VLDB Journal manuscript No.
(will be inserted by the editor)

Nearest and reverse nearest neighbor queries for moving objects

Rimantas Benetis, Christian S. Jensen*, **Gytis Karčiauskas, Simonas Šaltenis**

Aalborg University, Department of Computer Science, DK-9220 Aalborg Øst, Denmark

Received: date / Revised version: date

Abstract With the continued proliferation of wireless communications and advances in positioning technologies, algorithms for efficiently answering queries about large populations of moving objects are gaining in interest. This paper proposes algorithms for k nearest and reverse k nearest neighbor queries on the current and anticipated future positions of points moving continuously in the plane. The former type of query returns k objects nearest to a query object for each time point during a time interval, while the latter returns the objects that have a specified query object as one of their k closest neighbors, again for each time point during a time interval. In addition, algorithms for so-called persistent and continuous variants of these queries are provided. The algorithms are based on the indexing of object positions represented as linear functions of time. The results of empirical performance experiments are reported.

Key words Continuous queries – Incremental update – Location-based services – Mobile objects – Neighbor queries – Persistent queries

1 Introduction

We are currently experiencing rapid developments in key technology areas that combine to promise widespread use of mobile, personal information appliances, many 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 recent de-scrambling of the GPS signals and the US E911 mandate [8], 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 BotFighters game, by the 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 purely virtual world created on the limited displays of wireless devices [7].

To track and coordinate large numbers of continuously moving objects, their positions are stored in databases. This results in new challenges to database technology. The conventional assumption, that data remains constant unless it is explicitly modified, no longer holds when considering continuous data. To reduce the amount of updates needed to maintain a certain precision of positions stored in the database, moving point objects have been modeled as functions of time rather than simply as static positions [36]. Studies of GPS logs from vehicles show that representing positions as linear functions of time reduces the numbers of updates needed to maintain a reasonable precision by as much as a factor of three in comparison to using static positions [6].

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 (as will be discussed in Section 2.2), 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 relatively new and unexplored, objects that have the query object as their nearest neighbor have to be found. In the example 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 a reverse nearest neighbor of point 1 because point 1 is the point closest to point 2.

A straightforward solution for computing reverse nearest neighbor (*RNN*) queries is to check for each point whether it

* *Additional contact information for the contact author: e-mail: csj@cs.auc.dk, tel.: (+45) 96358900, fax: (+45) 98159889.*

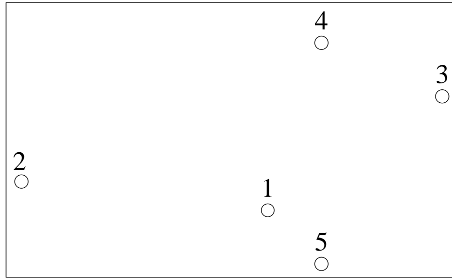

Fig. 1 Static points

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 our points are moving as depicted in Figure 2 then

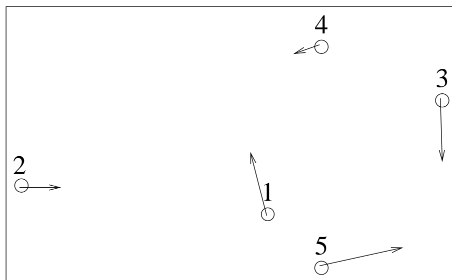

Fig. 2 Moving points

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 a mixed-reality game like the one mentioned at the beginning of the section, players may be "shooting" their nearest neighbors. Then players may be interested to know who their reverse nearest neighbors are in order to dodge their fire.

Solutions have been proposed for efficiently answering reverse nearest neighbor queries for non-moving points [16, 30,35], 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 study that has explored algorithms for a moving query point and moving data points [31].

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 and the query time interval starts at or after the current time (i.e., we do not consider querying of historical data). As a solution to a significant subproblem, an algorithm for answering *NN* queries for continuously moving points is also proposed.

The paper is a substantially revised and extended version of an earlier paper [3]. Main additions include support for *kNN* and *RkNN* queries (where $k > 1$), and support for and experimental evaluation of two kinds of index traversals and two types of search metrics. Also included is support for so-called persistent *kNN* and *RkNN* queries—incremental update techniques are introduced for this purpose. Next, support for so-called continuous current-time queries is included. A new, expanded empirical performance study of the presented types of queries is reported. Finally, material on distance computation for moving points and time-parameterized rectangles is included.

In the next section, the problem that this paper addresses 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. An appendix offers detail on the computation of distances between moving points and time-parameterized rectangles.

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 position of a point is (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 position of the point will be $(x + (t - t_0)v_x, y + (t - t_0)v_y)$, unless a new $(\text{position}, \text{velocity})$ 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 position (x, y) and its velocity (v_x, v_y) at some specific time; (2) a query point q; 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. Intuitively, we use NN_j and RNN_j for containing the *NN* and *RNN* query results, respectively, during the j-th time interval. More precisely, 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 at each time point during time interval T_j, and RNN_j is the set of the reverse nearest neighbors to q at each time point during time 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 $T_j)$. Here, $d(p_1, p_2)$ is the distance between points p_1 and p_2 and symbol $\setminus$ denotes set difference. Although any metric distance function will work, we use Euclidean distance for specificity.

We also consider the more general k nearest neighbor (*kNN*) and reverse k nearest neighbor (*RkNN*) queries. The answer to a *kNN* query has the same structure as the answer to an *NN* query, but instead of sets NN_j, each of which usually contains one element, a *kNN* answer has ordered lists $NN_j = (p_{j1}, p_{j2}, \ldots, p_{jk})$, each containing exactly k points (assuming $|S| \geq k$). The points in each list are ordered by their distance to q, so that p_{j1} is the closest point and p_{jk} is the k-th closest point to q during T_j. More formally, $\forall j\ (d(q,p_{j1}) \leq d(q,p_{j2}) \leq \cdots \leq d(q,p_{jk}) \wedge\ \forall r \in S \setminus NN_j\ (d(q,r) \geq d(q,p_{jk})$ during all of $T_j))$. Note that during T_j, there can be more than one point with a distance to q that is exactly equal to $d(q,p_{jk})$. For simplicity, an arbitrary subset of such points of size $k - |\{p \in S \mid d(q,p) < d(q,p_{jk})\}|$ is included in NN_j.

In the answer to an *RkNN* query, each set RNN_j contains all points such that each has query point q among its k nearest neighbors. More formally, $\forall j\ \forall p \in RNN_j\ \ (|\{r \in S \mid (d(p,r) < d(p,q)\}| < k$ during all of $T_j)$. Note that if, at some specific time point, point q is a k-th nearest neighbor of point p, according to the definition of the *kNN* query, q may still not be included in the answer set of *kNN*(p). This may happen if there are more than k points with a distance to p that is smaller than or equal to the distance from q to p. Nevertheless, in such situations p will always be included in the answer of the *RkNN*(q) query.

Next, observe that all the query answers are temporal, i.e., the future time interval $[t^{\vdash}; t^{\dashv}]$ is divided into disjoint intervals T_j during which different answer sets (the NN_j and RNN_j) are valid. Some of these answers may become invalidated if some of the points in the database are updated before $t^{\dashv}$. The straightforward solution would call for recomputing the answer each time the database is updated. In this paper, we present a more efficient algorithm that maintains the answer to a query when updates to the data set are performed. According to the terminology introduced by Sistla et al. [27], we use the term *persistent* for queries with answer sets that are maintained under updates.

In practice, 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. The answer to such a query should be maintained both because of the updates and because of the continuously changing query time interval. In particular, we investigate how to support *continuous* (and persistent) current-time queries ($\Delta = 0$).

2.2 Related work

Nearest neighbor queries and reverse nearest neighbor queries are intimately related. 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 have been proposed for efficient processing of nearest neighbor queries for stationary points. The majority of the methods use index structures, and some proposals rely on index structures built specifically for nearest neighbor queries. As an example, Berchtold et al. [4] propose a method based on Voronoi cells [20].

Branch-and-bound methods work on index structures originally designed for range queries. Perhaps the most influential method in this category is an algorithm, proposed by Roussopoulos et al. [22], for finding the k nearest neighbors. In this solution, an R-tree [9] indexes the points, and *depth-first* traversal of the tree is used. During the traversal, entries in the nodes of the tree are ordered and pruned based on a number of heuristics. Cheung and Fu [5] simplified this algorithm without reducing its efficiency. Other methods that use branch-and-bound algorithms modify the index structures to better suit the nearest neighbor problem, especially when applied for high-dimensional data [14, 34].

Next, 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 [12, 11]. They use a global priority queue of the objects to be visited in an R-tree. More specifically, Hjaltason and Samet [12] propose an incremental nearest neighbor algorithm, which uses a priority queue of the objects to be visited in an R*-tree [2]. They show that such a *best-first* traversal is optimal for a given R-tree. A very similar algorithm was proposed by Henrich [11], which employs two priority queues. For high-dimensional data, multi-step nearest neighbor query processing techniques are usually used [17, 25].

Kollios et al. [15] 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.

The work of Albers et al. [1], who investigate Voronoi diagrams of continuously moving points, relates to the problem of nearest neighbor queries. Even though such diagrams change continuously as points move, their topological structures change only when certain discrete events occur. The authors show a non-trivial upper bound of the number of such events. They also provide an algorithm to maintain such continuously changing Voronoi diagrams.

Song and Roussopoulos [29] propose a solution for finding the k nearest neighbors for a moving query point. However, the data points are assumed to be static. In addition, and in contrast to our approach, time is not assumed to be continuous—a periodical sampling technique is used instead. The time period is divided into n equal-length intervals. When computing the result set for some sample, the algorithm tries to reuse the information contained in the result sets of the previous samples.

The two works most closely related to ours are by Raptopoulou et al. [21] and by Tao et al. [31]. Both of these works consider the nearest neighbor problem for a query point moving on a line segment and for static or moving data points. In a manner similar to what is described in Section 2.1, the answer to a *kNN* query is temporal. In contrast to our work, both works do not consider the maintenance of query answers under updates, and reverse nearest neighbor queries are not considered. Also, compared to our work, Raptopoulou et al. consider simplified and less effective heuristics for directing and pruning the search in the TPR-tree.

In the above-mentioned study, Tao et al. also consider the general concept of so-called time-parameterized queries. The authors show how these queries can be processed using a tailored algorithm for nearest neighbor queries, such as the algorithm of Roussopoulos et al. [22]. This framework can be used to process time-parameterized nearest neighbor queries for moving objects, but each answer would include only the first time interval from the answer set as defined in Section 2.1.

2.2.2 Reverse nearest neighbor queries Several different solutions have been proposed for computing *RNN* queries for non-moving points in two and higher dimensional spaces. Stanoi et al. [30] present a solution for answering *RNN* queries in two-dimensional space. Their algorithm is based on the following observations [28]. 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 3. Assume also that each region S_i includes only one of its bordering half-lines. Then, there exists 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.

The same kind of observation leads to the following property. Let p be an *NN* point of q among the points in S_i. Then, either q is an *NN* point of p (and then p is an *RNN* point of q), or q has no *RNN* point in S_i. Stanoi et al. prove this property [30].

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.

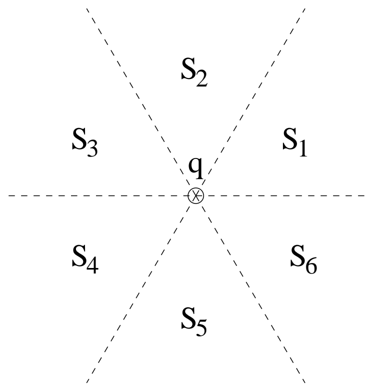

Fig. 3 Division of the space around query point q

In another solution for answering *RNN* queries, Korn and Muthukrishnan [16] use two R-trees for the querying, insertion, and deletion of points. In the first, 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 second, the NN-tree, is simply an R*-tree [2] that stores the data points. Yang and Lin [35] improve the solution of Korn and Muthukrishnan by introducing an Rdnn-tree, which makes it 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. Maheshwari et al. [19] propose main memory data structures for answering *RNN* queries in two dimensions. Their structures maintain for each point the distance to its nearest neighbor.

In contrast to the approach of Stanoi et al., updates of the database are problematic in the last three approaches mentioned. On the other hand, the approach of Stanoi et al. does not easily scale up to more than two dimensions because the number of regions where *RNN* candidates are found increases exponentially with the dimensionality [26]. To alleviate this problem, Singh et al. [26] propose an algorithm where *RkNN* candidates are found by performing a regular *kNN* query. The disadvantage of such an approach is that it does not always find all *RkNN* points. The recent approach by Tao et al. [33] fixes this problem. Their so-called TPL algorithm, similarly to the approach of Stanoi et al., works according to two phases—a filtering phase and a refinement phase—but no subdivision of the underlying space into regions is necessary in the refinement phase. Thus, the algorithm gracefully scales to more than two dimensions.

