Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects

Rimantas Benetis Christian S. Jensen Gytis Karčiauskas Simonas Šaltenis
Department of Computer Science
Aalborg University
DK-9220 Aalborg Øst, DENMARK
{rbenetis,csj,gytis,simas}@cs.auc.dk

Abstract

With the proliferation of wireless communications and the rapid advances in technologies for tracking the positions of continuously moving objects, algorithms for efficiently answering queries about large numbers of moving objects increasingly are needed. One such query is the reverse nearest neighbor (RNN) query that returns the objects that have a query object as their closest object. While algorithms have been proposed that compute RNN queries for non-moving objects, there have been no proposals for answering RNN queries for continuously moving objects. Another such query is the nearest neighbor (NN) query, which has been studied extensively and in many contexts. Like the RNN query, the NN query has not been explored for moving query and data points.

This paper proposes an algorithm for answering RNN queries for continuously moving points in the plane. As a part of the solution to this problem and as a separate contribution, an algorithm for answering NN queries for continuously moving points is also proposed. The results of performance experiments are reported.

1 Introduction

We are currently experiencing rapid developments in key technology areas that combine to promise widespread use of mobile, personal information appliances, most of which will be on-line, i.e., on the Internet. Industry analysts uniformly predict that wireless, mobile Internet terminals will outnumber the desktop computers on the Internet.

This proliferation of devices offers companies the opportunity to provide a diverse range of e-services, many of which will exploit knowledge of the user's changing location. Location awareness is enabled by a combination of political developments, e.g., the de-scrambling of the GPS signals and the US E911 mandate, and the continued advances in both infrastructure-based and handset-based positioning technologies.

The area of location-based games offers good examples of services where the positions of the mobile users play a central role. In the recently released BotFighters game, by Swedish company It's Alive, players get points for finding and "shooting"other players via their mobile phones. Only players close by can be shot. In such mixed-reality games, the real physical world becomes the backdrop of the game, instead of the world created on the limited displays of wireless devices [5].

To track and coordinate large numbers of continuously moving objects, their positions are stored in databases. Here, the conventional assumption, that data remains constant unless it is explicitly modified, no longer holds. An update is needed when the real position of an object deviates from that stored in the database by an application-dependent threshold. Modeling the position of an object as a static point either leads to very frequent updates or a very outdated database. To reduce the amount of updates needed, the positions of moving point objects have instead been modeled as functions of time. This makes the recorded positions more resilient to object movement, so that they may be expected to approximately capture the actual positions for longer time periods.

We consider the computation of nearest neighbor (*NN*) and reverse nearest neighbor (*RNN*) queries in this setting. In the *NN* problem, which has been investigated extensively in other settings, the objects in the database that are nearer to a given query object than any other objects in the database have to be found. In the *RNN* problem, which is new and largely unexplored, objects that have the query object as their nearest neighbor have to be found. In the example to the left in Figure 1, the *RNN* query for point 1 returns points 2 and 5. Points 3 and 4 are not returned because they have each other as their nearest neighbors. Note that even though point 2 is not a nearest neighbor of point 1, point 2 is the reverse nearest neighbor of point 1 because point 1 is the closest to point 2.

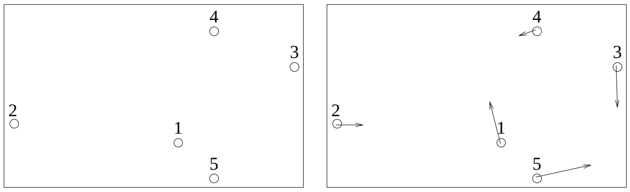

Figure 1. Static and Moving Points

A straightforward solution to computing reverse nearest neighbor (*RNN*) queries is to check for each point whether it has a given query point as its nearest neighbor. However, this approach is unacceptable when the number of points is large.

The situation is complicated further when the query and data points are moving rather than static and we want to know the reverse nearest neighbors during some time interval. For example, if points are moving as depicted to the right in Figure 1, then after some time, point 4 becomes a reverse nearest neighbor of point 1, and point 3 becomes a nearest neighbor of point 5, meaning that point 5 is no longer a reverse nearest neighbor of point 1.

Reverse nearest neighbors can be useful in applications where moving objects agree to provide some kind of service to each other. Whenever a service is needed an object requests it from its nearest neighbor. An object then may need to know how many objects it is supposed to serve in the near future and where those objects are. The examples of moving objects could be soldiers in a battlefield, tourists in dangerous environments, or mobile communication devices in wireless ad-hoc networks. In mixed-reality games like the one mentioned earlier, players may be "shooting" their nearest neighbors. Then a player may be interested to know who are her reverse nearest neighbors in order to dodge their fire.

There are proposed solutions for efficiently answering reverse nearest neighbor queries for non-moving points [12, 23, 25], but we are not aware of any algorithms for moving points. While much work has been conducted on algorithms for nearest neighbor queries, we are aware of only one work that has explored algorithms for a moving query point and static data points [22] and of no solutions for moving data and query points in two or higher dimensional space.

This paper proposes an algorithm that efficiently computes *RNN* queries for a query point during a specified time interval assuming the query and data points are continuously moving in the plane. As a solution to a subproblem, an algorithm for answering *NN* queries for continuously moving points is also proposed.

In the next section, the problem addressed by the paper is defined and related work is covered in further detail. In Section 3 our algorithms are presented. In Section 4 the results of the experiments are given, and Section 5 offers a summary and directions for future research.

2 Problem Statement and Related Work

We first describe the data and queries that are considered in this paper. Then we survey the existing solutions to the most related problems.

2.1 Problem Statement

We consider two-dimensional space and model the positions of two-dimensional moving points as linear functions of time. That is, if at time t_0 the coordinates of a point are (x, y) and its velocity vector is $\bar{v} = (v_x, v_y)$, then it is assumed that at any time $t \geq t_0$ the coordinates of the point will be $(x + (t - t_0)v_x, y + (t - t_0)v_y)$, unless a new (position, velocity) pair for the point is reported.

