g \subset R)$. Note that if cell $g[r, c]$ satisfies $r_s < r < r_e$ and $c_s < c < c_e$, then g is completely contained in R, and therefore, all moving objects m such that $m \triangleright g$ are also $m \triangleright R$. After the merge, for each moving object m in cells g such

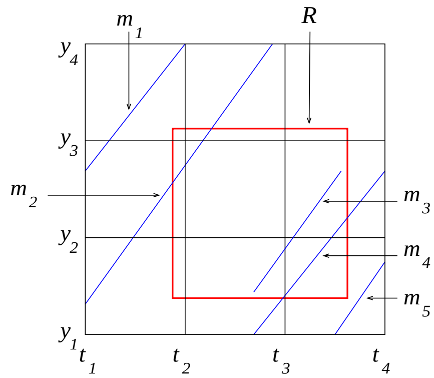
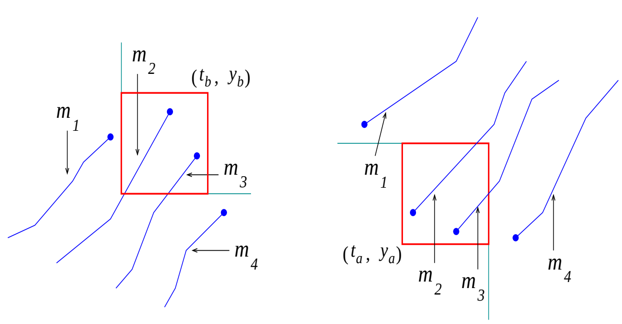


Figure 2: Range Query

that $g \nsubseteq R$ and $g \cap R \neq \phi$, we need to check if m intersects with R. Only if $m \triangleright R$, we add it to the answer set.

To check if $m \triangleright R$, we devise a very efficient filtering algorithm, *isIntersected*: Note that a polyline which corresponds to the trajectory of a moving object is represented as a singly linked list of points with an extra pointer to the last point and a range R is represented by two points, (t_a, y_a) and (t_b, y_b). Note also that the polyline is monotonically increasing since we assume that a moving object moves forward as time evolves. With these properties in mind, the basic idea of the filtering algorithm is to minimize the number of comparisons by pruning unnecessary comparisons. More specifically, we first try to see if we can trivially determine $m \triangleright R$. If not, we start to scan the linked list, and as soon as we can determine whether or not $m \triangleright R$, we return the answer eliminating further comparisons.

Let us now explain the algorithm *isIntersected* in more detail. Before starting to scan the linked list, we first examine the last point, (t_e, y_e), of the corresponding polyline to see if we can trivially determine $m \triangleright R$. As m_1 and m_4 in Figure 3(a) where the last points are depicted by dots, if $t_e < t_a$ or $y_e < y_a$, then $m \ntriangleright R$. On the other hand, if $(t_e, y_e) \in R$ ($t_a \leq t_e \leq t_b$ and $y_a \leq y_e \leq y_b$) such as m_2 and m_3 in Figure 3(a), obviously $m \triangleright R$. Note that since we already checked if $t_e < t_a$ or $y_e < y_a$, we only need to check if $t_e \leq t_b$ and $y_e \leq y_b$ to see if $(t_e, y_e) \in R$. Similarly, we can check trivial cases for the starting point (t_s, y_s) as in Figure 3(b). If $t_s > t_b$ or $y_s > y_b$, then $m \ntriangleright R$ and if $(t_s, y_s) \in R$, $m \triangleright R$.



(a) Trivial cases for the last point (b) Trivial cases for the first point

Figure 3: Trivial Cases

After checking these base cases, if we still cannot determine if $m \triangleright R$, we start the scan of the linked list. There are only two possible cases left for m to intersect with R as illustrated in Figure 4 (unless otherwise stated, $m_1, \ldots, m_7$ we refer from now on are in Figure 4). One is that $m \triangleright R$ at the left boundary such as m_1 and m_2 and the other is that $m \triangleright R$ at the lower boundary such as m_3 and m_4. Note that m cannot intersect with R at either the upper or the right boundary without intersecting at either the lower or the left boundary since we already check such cases as the base cases. In other words, the starting point of m should be in the area $A - B$, where $A = \{(t, y) | t \leq t_b \ \wedge \ y \leq y_b\}$ and $B = \{(t, y) | t_a \leq t \ \wedge \ y_a \leq y\}$, and the ending point of m should be in the area $B - A$. Now,