None of the above-mentioned methods handle continuously moving points and thus do not consider temporal query

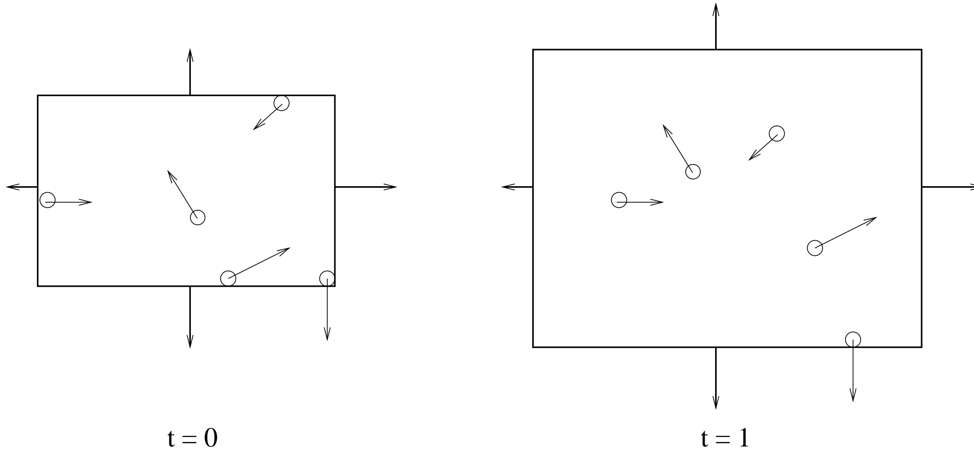

Fig. 4 Example time-parameterized bounding rectangle

answers. Persistent and continuous queries are also not supported.

3 Algorithms

This section first briefly describes the main ideas of the TPR-tree [24], 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 *kNN* and *RkNN* queries using the TPR-tree are presented in Sections 3.3 and 3.4. For clarity, the algorithms for $k = 1$ are presented first, followed by the more general algorithms. Section 3.5 presents a simple example that illustrates the computation of an *RNN* query. The next two subsections describe the algorithms that maintain the answer sets of *kNN* and *RkNN* queries under insertions and deletions. Finally, Section 3.8 covers the strategy for efficiently performing the continuous current-time query.

3.1 TPR-tree

We use the TPR-tree (Time Parameterized R-tree) [24], 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 [2], 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. Velocities are associated with the edges of bounding rectangles 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 are computed as follows:

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

Figure 4 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 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. Similarly, all algorithms presented in this paper should work without modifications for the TPR*-tree [32], which is an index that improves upon the TPR-tree, by the means of more advanced insertion and deletion algorithms.

3.2 Preliminaries

Our *RNN* algorithm is based on the proposal of Stanoi et al. [30], described in Section 2.2.2. This algorithm uses the R-tree and requires no specialized index structures. The proposals by Korn and Muthukrishnan [16] and Yang and Lin [35] 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. Similar to the approach of Stanoi et al., the recently proposed TPL algorithm [33] also does not require specialized index structures. It is an interesting future research topic to explore how the TPL algorithm can be adapted to work with continuously moving points using the techniques presented in this paper.

The idea 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 Algorithms for finding nearest neighbors

First we present an algorithm for finding the nearest neighbors of a query point. Then we show how the algorithm can be adapted to find the k nearest neighbors.

3.3.1 FindNN algorithm Our algorithm for finding the nearest neighbors for continuously moving points in the plane is based on the algorithms proposed by Roussopoulos et al. [22] and Hjaltason and Samet [12]. The former 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 [5] and, later, Hjaltason and Samet [12] prove that, given the $mindist$-based ordering of the tree traversal, the pruning that is obtained by Roussopoulos et al. can be achieved without the use of $minmaxdist$. This suggests that $minmaxdist$ can also be disregarded in our setting without any effect on the pruning. However, the proofs do not seem to be straightforwardly extendable to our setting, where $mindist$ is extended to take into account temporal evolution. We nevertheless choose to disregard $minmaxdist$. The reason is that this function is based on the assumption that bounding rectangles are always minimum [22], which is not true in the TPR-tree (cf. Figure 4). This means that we cannot straightforwardly adapt $minmaxdist$ to our setting. Thus, as described in the following, we construct and use a temporal version of the $mindist$ function, both for directing the tree traversal and for the pruning.

In describing our algorithm, 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.

As will be seen in the following, our algorithms use squared Euclidean-distance functions. Functions that express Euclidean distances between linearly moving points are square roots of quadratic polynomials. As we are interested only in the relative orders of the values of these functions, not the absolute values, we use the simpler, squared functions.

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 partitioned into a finite number of 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 in Appendix A.

The algorithm maintains a list of intervals T_j as mentioned in Section 2.1. Let us call this list the *answer list*. Initially the list contains a single interval $[t^{\vdash}; t^{\dashv}]$, which is subdivided as the algorithm progresses. Each interval T_j in the answer 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 squared distance between q and $min_q(t)$ at time t. The distance $min_q(t)$ is used to prune nodes with a bounding rectangle further away from q than $min_q(t)$ during the whole query time interval.

The algorithm is presented in Figure 5. The order of the tree traversal is determined by the min-priority queue Q that has two main operations: $pop()$, which returns an entry with the smallest key, and $push(e, M, level)$, which inserts e into the queue with a key that is constructed from metric M and tree level $level$ of e. (Metric M, to be covered in detail shortly, intuitively computes a representative distance between its two arguments during the query time interval.) If only metric M is used as the key, the algorithm performs a *best-first* traversal, which, in each step, visits an entry with the smallest metric (as done by Hjaltason and Samet [12]). If the key is a concatenation of $level$ and M, with the level number increasing when going from the leaves of the tree towards the root, the algorithm performs a *depth-first* traversal with entries in each node of the tree being visited in the order of increasing metric M (as done by Roussopoulos et al. [22]).

As noted earlier, we use a temporal version of $mindist$ as the metric M that directs the traversal. Given a time interval $[t^{\vdash}; t^{\dashv}]$ and a bounding rectangle R, there are two natural ways to compute a temporal version of $mindist$. One approach is to compute the integral of $d_q(R, t)$:

$$M(R, q) = \int_{t^{\vdash}}^{t^{\dashv}} d_q(R, t)dt$$

This metric, termed the *integral* metric, corresponds to the average of the squared distance between R and q (multiplied by the length of $[t^{\vdash}; t^{\dashv}]$). The other approach is to use the minimum of the squared distance $d_q(R, t)$:

$$M(R, q) = \min_{t \in [t^{\vdash}; t^{\dashv}]} d_q(R, t)$$

This metric, termed the *min* metric, can be computed by comparing the values of the squared distances at the end-points of the interval and at the point where the time derivative of $d_q(R, t)$ is zero. If for two rectangles R_1 and R_2, $d_q(R_1, t)$ and $d_q(R_2, t)$ are zero for some times during $[t^{\vdash}; t^{\dashv}]$ then if $d_q(R_1, t)$ is zero for a longer time period than $d_q(R_2, t)$, we say that the *min* metric of R_1 is smaller than the *min* metric of R_2.

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 Initialize a min-priority queue Q: insert into Q a pointer to the root of the TPR-tree.
3 While Q is not empty:
 3.1 Remove the top of Q: $e \leftarrow Q.pop()$. Let R be the bounding rectangle of e.
 3.2 If $\forall t \in [t^{\vdash};t^{\dashv}](d_q(R,t) > dmin_q(t))$, prune entry e (i.e., do nothing).
 3.3 Else if e points to a non-leaf node, for each entry $e_i = (R_i, ptr_i)$ in this node compute the metric $M_i = M(R_i, q)$ and add e_i to the queue: $Q.push(e_i, M_i, level(e_i))$.
 3.4 Else if e points to a leaf node, for each p contained in it, such that $p \neq q$:
  3.4.1 If $\forall t \in [t^{\vdash};t^{\dashv}](d_q(p,t) > dmin_q(t))$, skip p.
  3.4.2 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 $\forall t \in T'\ (min_q(t) \leftarrow min_q(t) \cup \{p\})$.

Fig. 5 Algorithm computing nearest neighbors for moving objects in the plane

Figure 6 plots the squared distance between a query point and two bounding rectangles. If the *min* metric is used, R_1 will have the smallest metric, if the *integral* metric is used, R_2 will have the smallest metric. As the figure shows, the *min* metric favors, when it is used for guiding the traversal, bounding rectangles that may contain nearest neighbors during *some* time points, while the *integral* metric favors bounding rectangles that contain points which are likely to reduce the pruning distance $dmin_q(t)$ during large parts of the interval $[t^{\vdash};t^{\dashv}]$.

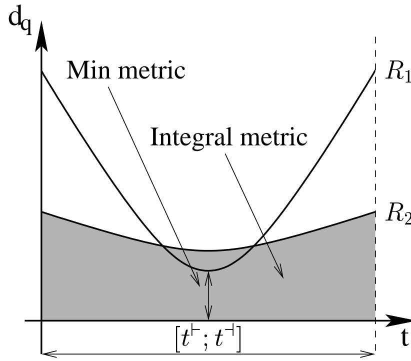

Fig. 6 *Integral* and *min* metrics

The two types of tree traversals combined with the two types of metrics yield four variants of the **FindNN** algorithm. We explore these variants in the performance experiments reported in Section 4.

3.3.2 Constructing the answer list Steps 3.2, 3.4.1, and 3.4.2 of algorithm **FindNN** construct the answer list as the tree is traversed and use the answer list for pruning. The steps are presented in a declarative way in Figure 5. In this section we discuss in greater detail the implementation of these steps, which involve scanning through a list (or two) of time intervals and solving quadratic inequalities for each interval.

More specifically, in step 3.2, the algorithm described in Appendix A is executed. This algorithm divides the original query interval into at most five (for two-dimensional data) subintervals, as indicated by the numbers in Figure 33. Note that this subdivision has no relation to the subdivision recorded in the answer list. Each of the produced subintervals has the three parameters (a_R, b_R, c_R) that define the quadratic function that expresses the distance from the query point to R.

After this step, there are two subdivisions of the query interval: the one just produced and the answer list. They are combined into one subdivision by sorting together the time points in both subdivisions. For example, if the query time interval was $[0, 10)$, the answer list was $[0, 6), [6, 10)$, and the subdivision produced by R was $[0, 3), [3, 10)$, we get the new subdivision $[0, 3), [3, 6), [6, 10)$. Associated with each of the intervals in this subdivision are both the original quadratic function $dmin_q(t)$ (expressed by the parameters a, b, and c) and the quadratic function of the distance to R (expressed by the parameters a_R, b_R, and c_R). For each interval I in this combined subdivision, the quadratic inequality $a_R t^2 + b_R t + c_R < at^2 + bt + c$ is solved to compare the distance from the query point q to R and the distance from q to point(s) in the answer list. The inequality can have at most two roots, which can be inside or outside of the interval I. This indicates whether some part of I exists where R gets closer to the query than point(s) in the answer list. If this is so for at least one interval I, we go deeper into the subtree rooted at the entry with R (step 3.3). Thus, 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.

At the leaf level, in steps 3.4.1 and 3.4.2, we similarly solve quadratic inequalities for each interval in the answer list. In this case, two subdivisions do not have to be combined. This is so because the distance between the query point q and the data point p can be described by a single quadratic function (expressed by the parameters a_p, b_p, and c_p). For each interval I in the answer list, the solution of the quadratic inequality $d_q(p,t) < dmin_q(t)$ may again produce at most two roots, which may result in subdivision of I into at most three subintervals. During the intervals in I, when the inequality holds, we replace the original parameters a, b, and c with the new parameters a_p, b_p, and c_p. This way, new intervals are introduced in the answer list in step 3.4.2. Processing all intervals I produces the new version of the answer list.

After the traversal of the tree, the following holds for each T_j in the answer list: $\forall t \in T_j(NN_j = min_q(t))$.

3.3.3 FindkNN algorithm The algorithm presented in the previous section can be extended to find k nearest neighbors. As described in Section 2.1, the result of such an algorithm, which we term **FindkNN**, is a set $\{\langle NN_j, T_j\rangle\}$, where each NN_j is an ordered list of k points that are closest to the query point during time interval T_j.