With this assumption, the nearest neighbor (*NN*) and reverse nearest neighbor (*RNN*) query problems for continuously moving points in the plane can be formulated as follows.

Assume (1) a set S of moving points, where each point is specified by its coordinates (x, y) and its velocity vector (v_x, v_y) at some specific time; (2) a query point $q \in S$; and (3) a query time interval $[t^{\vdash}; t^{\dashv}]$, where $t^{\vdash} \geq t_{current}$, and $t_{current}$ is the time when the query is issued.

Let NN_j and RNN_j denote sets of moving points and T_j denote a time interval. The NN query returns the set $\{\langle NN_j, T_j \rangle\}$, and the *RNN* query returns the set $\{\langle RNN_j, T_j \rangle\}$. These sets satisfy the conditions $\bigcup_j T_j = [t^{\vdash}; t^{\dashv}]$ and $i \neq j \Rightarrow T_i \cap T_j = \emptyset$. In addition, each point in NN_j is a nearest neighbor to q during all of interval T_j, and RNN_j is the set of the reverse nearest neighbors to q during all of interval T_j. That is, $\forall j\ \forall p \in NN_j\ \forall r \in S \setminus \{p\}\ (d(q,p) \leq d(q,r))$ and $\forall j\ \forall p \in RNN_j\ \forall r \in S \setminus \{p\}\ (d(q,p) \leq d(p,r))$ during all of T_j, where $d(p_1, p_2)$ is the Euclidean distance between points p_1 and p_2.

The requirement that the query point q belongs to data set S is natural for *RNN* queries—the points from S are "looking" for their neighbors among the other points in S. Nevertheless, none of the solutions presented in this paper rely inherently on this assumption. Thus, q could as well be a point not belonging to S.

Observe that the query answer is temporal, i.e., the future time interval $[t^{\vdash}; t^{\dashv}]$ is divided into disjoint intervals T_j during which different answer sets (NN_j, RNN_j) are valid. Some of these answers may become invalidated if some of the points in the database are updated before $t^{\dashv}$.

According to the terminology of Sistla et al. [20], we term queries with answer sets that are maintained under updates *persistent*. It may be useful to change the query time interval in step with the continuously changing current time, i.e., it may be useful to have $[t^{\vdash}; t^{\dashv}] = [now, now + \Delta]$, where now is the continuously changing current time. Such a query is termed *continuous*. Algorithms for updates and persistent and continuous queries are available in the extended version of this paper [2].

2.2 Related Work

Reverse nearest neighbor queries are intimately related to nearest neighbor queries. In this section, we first overview the existing proposals for answering nearest neighbor queries, for both stationary and moving points. Then, we discuss the proposals related to reverse nearest neighbor queries.

2.2.1 Nearest Neighbor Queries

A number of methods were proposed for efficient processing of nearest neighbor queries for stationary points. The majority of the methods use index structures. Some proposals rely on index structures built specifically for nearest neighbor queries [3]. Branch-and-bound methods work on index structures originally designed for range queries. Perhaps the most influential in this category is an algorithm for finding the k nearest neighbors proposed by Roussopoulos et al. [16]. In this solution, an R-tree [6] indexes the points, and traversal of the tree is ordered and pruned based on a number of heuristics. Cheung and Fu [4] simplified this algorithm without reducing its efficiency. Other branch-and-bound methods modify the index structures to better suit the nearest neighbor problem [10, 24]. A number of incremental algorithms for similarity ranking have been proposed that can efficiently compute the $(k + 1)$-st nearest neighbor, after the k nearest neighbors are returned [9, 8]. They use a global priority queue of the objects to be visited in an R-tree.

Kollios et al. [11] propose an elegant solution for answering nearest neighbor queries for moving objects in one-dimensional space. Their algorithm uses a duality transformation, where the future trajectory of a moving point $x(t) = x_0 + v_x t$ is transformed into a point (x_0, v_x) in a so-called dual space. The solution is generalized to the "1.5-dimensional" case where the objects are moving in the plane, but with their movements being restricted to a number of line segments (e.g., corresponding to a road network). However, a query with a time interval predicate returns the single object that gets the closest to the query object during the specified time interval. It does not return the nearest neighbors for each time point during that time interval (cf. the problem formulation in Section 2.1). Moreover, this solution cannot be straightforwardly extended to the two-dimensional case, where the trajectories of the points become lines in three-dimensional space.

Most recently, Song and Roussopoulos [22] have proposed a solution for finding the k nearest neighbors for a moving query point. However, the data points are assumed to be static. In addition, in contrast to our approach, time is not assumed to be continuous—a periodical sampling technique is used instead. When computing the result set for some sample, the algorithm tries to reuse the information contained in the result sets of the previous samples.

2.2.2 Reverse Nearest Neighbor Queries

To our knowledge, three solutions exist for answering *RNN* queries for non-moving points in two and higher dimensional spaces. Stanoi et al. [23] present a solution for answering *RNN* queries in two-dimensional space. Their algorithm is based on the following observations [21]. Let the space around the query point q be divided into six equal regions $S_i (1 \leq i \leq 6)$ by straight lines intersecting at q, as shown in Figure 2. Assume also that each region S_i includes only one of its bordering half-lines. Then, there exist at most six *RNN* points for q, and they are distributed so that there exists at most one *RNN* point in each region S_i.