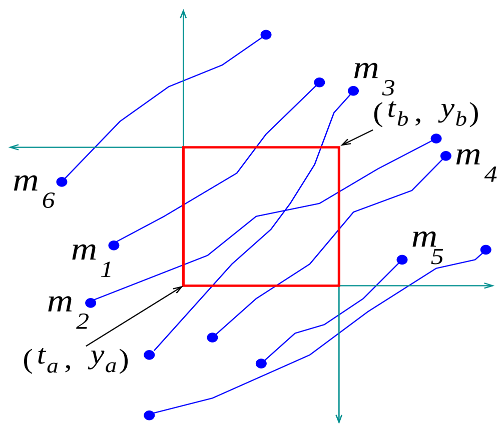


Figure 4: Filtering

we first try to see if $m \triangleright R$ at the left boundary scanning the linked list looking for the two points, $\alpha = (t_\alpha, y_\alpha)$ and $\beta = (t_\beta, y_\beta)$ such that $t_\alpha \leq t_a \leq t_\beta$. Unless the t-coordinate of the starting of m is greater than t_a such as m_4 and m_5 in which case we omit this step and process the next step, we are guaranteed to find such points. Once we find α and β, we can compute the location (y_m) of m at t_a. If y_m is greater than y_b(such as m_6), $m \not\triangleright R$ since the polyline is monotonically increasing. And if y_m is in between y_a and y_b, $m \triangleright R$. Otherwise($y_m < y_a$), we cannot determine(compare m_3 and m_7). Therefore as next step, we continue scanning looking for another two points, $\gamma = (t_\gamma, y_\gamma)$ and $\delta = (t_\delta, y_\delta)$ such that $y_\gamma \leq y_a \leq y_\delta$, and compute the t-coordinate (t_m) of m at y_a. If t_m is less than or equal to t_b, we can conclude that $m \triangleright R$. Note that we process this step because either $t_a < t_s$ or $y_m < y_a$. If $t_a < t_s$, then $t_a < t_m$ since any t-coordinate of m is greater than t_s. If $y_m < y_a$, then $t_a < t_m$ since the point (t_m, y_a) on the polyline is at the upper right direction from the point (t_a, y_m) due to the monotonicity of the polyline. Therefore, if $t_m \leq t_b$, then $t_a \leq t_m \leq t_b$. However, if $t_b < t_m$, $m \not\triangleright R$ since again the polyline is monotonically increasing and we do not need further comparisons, thus, the algorithm terminates here. Asymptotically, *isIntersected* is $O(n)$ with respect to the number of points in the linked list, n. We summarize this filtering algorithm in Figure 5.

Since a moving object may be inserted into multiple cells, the basic algorithm may apply the filtering algorithm redundantly to the same moving object. To address this problem, we develop the following algorithm: First, the algorithm retrieves moving objects from cells that are completely contained in R and keep them in a separate list, T, as a temporary result. In addition to this temporary list T, we also keep another temporary list, V, for objects we already checked and found out not to be intersected with R. We keep T and V in the order of the identifiers of objects so that we can use the binary search algorithm for searching and adding moving objects (For the simplicity of the implementation, java.util.Vector class is used for T and V). Second, for each moving object m

isIntersected(m, $R((t_a, y_a), (t_b, y_b))$):
(t_e, y_e) = last point;
if($t_e < t_a \ \vee \ y_e < y_a$)
    return false;
else if($t_e \leq t_b \ \wedge \ y_e \leq y_b$)
    return true;
(f_t, f_y) = first point;
if($t_b < f_t \ \vee \ y_b < f_y$)
    return false;
else if($t_a \leq f_t \ \wedge \ y_a \leq f_y$)
    return true;
// go to the right column $\nearrow$

// continued from the left column
if($f_t < t_a$)
    while(not done)
        (s_t, s_y) = next point;
        if($t_a \leq s_t$) break;
        else $(f_t, f_y) = (s_t, s_y)$;
    $v = (s_y - f_y)/(s_t - f_t)$;
    $y_m = v * (t_a - f_t) + f_y$;
    if($y_b < y_m$) return false;
    else if($y_a \leq y_m$) return true;
while(not done)
    if($t_b \leq s_t$) break;
    else $(f_t, f_y) = (s_t, s_y)$;
    (s_t, s_y) = next point;
$v = (s_y - f_y)/(s_t - f_t)$;
$t_m = (y_a - f_y)/v + f_t$;
if($t_m \leq t_b$) return true;
else return false;

Figure 5: Filtering Algorithm

in cells that are partially contained in R, if m is not already in either T or V, we check if $m \triangleright R$ using the filtering algorithm *isIntersected*. If m indeed intersects with R, add it to T. Otherwise, add it to V. This way we can efficiently retrieve moving objects which actually intersect with R. We call this algorithm an *exact* algorithm. We compare the three algorithms(approximate, basic, exact) in Section 4.

3.2 Nearest Neighbor Queries

Nearest neighbor queries have received significant interest recently [10, 1, 4] in the context of multidimensional databases. A traditional k nearest neighbor query executes against a static database given a static query point. In this paper, however, we concentrate on k nearest neighbor queries in the context of moving objects such as *Given a query point* $p = (t_p, y_p)$, *find* k *moving objects near* p. We can categorize the nearest neighbor queries in the context of moving objects into two types, static and dynamic. A static nearest neighbor query is a query that is executed against a static database such as a spatial database given a moving query point. An example of such query is *Given a moving object* m *and time* t_p, *find five nearest gas stations to* m. We could process this query by first computing the location of m at t_p and querying against a spatial database which stores the information about gas stations. However, once we compute the location y where m would be at time t_p, this query becomes a traditional k nearest neighbor query. There are many related work [10, 1, 4] for this type of queries, and therefore will not be addressed in this paper. On the other hand, a dynamic nearest neighbor query is executed against a dynamic database such as a moving objects database given a query point. The query point could be either static or dynamic. The dynamic query point, however, has to be converted into a static point at the time of query execution. Therefore, we do not differentiate between static and dynamic query points.