The overall structure of algorithm **FindkNN** is the same as that of **FindNN**. The answer list representing the subdivision of the query time interval is built as the algorithm traverses the tree. Each interval T_j in the answer list has associated with it an ordered list of points $min_q = (p_1, p_2, \ldots, p_l)$, where p_1 is the nearest neighbor of q and p_l is the l-th nearest neighbor of q during this interval among the points visited so far. At the beginning of the tree traversal, l is equal to the number of visited data points, but it stops at k when k data points have been visited. The squared distance function $d_q(p_i, t)$—in the form of the three parameters a_i, b_i, and c_i—is stored with each point p_i $(i = 1, \ldots, l)$.

We define $min_q(t)$ to be the list min_q associated with the answer list interval to which t belongs. We use the notation $min_q(t)[i]$ to access the point p_i in the list $min_q(t)$. We also define $dmin_q(t) = d_q(min_q(t)[k], t)$, if $l = k$, and $dmin_q(t) = \infty$, if $l < k$.

With this notation in place, the pseudo code of algorithm **FindkNN** is shown in Figure 7. Note that steps 3.2 and 3.4.1 involve solving quadratic inequalities as described in Section 3.3.2. In step 3.4.2, those time intervals from the answer set for which the inequality $d_q(p, t) < dmin_q(t)$ holds during only part of the interval are divided into two or three intervals, copying the corresponding list min_q and changing the k-th (or $(l+1)$-st) element of it where necessary. Similarly, in **CorrectOrder**, the intervals from the answer set are subdivided further, and points $min_q(t)[i]$ and $min_q(t)[i-1]$ are exchanged only for the subintervals during which the ordering of these points is wrong.

Figure 8 demonstrates how an answer list of two intervals (T_1 and T_2) is modified when visiting a data point p. Here

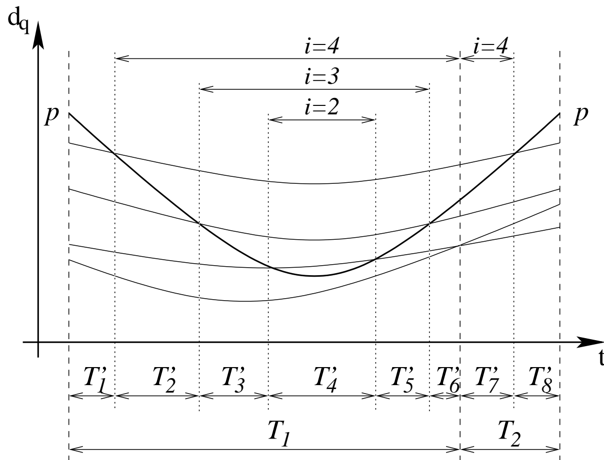

Fig. 8 Subdivision of the answer list intervals when visiting point p

$k = 4$ and for each of the four points in the answer list, as well as p, the squared distance to the query point is plotted against time. Subintervals T'_1 and T'_8 are introduced in step 3.4.2.2 of the algorithm. The remaining parts of T_1 and T_2 are passed to **CorrectOrder**, which subdivides T_1 further. The top of the figure demonstrates the different values of T and i parameters passed to the recursive invocations of **CorrectOrder**.

3.4 Algorithms for finding reverse nearest neighbors

In this section, we present the algorithms for finding the reverse nearest neighbors and the reverse k nearest neighbors of a query point.

3.4.1 FindRNN algorithm Algorithm **FindRNN** computes the reverse nearest neighbors for a continuously moving point in the plane. The notation used is the same as in the previous sections. The algorithm, shown in Figure 9, 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 algorithms we use this format in the rest of the paper. 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 time intervals formed.

To reduce the disk I/O incurred by the algorithm, all the six sets of candidate *RNN* points (the answer lists B_i) are found in a single index traversal. In steps 3.2 and 3.4.1 of the **FindNN** algorithm (cf. Figure 5) called from step 1 of **FindRNN**, a rectangle or a point is pruned only if the condition is satisfied for the answer sets of all six regions. In addition, the computation of the squared distance between a bounding rectangle and a query is modified, so that only the part of the rectangle that is inside the region under consideration is taken into account.

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}.

All the *RNN* candidates nn_{ij} found in the first traversal are verified also in one traversal. To make this possible, we use either $\sum_{i,j} M(R, nn_{ij})$ (for the *integral* metric) or $\min_{i,j} M(R, nn_{ij})$ (for the *min* metric) as the aggregate metric in step 3.3 of **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 **FindRNN**, we observe that in the worst case, all nodes of the tree are visited to find the nearest neighbors using **FindNN**, which is performed twice. As noted by Hjaltason and Samet [12], this

FindkNN$(q,\ [t^{\vdash}; t^{\dashv}],\ k)$:
1 Set $l \leftarrow 0$; $\forall t \in [t^{\vdash}; t^{\dashv}]$, set $min_q(t) \leftarrow ()$.
Steps 2–3.3 are the same as in **FindNN**.
3.4 Else if e points to a leaf node, for each p contained in it, such that $p \neq q$:
3.4.1 If $\forall t \in [t^{\vdash}; t^{\dashv}](d_q(p,t) \geq dmin_q(t))$, skip p.
3.4.2 $\forall t \in T',\ T' \subset [t^{\vdash}; t^{\dashv}](d_q(p,t) < dmin_q(t))$:
3.4.2.1 If $l < k$, set $l \leftarrow l + 1$ and $\forall t \in T'$ $(min_q(t)[l] \leftarrow p)$.
3.4.2.2 Else $\forall t \in T'$ $(min_q(t)[k] \leftarrow p)$.
3.4.2.3 Call **CorrectOrder**(T', l).
CorrectOrder$(T,\ i)$:
1 If $i > 1$ and $\exists T' \subset T$ such that $\forall t \in T'(d_q(min_q(t)[i], t) < d_q(min_q(t)[i-1], t))$:
1.1 $\forall t \in T'$, exchange $min_q(t)[i]$ and $min_q(t)[i-1]$. Call **CorrectOrder**$(T', i-1)$.

Fig. 7 Algorithm computing k nearest neighbors for moving objects in the plane

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 **FindNN**$(q, [t^{\vdash}; t^{\dashv}])$ for region S_i only. A version of algorithm **FindNN** is used where steps 3.2 and 3.4 are modified to consider only time intervals when R or 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 **FindNN**(nn_{ij}, T_{ij}) to check when during time interval T_{ij}, q is the *NN* point of nn_{ij}. The algorithm **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, in step 3.4.2, 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 $\forall t \in T'$ $(d_{nn_{ij}}(p,t) < d_{nn_{ij}}(q,t))$. **FindNN** stops if its answer list becomes empty.
3.2 If **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 $LRNN$.

Fig. 9 Algorithm computing reverse nearest neighbors for moving objects in the plane

FindRkNN$(q,\ [t^{\vdash}; t^{\dashv}],\ k)$:
Steps 1 and 2 are the same as in **FindRNN**, only **FindkNN** is used instead of **FindNN**.
3 For each B_i, for each $\langle NN_{ij},\ T_{ij}\rangle \in B_i$, and for each $nn \in NN_{ij}$, do:
3.1 Call **FindkNN**(nn, T_{ij}, k) to check when during time interval T_{ij}, q is among the k *NN* points of nn. The algorithm **FindkNN** is modified by setting $min_{nn}(t) \leftarrow (q)\ \forall t \in T_{ij}$ in place of $min_{nn}(t) \leftarrow ()$ in step 1. Note that, if $k > 1$, according to the definition from Section 3.3.3, $dmin_{nn}(t) = \infty$ initially. 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, in the list of nearest neighbors associated with this interval, $p_k = q$ is replaced by another point p in step 3.4.2.2. **FindkNN** stops if its answer list becomes empty.
3.2 If **FindkNN**(nn, T_{ij}, k) returns a non-empty answer list, for each $\langle NN, T'\rangle$ in this list, find the position r of q in the list *NN* and add $\langle nn, r, T'\rangle$ to *LRNN*. Any two elements of *LRNN* with the same *nn* and r and adjacent time intervals T_1 and T_2 are coalesced into one element with the time interval $T_1 \cup T_2$.

Fig. 10 Algorithm computing reverse k nearest neighbors for moving objects in the plane

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 **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 together. In our performance studies in Section 4, we investigate the observed average number of I/Os and the average sizes of result sets.

3.4.2 FindRkNN algorithm By using algorithm **FindkNN**, algorithm **FindRNN** can be extended easily to find the reverse k nearest neighbors. Similarly to the case of $k = 1$, it is easy to show that a point that has the query point among its k nearest neighbors can only be one of the k nearest neighbors of q in one of the six regions S_i. Figure 10 captures the differences between **FindRNN** and **FindRkNN**.

Note that in the algorithm **FindkNN** used from step 1 of **FindRkNN**, the lists min_q of the answer list may have different lengths. In a stand-alone version of **FindkNN**, whenever the l-th data point is visited in the initial stages of tree traversal (when $l < k$), it contributes to all lists min_q in the answer list. In the modified version of **FindkNN**, a visited data point can contribute to a list $min_q(t)$ only if the point is inside the searched region S_i at time t.

Note also that when compared with the elements of the *LRNN* list returned by **FindRNN**, the elements of *LRNN* returned by **FindRkNN** have an additional element—the

rank of the reverse nearest neighbor. A reverse nearest neighbor has rank r, if q is its r-th nearest neighbor. While the ranks are not required by the definition of the *RkNN* query given in Section 2.1, they are helpful for efficiently maintaining the results of the query, as will be described in Section 3.7.

3.5 Query example

To illustrate how an *RNN* query is performed, Figure 11 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, $d_{P_1}(R_2, t)$ is the distance between point 1 and rectangle 2.

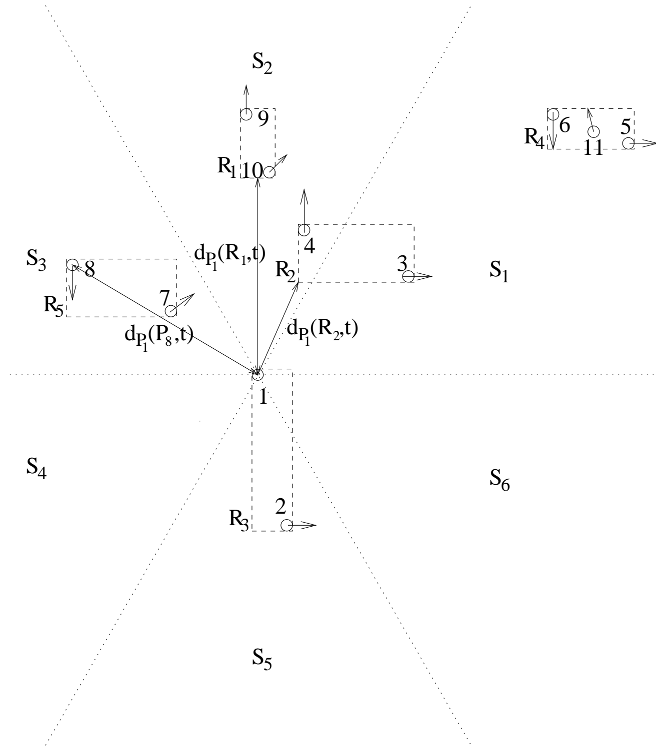

Fig. 11 Example query

If an *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 12 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 $t = 1.5$.

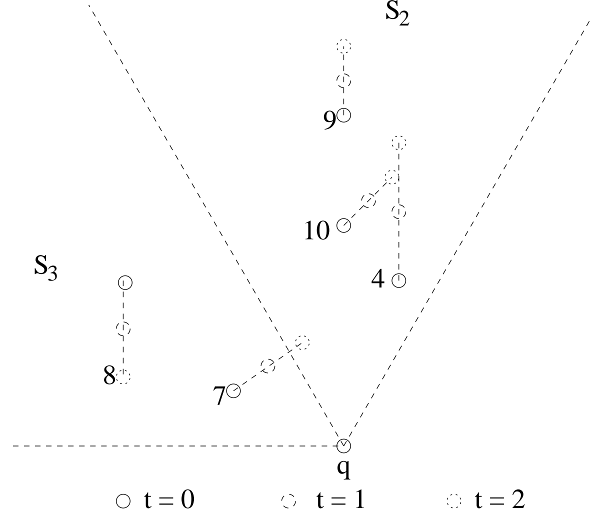

Fig. 12 Simplified example query

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 point closest to point 4, and, similarly, during time interval $[1.5; 2]$, point 7, but not point 1, is the point closest to point 8.