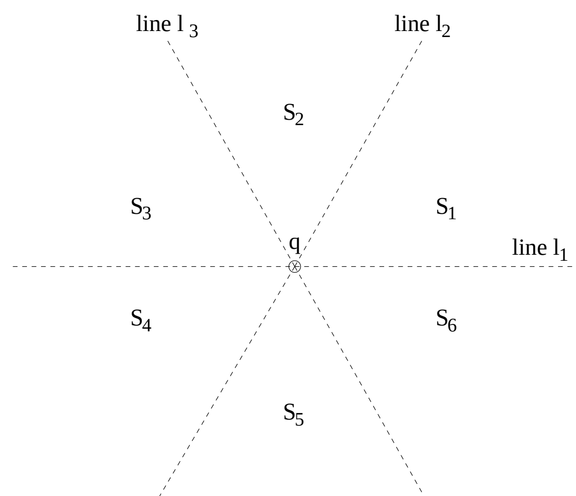

Figure 2. Division of the Space Around Query Point q

The same kind of observation leads to the following property. Let p be a *NN* point of q among points in S_i. Then, either q is the *NN* point of p (and then p is the *RNN* point of q), or q has no *RNN* point in S_i. Stanoi et al. prove this property [23].

These observations enable a reduction of the *RNN* problem to the *NN* problem. For each region S_i, an *NN* point of q in that region is found. We term it an *RNN* candidate. If there are more than one *NN* point in some S_i, they are not *RNN* candidates. For each of the candidate points, it is checked whether q is the nearest neighbor of that point. The answer to the $RNN(q)$ query consists of those candidate points that have q as their nearest neighbor.

In another solution for answering *RNN* queries, Korn and Muthukrishnan [12] use two R-trees for the querying, insertion, and deletion of points. In the RNN-tree, the minimum bounding rectangles of circles having a point as their center and the distance to the nearest neighbor of that point as their radius are stored. The NN-tree is simply an R*-tree [1] where the data points are stored. Yang and Lin [25] improve the solution of Korn and Muthukrishnan by introducing the Rdnn-tree, which makes possible to answer both *RNN* queries and *NN* queries using a single tree. Structurally, the Rdnn-tree is an R*-tree where each leaf entry is augmented with the distance to its nearest neighbor (dnn) and where a non-leaf entry stores the maximum of its children's dnn's.

None of the above-mentioned methods handle continuously moving points. In the next section, before presenting our method, we discuss the extendibility of these methods to support continuously moving points.

3 Algorithms

This section first describes the main ideas of the TPR-tree [18], which is used to index continuously moving points. Then, we briefly discuss the suitability of the methods described in Section 2.2.2 as the basis for our solution. The algorithms for answering the *NN* and *RNN* queries using the TPR-tree are presented next, followed by a simple example of a query.

3.1 TPR-tree

We use the TPR-tree (Time Parameterized R-tree [18]) as an underlying index structure. The TPR-tree indexes continuously moving points in one, two, or three dimensions. It employs the basic structure of the R*-tree [1], but both the indexed points and the bounding rectangles are augmented with velocity vectors. This way, bounding rectangles are time parameterized—they can be computed for different time points. The velocities of the edges of bounding rectangles are chosen so that the enclosed moving objects, be they points or other rectangles, remain inside the bounding rectangles at all times in the future. More specifically, if a number of points p_i are bounded at time t, the spatial and velocity extents of a bounding rectangle along the x axis is computed as follows:

$$x^{\vdash}(t) = \min_i\{p_i.x(t)\}; \quad x^{\dashv}(t) = \max_i\{p_i.x(t)\};$$
$$v_x^{\vdash} = \min_i\{p_i.v_x\}; \quad v_x^{\dashv} = \max_i\{p_i.v_x\}.$$

Figure 3 shows an example of the evolution of a bounding rectangle in the TPR-tree computed at $t = 0$. Note that, in contrast to R-trees, bounding rectangles in the TPR-tree are not minimum at all times. In most cases, they are minimum only at the time when they are computed. Other than

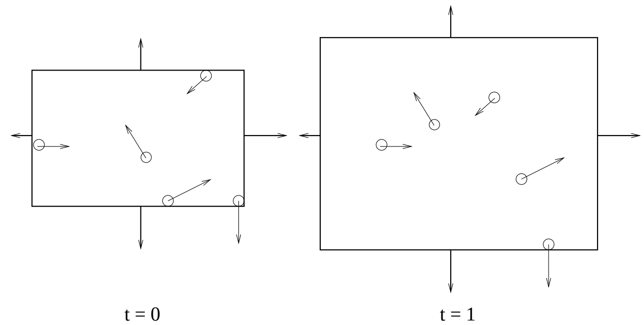

Figure 3. Time-Parameterized Bounding Rectangle

that, the TPR-tree can be interpreted as an R-tree for any specific time, t. This suggests that the algorithms that are based on the R-tree should be easily "portable" to the TPR-tree.

3.2 Preliminaries

Our *RNN* algorithm is based on the proposal of Stanoi et al. [23], described in Section 2.2.2. This algorithm uses the R-tree and does not require any specialized index structures. The other two proposals mentioned in Section 2.2.2 store, in one form or another, information about the nearest neighbor(s) of each point. With moving points, such information changes as time passes, even if no updates of objects occur. By not storing such information in the index, we avoid the overhead of its maintenance.

The sketch of the algorithm is analogous to the one described in Section 2.2.2. Our *RNN* algorithm first uses the *NN* algorithm to find the *NN* point in each S_i. For each of these candidate points, the algorithm assigns a validity time interval, which is part of the query time interval. Then, the *NN* algorithm is used again, this time unconstrained by the regions S_i, to check when, during each of these intervals, the candidate points have the query point as their nearest neighbor.

3.3 Algorithm for Finding Nearest Neighbors

Our algorithm for finding the nearest neighbors for continuously moving points in the plane is based on the algorithm proposed by Roussopoulos et al. [16]. That algorithm traverses the tree in depth-first order. Two metrics are used to direct and prune the search. The order in which the children of a node are visited is determined using the function $mindist(q, R)$, which computes the minimum distance between the bounding rectangle R of a child node and the query point q. Another function, $minmaxdist(q, R)$, which gives an upper bound of the smallest distance from q to points in R, assists in pruning the search.

