Nearest Neighbor Queries in a Mobile Environment

George Kollios[1], Dimitrios Gunopulos[2], and Vassilis J. Tsotras[2]

[1] Polytechnic University
Dept. of Computer and Information Science
Six MetroTech Center
Brooklyn, NY 11201-3840, USA
gkollios@db.poly.edu
[2] University of California, Riverside
Department of Computer Science and Engineering
Riverside, CA 92521, USA
dg@cs.ucr.edu, tsotras@cs.ucr.edu

Abstract. Nearest neighbor queries have received much interest in recent years due to their increased importance in advanced database applications. However, past work has addressed such queries in a static setting. In this paper we consider instead a dynamic setting where data objects move continuously. Such a mobile spatiotemporal environment is motivated by real life applications in traffic management, intelligent navigation and cellular communication systems. We consider two versions of nearest neighbor queries depending on whether the temporal predicate is a single time instant or an interval. For example: "find the closest object to a given object o after 10 minutes from now", or, "find the object that will be the closest to object o between 10 and 15 minutes from now". Since data objects move continuously it is inefficient to update the database about their position at each time instant. Instead our approach is to employ methods that store the motion function of each object and answer nearest neighbor queries by efficiently searching through these methods.

1 Introduction

A spatiotemporal database system manages data whose geometry changes over time. There are many applications that create such data, including global change (as in climate or land cover changes), transportation (traffic surveillance data, intelligent transportation systems), social (demographic, health, etc.), and multimedia (animated movies) applications. In general one could consider two spatial attributes of spatiotemporal objects which are time dependent, namely: position (i.e., the object's location inside some reference space) and extent (i.e., the area or volume the object occupies in the reference space)[8]. Depending on the application, one or both spatial attributes may change over time. Examples include: an airplane flying around the globe, a car traveling on a highway, the land covered by a forest as it grows/shrinks over time, or an object that concurrently moves and changes its size in an animated movie. For the purposes of this paper we concentrate on applications with objects which change position over time but whose extent remains unchanged. Hence for our purposes we represent such objects as points moving in some reference space ("mobile points").

The usual assumption in traditional database management systems is that data stored in the database remains constant until explicitly changed by an update. For example, if a price field is $5, it remains $5 until explicitly updated. This model is appropriate when data changes in discrete steps, but it is inefficient for applications with continuously changing data [20]. Consider for example a database keeping the position of mobile objects (like automobiles). The primary goal of this database is to correctly represent reality as objects move. Since objects move continuously updating the database about each object's position at each unit of time is clearly an inefficient and infeasible solution due to the prohibitively large update overhead. Updating the database only at few, representative time instants limits query accuracy.

A better approach is to abstract each object's position as a function of time $f(t)$, and update the database only when the parameters of f change (for example when the speed or the direction of a car changes). Using $f(t)$ the database can then compute the position of the mobile object at any time in the future (under the current knowledge about the motion characteristics of the database objects). Storing the motion function minimizes the update overhead, but it also introduces many novel problems since the database is not directly storing data values but functions to compute these values. Motion databases have recently attracted the interest of the database community. There is already a GIS system [2] that supports tracking and querying mobile objects. In the research front, [20–22, 8, 6] present Spatio-Temporal models and languages for querying the locations of such objects. Recently, [16] presents access methods for indexing mobile objects; the focus is on spatiotemporal range queries for objects moving in one and two dimensions. For example: "find the objects that will be inside a given query region P after 10 minutes from now".

In this paper we concentrate instead on answering nearest neighbor queries among the future locations of mobile objects. An example of such a spatio-temporal query is: "Report the object that will be the closest to an object o after 10 minutes from now". Since object o moves and it's motion information is known, the above query is equivalent to finding the object (except from object o) which will be the closest to position P after 10 minutes from now, where P is the position o will be in 10 minutes. Note that the answer to this query is tentative in the sense that it is computed based on the current knowledge stored in the database about the mobile objects' future motion functions. If this knowledge changes, the same query would produce a different answer.

We also examine nearest neighbor queries where instead of a time instant, an interval predicate in future is given as in: "Report the object that will be the closest to object o between 10 and 20 minutes from now". We will answer this query by reducing it to a combination of range queries and the above simple nearest neighbor queries.

We are not interested in providing new indices specifically designed for neighbor queries, as such would be of limited usefulness. In practice, neighbor queries are addressed by using traditional selection based indices (R-trees etc.) and modifying the search algorithm so that neighbor queries are also answered. We will follow the same approach here, thus utilizing the indexing techniques we have proposed in [16] and adapting them for neighbor searching.