3.6 *Updating the answers to the* NN *algorithms*

In the following two sections, we present algorithms that render the *NN* and *RNN* queries persistent. The algorithms incrementally update the answer set of a query when a point is inserted into or deleted from the database without re-calculating the answer set from scratch. We start with the algorithms for maintaining the result of an *NN* query.

Inserting a new point is the same as visiting a new point in a leaf node of the tree. Thus, to maintain the query result when point p is inserted, it suffices to perform step 3.4 of $\mathbf{FindNN}$ or $\mathbf{FindkNN}$. If the beginning of the query time interval is already in the past, only the remaining part of it that starts from the current time is maintained, i.e., $[t^{\vdash}; t^{\dashv}]$ is replaced by $[\max\{t_{insert}, t^{\vdash}\}, t^{\dashv}]$.

To maintain the result of an *NN* query when a point p is deleted is also simple. If p is not in any of the sets associated with time intervals of the query result, then nothing has to be done. Otherwise, for the elements of the answer list $\langle T_j, NN_j\rangle$ such that $NN_j = \{p\}$, $\mathbf{FindNN}(q, T_j)$ has to be performed. We call such time intervals of the result set the *affected* time intervals. When the result of the *kNN* query is maintained, step 1 of $\mathbf{FindkNN}(q, T_j, k)$ is skipped, and $\{\langle T_j, NN_j\rangle\}$, with point p removed, is used as the initial result list for affected time interval T_j.

Observe that only deletion involves accessing the index to maintain a query result; and this happens only when the deleted point is in the result. Also, the tree traversals associated with different, affected time intervals T_j can be combined into one traversal—in the same way as for the second traversal of the **FindRNN** algorithm (see Section 3.4.1).

3.7 Updating the answers of the RNN *algorithms*

Maintaining the results of *RNN* (and *RkNN*) queries is more difficult than maintaining the results of *NN* queries. We proceed to describe separately how insertions and deletions are processed. In each case, we first consider the simpler case of the *RNN* query, then describe the algorithms for the more complex *RkNN* query.

3.7.1 Insertion of a point The algorithm for updating the answer to a query when a new point is inserted consists of two parts. First, we have to check whether the newly inserted point becomes an *RNN* point of q. Then, we have to check whether the new point invalidates some of the existing *RNN* points, which occurs if the new point is closer to these points than is q.

Suppose that point p is inserted at time t_{insert}, where $t_{current} \leq t_{insert} \leq t^{\dashv}$. Recall that the query is assumed to be issued at time $t_{current}$ and that the query interval ends at $t^{\dashv}$.

The algorithm for maintaining the result of an *RNN* query when insertion is performed is shown in Figure 13. To understand the notation used in step 2 of the algorithm, observe that for each i, there is at most one non-empty time interval during which point p is in S_i. Interval T_i denotes the intersection of this possibly empty interval with the time during which to update the answer. For each region S_i with a non-empty T_i, this step checks if point p becomes an *NN* point of q in that region. If it does, the corresponding B_i list is updated and it is checked for the inclusion of p into $LRNN$. In step 3, those points that have p as their new *NN* point at some time during $[t_0; t^{\dashv}]$ are deleted from $LRNN$ for the corresponding time intervals.

The corresponding algorithm for maintaining the result of an *RkNN* query has the same structure (see Figure 14). In step 2, for each region S_i with a non-empty T_i, the algorithm checks if there are times when p becomes closer to q than the furthest of the k nearest neighbors in that region. If so, the corresponding B_i list is updated, and it is checked for the inclusion of p into $LRNN$. Step3 differs from the corresponding step in Figure 13 in that reverse nearest neighbors are not always removed from the answer list for time periods when p gets closer to them than q. In such cases, only their rank is incremented by one for the corresponding time intervals. Only when the rank of an *RNN* point gets larger than k during some time interval, the *RNN* point is removed from the answer for the corresponding time interval.

Observe that the lists of nearest neighbors B_i are used and updated in both algorithms. Thus, if persistent queries have to be efficiently supported, these lists must be retained after the completion of algorithm **FindRNN**. In addition, the squared-distance functions (expressed by the three parameters described in Section 3.3.1) associated with each of the elements in the B_i and $LRNN$ must be retained.

The algorithms described involve one index traversal in step 2.1, although this traversal should occur only rarely. It is performed only when the inserted point is closer to q than the current nearest neighbors at some time during $[t_0; t^{\dashv}]$. We investigate the amortized cost of the algorithm in empirical performance experiments.

3.7.2 Deletion of a point Three computations are involved when maintaining the answer set $LRNN$ of a query when a point p is deleted. First, if p was in the answer set, it should be removed. Second, to correctly maintain the lists B_i of nearest neighbors, these must be searched for p, which is removed if found. For the time intervals during which p was a nearest neighbor, new *NN* points should be found and checked for inclusion into $LRNN$. Third, those *RNN* candidates from the lists B_i that are not included in $LRNN$ (or are included with reduced time intervals) should be rechecked by the algorithm; this is so because some of them may not have been included into $LRNN$ due to p being their nearest neighbor (with q possibly being their second-nearest neighbor).

We use $\overline{LRNN}$ to denote the list of the above-mentioned candidate points with associated time intervals during which they are not reverse nearest neighbors. More formally:

$$\overline{LRNN} = \{\langle p_l, T_l \rangle \mid \exists\, i, j\ (\langle nn_{ij}, T_{ij} \rangle \in B_i\ \wedge\ p_l = nn_{ij}\ \wedge\ T_l \subseteq T_{ij})\ \wedge\ \nexists\, \langle p', T' \rangle \in LRNN\ (p_l = p'\ \wedge\ T_l \cap T' \neq \emptyset)\}$$

List $\overline{LRNN}$ can be computed by sorting the start and end times of the time intervals in $LRNN$ and the B_i lists, then performing a "time sweep." A binary search tree can be used to store the IDs of all points that have their corresponding time intervals intersect the sweep line. This way, all time intervals from the B_i lists can be subtracted efficiently from the corresponding intersecting time intervals from $LRNN$.

Suppose a data point p (i.e., $p \neq q$) is deleted at time t_{delete} ($t_{current} \leq t_{delete} \leq t^{\dashv}$). The algorithm for maintaining the result of an *RNN* query is given in Figure 15.

In step 2 of the algorithm, p is removed from $LRNN$. In step 3, for each region S_i, p is removed from the list of the nearest neighbors of q in that region for the time period when p is no longer in the set of data points. Also, for each entry removed, new *NN* points of q are found in that region during the time interval when p was the nearest neighbor of q in that region.

In step 4, the points that had p as their nearest neighbor, and q as their second nearest neighbor, are included into $LRNN$. Note that in this step, all new *RNN* candidate points added to the B_i lists in step 3 are also checked for inclusion into $LRNN$. This happens because such points are included in $\overline{LRNN}$ (by definition) and, for each such point p_l, the in-

$\mathbf{Insert}(q,\ [t^{\vdash};t^{\dashv}],\ LRNN,\ p,\ t_{insert})$:
1 Set $t_0 \leftarrow \max\{t_{insert}, t^{\vdash}\}$.
2 Let $T_i \subset [t_0;t^{\dashv}]$ be the time interval when p is in region S_i. For each i such that $T_i \neq \emptyset$, do:
For each $\langle nn_{ij},\ T_{ij}\rangle \in B_i$, such that $T_{ij} \cap T_i \neq \emptyset$, do:
Let $T' = T_{ij} \cap T_i$. Let $T'' \subset T'$ be the time interval during which $d(q,p) < d(q,nn_{ij})$. If $T'' \neq \emptyset$:
2.1 Add $\langle p,\ T''\rangle$ to B_i. Check for inclusion of $\langle p,\ T''\rangle$ into $LRNN$, as described in step 3 of $\mathbf{FindRNN}$.
2.2 Change $\langle nn_{ij},\ T_{ij}\rangle$ to $\langle nn_{ij},\ T_{ij} \setminus T''\rangle$ in B_i.
3 For each $\langle p_l, T_l\rangle \in LRNN$ such that $p_l \neq p$, do:
Let $T' \subset T_l \cap [t_0;t^{\dashv}]$ be the time interval during which $d(p_l,p) < d(p_l,q)$. If $T' \neq \emptyset$, change $\langle p_l, T_l\rangle$ to $\langle p_l, T_l \setminus T'\rangle$.

Fig. 13 Incremental maintenance of *RNN* query answers during insertions of data points

$\mathbf{Insert}(q,\ [t^{\vdash};t^{\dashv}],\ LRNN,\ p,\ t_{insert},\ k)$:
1 Set $t_0 \leftarrow \max\{t_{insert}, t^{\vdash}\}$.
2 Let $T_i \subset [t_0;t^{\dashv}]$ be the time interval when p is in region S_i. For each i such that $T_i \neq \emptyset$, do:
For each $\langle (p_1,p_2,\ldots,p_k),\ T_{ij}\rangle \in B_i$, such that $T_{ij} \cap T_i \neq \emptyset$, do:
Let $T' = T_{ij} \cap T_i$. Let $T'' \subset T'$ be the time interval during which $d(q,p) < d(q,p_k)$. If $T'' \neq \emptyset$:
2.1 Add $\langle (p_1,p_2,\ldots,p_{k-1},p),\ T''\rangle$ to B_i. Call $\mathbf{CorrectOrder}(T'',k)$ on answer list B_i. Check for inclusion of $\langle p,\ T''\rangle$ into $LRNN$, as described in step 3 of $\mathbf{FindRkNN}$.
2.2 Change $\langle (p_1,p_2,\ldots,p_k),\ T_{ij}\rangle$ to $\langle (p_1,p_2,\ldots,p_k),\ T_{ij} \setminus T''\rangle$ in B_i.
3 For each $\langle p_l, r_l, T_l\rangle \in LRNN$ such that $p_l \neq p$, do:
Let $T' \subset T_l \cap [t_0;t^{\dashv}]$ be the time interval during which $d(p_l,p) < d(p_l,q)$. If $T' \neq \emptyset$, change $\langle p_l, r_l, T_l\rangle$ to $\langle p_l, r_l, T_l \setminus T'\rangle$ and, if $r_l < k$, add $\langle p_l, r_l+1, T'\rangle$ to *LRNN*.

Fig. 14 Incremental maintenance of *RkNN* query answers during insertions of data points

$\mathbf{Delete}(q,\ [t^{\vdash};t^{\dashv}],\ LRNN,\ p,\ t_{delete})$:
1 Set $t_0 \leftarrow \max\{t_{delete}, t^{\vdash}\}$.
2 For each $\langle p_l, T_l\rangle \in LRNN$, such that $p_l = p$ and $T_l \cap [t_0;t^{\dashv}] \neq \emptyset$, do:
Change $\langle p_l, T_l\rangle$ to $\langle p_l, T'\rangle$, where $T' = T_l \setminus [t_0;t^{\dashv}]$. If $T' = \emptyset$, remove $\langle p_l, T'\rangle$ from $LRNN$.
3 For each $\langle nn_{ij},\ T_{ij}\rangle \in B_i$ such that $nn_{ij} = p$, do:
3.1 Remove $\langle nn_{ij},\ T_{ij}\rangle$ from B_i.
3.2 Call $\mathbf{FindNN}(q,\ T_{ij})$ for the region S_i. Add the returned points with their corresponding time intervals to B_i.
4 Compute $\overline{LRNN}$. For each $\langle p_l, T_l\rangle \in \overline{LRNN}$ do:
Let $T' \subset T_l \cap [t_0;t^{\dashv}]$ be the time interval during which inequality $d(p_l,p) < d(p_l,q)$ holds. If $T' \neq \emptyset$, check for the inclusion of $\langle p_l, T'\rangle$ into $LRNN$, as described in step 3 of $\mathbf{FindRNN}$.

Fig. 15 Incremental maintenance of *RNN* query answers during deletions of data points

equality $d(p_l,p) < d(p_l,q)$ holds during the corresponding time interval.

Figure 16 shows the a modified version of the algorithm that is able to maintain the result of an *RkNN* query. The first major modification is the additional step 3 in Figure 16, which is not present in Figure 15. This step updates the ranks of those reverse nearest neighbors that, during some intervals of time, are closer to p than to q. When p is removed from in-between such an *RNN* point and q, the rank of the *RNN* point should be decreased by one for the corresponding time interval.

Another difference between the two algorithms is that in step 4.2 in Figure 16, the algorithm $\mathbf{FindkNN}$ does not have to start from scratch—the $k-1$ nearest neighbors of q remain the same during T_{ij}.

