Algorithms for Nearest Neighbor Search on Moving Object Trajectories[*]

Elias Frentzos[1], Kostas Gratsias[1,2], Nikos Pelekis[1], Yannis Theodoridis[1,2,†]

[(1)] Information Systems Laboratory
Department of Informatics, University of Piraeus
Piraeus, Greece
URL: http://isl.cs.unipi.gr/db
E-mail: {efrentzo, gratsias, npelekis, ytheod}@unipi.gr

[(2)] Data and Knowledge Engineering Group
R. A. Computer Technology Institute
Patras, Greece
URL: http://dke.cti.gr
E-mail: {gratsias, ytheod}@cti.gr

Abstract

Nearest Neighbor (NN) search has been in the core of spatial and spatiotemporal database research during the last decade. The literature on NN query processing algorithms so far deals with either stationary or moving query points over static datasets or future (predicted) locations over a set of continuously moving points. With the increasing number of Mobile Location Services (MLS), the need for effective *k*-NN query processing over historical trajectory data has become the vehicle for data analysis, thus improving existing or even proposing new services. In this paper, we investigate mechanisms to perform NN search on R-tree-like structures storing historical information about moving object trajectories. The proposed (depth-first and best-first) algorithms vary with respect to the type of the query object (stationary or moving point) as well as the type of the query result (historical continuous or not), thus resulting in four types of NN queries. We also propose novel metrics to support our search ordering and pruning strategies. Using the implementation of the proposed algorithms on two members of the R-tree family for trajectory data (namely, the TB-tree and the 3D-R-tree), we demonstrate their scalability and efficiency through an extensive experimental study using large synthetic and real datasets.

1. Introduction

With the integration of wireless communications and positioning technologies, the concept of Moving Object Databases (MOD) has become increasingly important, and has posed a great challenge to the database community. In such implicitly formulated location-aware environments, moving objects are continuously changing locations; nevertheless existing DBMSs are not well equipped to handle continuously changing data. Emerging location-dependent services call for new query processing algorithms and techniques to deal with both the spatial and temporal domains. Examples of these new services include traffic monitoring, nearby information accessing and enhanced 911 services.

Unlike traditional databases, MODs have some distinctive characteristics: First of all, several spatiotemporal queries in MODs are by nature continuous. In contrast to snapshot queries, which are invoked

[*] This work is an extended version of [FGPT05].
† Contact author's address: 80 Karaoli-Dimitriou St., GR-18534 Piraeus, Greece. Tel: +30-2104142449, Fax: +30-2104142264.

only once, continuous queries require continuous evaluation as the query result becomes invalid after a short period of time. Also, we typically have to deal with large volumes of historical data which correspond to a large number of mobile and stationary objects. As a consequence, querying functionality embedded in an extensible DBMS that supports moving objects has to present robust behavior in the above mentioned issues.

An important class of queries that is definitely useful for MOD processing is the so-called *k* nearest neighbor (*k*-NN) queries, where one is interested in finding the *k* closest trajectories to a predefined query object *Q*. To our knowledge, in the literature such queries primarily deal with either static ([RKV95], [CF98], [HS99]) or continuously moving query points ([SR01], [TPS02]) over stationary datasets, or queries about the future or current positions of a set of continuously moving points ([BJKS02], [TP02], [ISS03], [YPK05], [XMA05], [MPH05]). Apparently, these types of queries do not cover NN search on historical trajectories.

The challenge accepted in this paper is to describe diverse mechanisms to perform *k*-NN search on R-tree-like structures [MNPT05] storing historical information. To illustrate the problem, consider an application tracking the positions of rare species of wild animals. Such an application is composed of a MOD storing the location dependent data, together with a spatial index for searching and answering *k*-NN queries in an efficient manner. Experts in the field would be advantaged if they could pose a query like "*find the nearest trajectories of animals to some stationary point (lab, source of food or other non-emigrational species) from which this species passed during March*". Now imagine that the expert's wish is to pose the same query with the difference that the query object *Q* is not a stationary point but a moving animal moving from location P_1 to P_2 during a period of time. This query gives us rise to deduce a more generic query where the expert may wish to set another trajectory of the same or relative class of species as the query object *Q*. It is self-evident that by these types of queries an expert may figure out motion habits and patterns of wild species or deviations from natural emigration, which could be interrelated with environmental and/or ecological changes or destructions. Having in mind that MOD users are usually interested in continuous types of queries, the two previously discussed queries are extended to their continuous counterparts. In their continuous variation, each query returns a time-varying number (denoting the nearest distance, which depends on time) along with a collection of trajectory ids and the appropriate time intervals for which each moving object is valid $\{O_1[t_1, t_2), O_2[t_2, t_3), \ldots\}$.