As the number of mobile objects in the applications we consider (traffic monitoring, mobile communications, etc.) can be rather large we are interested in external memory solutions. While in general an object could move anywhere in the 3-dimensional space using some rather complex motion, we limit our treatment to objects moving in 1-dimensional space (a line) and whose location is described by a linear function of time. There is a strong motivation for such an approach based on the real-world applications we have in mind: straight lines are usually the faster way to get from one point to another; cars move in networks of highways which can be approximated by connected straight line segments on a plane; this is also true for routes taken by airplanes or ships. In addition, solving this simpler problem may provide intuition for addressing the more difficult nearest neighbor query among objects moving in 2- or 3-dimensional space.

2 Problem Description

We consider a database that keeps track of mobile objects moving on a finite line segment. We model the objects as points that move with a constant velocity starting from a specific location at a specific time instant. Using this information we can compute the location of an object at any time in the future for as long as its movement characteristics remain the same. Thus, an object started from location y_0 at time t_0 with velocity v (v can be positive or negative) will be in location $y_0 + v(t - t_0)$ at time $t > t_0$. Objects are responsible to update their motion information, every time when the speed or direction changes. When an object has reached the line segment limits, it has to issue an update. Such update can be either a deletion (the object is removed from the collection) or a reflection (direction and possibly speed change). Finally, we allow to insert a new object or to delete an existing one anywhere in the line segment, eg. the system is dynamic.

We first examine nearest neighbor queries described by a tuple (y_q, t_q) as in: "Report the object that will be closer to the point y_q at the time instant t_q (where $t_{now} \leq t_q$)". Section 3 describes two geometric representations of the problem, the primal plane (where object trajectories are represented as long lines in the time-position plane) and the dual plane (where a trajectory becomes a point). Two dual plane transformations are examined (Hough-X and Hough-Y). We present efficient algorithms for answering near neighbor queries in the primal and dual planes in section 4. A performance study appears in section 5. In section 6 we examine the nearest neighbor query with a time interval predicate ("Report the object that will be closer to the point y_q during the time interval $[t_{1q}, t_{2q}]$ (where $t_{now} \leq t_{1q}$)"). We also discuss how our 1-dimensional results can be extended to apply on a limited 2-dimensional case, where the objects are restricted to moving in a given collection of line segments (like roads comprising a highway system). We call this the 1.5 dimensional case and discuss it in section 7. Finally, related work appears in section 8, while section 9 summarizes our findings and presents problems for future research.

3 Geometric representations

First, we partition the mobile objects into two categories, the objects with low speed $v \approx 0$ and the objects with speed between a minimum v_{min} and maximum speed v_{max}. We consider here the "moving" objects, eg. the objects with speed greater than v_{min}. The case of static or almost static objects can be solved using traditional approaches and we don't discuss this issue further.

The problem is to index the mobile objects in order to efficiently answer nearest neighbor queries over their locations into the future. The location of each object is described as a linear function of time, namely the location $y_i(t)$ of the object o_i at time t is equal to $v_i(t - t_{i_0}) + y_{i_0}$, where v_i is the velocity of the object and y_{i_0} is its location at t_{i_0}. We assume that the objects move on the $y-$axis between 0 and y_{max} and that an object can update its motion information whenever it changes. We treat an update as a deletion of the old information and an insertion of the new one. As in [16] we describe two geometric representations of the problem.

3.1 Space-time representation

In this representation we plot the trajectories of the mobile objects as lines in the time-location (t, y) plane. The equation of each line is $y(t) = vt + a$ where v is the slope (the velocity in our case) and a is the intercept, that can be computed by the motion information. In fact a trajectory is not a line but a semi-line starting from the point (y_i, t_i). However since we ask queries for the present or for the future, assuming that the trajectory is a line does not affect the correctness of the answer. Figure 1 shows a number of trajectories in the plane.

The query is expressed as a point (y_q, t_q) in the 2-dimensional space. The answer is the object that corresponds to the line that is closer to this point at the time t_q. So we have to consider the distance of the trajectories to the query point along the line $t = t_q$. For example in Figure 1 the answer to the nearest neighbor query is the object o_6.

3.2 The dual space-time representation.

Duality is a transform frequently used in the computational geometry literature; in general it maps a hyper-plane h from R^d to a point in R^d and vice-versa. The duality transform is useful because it allows to formulate a problem in a more intuitive manner.