Although step 5 in Figure 16 is the same as step 4 in Figure 15, the definition of $\overline{LRNN}$ has to be modified to account for lists of points, instead of single points, associated with time intervals in the B_i result lists. More formally:

$$\overline{LRNN} = \{\langle p_l, T_l\rangle \mid \exists\, i,j,s\ (\langle (p_1,p_2,\ldots,p_k), T_{ij}\rangle \in B_i \wedge p_l = p_s \ \wedge\ T_l \subseteq T_{ij}) \wedge \nexists\, \langle p', r', T'\rangle \in LRNN\ (p_l = p' \ \wedge\ T_l \cap T' \neq \emptyset)\}$$

The same procedure as described for the case of $k = 1$ is used to compute the list $\overline{LRNN}$. Ranks are ignored in this computation.

In contrast to algorithm $\mathbf{Insert}$, algorithm $\mathbf{Delete}$ requires two index traversals in the worst case. One in step 3.2 and another in step 4 (Figure 15). Note that no tree traversals are performed if the deleted point is not in the B_i lists and is further away from the points in the B_i lists than the query point. We investigate the amortized cost of the algorithm in our performance experiments.

3.8 Continuous queries

As stated in Section 2.1, continuous queries are queries with time intervals that advance in step with the continuously progressing current time. In this section, we discuss how to sup-

Delete$(q,\ [t^{\vdash};t^{\dashv}],\ LRNN,\ p,\ t_{delete},\ k)$:
Steps 1 and 2 are analogous to the corresponding steps in Figure 15.
3 For each $\langle p_l, r_l, T_l\rangle \in LRNN$ do:
Let $T' \subset T_l \cap [t_0; t^{\dashv}]$ be the time interval during which inequality $d(p_l, p) < d(p_l, q)$ holds. If $T' \neq \emptyset$, change $\langle p_l, r_l, T_l\rangle$ to $\langle p_l, r_l, T_l \setminus T'\rangle$ and add $\langle p_l, r_l - 1, T'\rangle$ to *LRNN*.
4 For each $\langle (p_1, p_2, \ldots, p_k),\ T_{ij}\rangle \in B_i$ such that $\exists l\ (p_l = p)$, do:
4.1 Remove $\langle (p_1, p_2, \ldots, p_k), T_{ij}\rangle$ from B_i.
4.2 Call $\mathbf{FindkNN}(q, T_{ij}, k)$ for the region S_i. Instead of performing step 1 of **FindkNN**, use $\{\langle (p_1, \ldots, p_{l-1}, p_{l+1}, \ldots, p_k), T_{ij}\rangle\}$ as the initial answer list. Add the returned results to B_i.
Step 5 is the same as step 4 in Figure 15, only **FindRkNN** is used instead of **FindRNN**.

Fig. 16 Incremental maintenance of *RkNN* query answers during deletions of data points

port continuous current-time queries, i.e., those that have $t^{\vdash} = t^{\dashv} = now$.

A continuous current time query issued at time t_{issue} can be supported by computing a persistent query q_l with time interval $[t_{issue}; t_{issue}+l]$. The start and end times of the time intervals in the answer to this query are then the times of scheduled events that update the answer to the continuous query. These event times change as the answer to q_l is maintained under updates. At $t_{issue} + l$, a new persistent query with time interval of length l is computed.

The choice of an optimal l value involves a trade-off between the cost of the computation of q_l and the cost of maintaining its result. On the one hand, it involves a substantial I/O cost to compute even a query with $l = 0$, so we want to avoid frequent recomputations of queries with small l. On the other hand, although computing one or a few queries with large l is cost effective in itself, we must also take into account the cost of maintaining the larger answer set of q_l, which generates substantial additional I/O on each update. So, using queries with large l is also not likely to be efficient.

Let N be the number of moving points and U be the average time duration between two updates of a point. Assume also that we want to maintain the answer to a continuous query from the current time and for a large period of L time units into the future. Then, we want to find a value of l that minimizes function $C(l)$, defined next, that denotes the total cost of maintaining the continuous query.

$$C(l) = \frac{L}{l}\left(Q(l) + \frac{l}{U}NM(l)\right)$$

Here, $Q(l)$ is the cost of computing the persistent query q_l with time interval of length l and $M(l)$ is the amortized cost of a single update (a deletion followed by an insertion) that is required to maintain the answer to q_l. The ratio l/U expresses how many times a point is updated during the life-time of a persistent query and $(l/U)NM(l)$ gives the total cost of maintenance, when updating all N points. Let both $Q(l)$ and $M(l)$ be linear functions. (We verify this assumption in our performance experiments.) Then,

$$\begin{aligned} C(l) &= \frac{L}{l}\left(Q_0 + Q_f l + \frac{l}{U}N(M_0 + M_f l)\right) \\ &= \frac{L}{l}Q_0 + LQ_f + L\frac{N}{U}M_0 + L\frac{N}{U}M_f l. \end{aligned}$$

To minimize $C(l)$, we differentiate C and solve the equation $C'(l) = 0$:

$$C'(l) = L\left(\frac{NM_f}{U} - \frac{Q_0}{l^2}\right) = 0 \quad \Rightarrow \quad l = \sqrt{\frac{Q_0 U}{M_f N}}$$

Observe that Q_0 is the cost of computing q_l, when $l = 0$. The coefficient M_f specifies how fast the cost of one update grows when the length of the maintained persistent query grows. The result obtained is quite intuitive. Ratio U/N is the average time between two updates to the whole database. The larger it is (the smaller the frequency of updates), the cheaper the maintenance of the query result is and the larger l can be. Also, the larger the base cost (Q_0) involved in computing q_l is, the less frequently we want to compute q_l—making a larger l is desirable. Finally, the faster the cost of maintaining q_l grows with the growing l (the rate of growth expressed by M_f), the smaller an l we want.

Parameters Q_0 and M_f are dependent on N and other specifics of the data set, and approximate values for them could be maintained automatically by the query processor. This could be done by monitoring the performance of queries issued by users or by periodically performing a predefined suite of sample queries. Similarly, the value of U could be maintained automatically by monitoring the frequency of updates.

The presented cost model should be applicable to both nearest neighbor and reverse nearest neighbor continuous current-time queries. Our performance experiments, described in the next section (in Section 4.6, in particular), investigate and verify the applicability of this cost model.

4 Performance experiments

This section presents results of experiments with the algorithms presented in the previous section. Following a description of the experimental setup, Sections 4.2 and 4.3 study properties of the *NN* algorithms, with the second of these focusing on persistent *NN* queries. Then two sections consider the *RNN* algorithms. Finally, Section 4.6 considers the continuous versions of both *NN* and *RNN* queries.

4.1 Experimental setting

All algorithms presented in the previous section were implemented in C++, using a TPR-tree implementation based on GiST [10]. Specifically, the TPR-tree implementation with self-tuning time horizon was used [23]. We investigate the performance of the different algorithms in terms of the numbers 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 [18], with the root of a tree always being 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 that accommodates $30,000$ entries, each entry consisting of a moving point, a time interval, and a distance function expressed by three parameter values (cf. Section 3.3.1). This storage is used to record the answer sets and intermediary answer sets (the B_i lists) of persistent queries. The storage is large enough so that these sets always fit in main memory.

If the answer sets or intermediary answer sets were larger than the available main memory, they would need to be maintained on disk. However, this would result in very substantial I/O because access to the entries in these sets is very nonlocal. In the worst case, each time a point or a bounding rectangle is examined, the whole answer set would have to be read from the disk. Thus, our algorithms are not well suited for query answers that do not fit in main memory.

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 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 some of the experiments, we use uniform workloads, where point positions and velocities are distributed uniformly. The speeds of objects vary from 0 to 3 kilometers per time unit (minute). In most of the experiments, more realistic workloads are used, where simulated objects move in a fully connected 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 a destination, it chooses the next target destination at random. Unless specified otherwise, the experiments use workloads from simulations with 20 destinations. The network-based workload generation used in these experiments is described in more detail elsewhere [24].

In both types of workloads, the average interval in-between 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, the workloads are run for additional 60 time units with queries intermixed with the updates. Unless noted otherwise, 600 queries are issued—ten for each time unit.

Note that the update rate implied by this setting may be expected to be in the low end of what may be expected in real-life scenarios. Since our experiments explore the performance of queries, the setting is conservative—for scenarios with higher update rates, queries would be more efficient. This is due to the specifics of the TPR-tree, in which time-parameterized bounding rectangles are "tighter" when more updates happen, leading to better query performance.

For the experiments with the *NN* queries, a query point is generated in the same way as a new data point is generated. For the experiments with the *RNN* queries, each query corresponds to a randomly selected point from the currently active data set. Unless noted otherwise, $k = 1$, and the query time interval is a random interval contained in $[t_{issue}, t_{issue} + 30]$, where t_{issue} is the time when the query is issued.

Our performance graphs report average numbers of I/O operations per query. When query selectivity is given as an average numbers of time intervals in a result, the reported numbers of time intervals in a result is minimal, i.e., the implementations of the algorithms ensure that results are coalesced. For *kNN* queries, this means that for any two consecutive time intervals in the result, the associated *NN* points are different or their orderings are different. For *RkNN* queries this means that in the $LRNN$ answer set, no two elements with the same data point and rank have adjacent time intervals that can be merged into a single interval.

4.2 Properties of the NN algorithms

In the first round of experiments, we explore the four variations of the *NN* algorithm mentioned in Section 3.3.1: *best-first* traversal using the *min* metric, *best-first* traversal using the *integral* metric, *depth-first* traversal using the *min* metric, and *depth-first* traversal using the *integral* metric.

Figure 17 shows how the average number of I/O operations per query changes when the number of indexed points increases. The numbers of I/O operations for all four variants of the algorithm grow as the number of points increases. The results of the experiments show that this increase is proportional to the increase in the average selectivity of queries.

Figure 18 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 where the points and their velocities are distributed uniformly, which, intuitively, corresponds to a very large number of destinations.

Not considering the two extreme workloads—the simulation with two destinations and the uniform workload—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 those reported for range queries on the TPR-tree [24], although they are less pronounced.

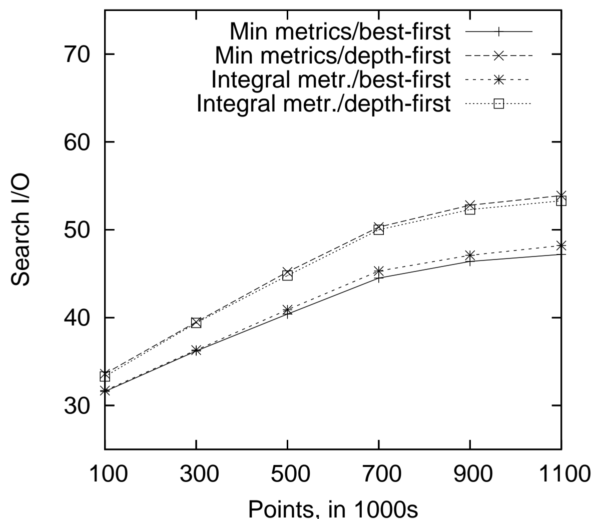

Fig. 17 *NN* query performance for varying number of points

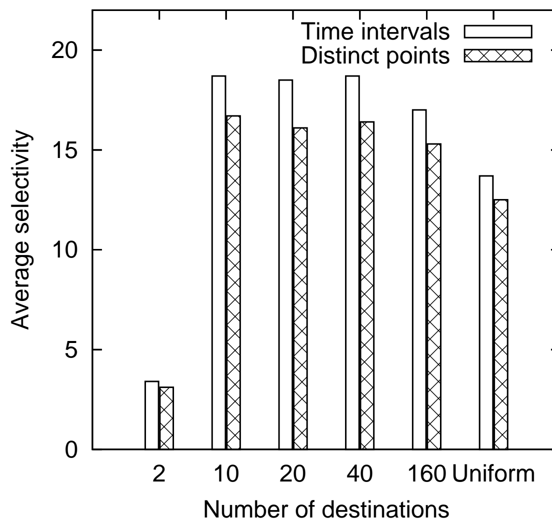

Fig. 19 *NN* query selectivity for varying number of destinations

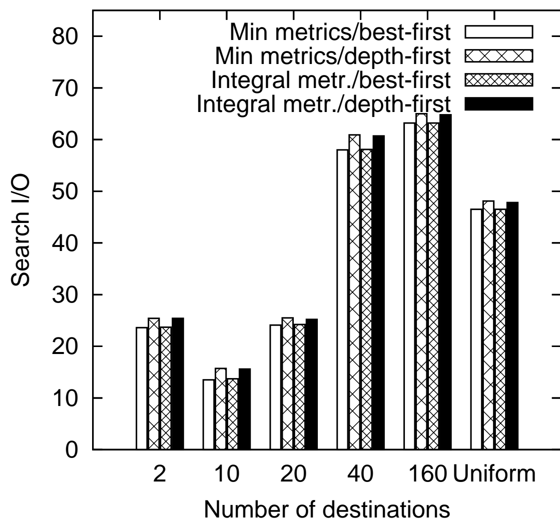

Fig. 18 *NN* query performance for varying number of destinations

Figure 19 explains why the cost of queries in the uniform workload is less than the cost of queries in the workloads with 40 and 160 destinations. In this graph, the average number of time intervals and the average number of distinct points in the query results are plotted. The graph shows that query results are smaller for the uniform workload; hence, less I/O operations are needed to retrieve these results. The smaller results for the uniform workload are most probably due to the specifics of the workload generation—more objects move at the maximum speed of 3 km/min in the non-uniform workloads. This means that temporal query results record more changes in these workloads.