Cheung and Fu [4] prove that, given the $mindist$-based ordering of the tree traversal, the pruning obtained by Roussopoulos et al. can be achieved without use of $minmaxdist$. Their argument does not seem to be straightforwardly extendible to our algorithm, where $mindist$ is extended to take into account temporal evolution. Nevertheless, because the $minmaxdist$ function is based on the assumption that bounding rectangles are always minimum [16], which is not the case in the TPR-tree (cf. Figure 3), we cannot adapt this function to our need.

In describing our algorithm, $\mathbf{FindNN}$, the following notation is used. The function $d_q(p,t)$ denotes the square of the Euclidean distance between query point q and point p at time t. Similarly, function $d_q(R,t)$ indicates the square of the distance between the query point q and the point on rectangle R that is the closest to point q at time t.

Because the movements of points are described by linear functions, for any time interval $[t^{\vdash};t^{\dashv}]$, $d_q(p,t) = at^2 + bt + c$, where $t \in [t^{\vdash};t^{\dashv}]$ and a, b, and c are constants dependent upon the positions and velocity vectors of p and q. Similarly, any time interval $[t^{\vdash};t^{\dashv}]$ can be subdivided into a finite number of non-intersecting intervals T_j so that $d_q(R,t) = a_k t^2 + b_k t + c_k$, where $t \in T_j$ and a_k, b_k, and c_k are constants dependent upon the positions and velocity vectors of R and q. Function $d_q(R,t)$ is zero for times when q is inside R. The details of how the interval is subdivided and how the constants a_k, b_k, and c_k are computed can be found elsewhere [2].

Nearest Neighbor Algorithm

$\mathbf{FindNN}(q,\ [t^{\vdash};t^{\dashv}])$:
1: $\forall t \in [t^{\vdash};t^{\dashv}]$, set $min_q(t) \leftarrow \emptyset$ and $dmin_q(t) \leftarrow \infty$.
2: Do a depth-first search in the TPR-tree, starting from the root. For each visited node:
2.1: If it is a non-leaf node, order all rectangles R in the node according to the metric $M(R,q) = \int_{t^{\vdash}}^{t^{\dashv}} d_q(R,t)dt$. The entries corresponding to rectangles with smaller $M(R,q)$ are visited first. For each R:
2.1.1: If $\forall t \in [t^{\vdash};t^{\dashv}](d_q(R,t) \geq dmin_q(t))$, prune rectangle R.
2.1.2: Else, go deeper into the node corresponding to R.
2.2: If it is a leaf node, for each p contained in it, such that $p \neq q$:
2.2.1: If $\forall t \in [t^{\vdash};t^{\dashv}](d_q(p,t) \geq dmin_q(t))$, skip p.
2.2.2: If $\forall t \in T',\ T' \subset [t^{\vdash};t^{\dashv}](d_q(p,t) < dmin_q(t))$, set $\forall t \in T'\ (min_q(t) \leftarrow \{p\},\ dmin_q(t) \leftarrow d_q(p,t))$. If $\forall t \in T',\ T' \subset [t^{\vdash};t^{\dashv}](d_q(p,t) = dmin_q(t))$, set $\forall t \in T'\ (min_q(t) \leftarrow min_q(t) \cup \{p\})$.

The algorithm maintains a list of intervals T_j as mentioned in Section 2.1. Initially the list contains a single interval $[t^{\vdash};t^{\dashv}]$, which is subdivided as the algorithm progresses. Each interval T_j in the list has associated with it (i) a point p_j, and possibly more points with the same distance from q as p_j, that is the nearest neighbor of q during this interval among the points visited so far and (ii) the squared distance $d_q(p_j,t)$ of point p_j to the query point expressed by the three parameters a, b, and c. In the description of the algorithm, we represent this list by two functions. For each $t \in [t^{\vdash};t^{\dashv}]$, function $min_q(t)$ denotes the points that are the closest to q at time t (typically, there will only be one such point), and $dmin_q(t)$ indicates the distance between q and $min_q(t)$ at time t.

Steps 2.1.1, 2.2.1, and 2.2.2 of the algorithm involve scanning through a list (or two) of time intervals and solving quadratic inequalities for each interval. In step 2.2.2, new intervals are introduced in the answer list. After the traversal of the tree, for each T_j in the answer list, $\forall t \in T_j(NN_j = min_q(t))$.

The idea behind metric M in step 2.1 is to visit first parts of the tree that are on average the closest to the query point q. The rectangle is pruned if there is no chance that it will contain a point that at some time during the query interval is closer to the query point q than the currently known closest point to q at that time.

3.4 Algorithm for Finding Reverse Nearest Neighbors

In this section, we describe algorithm $\mathbf{FindRNN}$ that computes the reverse nearest neighbors for a continuously moving point in the plane. The notation is the same as in the previous section. The algorithm produces a list $LRNN = \{\langle p_j, T_j \rangle\}$, where p_j is the reverse nearest neighbor of q during time interval T_j. Note that the format of $LRNN$ differs from the format of the answer to the RNN query, as defined in Section 2.1, where intervals T_j do not overlap and have sets of points associated with them. To simplify the description of the algorithm, we use this new format. Having $LRNN$, it is quite straightforward to transform it into the format described in Section 2.1 by sorting end points of time intervals in $LRNN$, and performing a "time sweep" to collect points for each of the formed time intervals.