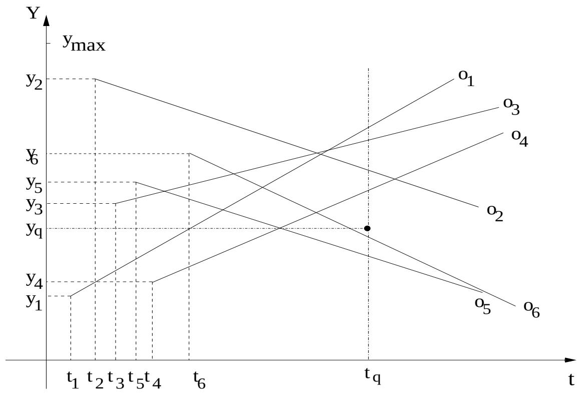

Fig. 1. Trajectories and query in (t, y) plane.

In our case we can map a line from the *primal* plane (t, y) to a point in the *dual* plane. In Figure 2, line l and point p are transformed to point l^* and line p^*. There is no unique duality transform, but a class of transforms with similar properties. Sometimes one transform is more convenient than another.

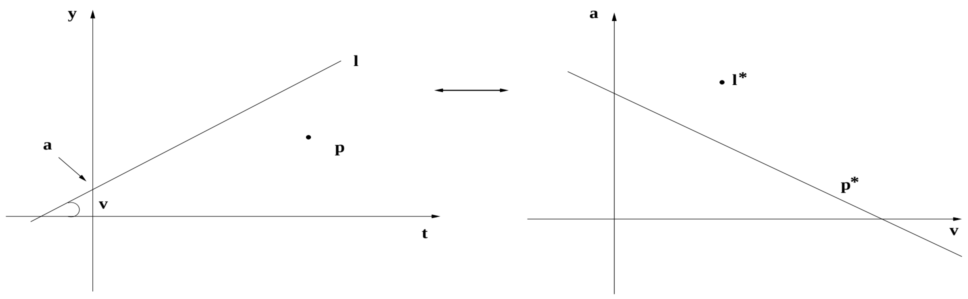

Fig. 2. Duality transform of a line and a point.

Consider a dual plane where one axis represents the slope of an object's trajectory and the other axis its y-intercept[1]. Thus the line with equation $y(t) = vt + a$ is represented by the point (v, a) in the dual space (this is called the Hough-X transform in [14]). While the values of v are between $-v_{max}$ and v_{max}, the values of the intercept are depended on the current time. If the current time is t_{now} then the range for a is $[-v_{max} \times t_{now}, y_{max} + v_{max} \times t_{now}]$.

The query is transformed to a line in the dual space. This line is the dual of the query point (t_q, y_q), thus it has the equation: Q_l: $a = -vt_q + y_q$ (Figure 3). Next we show the following lemma:

[1] The y-intercept is defined as the point where a given line intersects the y-axis. Similarly t-intercept is the point where a line intersect the t-axis.

Lemma 1. *The nearest neighbor to the query point (t_q, y_q), is the object whose dual point is closest to the line that is the dual of the query point.*

Proof. Assume a query point (s_1, t_1) and an object with trajectory $y = v_1 x + a_1$. The distance of this object to the query point at the time t_1 is $D = |v_1 t_1 + a_1 - s_1|$, and it is computed along the line $t = t_1$. In the dual plane the same distance has to be computed along the line $v = v_1$. If the Euclidean distance of the object to the query line in the dual space is d, then it is easy to show that $D = d\cos(\theta)$, where θ is the slope of the query line. $\square$

Thus we can use the Euclidean distance to compute the nearest neighbor in the dual space, although this is not true for the primal space.

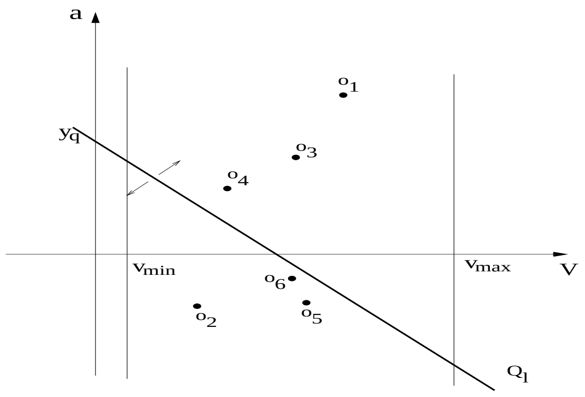

Fig. 3. Data objects and query in the Hough-X dual plane.

Note that the dual representation has the problem that the values of the intercept are unbounded. A simple solution to this problem is presented in [16].