On the other hand, the very small query results for workloads with two destinations does not decrease the cost of queries in these workloads. On the contrary, queries in these workloads are more expensive than those in the workloads with ten destinations (cf. Figure 18). This is explained by the fact that objects in a two-destination simulation move on one one-dimensional road and that the TPR-tree is not well suited for such one-dimensional datasets. For example, in our experiments, the overlaps among the bounding boxes of the TPR-tree at the end of the two-destination workload is 2.8 times larger than the overlaps at the end of the ten-destination workload.

Figures 17 and 18 show that the best performance is achieved by the variants of the *NN* algorithm that use the *best-first* tree traversal. It can also be observed that the performance differences among the four variants of the algorithm are quite small. To understand why this is so, and to learn whether the *NN* algorithm could possibly be significantly improved, we explored how many of the performed I/O operations corresponded to the reading of tree nodes with bounding rectangles that actually contained the query point at some time point during the corresponding query time interval. Such tree nodes, which we term *covering*, must necessarily be visited by any *NN* algorithm to produce the correct answer. Thus, given a specific TPR-tree, the number of I/O operations corresponding to the covering tree nodes gives the lower performance bound for a corresponding nearest neighbor query.

For the uniform workload, 45.4 out of 46.5 I/O operations corresponded to the covering tree nodes. For the 20-destination workload, the two numbers were 23 and 24.1. This demonstrates that to improve the performance of the *NN* algorithms, the underlying index structure has to be improved, not the query algorithms. The notable exception was the two-destination workload, where, in our experiments, only 16.4 out of 23.6 I/O operations corresponded to the covering tree nodes.

Figures 20 and 21 show how different variants of the *NN* algorithm perform when varying k and the length of the query time interval. For the experiment with the varying query time interval, all query intervals in a workload are of the same length and start at the time when the corresponding query is issued. The graphs show results that are consistent with

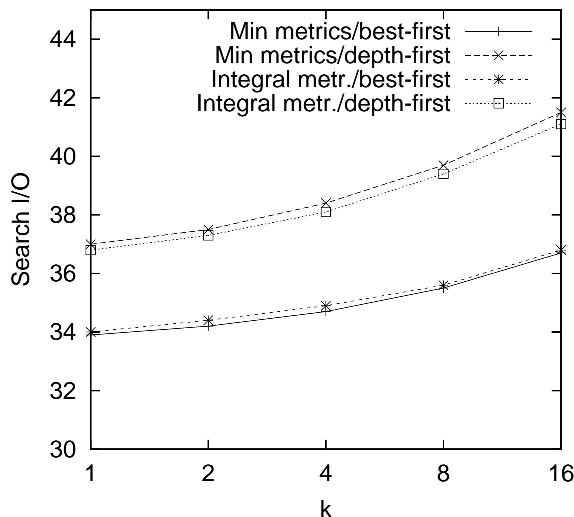

Fig. 20 *NN* query performance for varying k

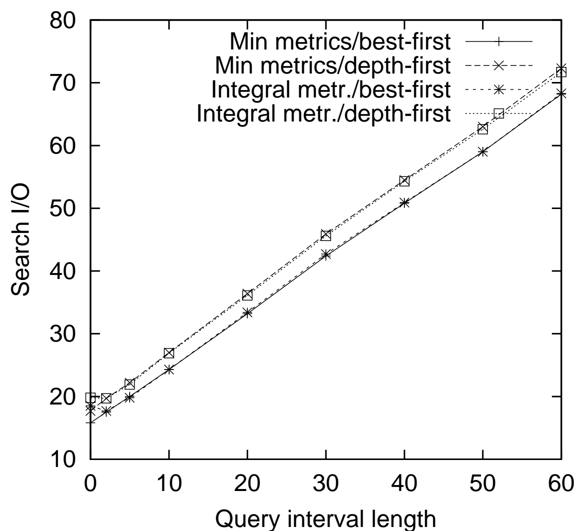

Fig. 21 *NN* query performance for varying query interval length

the results shown in Figure 18. Note that the performance of the *kNN* queries decreases only slightly with an increasing k. This is in spite of the increase of the average size of the returned query results—from 20.6 distinct points per result for $k = 1$ to 109 distinct points per result for $k = 16$.

The graph in Figure 21 validates the assumption from Section 3.8, that the cost of queries grows approximately linearly with the increasing query interval length.

4.3 Persistent NN queries

To evaluate the cost of maintaining the results of persistent *NN* queries, a round of workloads was run where the results of queries were maintained during insertions and deletions. Each workload contains 60 queries, one per time unit, starting at 120 time units after the start of the workload. The time interval of each query starts when the query is issued and all queries in a workload have the same query interval length. As described in Section 3.6, maintaining query results under insertions does not cost any I/O. Figure 22 shows the average cost of maintaining one query result when a deletion is performed.

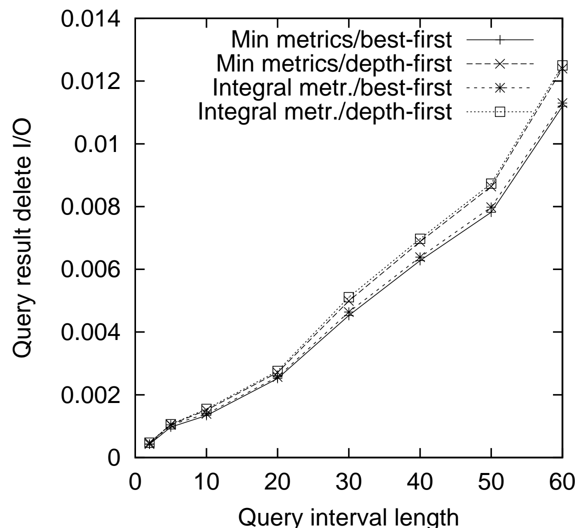

Fig. 22 The cost of maintaining *NN* queries of different length

The graphs show that the cost is low and, consistent with the other results, *best-first* that traversal slightly outperforms *depth-first* traversal. The average cost is low mainly because no I/O is necessary for most of the query result updates. For example, only 77 out of 853,228 query result deletions required I/O for the query interval length of 10 units. For the minimum metrics and best-first traversal, considering only these 77 deletions with non-zero I/O, the average cost of query result deletion is 15.3, which has to be compared to 25.4 I/Os on average for performing a query (see Figure 21).

Finally, note that when ignoring large query interval lengths, the average cost of query maintenance grows almost linearly, as assumed in Section 3.8.

4.4 Properties of the RNN algorithm

Another batch of experiments aims to explore a variety of the properties of the algorithms computing the reverse nearest neighbors. Based on the results of the experiments with the different variants of the *NN* algorithm, the *RNN* algorithms use the *NN* algorithm with the *best-first* traversal and the *min* metric.

To test the scalability of the *RNN* algorithm, the number of points in the database was varied. Figure 23 shows the average number of I/O operations per *RNN* query for workloads with varying database size.

The number of I/O operations increases with the number of data points. The increase is most probably due to two factors. First, as the size of the database increases, different queries are more likely to “touch” different parts of the

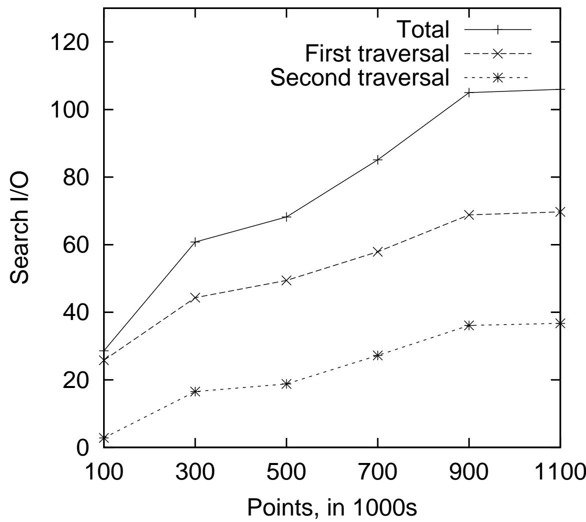

Fig. 23 *RNN* query performance for varying number of points

dataset, and the probability that one query will benefit from the disk pages left in the buffer from the execution of another query is reduced. Second, increasing numbers of *RNN* candidates are retrieved and checked.

Figure 24 plots the sizes of the B_i lists, which store the candidate *RNN* points, and the $LRNN$ set, which stores the final result, in terms of both the number of time intervals and the number of distinct points. The graph shows that while

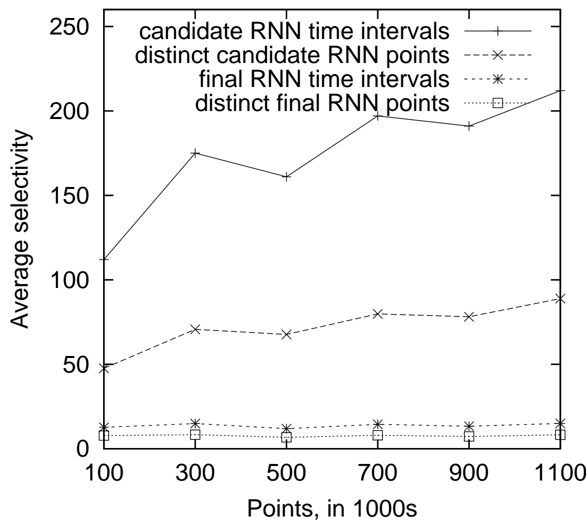

Fig. 24 Average selectivity of *RNN* queries for varying number of points

the average size of the final result remains almost the same as the database size increases, the number of *RNN* candidates increases. The graph also shows that only every sixth *RNN* candidate point becomes an *RNN* point (for the workload of 100,000 points). This ratio, of course, depends on the workload. In our experiments with the uniform workloads of 100,000 points, the ratio is about 3.3. Theoretically, the number of distinct *RNN* candidate points may be as large as the size of the dataset (see Section 3.4.1). However, for our workloads, the maximum sizes of the answer sets (both in terms of intervals and distinct points) are larger than the average answer set sizes by about a factor of 10, which makes them between two and three orders of magnitude smaller than the dataset size. For example, for the dataset of 1,100,000 points, the query with the largest number of candidate *RNN* time intervals required storing 2,165 time intervals with associated *RNN* candidates in the B_i lists.

Returning to Figure 23, observe that the second traversal of the tree, in which the candidates produced by the first traversal are verified, is cheaper than the first traversal, in which these candidates are found. This is as expected. Because although the second traversal involves multiple query points, all these query points are close to the original query point, making it very likely that the disk pages brought into the buffer by the first traversal can be reused in the second traversal, thus reducing the number of I/O operations.

Another factor also contributes to making the second traversal cheaper. 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 **FindNN** algorithm. The second traversal only needs to determine whether the point q is an *NN* point of 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 of q in some region S_i at some time, the distance between q and nn_{ij} is typically small. This enables more aggressive pruning of tree nodes during the second traversal of the TPR-tree.

Figure 25 shows the average number of I/O operations per *RNN* query when the number of destinations in the simulated network of routes is varied and for the uniform workload. Both the performance of the first traversal and the performance of the second traversal follow the same trends as observed for the *NN* queries in Section 4.2.

Figure 26 shows the performance of the *RkNN* queries for different values of k. As for the *kNN* queries, the performance decreases only slightly as k increases. Next, Figure 27 plots the average selectivity of the two traversals of the *RkNN* queries. Note that for $k = 16$, the average number of time intervals in the final answer surpasses the number of time intervals in the B_i lists of *RNN* candidates. This is possible because one time interval in a B_i list is associated with k *RkNN* candidates, each of which may turn into several *RkNN* points that are then associated with different time intervals. In the experiment with $k = 16$, we observed the largest combined size of the B_i lists for an *RkNN* query, namely $6,272$ time intervals (which exceeds the average by a factor of about 10).