To reduce the disk I/O incurred by the algorithm, all the six sets B_i are found in a single traversal of the index. Note that if, at some time, there is more than one nearest neighbor in some S_i, those nearest neighbors are nearer to each other than to the query point, meaning that S_i will hold no *RNN* points for that time. We thus assume in the following that, in sets B_i, each interval T_{ij} is associated with a single nearest neighbor point, nn_{ij}.

Reverse Nearest Neighbor Algorithm

$\mathbf{FindRNN}(q,\ [t^{\vdash}; t^{\dashv}])$:
1: For each of the six regions S_i, find a corresponding set of nearest neighbors B_i by calling $\mathbf{FindNN}(q,[t^{\vdash}; t^{\dashv}])$ for region S_i only. A version of algorithm $\mathbf{FindNN}$ is used were step 2.2.2 is modified to consider only time intervals when p is inside S_i.
2: Set $LRNN \leftarrow \emptyset$.
3: For each B_i and for each $\langle NN_{ij}, T_{ij}\rangle \in B_i$, if $|NN_{ij}| = 1$ (and $nn_{ij} \in NN_{ij}$), do:
3.1: Call $\mathbf{FindNN}(nn_{ij}, T_{ij})$ to check when during time interval T_{ij}, q is the *NN* point of nn_{ij}. The algorithm $\mathbf{FindNN}$ is modified by using $min_{nn_{ij}}(t) \leftarrow q$, $dmin_{nn_{ij}}(t) \leftarrow d_{nn_{ij}}(q,t)$ in place of $min_{nn_{ij}}(t) \leftarrow \emptyset$, $dmin_{nn_{ij}}(t) \leftarrow \infty$ in step 1. In addition, an interval $T' \subset T_{ij}$ is excluded from the list of time intervals and is not considered any longer as soon as a point p is found such that $\forall t \in T'\ (d_{nn_{ij}}(p,t) < d_{nn_{ij}}(q,t))$.
3.2: If $\mathbf{FindNN}(nn_{ij}, T_{ij})$ returns a non-empty answer, i.e., $\exists\, T' \subset T_{ij}$, such that q is an *NN* point of nn_{ij} during time interval T', add $\langle nn_{ij}, T'\rangle$ to $LRNN$.

All the *RNN* candidates nn_{ij} are also verified in one traversal. To make this possible, we use $\sum_{i,j} M(R, nn_{ij})$ as the metric for ordering the search in step 2.1 of $\mathbf{FindNN}$. In addition, a point or a rectangle is pruned only if it can be pruned for each of the query points nn_{ij}.

Thus, the index is traversed twice in total.

When analyzing the I/O complexity of $\mathbf{FindRNN}$, we observe that in the worst case, all nodes of the tree are visited to find the nearest neighbors using $\mathbf{FindNN}$, which is performed twice. As noted by Hjaltason and Samet [9], this is even the case for static points ($t^{\vdash} = t^{\dashv}$), where the size of the result set is constant. For points with linear movement, the worst case size of the result set of the *NN* query is $O(N)$ (where N is the database size). The size of the result set of $\mathbf{FindNN}$ is important because if the combined size of the sets B_i is too large, the B_i will not fit in main memory. In our performance studies in Section 4, we investigate the observed average number of I/Os and the average sizes of result sets.

3.5 Query Example

To illustrate how an *RNN* query is performed, Figure 4 depicts 11 points, with point 1 being the query point. The velocity of point 1 has been subtracted from the velocities of all the points, and the positions of the points are shown at time $t = 0$. The lowest-level bounding rectangles of the index on the points, R_1 to R_5, are shown. Each node in the TPR-tree has from 2 to 3 entries. As examples, some distances from point 1 are shown: $d_{P_1}(P_8, t)$ is the distance between point 1 and point 8, $d_{P_1}(R_1, t)$ is the distance between point 1 and rectangle 1, and $d_{P_1}(R_2, t)$ is the distance between point 1 and rectangle 2.

If the *RNN* query for the time interval $[0; 2]$ is issued, $dmin_{P_1}(t)$ for region S_1 is set to $d_{P_1}(P_3, t)$ after visiting rectangle 2, and because $d_{P_1}(R_4, t) > d_{P_1}(P_3, t)$ for all $t \in [0; 2]$, rectangle R_4 is pruned.

With the purpose of taking a closer look at how the *RNN* query is performed in regions S_2 and S_3, Figure 5 shows the positions of the points in regions S_2 and S_3 at time points $t = 0$, $t = 1$, and $t = 2$. Point 7 crosses the line delimiting regions S_2 and S_3 at time $t = 1.5$.

After the first tree-traversal, the *NN* points in region S_2 are $B_2 = \{\langle P_4, [0; 1.5]\rangle, \langle P_7, [1.5; 2]\rangle\}$, and in region S_3, they are $B_3 = \{\langle P_7, [0; 1.5]\rangle, \langle P_8, [1.5; 2]\rangle\}$. However, the list of *RNN* points *LRNN*, which is constructed during the second traversal of the TPR-tree while verifying candidate points 4, 7, and 8, is only $\{\langle P_7, [0; 1.5]\rangle, \langle P_7, [1.5; 2]\rangle\}$. This is because during time interval $[0; 1.5]$, point 10, but not point 1, is the closest to point 4, and, similarly, during time interval $[1.5; 2]$, point 7, but not point 1, is the closest to point 8.

4 Performance Experiments

The experimental setting for the performance experiments is described initially. Then follows an account for results of experiments that aim to elicit pertinent properties of the proposed algorithms.

4.1 Experimental Setting

The algorithms presented in this paper were implemented in C++, using a TPR-tree implementation based on GiST [7]. Specifically, the TPR-tree implementation with self-tuning time horizon was used [17]. We investigate the performance of algorithms in terms of the number of I/O operations they perform. The disk page size (and the size of a TPR-tree node) is set to 4k bytes, which results in 204 entries per leaf node in trees. An LRU page buffer of 50 pages is used [14], with the root of a tree always pinned in the buffer. The nodes changed during an index operation are marked as “dirty” in the buffer and are written to disk at the end of the operation or when they otherwise have to be removed from the buffer. In addition to the LRU page buffer, we use a main-memory resident storage area for recording the temporary answer sets of the first tree-traversals of *RNN* queries (the B_i lists).