Let us now give an example of a dynamic nearest neighbor query: Suppose you are the owner of

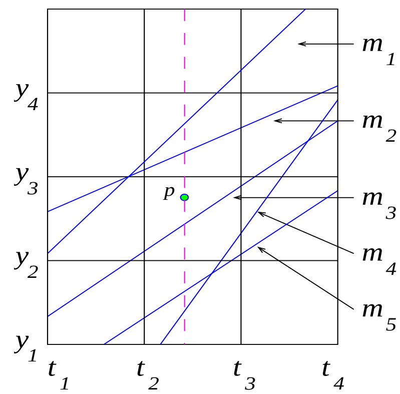


Figure 6: k Nearest Neighbor Query

a grocery store(y_p) and you are closing off the store at 4 p.m.(t_p). Before you close, you would like to sell off fresh vegetables, so you decide to advertise this to potential customers(k) near you at time t_p. A generic query of this type is *Given time t_p, find k moving objects near y_p* (in terms of location). In other words, *Find k smallest distances on y-direction from polylines to y_p at $t = t_p$ and return the corresponding moving objects*. As an example, see Figure 6 where there are five moving objects $\{m_1, ..., m_5\}$ and a query point $p = (t_p, y_p)$ is given which happens to be in the cell $g : [t_2, t_3] \times [y_2, y_3]$. Since cells store only the objects' identifier, we need to retrieve the polyline information from the Object Store to compute the distance between p and a moving object. We have to do this for all the moving objects in a cell in order to determine which one is closer to p since moving objects in a cell is not ordered by the distance to p. Furthermore, examining all the moving objects in the cell to which a query point p belongs may not be sufficient to answer a k nearest query. For example, assuming k is 3, if we examine only g in Figure 6, we would have had $\{m_3, m_5, m_4\}$ as an answer of 3 nearest moving objects to p whereas the actual answer should be $\{m_3, m_2, m_1\}$. The k nearest neighbor query algorithm we propose here addresses this issue.

We now explain how the algorithm works. To store k nearest neighbors in the order of distance to a moving object m from p at time t, we maintain a list KN and for efficient processing we keep a variable kD for the distance to the last object in KN from p. The distance, denoted by $d(m, p)$, can be computed as follows:

$$d(m, p) = |y_p - y_m|, \text{ where } y_m = \text{location of } m \text{ at } t_p.$$

As in the exact range query algorithm, we maintain a temporary list V. However, unlike the exact range query algorithm, V contains all the moving objects we already checked, whether they belong to KN or not. We then find a cell to which the query point p belongs and retrieve the first k moving objects that we encounter inserting them into KN updating kD. Note that we maintain KN in the order of distance and the maximum number of objects that KN keeps is k, $|KN| \leq k$. Therefore, after we insert a moving object into KN, if $|KN| = k + 1$, we discard the last element in KN so that $|KN|$ remains k. For the rest of the moving objects $m \triangleright g$, we compute $d(m, p)$ and if $d(m, p) < kD$, we insert it into KN and update kD accordingly. We have to do this for all the moving objects in g since the cell stores the objects in the order of their identifiers not by the distance to p.

After checking the cell g such that $p \in g$, we need to decide whether we need to search further to next cell or not since as seen in Figure 6, checking only one cell may not be sufficient. The order we choose next cell is in the order of distance to the border of the cell from p. For example, p in Figure

```
k Nearest Neighbor Algorithm(k, p = (t, y)):
kD = ∞;
initialize KN; // for kNeighbors;
initialize V; // for visited;
g = cell such that p ∈ g;
while(not done) {
    gD = distance to g from p;
    if(|KN| = k ∧ kD ≤ gD)
        break;
    for each m ▷ g
        if(m ∉ V)
            updateKND(KN, kD, m, k);
            V = V ∪ m;
    g = next cell to search;
}
return KN;
```

```
updateKND(KN, kD, m, k):
d_m = d(m, p);
if(KN = φ)
    KN = KN ∪ (m, d_m);
    kD = d_m;
if(|KN| = k ∧ d_m ≥ kD)
    return; // nothing to update
fD = distance of the first element in KN;
if(d < fD)
    KN = KN ∪ (m, d_m);
else
    for i = 1 to k
        if(|KN| = i)
            KN = KN ∪ (m, d_m);
            break;
        iD = distance of the ith one in KN;
        if(d < iD)
            KN = KN ∪ (m, d_m);
            break;
if(|KN| = k + 1)
    KN = KN− lastElement;
kD = distance of the last element in KN;
```

Figure 7: k Nearest Neighbor Query Algorithm

6 is closer to the cell above than the one below, so that we search $[t_2, t_3] \times [y_3, y_4]$ first, and then $[t_2, t_3] \times [y_1, y_2]$ next, and so on. Once we discover that searching further cells would not change KN, we can return KN as the answer and terminate the algorithm. To decide whether we search the next cell or not, we first compute the distance to the next cell from p, which is the distance to the lower boundary of the cell from p if the cell is in the upper direction of p or to the upper boundary if it is in the lower direction. If this distance is greater than or equal to kD and we already have k nearest neighbors in KN, then we could not find any more objects by searching the cell.