Figure 28 shows the average number of I/O operations per query for varying query interval lengths. The setup of the experiment is the same as that of the corresponding experiment for *NN* queries (cf. Section 4.2). The graph shows that for small query interval lengths, the second traversal incurs almost no I/O, which confirms the importance of a buffer. The

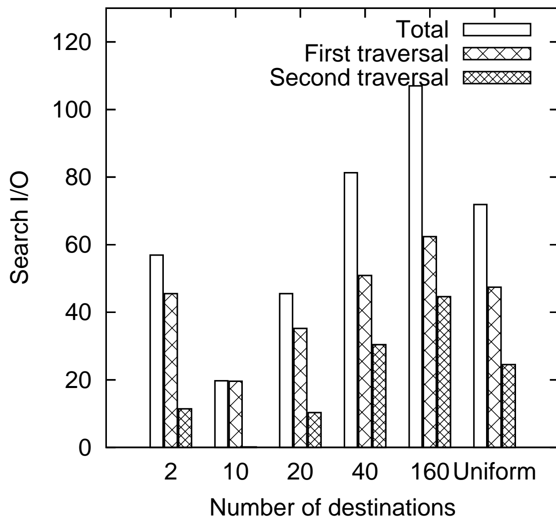

Fig. 25 *RNN* query performance for varying number of destinations

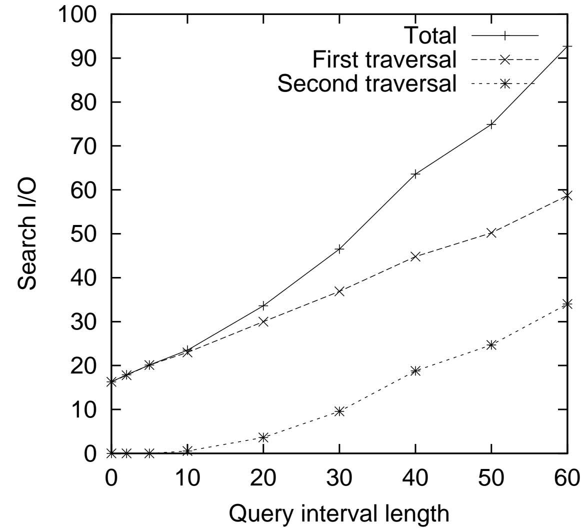

Fig. 28 *RNN* query performance for varying query interval length

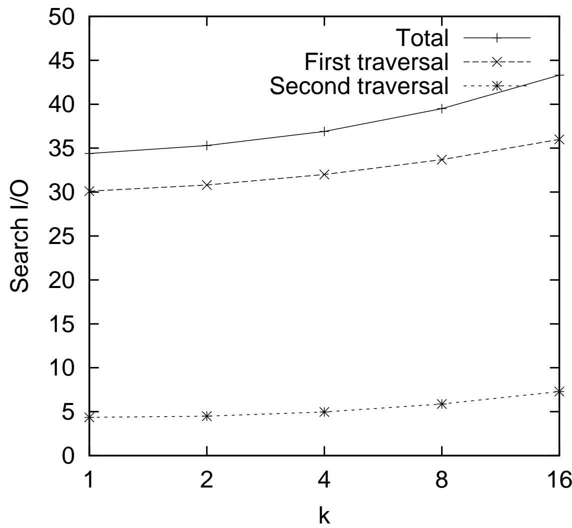

Fig. 26 *RNN* query performance for varying k

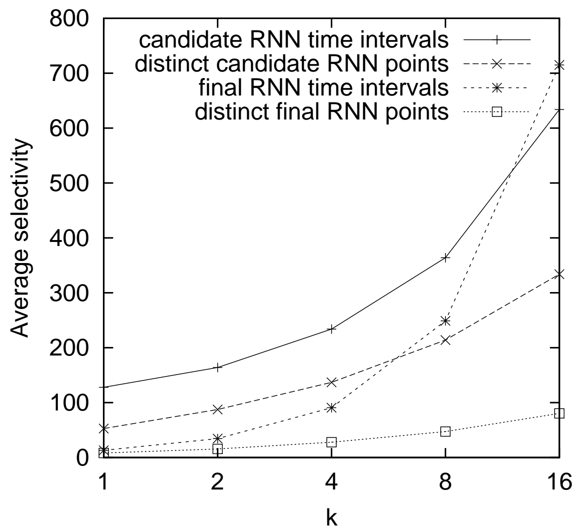

Fig. 27 *RNN* query selectivity for varying k

number of I/O operations increases approximately linearly with the query interval length. The experiment also shows that the number of query results also increases approximately linearly.

4.5 Persistent RNN queries

To investigate the cost of maintaining persistent *RNN* queries, we varied the query interval lengths in an experiment with a setup that is the same as for the analogous experiment with the persistent *NN* queries (cf. Section 4.3). Figure 29 shows the average, amortized cost per single insertion or deletion of maintaining one query result set.

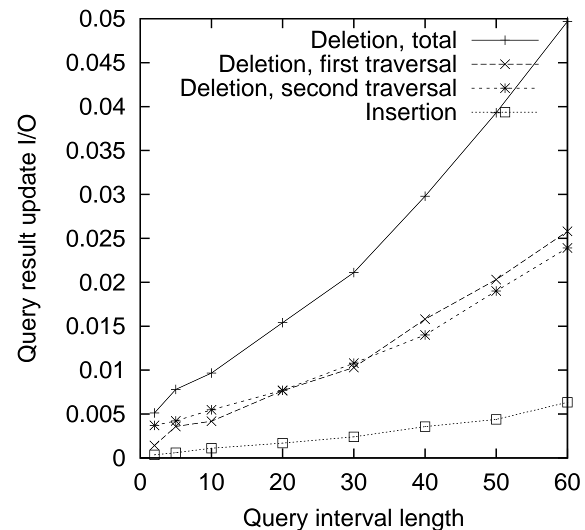

Fig. 29 The cost of maintaining *RNN* queries of different lengths

The graph demonstrates that maintaining *RNN* query results under insertions incurs very little amortized I/O. As men-

tioned at the end of Section 3.7.1, this is because traversals of the tree in algorithm **Insert** are quite rare. Interestingly, deletions are much more costly. Algorithm **Delete** involves two tree traversals—one to find new candidate *RNN* points, if one was deleted, and another to check whether some of the *RNN* candidates become actual *RNN* points, when a point close to these candidates is deleted. As the graph shows, contrary to the *RNN* query algorithm, the two traversals have similar amortized cost. A possible reason for this behavior is that the probability that some *RNN* candidate is deleted is lower than the probability that a point is deleted that at some time during the query interval gets close to some *RNN* candidate. The latter condition requires rechecking of such *RNN* candidates (step 4 in Figure 15). Thus, the second traversal is performed more often than the first traversal.

Comparison of the absolute numbers in Figures 22 and 29 shows that maintaining the results of *RNN* queries is more expensive than maintaining the results of *NN* queries, and is so by more than an order of magnitude. This is mainly so because a larger number of *RNN* query result updates incur non-zero I/O than *NN* query result updates do. For example, for query interval length 10, 2,013 out of 874,983 *RNN* query result deletions incurred I/O, while only 77 out of the similar number of *NN* query result deletions incurred I/O (see Section 4.3). On the other hand, each non-zero-I/O query result deletion costs less for *RNN* queries than for *NN* queries (4.2 I/Os vs. 15.3 I/Os for the same settings).

Figure 29 also shows that the amortized cost per update increases approximately linearly with the length of the maintained query interval. This is as could be expected.

4.6 Continuous queries

Section 3.8 describes a cost model for choosing the optimum query re-computation interval length l when maintaining a continuous current-time query. Recall that a continuous query is maintained by means of a persistent query that extends from the time it is issued and l time units into the future. With a small l, a new persistent query must be computed frequently, which is expensive. But each query result is also relatively small and thus cheap to maintain as updates to the underlying data occur. With a large l, few persistent queries need to be computed, but the ones that are computed have large results that are expensive to maintain. So a small l is expected to result in high recomputation cost and low maintenance cost, while the opposite is expected for a large l.

To empirically understand the effect of different l values, we performed a series of experiments where we varied the length of the query re-computation interval. For each l value used, 20 queries were issued at time 120 and then maintained for 60 time units.

Figures 30 and 31 show the amortized cost per single update operation (insertion or deletion) while maintaining, respectively, one *NN* and one *RNN* continuous query for workloads of $100,000$ and $500,000$ points. Observe that the amortized cost per single update is lower for the larger dataset.

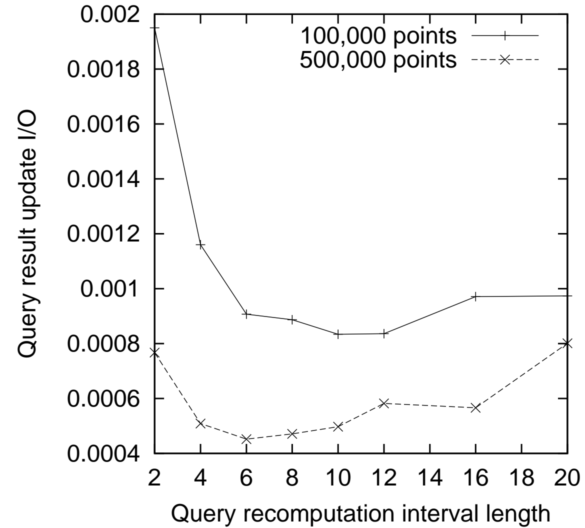

Fig. 30 The cost of maintaining continuous current time *NN* queries

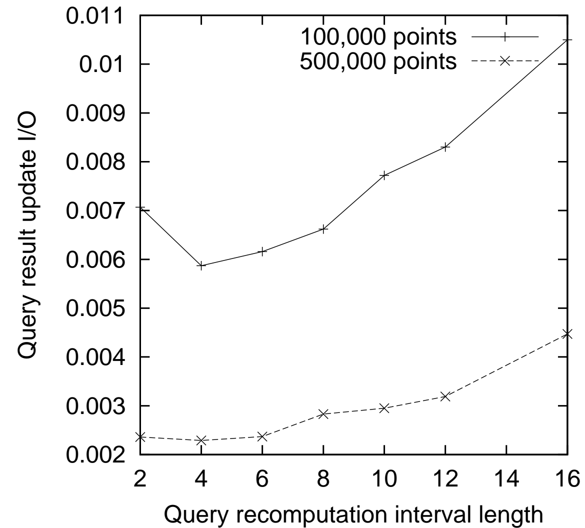

Fig. 31 The cost of maintaining continuous current time *RNN* queries

This is so because a larger number of points leads to a lower probability that a specific update will have an effect on a specific query result. In spite of this, when the total cost of query maintenance is computed by adding the maintenance costs for all updates, the $500,000$ points workload has the larger total query maintenance cost than the $100,000$ points workload, although the increase is by less than a factor of 5.

For the nearest neighbor queries, the best l for the workload with $100,000$ points seems to be approximately 10. For the workload with $500,000$ points, the best l value is approximately 6. For the reverse nearest neighbors, the best l for both workloads seems to be approximately 4.

The performance experiments with varying query interval lengths (cf. Sections 4.2–4.5) validated the assumptions that functions $Q(l)$ and $M(l)$ from the cost model presented in Section 3.8 are approximately linear. To compare the empirically observed optimal values of l with the ones computed

using the cost model presented in Section 3.8, we estimated the values of parameters Q_0 and M_f from the performance experiments with varying query interval length.

For the *NN* queries (see Figures 21 and 22), $Q_0 \approx 15.8$ and $M_f \approx 0.00015$. According to our workload generation parameters, $U = 60$. For $100,000$ points, the cost model gives $l \approx 7.9$, which is quite close to the empirically observed value of 10. For the *RNN* queries (see Figures 28 and 29), $Q_0 \approx 16.3$ and $M_f \approx 0.00063$. This gives $l \approx 3.9$, which agrees very well with the empirically observed value of 4. These results indicate that the mathematical cost model is practical.

5 Summary and future work

Rapid technological advances promise to enable the tracking of the positions of large populations of continuously moving objects. Consequently, efficient algorithms for answering various queries about continuously moving objects are of interest. Algorithms have previously been suggested for answering nearest neighbor and reverse nearest neighbor queries for non-moving objects, but no solutions have been proposed for efficiently answering these queries for large populations of continuously moving objects.