To solve the problem we use our assumption that when an object crosses a border it issues an update (i.e. it is deleted or reflected). Combining this assumption with the minimal speed, we can assure that all objects have updated their motion information at least once during the last T_{period} time instants, where $T_{period} = \frac{y_{max}}{v_{min}}$. We can then use two distinct index structures. The first index stores all objects that have issued their last update in the period $[0, T_{period}]$. The second index stores objects that have issued their last update in the interval $[T_{period}, 2T_{period}]$. Each object is stored only once, either in the first or in the second index. Before time T_{period}, all objects are stored in the first index. However, every object that issues an update after T_{period} is deleted from the first index and it is inserted in the second index. The intercept of the first index is computed by using the line $t = 0$ and for the second index using the line $t = T_{period}$. Thus we are sure that the intercept will always have values between 0 and $v_{max} \times T_{period}$. To query the database we use both indices. After time $2T_{period}$ we know that the first index is empty and all objects are stored in the second index, since every object have issued at least one update from T_{period} to $2T_{period}$. At that time we remove the empty index and we initiate a new one with time period $[2T_{period}, 3T_{period}]$. We continue in the same way and every T_{period} time instants we initiate a new index and we remove an empty one. Using this method the intercept is bounded while the performance of the index structures remain asymptotically the same as if we had only one structure.

Another way to represent a line $y = vt + a$, is to write the equation as $t = \frac{1}{v}y - \frac{a}{v}$. Then we can map this line to a point in the dual plane with coordinates $n = \frac{1}{v}$ and $b = -\frac{a}{v}$ (Hough-Y in [14]).

4 Algorithms to answer nearest neighbor queries

An obvious approach would be to check all the objects and return the closest to the query point. However this method is inefficient since the number of mobile objects can be large. Hence, we discuss methods that avoid a linear scan on the data set by using index structures. For each representation we present possible indices and methods to use these indices for nearest neighbor search.

4.1 Using traditional methods

In the space-time representation we can use a Spatial Access Method (SAM) to index the lines or line segments. Possible methods include R-tree and Quadtree based indices. Then we can use the algorithms proposed by Hjaltason and Samet [12] or by Roussopoulos at. al. [17] to find the lines that are closer to the query point. These algorithms work for any hierarchical index structure which uses recursive partitioning. Note that in [5] it was shown that the algorithm in [12] is the optimal algorithm for nearest neighbor search, in the sense that we access the smallest number of pages to answer a nearest neighbor query, given an underlying index structure. The main idea of these algorithms is to traverse the hierarchical structure in a top-down fashion and keep for every visited partition a list of subpartitions ordered by their distance to the query point. We then visit the subpartitions in sorted order. We can prune some subpartitions, when the minimum distance of these partitions to the query point is larger than the distance of an already found object. In [12], a global priority queue is used to keep both the subpartitions and the objects found up to know in a sorted order. It is obvious that we can use the above algorithms to process our query, since these algorithms work for any type of objects. The only difference is the way we compute the distance of the objects to query point. We don't use the Euclidean distance but the distance along the line $t = t_q$.

The main problem of this approach is that there are no efficient index structures for indexing lines or very large line segments. In case of hierarchical methods with overlapping partitions, like R-tree[11] or R*-tree[4], the overlapping is extensive, resulting in a very poor query performance. On the other hand, methods that partition the data space into disjoin cells and use clipping(like PMR-quadtree[18] or R+-tree[19]), will have a very high space consumption (in our case probably superlinear).

4.2 Using *kd*-tree methods in the dual

There is a large number of access methods that have been proposed to index point data[10]. All these structures were designed to address *orthogonal* range queries, eg. queries expressed as multidimensional hyper-rectangles. However, most of them can be used to answer nearest neighbor queries as well.

We use this approach to answer the nearest neighbor query in the dual Hough-X space (Figure 3). An index structure based on *kd*-trees is used to store the dual points. In our experiments we used the hB^{Π}-tree [9]. Next a traversal algorithm similar to the algorithm presented in [12] is used to find the answer. The only difference here is that we compute the distance of the regions or the data points to the line that represents the query in the dual space and not to a point.