However, instead of the nearest neighbors in terms of location, if the nearest neighbors in terms of time are of interest, then we can apply the same algorithm with a different distance metric

$$d(m, p) = |t_p - t_m|, \text{ where } t_m = \text{time when } m \text{ is at } y_p.$$

We summarize the algorithm in Figure 7. In the next section, we report the performance study of this k nearest neighbor query algorithm as well as the insertion and various range query algorithms.

4 Performance Study

In this section, we first explain the experimental settings and in Section 4.2 report the performance study of the insertion algorithm, which is the crucial part of the system. We then compare the three range query algorithms and report the results in Section 4.3 where we also report the performance of the k nearest neighbor algorithm.

Parameters	ADT	MPH	AMT	MPM	Maximum Speed	vehicles/mile
Uncongested	Under 15,000	60	Under 10.42	1	2	10.42(11)
Moderate	15,001 - 17,500	45	10.42 - 12.15	0.75	0.82	16.2(17)
Heavy	17,501 - 20,000	38	12.15 - 13.89	0.63	0.69	22
Severe	20,001 - 25,000	35	13.89 - 17.36	0.58	0.63	29.9(30)
Extreme	Over 25,000	32	Over 17.36	0.53	0.58	32.8(33)

Table 1: Daily Traffic Volume Per Lane and Estimated Speed on Freeway

4.1 Experimental Settings

Since the system we implement is a main memory solution for now in that we keep all the data in main memory, we measure the performance of the system in terms of running time. For fairness, we use only one machine which is dedicated to this system and when the system is running, it uses the machine exclusively. The machine we used is a PC with a Pentium III 800MHz, 256MB, and on top of Windows 2000. We implemented the system in Java and run it with the JVM version 1.3.0.

There are basically two variables that could affect the performance of the insertion. One is the size of the grid, which is the number of rows and columns. In the performance tests, we use three different sizes of the grid, 20×20, 50×50, and 100×100. In the grids, a row corresponds to a section of the road(1-dimensional line) on which moving objects move and a column corresponds to time. Thus, if a cell is used to monitor a road of one mile long during one minute period, 100×100 grid can manage information about moving objects on a road of one hundred miles long for the next one hundred minutes. 50×50 grid can monitor a road for 50 minutes if we use the same granularity, however, it could monitor 100 miles for the next 100 minutes if we choose the granularity so that a cell monitors two miles for two minutes.

The other variable that could affect the performance of the insertion is the thresholds of the maximum velocity function which returns the maximum velocity at which a moving object can move depending on the number of moving objects a cell currently maintains. To choose the right maximum velocity function, the following three questions arise: *What is the right number of thresholds?*, *What is the right velocities for each threshold?*, and *What are the appropriate number of moving objects a cell can store before it reaches a threshold?*. To answer these questions, we refer to the 2001 Urban Mobility Report from the Texas Transportation Institute [12]. In the report, they categorize the congestion into 5 levels: uncongested, moderate, heavy, severe, and extreme. They also observed that as the daily traffic volume (the average number of vehicles observed in a day at a location) increases, the average speed of vehicles decreases. Table 1 shows Average Daily Traffic volume per lane(ADT) and the average speed in Miles Per Hour(MPH) for 5 congestion levels.

In the table, AMT means average minutely traffic and MPM means miles per minute. From these numbers, we can compute the number of vehicles in a mile. For the maximum speed, we increase the average speed (MPM) by 10 % but not faster than the average speed of the previous level. We set 2 MPM for the maximum speed of the uncongested level, which could correspond to 120 miles per hour which we believe is fast enough to be infinity. For example, the first 11 vehicles in a cell can drive as fast as they want as long as it is slower than 2 MPM and when 12th vehicle is added to the cell, every vehicle has to reduce its speed to 0.82 MPM and so on. In the next section, we show the performance result of the insertion algorithm.