The performance studies are based on synthetically generated workloads that intermix update operations and queries. To generate the workloads, we simulate N objects moving in a region of space with dimensions 1000 × 1000

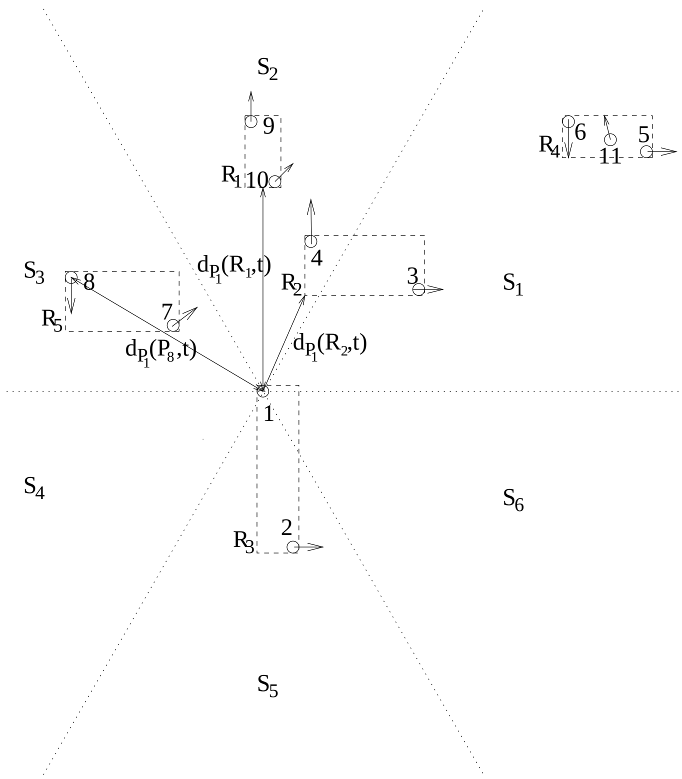

Figure 4. Example Query

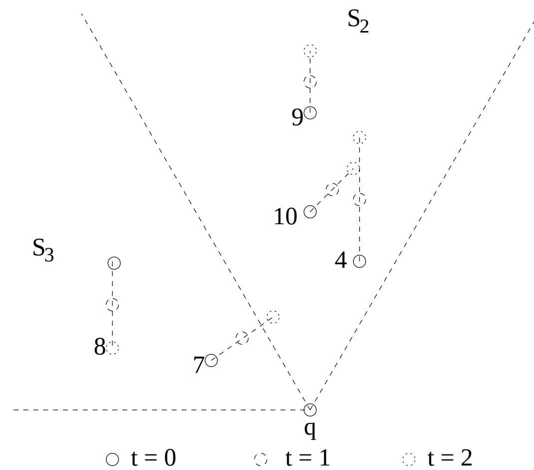

Figure 5. Simplified Example Query

kilometers. Whenever an object reports its movement, the old information pertaining to the object is deleted from the index (assuming this is not the first reported movement from this object), and the new information is inserted into the index.

Two types of workloads were used in the experiments. In most of the experiments, we use uniform workloads, where positions of points and their velocities are distributed uniformly. The speeds of objects vary from 0 to 3 kilometers per time unit (minute). In other experiments, more realistic workloads are used, where objects move in a network of two-way routes, interconnecting a number of destinations uniformly distributed in the plane. Points start at random positions on routes and are assigned with equal probability to one of three groups of points with maximum speeds of 0.75, 1.5, and 3 km/min. Whenever an object reaches one of the destinations, it chooses the next target destination at random. The network-based workload generation used in these experiments is described in more detail elsewhere [18].

In both types of workloads, the average interval between two successive updates of an object is equal to 60 time units. Unless noted otherwise, the number of points is 100,000. Workloads are run for 120 time units to populate the index. Then, queries are introduced, intermixed with additional updates. Each query corresponds to a randomly selected point from the currently active data set. Our performance graphs report average numbers of I/O operations per query.

4.2 Properties of the Nearest Neighbor and Reverse Nearest Neighbor Algorithms

In the first round of the experiments, a variety of the properties of the algorithms computing nearest and reverse nearest neighbors are explored.

Figure 6 shows the average number of I/O operations per query when varying the number of points in the database. In this experiment, after the initial phase of 120 time units, the workloads are run for an additional 10 time units. During this period, 500 queries are issued. For each query, its time interval starts at the time of issue, and the length of the interval varies from 0 to 30 time units.

The number of I/O operations increases almost linearly with the number of data points. Figure 7 shows that the size of an average result increases similarly.

It is interesting to observe that the second traversal of the tree, in which the candidates produced by the first traversal are verified, is more expensive than the first traversal, in which these candidates are found. The main reason for this behavior is that while there is only one query point during the first traversal, during the second traversal, there is a number of *RNN* candidates (from the different regions S_i