4.3 Using *B*+-trees in the dual

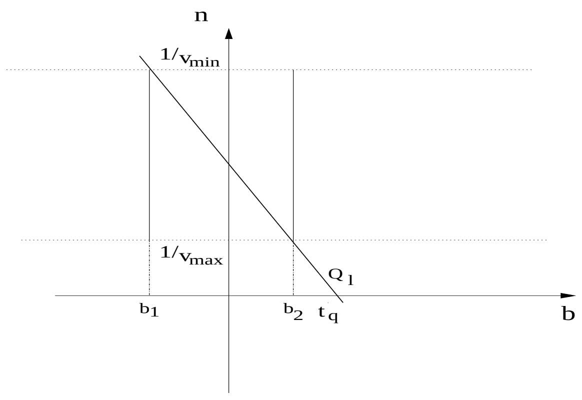

Fig. 4. Query in the Hough-Y dual plane.

A different approach is based on the *query approximation* method presented in [16]. Consider the Hough-Y dual plane and a query line in that plane with equation: $b = -y_q n + t_q$. We assume that the b coordinate is computed using a general horizontal line with equation $y = y_r$, where y_r is a value between 0 and y_{max}(note that the t-axis is the line $y = 0$). We can use an one dimensional data structure (B+-tree) to index the dual points using only the b coordinate. The query line is mapped to an 1-dimensional range on this structure, for example in the Figure 4 the range is (b_1, b_2) where $b_1 = t_q - \frac{y_q - y_r}{v_{min}}$ and $b_2 = t_q - \frac{y_q - y_r}{v_{max}}$. Now we can use this structure to answer the nearest neighbor query. Note however that the size of the range query is proportional to $|y_q - y_r|$. So, we can decrease the size of this range if we keep c index structures at equidistant y_r's (see Figure 5). While we can still get the correct answer with $c = 1$ using some limited replication (small $c > 1$) we get better performance.

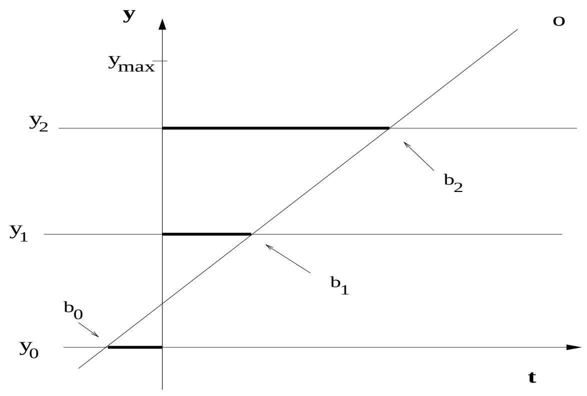

Fig. 5. Using three indices to store a moving object.

All c indices contain the same information about the objects, but use different y_r's. The i-th index stores the b coordinates of the data points using $y = \frac{y_{max}}{c} \times i$, $i = 0, .., c-1$. However the index that is closer to the query point has a smaller range to search, and therefore we expect that the query time to process the nearest neighbor query will be smaller. In Figure 6 we describe an algorithm that is used to answer the nearest neighbor query using the above one dimensional structures.

1. Choose the structure that is the closest to the query point.
2. Find the range (b_1, b_2) where the query is mapped. Find the closest object to the query point y_q inside this range. This is the nearest neighbor (NN) so far.
3. Start visiting the previous and the next pages and update the NN if necessary. Stop condition: check if it is possible to find a nearest point than the current NN in the next pages using v_{min} and in the previous pages using the v_{max}. Stop when this is not possible.

Fig. 6. Algorithm to compute the nearest neighbor for mobile objects

5 A Performance Study

We present initial results for the nearest neighbor query, comparing the 1-dimensional structures, the *kd*-tree method and a traditional R-tree based approach. First we describe the way experimental data is generated. At time $t = 0$ we generated the initial locations of N mobile points uniformly distributed on the terrain $[0, 1000]$. We varied N from $100k$ to $500k$. The speeds were generated uniformly from $v_{min} = 0.16$ to $v_{max} = 1.66$ and the direction randomly positive or negative.[2] Then all points start moving. When a point reaches a border simply

[2] Note that 0.16 miles/min is equal to 10 miles/hour and 1.66 miles/min is equal to 100 miles/hour.

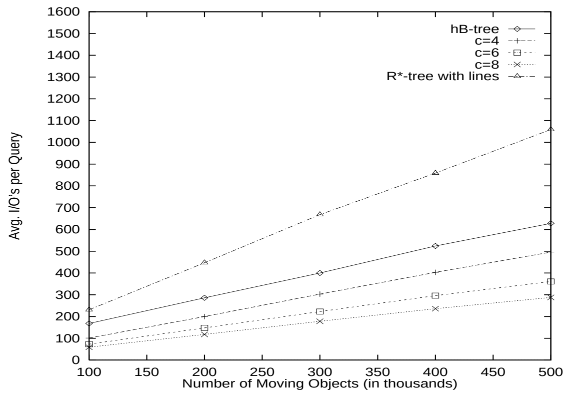

Fig. 7. Query Performance for Nearest Neighbor Queries.

it changes its direction. At each time instant we choose 200 objects randomly and we randomly change their speed and/or direction. We generate 10 different time instants that represent the times when queries are executed. At each such time instant we execute 200 nearest neighbor queries, where the y-coordinate is chosen uniformly between 0 and 1000 and the time instant between t_{now} and $t_{now} + 60$. We run the above scenario using a particular access method for 2000 time instants.