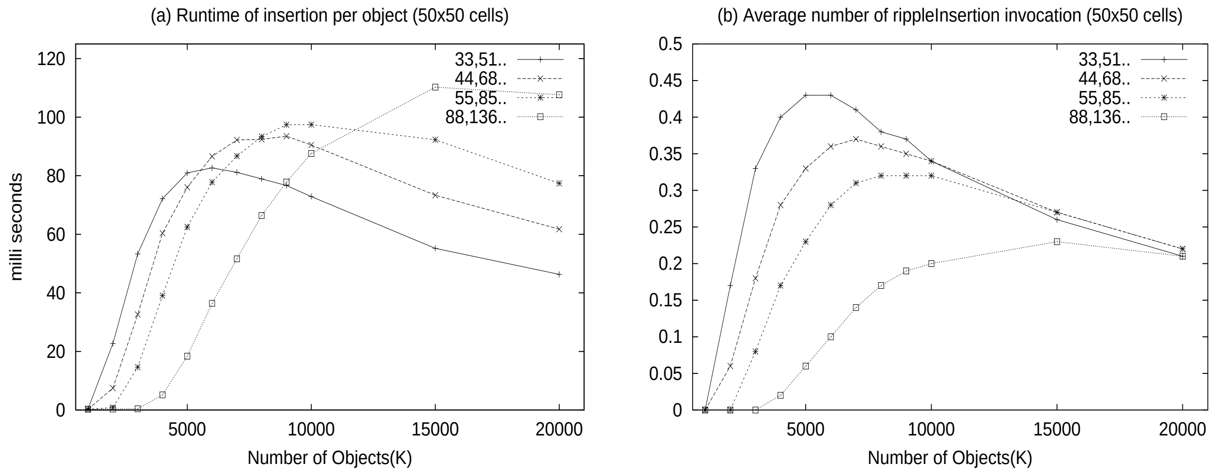


Figure 8: Fixed Grid Size: 50×50

4.2 Insertion

In this section, we study the performance of the insertion algorithm. We use three measurements, running time, the average number of intersected cells per object, and the average number of the invocation of the RippleInsertion method, which in fact is closely related with the running time. As we mentioned in the previous section, there are two variables that could affect the performance of the insertion, which are the size of the grid and thresholds. Therefore we show how these two variables affect the system performance.

First, we fix the grid size to be 50×50 and use the thresholds $L = (11, 17, 22, 30, 33)$ in Table 1 as a base case. We use one maximum velocity function for a test assuming all the sections have the same capacity throughout the entire road (could be 50 miles or 100 miles depending on the granularity). We then vary the thresholds from $3L = (33, 51, 66, 90, 99)$ to $8L = (88, 136, 176, 240, 264)$. For each thresholds, we initialize the system, generate four random numbers (s, t_s, e, v) for a moving object, and insert $N(= 1000, 2000, ..., 9000, 10000, 15000, 20000)$ objects into the system. We repeat these steps 10 times (5 times for 15000 and 20000 cases) and take the average. Figure 8(a) and 8(b) show the average running time of one insertion and the average number of invocations of RippleInsertion method per insertion, respectively. Note that as the number of the RippleInsertion method calls increases, the average running time also increases, which indicates that the RippleInsertion method has a significant impact on the performance of the system. Note that the higher the thresholds are, the less the number of the RippleInsertion method calls were invoked. This is because with higher thresholds, the grid has more capacity to hold more objects in a cell, that way less number of the method calls was incurred. For every test case, the average running time of an insertion starts to decrease at some point where the system is saturated. As Figure 8(a) shows, the saturation point comes later if the system has more capacity(higher thresholds).

Second, we fix the threshold to $4L = (44, 68, 88, 120, 132)$ and test with three grid sizes, 20×20, 50×50, and 100×100. Figure 9(a) shows the average running time of insertion per objects in terms of milli seconds. The running time with the grid size 20×20 is no more than 7.48 milli seconds per insertion which happens at the insertion of 3000 objects. This is due to the fact that the grid was saturated at the early stage(at 3000 insertions), after that not many RippleInsertion method calls occurred. In the mean time, the running time with the grid size 100×100 takes much time, not only