This paper proposed algorithms that enable the computation of nearest and k nearest neighbor queries as well as reverse and reverse k nearest neighbor queries for this setting. Each such query takes as parameter a time interval that extend from some time not preceding the current time and until some later time, and the algorithm computing the query produces a result that contains the nearest or reverse nearest neighbors for each point in time during this interval. Three variants of these types of queries are supported: A standard (one-time) variant that simply returns its result; a persistent variant, where the result is updated incrementally to account for updates of the underlying data; and a continuous variant, where the result as of the changing current time is maintained for a duration of time.

The algorithms utilize the standard TPR-tree [24] as an index on the argument moving objects. This means that a single index structure, be it the TPR-tree, the TPR*-tree, or a similar index, can be used for range queries, nearest neighbor queries, and reverse nearest neighbor queries. Variants of the algorithms were developed that use depth-first and best-first search in the index structure. A comprehensive empirical performance study was conducted that offered insight into a range of performance-related properties of all the different algorithms. Key findings include that best-first search is slightly better than depth-first search; that performance decreases linearly with growing query-interval length, but is relatively unaffected by increases in k; and that the amortized cost of an update for persistent-query maintenance is very low, particularly for nearest neighbor queries.

The presented reverse nearest neighbor algorithms are suitable for the *monochromatic* case [16] 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, e.g., tourists and rescue workers), and a reverse nearest neighbor query asks for points that belong to the opposite category than the query point and that 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 reverse (first) nearest neighbor points at a single point in time. An interesting future research direction is to develop algorithms for efficiently answering reverse nearest neighbor queries for continuously moving bichromatic points.

Next, we have only considered metric distance functions in this paper. But settings exist where other notions of distance are also meaningful. For example, the objects considered may be assumed to move along some underlying transportation network structure—they may be vehicles in a road network. Or the objects may move more freely, with different types of infrastructure, such as lakes, mountains, or farmland, prohibiting movement in some areas. While Euclidean distance may be relevant in such settings, it is also highly relevant to study how to handle the complexities arising from the non-Euclidean and non-metric distance functions that exist in such settings.

Acknowledgments

This research was supported in part by grants from the Danish National Centre for IT research, the Nordic Academy for Advanced Study, and the Nykredit Corporation.

References

1. Albers G, Guibas LJ, Mitchell JSB, Roos T (1998) Voronoi diagrams of moving points. International Journal of Computational Geometry and Applications 8(3): 365–380
2. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R*-tree: An efficient and robust access method for points and rectangles. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 322–331
3. Benetis R, Jensen CS, Karciauskas G, Šaltenis S (2002) Nearest neighbor and reverse nearest neighbor queries for moving objects. In: Proceedings of the International Data Engineering and Applications Symposium, pp 44–53
4. Berchtold S, Ertl B, Keim DA, Kriegel HP, Seidl T (1998) Fast nearest neighbor search in high-dimensional space. In: Proceedings of the International Conference on Data Engineering, pp 209–218
5. Cheung KL, Fu AW-C (1998) Enhanced nearest neighbour search on the R-tree. ACM SIGMOD Record 27(3): 16–21
6. Čivilis A, Jensen CS, J, Pakalnis S. Techniques for Efficient Tracking of Road-Network-Based Moving Objects. IEEE Transactions on Knowledge and Data Engineering 17(5), 15 pages, to appear
7. Elliott J (2001) Text messages turn towns into giant computer game. Sunday Times, April 29
8. Federal Communications Commission. Enhanced 911. URL: http://www.fcc.gov/911/enhanced/. Current as of June 1, 2003.

9. Guttman A (1984) R-trees: A dynamic index structure for spatial searching. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 47–57
10. Hellerstein JM, Naughton JF, Pfeffer A (1995) Generalized search trees for database systems. In: Proceedings of the VLDB Conference, pp 562–573
11. Henrich A (1994) A distance scan algorithm for spatial access structures. In: Proceedings of the Second ACM Workshop on Geographic Information Systems, pp 136–143
12. Hjaltason GR, Samet H (1999) Distance browsing in spatial databases. ACM Transactions on Database Systems 24(2): 265–318
13. Jensen CS (2002) (ed) Indexing of Moving Objects. Special issue of the IEEE Data Engineering Bulletin 25(2)
14. Katayama N, Satoh S (1997) The SR-tree: An index structure for high-dimensional nearest neighbor queries. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 369–380
15. Kollios G, Gunopulos D, Tsotras VJ (1999) Nearest neighbor queries in a mobile environment. In: Proceedings of the International Workshop on Spatio-Temporal Database Management, pp 119–134
16. Korn F, Muthukrishnan S (2000) Influence sets based on reverse nearest neighbor queries. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 201–212
17. Korn F, Sidiropoulos N, Faloutsos C, Siegel E, Protopapas Z (1996) Fast nearest neighbor search in medical image databases. In: Proceedings of the VLDB Conference, pp 215–226
18. Leutenegger ST, Lopez MA (1998) The effect of buffering on the performance of R-trees. In: Proceedings of the International Conference on Data Engineering, pp 164–171
19. Maheshwari A, Vahrenhold J, Zeh N (2002) On reverse nearest neighbor queries. In: Proceedings of the Canadian Conference on Computational Geometry, pp 128–132
20. Preparata FP, Shamos MI (1993) Computational geometry. An introduction. Texts and Monographs in Computer Science. 5th corrected ed., Springer
21. Raptopoulou K, Papadopoulos A, Manolopoulos Y (2003) Fast Nearest-Neighbor Query Processing in Moving-Object Databases. GeoInformatica 7(2): 113–137
22. Roussopoulos N, Kelley S, Vincent F (1995) Nearest neighbor queries. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 71–79
23. Šaltenis S, Jensen CS (2002) Indexing of moving objects for location-based services. In: Proceedings of the International Conference on Data Engineering, pp 463–472
24. Šaltenis S, Jensen CS, Leutenegger ST, Lopez MA (2000) Indexing the positions of continuously moving objects. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 331–342
25. Seidl T, Kriegel HP (1998) Optimal multi-step k-nearest neighbor search. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 154–165
26. Singh A, Ferhatosmanoglu H, Tosun A (2003) High dimensional reverse nearest neighbor queries. In: Proceedings of ACM CIKM International Conference on Information and Knowledge Management, pp 91–98
27. Sistla AP, Wolfson O, Chamberlain S, Dao S (1997) Modeling and querying moving objects. In: Proceedings of the International Conference on Data Engineering, pp 422–432
28. Smid M (1997) Closest point problems in computational geometry. In: Sack JR, Urrutia J (eds) Handbook on computational geometry. Elsevier Science Publishing, pp 877–935
29. Song Z, Roussopoulos N (2001) K-nearest neighbor search for moving query point. In: Proceedings of the International Symposium on Spatial and Temporal Databases, pp 79–96
30. Stanoi I, Agrawal D, El Abbadi A (2000) 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
31. Tao Y, Papadias D (2003) Spatial queries in dynamic environments. ACM TODS 28(2): 101–139
32. Tao Y, Papadias D, Sun J (2003) The TPR*-tree: an optimized spatio-temporal access method for predictive queries In: Proceedings of the VLDB Conference, pp 790–801
33. Tao Y, Papadias D, Lian X (2004) Reverse kNN search in arbitrary dimensionality In: Proceedings of the VLDB Conference, pp 744–755
34. White DA, Jain R (1996) Similarity indexing with the SS-tree. In: Proceedings of the International Conference on Data Engineering, pp 516–523
35. Yang C, Lin K-Ip (2001) An index structure for efficient reverse nearest neighbor queries. In: Proceedings of the International Conference on Data Engineering, pp 485–492
36. Wolfson O, Xu B, Chamberlain S, Jiang L (1998) Moving objects databases: Issues and solutions. In: Proceedings of the International Conference on Scientific and Statistical Database Management, pp 111–122

A Distance Computation for Moving Points and Time-Parameterized Rectangles

First, we provide the formula for the squared distance $d_q(p,t)$ between two d-dimensional moving points:

$$q = (x_1, x_2, \ldots, x_d, v_1, v_2, \ldots, v_d)$$
$$p = (y_1, y_2, \ldots, y_d, w_1, w_2, \ldots, w_d)$$

Here, the x_i and y_i, when not used as functions, are coordinates at time $t = 0$. The squared distance is then given as follows:

$$\begin{aligned} d_q(p,t) &= \sum_{i=1}^{d}(x_i(t) - y_i(t))^2 = \sum_{i=1}^{d}(x_i + v_i t - y_i - w_i t)^2 \\ &= t^2 \sum_{i=1}^{d}(v_i - w_i)^2 + 2t \sum_{i=1}^{d}(x_i - y_i)(v_i - w_i) \\ &\quad + \sum_{i=1}^{d}(x_i - y_i)^2 \end{aligned}$$

Next, let a time-parameterized rectangle be given as follows:

$$\begin{aligned} R = (&[x_1^{\vdash}; x_1^{\dashv}], [x_2^{\vdash}; x_2^{\dashv}], \ldots, [x_d^{\vdash}; x_d^{\dashv}], \\ &[v_1^{\vdash}; v_1^{\dashv}], [v_2^{\vdash}; v_2^{\dashv}], \ldots, [v_d^{\vdash}; v_d^{\dashv}]) \end{aligned}$$

The shortest squared distance $d_q(R,t)$ between moving point q and time-parameterized rectangle R during time interval

Distance$(q,\ R,\ [t^{\vdash}; t^{\dashv}])$:
1 Set $E \leftarrow \emptyset$.
2 For each dimension $i = 1, \ldots, d$, do:
If $v_i \neq v_i^{\vdash}$ and $t_i^{\vdash} = (x_i - x_i^{\vdash})/(v_i^{\vdash} - v_i) \in [t^{\vdash}; t^{\dashv}]$, add $t_i^{\vdash}$ to E.
If $v_i \neq v_i^{\dashv}$ and $t_i^{\dashv} = (x_i - x_i^{\dashv})/(v_i^{\dashv} - v_i) \in [t^{\vdash}; t^{\dashv}]$, add $t_i^{\dashv}$ to E.
3 Sort E. The elements of E divide $[t^{\vdash}; t^{\dashv}]$ into at most $2d + 1$ intervals. For each such interval T_j:

$$d_q(R,t) = \sum_{i=1}^{d} d_{q,i}(R,t),$$

where

$$d_{q,i}(R,t) = \begin{cases} t^2(v_i^{\vdash} - v_i)^2 + 2t(x_i^{\vdash} - x_i)(v_i^{\vdash} - v_i) + (x_i^{\vdash} - x_i)^2 & \textbf{if } \forall t \in T_j (x_i + v_i t \leq x_i^{\vdash} + v_i^{\vdash} t) \\ t^2(v_i^{\dashv} - v_i)^2 + 2t(x_i^{\dashv} - x_i)(v_i^{\dashv} - v_i) + (x_i^{\dashv} - x_i)^2 & \textbf{if } \forall t \in T_j (x_i + v_i t \geq x_i^{\dashv} + v_i^{\dashv} t) \\ 0 & \textbf{otherwise} \end{cases}$$

Fig. 32 Distance computation

$[t^{\vdash}; t^{\dashv}]$ is a piece-wise quadratic function. The algorithm computing this function is given in Figure 32.

In step 2, the algorithm computes the times when the moving point q crosses the moving hyper-planes $x_i = x_i^{\vdash}(t)$ and $x_i = x_i^{\dashv}(t)$—the extensions of those two of R's opposite sides that are perpendicular to the x_i axis (see Figure 33, which also enumerates the $2d+1$ possible subdivisions). Note

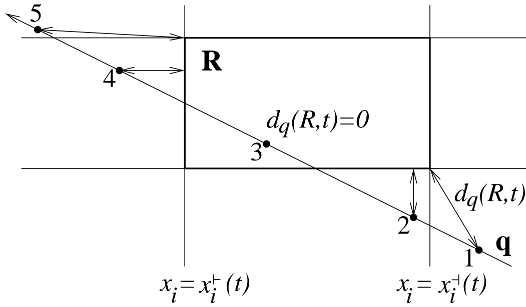

Fig. 33 Distance between a moving point q and a time-parameterized rectangle R

that here, $t_i^{\vdash}$ is not necessarily less than $t_i^{\dashv}$. In step 3, during each of the T_j, q does not cross any of the above-mentioned hyperplanes. From the formulas in step 3, it is quite straightforward to obtain the parameters a, b, and c mentioned in Section 3.3.1.

Observe that for the time periods where q is inside R, $d_q(R,t) = 0$.