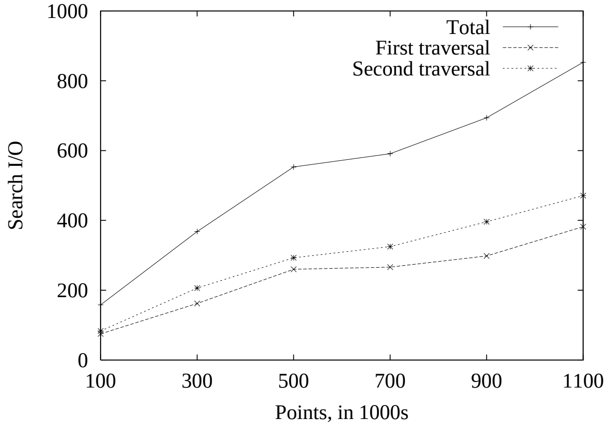

Figure 6. Query Performance for Varying Number of Points

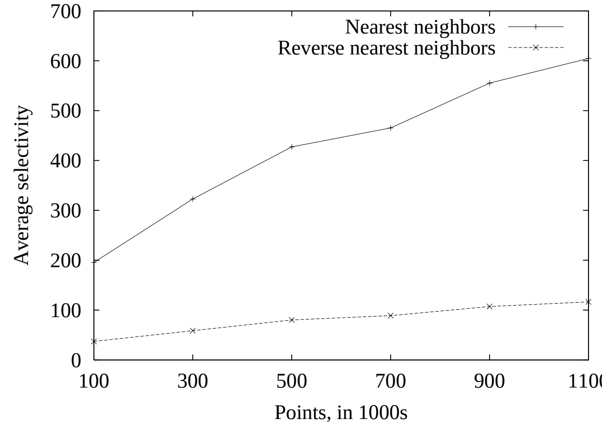

Figure 7. Average Selectivity of Queries for Varying Number of Points

and during different parts of the query interval) that serve as *NN* query points. This argument alone would perhaps lead us to expect a larger difference between the costs of the two traversals.

The relatively small difference between the two traversals occurs because during the first traversal, there is no initial upper bound for the distance between the query point q and the *RNN* candidate point, i.e., $dmin_q(t)$ is initially set to ∞ in the $\mathbf{FindNN}$ algorithm. The second traversal only needs to determine whether the point q is an *NN* point to the candidate points; and for each candidate point, there is an initial upper bound for $dmin_{nn_{ij}}(t)$, namely the distance between the point q and that candidate point, nn_{ij}. Further, since nn_{ij} is the *NN* point to q in some region S_i at some time, the distance between q and nn_{ij} is typically small. This enables a more aggressive pruning of tree nodes during the second traversal of the TPR-tree.

To learn whether the nearest neighbor (and reverse nearest neighbor) algorithm could possibly be significantly improved by changing the tree traversal order or by somehow improving the pruning, we explored how many of the visited bounding rectangles actually contained the query point at some time point during the corresponding query time interval. If several queries were performed in one tree traversal, we observed whether the bounding rectangle contained any of the query points. Tree nodes corresponding to such bounding rectangles must necessarily be visited by any *NN* algorithm to produce a correct answer. Thus, given a specific TPR-tree, the number of such bounding rectangles gives the lower performance bound for a corresponding nearest neighbor query.

In experiments with 100,000 points, during the first traversal, a total of 32 I/Os out of the average of 75 I/Os corresponded to "necessary" bounding rectangles. For the second traversal, the numbers were 39 I/Os out of 83 I/Os. This shows, that under the most optimistic assumptions, the algorithm can be improved by no more than approximately a factor of two.

Figure 7 plots the average number of entries in the result sets of queries after the first traversal of the tree, which finds nearest neighbors, and after the second traversal, which finds reverse nearest neighbors. Note that a single point in the answer set may have more than one time interval associated with it. The graphs show that on average, only one out of five candidate *RNN* points is found to be a real *RNN* point. To investigate how much memory is needed for storing candidate *RNN* points (the B_i lists), we also recorded the maximum size of the answer sets in our experiments. It was no more than five times the average sizes reported in Figure 7 (i.e., at most ca. 100k bytes).

Figure 8 shows the average number of I/O operations per query when the number of destinations in the simulated network of routes is varied. "Uniform" indicates the case when the points and their velocities are distributed uniformly, which, intuitively, corresponds to a very large number of destinations. Each workload contained 500 queries, generated in the same way as for the previous experiment.

The number of I/O operations tends to increase with the number of destinations, i.e., as the workloads get more "uniform." The results are consistent with, although not as pronounced as, those reported for range queries on the TPR-tree [18]. Observe that while the performance of the second traversal shows the above-mentioned trend, data skew seems to not affect the performance of the first traversal. A possible explanation is that when moving points are concentrated on a small number of routes, the good quality of the TPR-tree is offset by the fact that there can be regions S_i that have no points inside of them, but contain parts of bounding rectangles. In such cases, $dmin_q(t)$ in $\mathbf{FindNN}$ always remains ∞ and those bounding rectangles cannot be

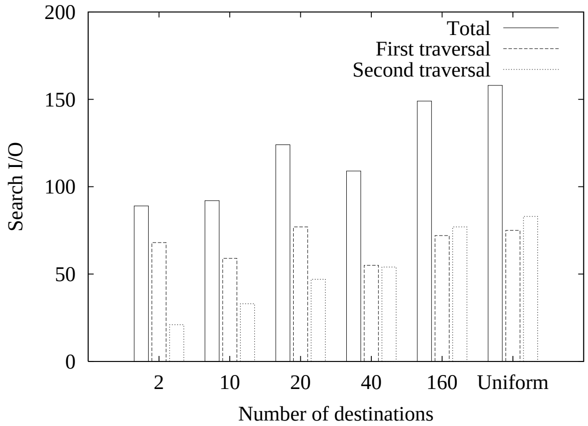

Figure 8. Query Performance for Varying Number of Destinations

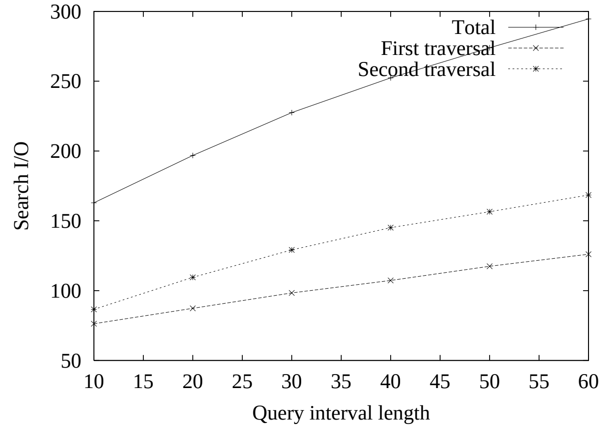

Figure 9. Query Performance for Varying Query Interval Length

pruned.