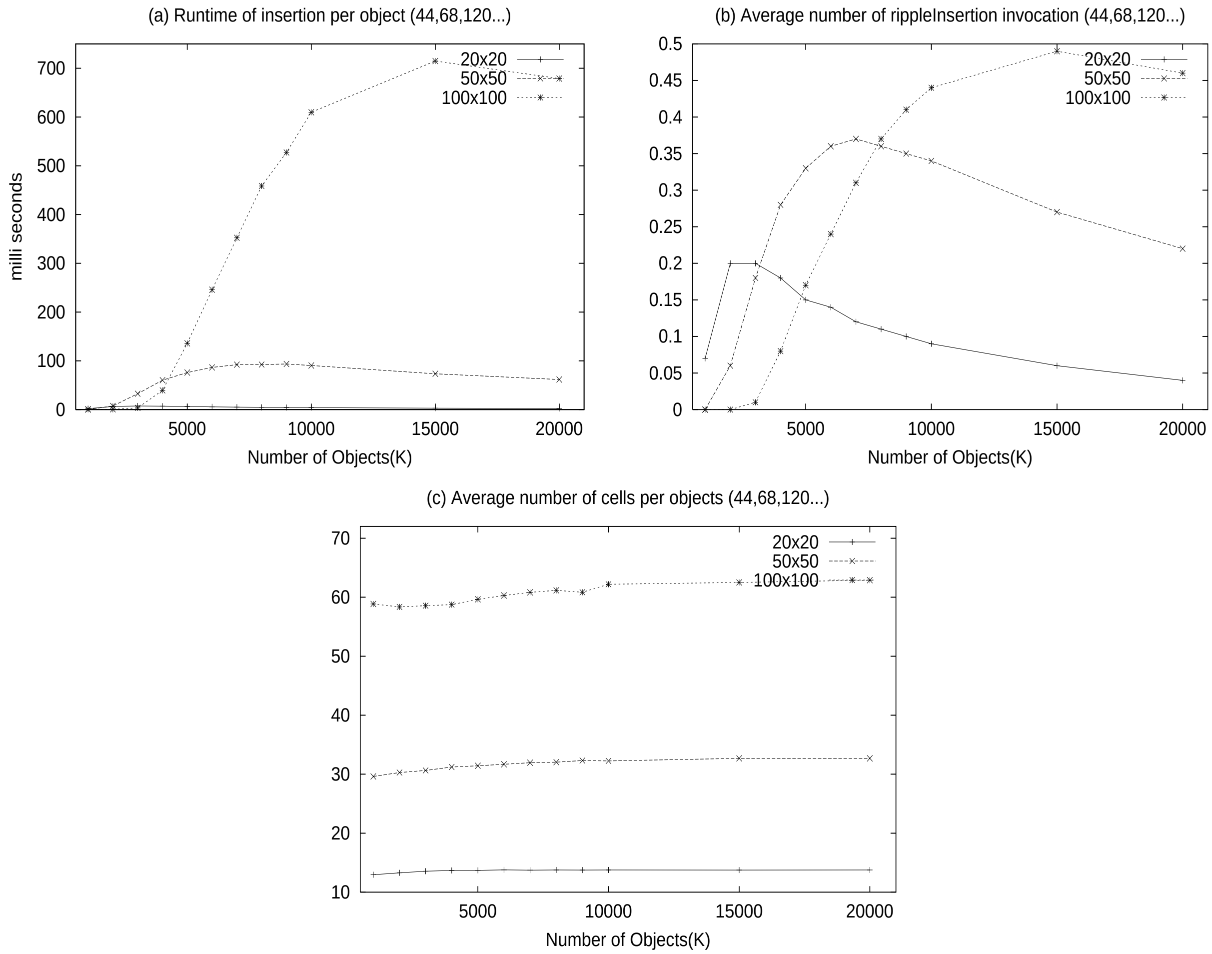


Figure 9: Fixed Thresholds : $(44, 68, 88, 120, 132)$

because the number of the method calls increases but also because when the RippleInsertion method was invoked, it has much more cells to process than with the grid size 20×20 or 50×50. In Figure 9(c), we compare the average number of cells that a moving object intersect with. They all maintain about 60 % of their grid size. This is due to the fact that we generate moving objects randomly and uniformly, as a result the average trip distances turns out to be similar in any cases.

From the experimentation, we could conclude that the grid size 20×20 is too small to handle many objects. Since the system with the grid size 100×100 takes about 700 milli seconds to insert a moving object at some cases, 100×100 is too big to be practical if we want to use it to monitor more than 15000 objects. Unlike the grid size, there is not much we can do about the thresholds size, which depends on the section of the road the system intends to monitor and manage. In the next section, we study the performance of various range query algorithms and a k nearest neighbor algorithm assuming that the grid size is 50×50 with the thresholds set to $4L = (44, 68, 88, 120, 132)$.

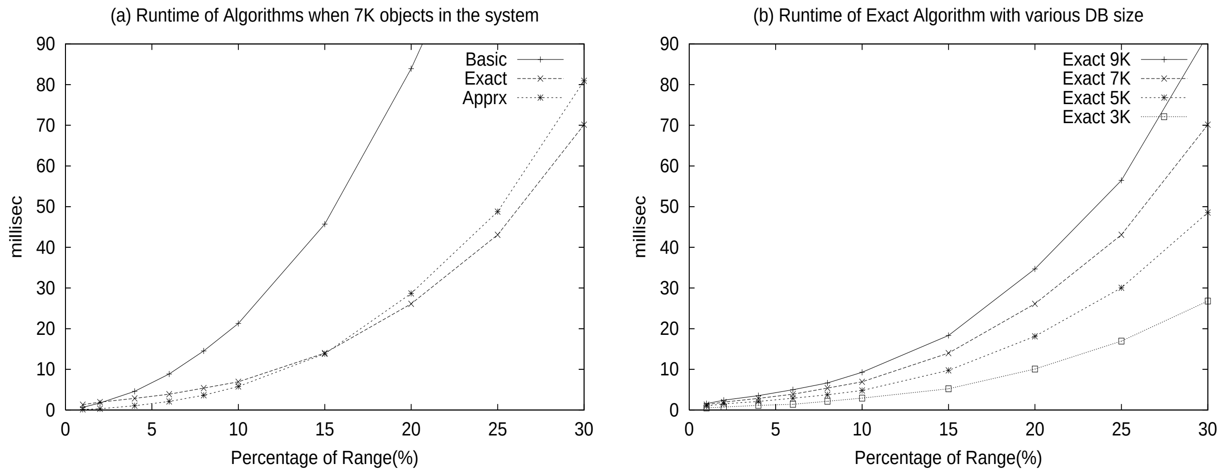


Figure 10: Various Range Query Algorithms

4.3 Range Query Algorithms and a k Nearest Neighbor Algorithm

In this section, we compare three kinds of range query algorithms and report the performance study of those algorithms, which shows quite encouraging results. The first algorithm is the approximate algorithm, which returns all the moving objects in the cells that are intersected with a range R. We expect this one to be fast because there is no filtering involved, it just merges the IDs in the cells which store them in a skip list [9]. The second one is the basic algorithm that works the same as the approximate algorithm except that it filters out the moving objects which do not intersect with R. The third and the last one is the exact algorithm we described in Section 3.1. We first explain the experimental settings and then the results.