To verify that indexing mobile objects as line segments is not efficient, we stored the trajectories in an R*-tree. We fixed the page size to 4096 bytes. All methods had page capacity equal to 340 except the R-tree where the page capacity was 204. We consider a simple buffering scheme for the results we present here. For each tree we buffer the path from the root to a leaf node, thus the buffer size is only 3 or 4 pages. For the queries we always clear the buffer pool before we run a query. An update is performed when the motion information of an object changes.

In Figure 7 we present the results for the average number of I/O's per nearest neighbor query. The approximation method used $c = 4, 6$ and 8 B+-trees. As anticipated, the line segments method with R*-trees has the worst performance. Also, the approximation method outperforms the hB^{Π}-tree.

In Figures 8 and 9 we plot the space consumption and the average number of I/O's per update respectively. We did not report the update performance for the R*-tree method because it was very high (more than 90 I/O's per update). The update and space performance of the hB^{Π}-tree is better than the other methods since its objects are stored only once and better clustered than the R*-tree. The update performance of the hB^{Π}-tree and the approximation approach remain constant for different number of mobile objects. The space of all methods is linear to the number of objects. The approximation approach uses more space

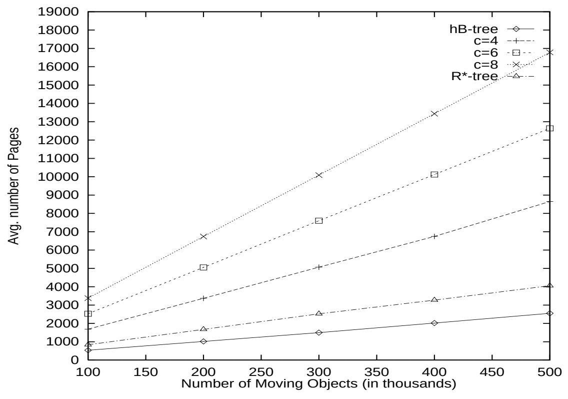

Fig. 8. Space Consumption.

due to the use of c observation indices. There is a tradeoff between c and the query/update performance.

6 Nearest Neighbor Queries with time interval predicate

Another interesting nearest neighbor query is a query where a time interval is given instead of a time point. This time interval and the trajectory of the query object create a line segment in the space-time plane. The answer to the query is the object that comes closest to the query object during this time interval, or equivalently to the query line segment. Note that this problem is important only in a spatiotemporal environment. For example in Figure 10, o_4 is the query object and $[t_{1q}, t_{2q}]$ the time interval. The answer to this query are the objects o_5 and o_6, since at some point during the time interval, these objects have zero distance from the query object. However if we ask for the three nearest neighbors, then the objects o_7 belongs also to the result. A method to answer the query is to find first the lines that intersect the query line segment. If there is no such line, then we run two nearest neighbor queries, one on each of the two ends of the line segment and keep the best answer. This is correct because the trajectories of the mobile objects are straight lines. In particular we can show that the time where an object is closer to the query object, has to be one of the time instants t_{1q} or t_{2q}, if the trajectories of the mobile objects have no common point during the query interval $[t_{1q}, t_{2q}]$.

In the primal plane we can modify the existing algorithms for query points and find the lines closest to the line segment. However the performance of these methods will be at least as bad as the case for query point.

In the dual plane we can use the method described above. The query to find the lines that intersect the query line segment is a non-orthogonal range query. To answer this query we can use the methods proposed in [16]. If no answer is

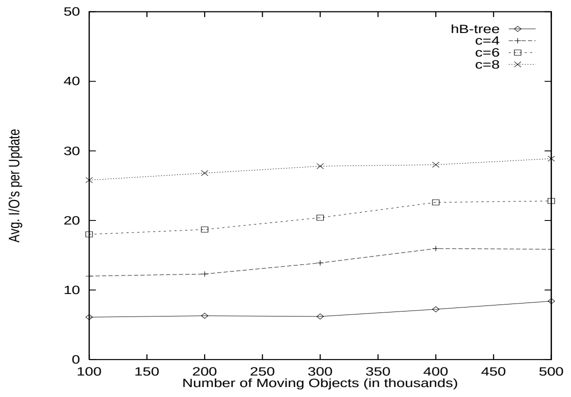

Fig. 9. Update Performance.

found, the we run two nearest neighbor queries for the time instants t_1 and t_2 and keep the closest object.