Figure 9 shows the average number of I/O operations per query for varying query interval lengths. The number of I/O operations increases approximately linearly with the query interval length. The experiment also showed that the number of results returned increases linearly.

5 Summary and Future Work

Rapidly advancing technologies make it possible to track the positions of large numbers of continuously moving objects. Because of this, efficient algorithms for answering various queries about continuously moving objects are needed. Algorithms have previously been suggested for answering *RNN* and *NN* queries for non-moving objects, but no solutions have been proposed for efficiently answering these queries when large numbers of objects are moving continuously. In this paper, we have proposed an algorithm for answering *RNN* queries for large numbers of continuously moving points in the plane. As a solution to a subproblem, an algorithm for answering *NN* queries for continuously moving points in the plane has been proposed. An experimental study was performed that revealed a number of interesting properties of the proposed algorithms.

As the indexing structure for continuously moving points, the TPR-tree [18] has been used. This means that the same index structure can be used for range queries, nearest neighbor queries, and reverse nearest neighbor queries.

The presented *RNN* query algorithm is suitable for the *monochromatic* case [12] only—all the points are assumed to be of the same category. In the *bichromatic* case, there are two kinds of points (i.e., “clients” and “servers,” corresponding to tourists and rescue workers), and an *RNN* query asks for points that belong to the opposite category than the query point and have the query point as the closest from all the points that are in the same category as the query point. The approach of dividing the plane into six regions does not work for the bichromatic case—a point can have more than six *RNN* points. An interesting future research direction is to develop an algorithm for efficiently answering *RNN* queries for continuously moving bichromatic points.

Sometimes it is important to know not only the objects that have the query object as their nearest neighbor (a simple *RNN* query) but also the objects that have the query object as their second nearest, third nearest neighbor (second, third order *RNN* query), etc. Processing of higher order *RNN* queries could be another possible extension of the proposed algorithm.

In reality, the objects most often move along some underlying route structure, for example, cars in a road network. Even if objects move freely, another type of infrastructure could exist that prohibits movement in some areas, such as lakes or mountains. How to handle the complexities arising from the non-Euclidean distance functions inherent to such environments is an interesting research direction.

Acknowledgments

This research was supported in part by the Danish Technical Research Council through grant 9700780, by the Wireless Information Management network, funded by the Nordic Academy for Advanced Study through grant 000389, and by a grant from the Nykredit corporation.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Ac-

cess Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pp. 322–331, 1990.

[2] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Šaltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. TimeCenter TR-66, avaiable via <www.cs.auc.TimeCenter>.

[3] S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl. Fast Nearest Neighbor Search in High-Dimensional Space. In *Proceedings of the 14th International Conference on Data Engineering*, pp. 209–218, 1998.

[4] K. L. Cheung and A. W. Fu. Enhanced Nearest Neighbour Search on the R-tree. *SIGMOD Record*, 27(3): 16–21, 1998.

[5] J. Elliott. Text Messages Turn Towns into Giant Computer Game. *Sunday Times*, April 29, 2001.

[6] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pp. 47–57, 1984.

[7] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the 21st VLDB Conference*, pp. 562–573, 1995.

[8] A. Henrich. A Distance Scan Algorithm for Spatial Access Structures. In *Proceedings of the Second ACM Workshop on Geographic Information Systems*, pp. 136–143, 1994.

[9] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. In *ACM Transactions on Database Systems*, 24(2): 265–318, 1999.

[10] N. Katayama and S. Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 369–380, 1997.

[11] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest Neighbor Queries in a Mobile Environment. In *Proceedings of the International Workshop on Spatio-Temporal Database Management*, pp. 119–134, 1999.

[12] F. Korn and S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 201–212, 2000.

[13] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast Nearest Neighbor Search in Medical Image Databases. In *Proceedings of the 22nd VLDB Conference*, pp. 215–226, 1996.

[14] S. T. Leutenegger and M. A. Lopez. The Effect of Buffering on the Performance of R-Trees. In *Proceedings of the 14th International Conference on Data Engineering*, pp. 164–171, 1998.

[15] F. P. Preparata and M. I. Shamos. *Computational Geometry. An Introduction.* Texts and Monographs in Computer Science. 5th corrected ed., Springer, 1993.

[16] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 71–79, 1995.

[17] S. Šaltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proceedings of the 18th International Conference on Data Engineering*, pp. 463–472, 2002.

[18] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 331–342, 2000.

[19] T. Seidl and H.-P. Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pp. 154–165, 1998.

[20] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proceedings of the 13th International Conference on Data Engineering*, pp. 422–432, 1997.

[21] M. Smid. Closest Point Problems in Computational Geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry.* Elsevier Science Publishing, pp. 877–935, 1997.

[22] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proceedings of the 7th International Symposium on Spatial and Temporal Databases*, pp. 79–96, 2001.

[23] I. Stanoi, D. Agrawal, and A. El Abbadi. Reverse Nearest Neighbor Queries for Dynamic Databases. In *Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pp. 44–53, 2000.

[24] D. A. White and R. Jain. Similarity Indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pp. 516–523, 1996.

[25] C. Yang and K.-Ip Lin. An Index Structure for Efficient Reverse Nearest Neighbor Queries. In *Proceedings of the 17th International Conference on Data Engineering*, pp. 485–492, 2001.