We assume that the grid size is 50×50 and the threshold is $4L$. After initializing the system, we insert $N(= 3000, 5000, 7000, 9000)$ objects, and execute 300 range queries. In each range query, we generate a random range R ranging from 1% to 30% of the size of row and column of the grid. For example, if the grid cover 50 miles for 50 minutes, a 10% range would mean the range of 5 miles for 5 minutes. We then reinitialize the system, repeat these steps 10 times, and take the average.

Figure 10(a) shows the running time of three algorithms for various ranges. We observed the same trend which was not affected by the number of objects in the system. Hence, we analyze only the case when the system contains 7000 objects. Obviously, as the percentage increases, the running time also increases because it needs to retrieve more objects. Interestingly enough, however, the basic algorithm outperforms the exact algorithm in 1% and 2% range query. The reason for this is due to the fact that the exact algorithm keeps two temporary lists T for answers found out so far and V for objects already checked and found out not to be intersected with R. Accessing the two lists before calling *isIntersected* method and inserting a moving object into either of the lists at the end of the intersection checking cause some overhead on the overall performance of the query processing when the number of objects to be processed is small. However, this overhead is amortized when the size of a range query is larger than 2%. In fact, the exact algorithm is so efficient that it outperforms the approximate algorithm when the size of a range query is greater than 15%. The performance gap diverges as the size of a range query grows. In Figure 10(b), we show the result of running the exact algorithm with various N, the number of objects in the system. The running time increases as the percentage of the range increases and also as N increases. However, the growth rate of the running

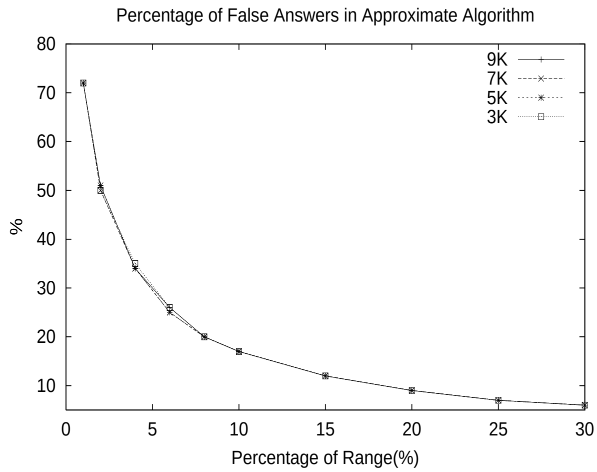


Figure 11: Overhead of Approximate Algorithm

time is almost linear with respect to N.

The approximate algorithm is about 8 times faster than the exact algorithm when querying 1% range with $N = 7000$, but the factor decreases to about 2 when querying over 6% range. However, as the size of a range grows, the performance gap is being reduced and when querying greater than 15% range, the exact algorithm begins to outperform the approximate algorithm and doing better and better thereafter. To determine which algorithm to use, we need to consider the accuracy of the candidate algorithms as well as the speed. Figure 11 shows the percentage of false answers that the approximate algorithm returns for various size of range queries. The trend is very much the same independent of how many objects in the system. For a 1% range query, 72% of the answer set that the approximate algorithm returns are false answers. In other words, it returns 2.57 times more false answers than true answers. The percentage of false answers decrease to 6% for a 30% range query.

Figure 10 and 11 reveal that there are tradeoffs between the performance and the accuracy. The approximate algorithm gives the best performance when the size of a range is smaller than 15% of the domain space but it returns relatively large amount of false answers. Between two algorithms that return actual answers, if the size of a range is small (1 or 2%), the basic algorithm outperforms the exact algorithm. Otherwise, the exact algorithm should be used. As the size of a range grows to 15%, the performance difference between the approximate and the exact algorithm is rapidly diminished while the percentage of false answers the approximate algorithm returns is still over 12%. Moreover, the time difference between the two algorithms is less than 2 milli seconds throughout the experiment. In the meantime, if the size is greater than 15%, the performance difference is even reversed while the approximate algorithm still returns 6% false answers. Therefore, we can conclude that the exact algorithm should be used for more than 2% range queries and the basic algorithm should be used for less than 2% range queries unless used by an application that rather tolerates 72% false answers than waiting 2 more milli seconds.