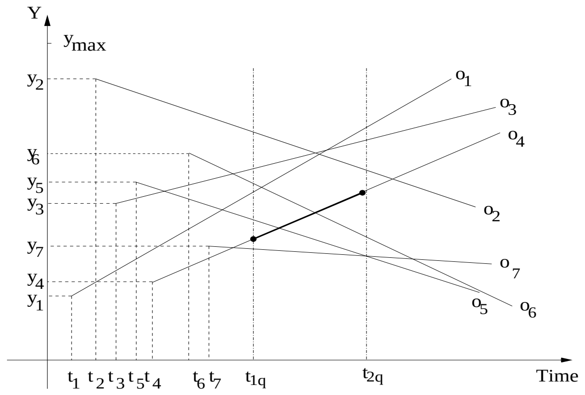

Fig. 10. Data objects and query with time interval predicate.

7 The 1.5-dimensional case

There is an interesting restricted version of the general 2-dimensional problem where our 1-dimensional algorithms can be easily extended. We consider objects moving in the plane but their movement is restricted on using a given collection of line segments (routes, roads) on the finite terrain. Due to its restriction, we call this case the 1.5-dimensional problem. There is a strong motivation for such an environment: in many applications objects (cars, airplanes etc.) move on a network of specific routes (highways, airways).

The 1.5-dimensional problem can be reduced to a number of 1-dimensional problems. In particular, we propose representing each predefined route as a sequence of connected (straight) line segments. The positions of these line segments on the terrain are indexed by a standard spatial access method (SAM). Given that the number of routes is usually much smaller than the number of objects moving on them, and that each route can be approximated by a small number of straight lines, maintaining such an index is straightforward (in addition, new routes are added rather infrequently.)

Indexing the objects moving on a given route is an 1-dimensional model and will use techniques from the previous section. Given a Nearest Neighbor query for a given two-dimensional point, we identify first the route that passes closest to the point. We could use the spatial access method and traditional NN techniques [12, 17] for finding this route (line segment) as the collection of routes is rather static. Once the route that passes closest to the query point is identified, we find the Nearest Neighbor of the query point among the objects on that route. This object must also be the nearest point to the orthogonal projection of the query point on the route line. To find it, we take the orthogonal projection of the query point onto the route, and solve the one dimensional problem of finding the Nearest Neighbor of the projection among the points on the route. Let d be the distance of the nearest neighbor on the closest route and the query point.

Thus we obtain an upper bound, d, on the distance of the true nearest neighbor point to the query. Clearly we have to consider only the points that are on routes that are closer than d to the query time. To find those routes we perform a range query on the SAM data structure we use to keep all routes. This will return all routes with L_{inf} distance from the query point less than d. We sort the routes according to distance from the query point and find the nearest neighbor to the query point on each route, updating the global nearest point when a closer point is found. We terminate this process if the distance of the next route to be considered is larger than the nearest point found so far.

8 Related Work

There is a lot of work on the Nearest Neighbor problem in spatial domains. However this work has concentrated on the static case, where points or objects remain in a fixed location.

Roussopoulos et al [17] give an R-tree based algorithm to find the point closest to the query point. Hjaltason and Samet [12] give a different algorithm based on Quadtrees. These algorithms can be extended to handle moving points, where the trajectory of a point is modeled by a line, but this straightforward modification does not work very well, as our experimental results show.

In both of the previous approaches the points are stored in external memory. Recent work on the Nearest Neighbor problem is mainly on main memory algorithms and for the high-dimensional setting. Both the exact nearest neighbor problem [15] and the approximate nearest neighbor problem [13] [3] have been considered. These algorithms however have either query times that are exponential to the dimensionality of the space, or require more than linear space. In addition these approaches cannot be easily modified to find the closest point to a line (instead of a point).

We also note that computational geometry work on arrangements is very relevant in our setting. The trajectories of the n points in the time-space plane define an arrangement, that is a partition of the plane into convex regions. A point y_q at time t_q will be in the interior of one of these regions. To find the line that is closest to it, we have to find the lines that form the boundary of the convex region the point is in, and examine them only. It can be shown that this can be done in $O(\log n)$ time. However the arrangement of the lines takes $O(n^2)$ time to compute, in the main memory model, and also requires $O(n^2)$ space to keep [1].

9 Summary and Future Work

In this paper we present efficient external memory data structures to solve the nearest neighbor problem in the setting of points that move with constant velocity. The queries we want to answer are of the form: given a point y_q and a time instant t_q, find the point that is closest to y_q at the time t_q, or, given a (moving) point y_q, and a time interval, find the point that come closest to y_q during this time interval. We consider mainly the one dimensional case, and extend out results to the 1.5-dimensional case.

We plan to work on the nearest neighbor problem when the points move in two dimensional space. The main problem in two dimensions is that the trajectories of the points are lines in three-dimensional space, and therefore taking the duality transformation does not help. Since we are using the Euclidean norm, we cannot solve the problem separately in the x-axis and y-axis and combine the results either. Many points may be closer to the query point in the x-coordinate or the y-coordinate than the the nearest point, and yet have larger Euclidean distances.

Another area for future work is giving an algorithm for the join operation, that is, find for all points the nearest neighbor at a query time.

10 Acknowledgement

We were introduced to the problems of indexing moving points by Ouri Wolfson at the 1997 Dagstuhl Seminar on Temporal Databases. The authors would like to thank O. Wolfson, C. Faloutsos and V.S. Subrahmanian for many helpful discussions. We would also like to thank G. Evangelidis for the hB-tree code.

References

1. P. Agarwal and M. Sharir. Arrangements and their applications. in *Handbook of Computational Geometry*, (J. Sack, ed.), North Holand, Amsterdam.

2. ArcView GIS. ArcView Tracking Analyst. 1998.
3. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman and A. Wu. An optimal algorithm for approximate nearest neighbor searching. Journal of the ACM, to appear.
4. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method For Points and Rectangles. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, May 1990.
5. S. Berchtold, C. Bohm, D. Keim, H.-P. Kriegel. A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space. In *Proc. 16th ACM-PODS*, pp. 78-86, 1997.
6. J. Chomicki and P. Revesz. A Geometric Framework for Specifying Spatiotemporal Objects. *Proc. 6th International Workshop on Time Representation and Reasoning*, May 1999.
7. D. Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, June 1979.
8. M. Erwig, R.H. Guting, M. Schneider and M. Vazirgianis. Spatio-temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. In *Proc. of ACM GIS Symposium '98.*
9. G. Evangelidis, D. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: A Modified hB-tree Supporting Concurrency, Recovery and Node Consolidation. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995, pages 551–561.
10. V. Gaede and O. Gunther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170-231, 1998.
11. A. Guttman. R-Trees: A Dynamic Index Structure For Spatial Searching. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 47–57, Boston, June 1984.
12. G. R. Hjaltason and H. Samet. Ranking in Spatial Databases. In *Proc. 4th Int. Symp. on Spatial Databases*, pp. 83-95, 1995.
13. P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proc.30th ACM-STOC*, pp. 604-613, 1998.
14. H. V. Jagadish. On Indexing Line Segments. In *Proc. 16th International Conference on Very Large Data Bases*, pages 614–625, Brisbane, Queensland, Australia, August 1990.
15. J. Kleinberg. Two Algorithms for Nearest-Neighbor Search in High Dimensions. In *Proc. 29th ACM-STOC*, pp. 599-608, 1997.
16. G. Kollios, D. Gunopulos and V.J. Tsotras. On Indexing Mobile Objects. In *Proc. 18th ACM-PODS*, 1999.
17. N. Roussopoulos, S. Kelley, F. Vincent. Nearest Neighbor Queries. In *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, pages 71-79, June 1992.
18. H. Samet. *The Design and Analysis of Spatial Data Structures.*, Addison-Wesley, Reading, MA, 1990.
19. T. Sellis, N. Roussopoulos and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proc. 13rd International Conference on Very Large Data Bases*, pages 507-518, Brighton, England, September 1987.
20. A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. Modeling and Querying Moving Objects. In *Proc. 13th IEEE International Conference on Data Engineering*, pages 422-432, Birmingham, U.K, April 1997.

21. O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, G. Mendez. Cost and Imprecision in Modeling the Position of Moving Objects. In *Proc. 14th IEEE International Conference on Data Engineering*, pages 588-596, Orlando, Florida, February 1998.
22. O. Wolfson, B. Xu, S. Chamberlain, L. Jiang. Moving Objects Databases: Issues and Solutions In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*. Capri, Italy, July 1998.