We now explain the result of performance study of the k nearest neighbor (kNN) algorithm we proposed in Section 3.2. After we execute range queries, we execute kNN queries 300 times for each k from 1 to 50. Figure 12(a) shows the average running time of the kNN queries over the various number of objects in the system. This figure shows that a query takes more time as the system contains more objects, and also takes more time as k increases since it obviously needs to retrieve

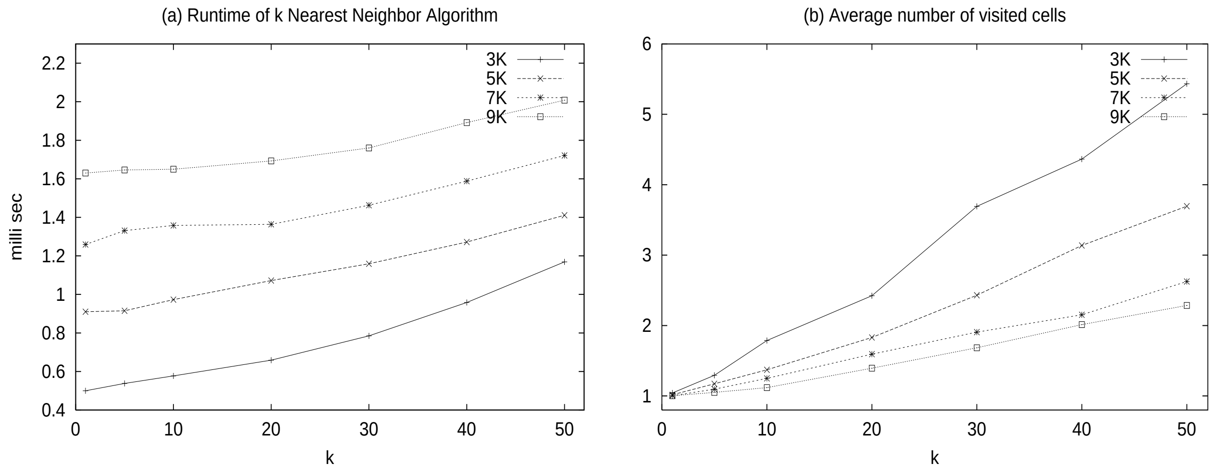


Figure 12: Runtime of k Nearest Neighbor Algorithm

more objects. All of the test cases, however, take less than 2 milli seconds. In Figure 12(b), we show the average number of cells the kNN algorithm visits before it terminates and returns the k nearest neighbors. It visits more cells as k increases. And if the system contains less number of objects, it needs to visit more cells. In other words, as the system gets sparser, the kNN algorithm visits more cells.

5 Conclusions and Future Work

We proposed an efficient range query algorithm part of which is a filtering algorithm *isIntersected* which determines if $m \triangleright R$ with only necessary computations. We also proposed a kNN query algorithm which takes less than 2 milli seconds in most test cases. From the performance study, we concluded that the grid size 20×20 is too small to handle many objects and 100×100 is too big to be practical if we want to use it to monitor more than 15000 objects. After that, we fixed the grid size to 50×50 and the thresholds $4L$ of the system, and study the three range query algorithms. Among them, the basic algorithm outperforms the exact algorithm when a range is small. Despite the filtering step, the exact algorithm outperforms the approximate algorithm when the size of a range is large and even if the size is small, the performance difference between the two algorithm is less than 2 milli seconds. Future work includes extending this system to 2-dimensional space so that we can answer time-dependent shortest path problems. Improvement of the insertion algorithm, especially RippleInsertion method, is also part of future work.

References

[1] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful. In *Int. Conf. on Database Theory*, 1999.

[2] H. Chon, D. Agrawal, and A. E. Abbadi. Using Space-Time Grid for Efficient Management of Moving Objects. In *MobiDE*, May 2001.

[3] H. Chon, D. Agrawal, and A. E.Abbadi. Storage and Retrieval of Moving Objects. In *Proceedings of the Int. Conf. on Mobile Data Management*, 2001.

[4] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. E. Abbadi. Approximate nearest neighbor searching in multimedia databases. In *Proc of Int. Conf. on Data Engineering*, 2001.

[5] E. Hoel and H. Samet. Efficient Processing of Spatial Queries in Line Segment Databases. In *Advances in Spatial Database - 2nd Symposium*, 1991.

[6] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Moving Objects. In *Proceedings of ACM Symp. on Principles of Database Systems*, 1999.

[7] R. C. Nelson and H. Samet. A Consistent Hierarchical Representation for Vector Data. In *ACM SIGGRAPH*, Aug. 1986.

[8] D. Pfoser, C. Jensen, and Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. In *Proceedings of the Int. Conf. on Very Large Data Bases*, 2000.

[9] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of ACM*, 33(6), 1990.

[10] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1995.

[11] S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2000.

[12] D. Schrank and T. Lomax. The 2001 Urban Mobility Report. Technical report, Texas Transportation Institute, 2001.

[13] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proceedings of the Int. Conf. on Data Engineering*, 1997.

[14] J. Tayeb, O. Ulusoy, and O. Wolfson. A Quadtree Based Dynamic Attribute Indexing Method. In *Proceedings of ACM Symp. on Principles of Database Systems*, 1998.