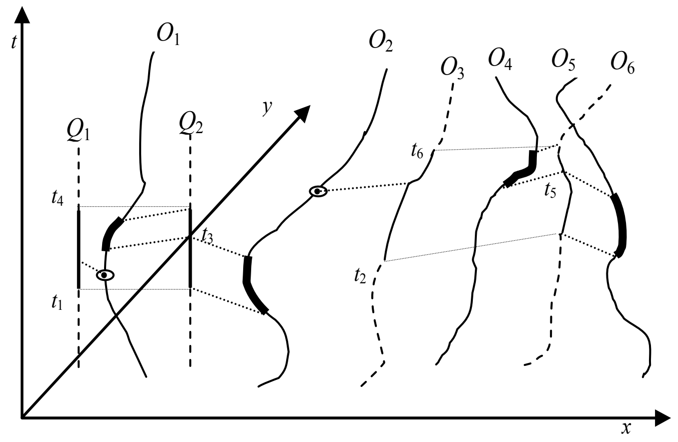


Figure 1: Historical continuous and non-continuous point and trajectory NN queries over moving objects trajectories

To make the previous example more intelligible, Figure 1 illustrates the trajectories of six moving animals $\{O_1, O_2, O_3, O_4, O_5, O_6\}$ along with two stationary points (Q_1 and Q_2) representing two sources of food.

Now, consider the following queries demonstrated in Figure 1 (Queries 2 and 4 are the continuous counterparts of Queries 1 and 3, respectively):

Query 1. *"Find which animal was nearest to the stationary food source Q_1 during the time period $[t_1,t_4]$"*, resulting to animal O_1.

Query 2. *"Find which animal was nearest to the stationary food source Q_2 at any time instance of the time period $[t_1,t_4]$",* resulting to a list of objects: O_2 for the interval $[t_1,t_3)$; O_1 for the interval $[t_3,t_4]$.

Query 3. *"Find which animal was nearest to animal O_3 during the time period $[t_2,t_6]$"*, resulting to O_2.

Query 4. *"Find which animal was nearest to animal O_6 at any time instance of the time period $[t_2,t_6]$"*, resulting to a list of objects: O_5 for the interval $[t_2,t_5)$; O_4 for the interval $[t_5,t_6]$.

Although queries 2 and 4 are continuous in nature (*at any time instance*) they cannot be characterized as pure continuous queries; with respect to the database engine, a continuous query is one that is submitted to the database only once and remains active, continuously updating the query result with the evolution of time, until its completion is declared by either a user's message or a predetermined query lifetime [BW01, MXA04, HXL05]. In this sense, queries 2 and 4 are snapshot queries. However, in order to differentiate them from queries 1 and 3 and also from pure continuous queries, in the rest of the paper, we will call them *Historical Continuous NN queries (HCNN).*

Posing the problem in a more human-centric context, consider an application analyzing the dynamics of urban and regional systems. The intention here is to assist the development of spatiotemporal decision support systems (STDSS) aimed at the planning profession. Such a case requires similar methodologies for comprehending, in space and time, the interrelations of the life courses of individuals. The life courses of most individuals are built around two interlocking successions of events: a residential trajectory and an occupational career. These patterns of events became more complex during last decades, creating new challenges for urban and regional planners. We believe that an expert may take advantage of the features provided by our nearest neighbor query processing algorithms and utilize them for analyzing human life courses.

To the best of our knowledge, this is the first work on *k*-NN query processing over historical trajectories of moving objects. Outlining the major issues that will be addressed in this paper, our main contributions are as follows:

- We propose query processing algorithms to perform NN search on R-tree-like structures storing historical information about moving objects. The description of our algorithms for different queries depends on the type of the query object (point or trajectory) as well as on whether the query itself is continuous or not. In particular, we present efficient depth-first and best-first (incremental) algorithms for historical NN queries as well as depth-first algorithms for their continuous counterparts. All the proposed algorithms are generalized to find the *k* nearest neighbors.
- We propose novel metrics to support our search ordering and pruning strategies. More specifically, the definition of the minimum distance metric MINDIST between points and rectangles, initially proposed in [RKV95] and extended in [TPS02], is further extended in order for our algorithms to calculate the minimum distance between trajectories and rectangles efficiently.
- We conduct a comprehensive set of experiments over large synthetic and real datasets demonstrating that the algorithms are highly scalable and efficient in terms of node accesses, execution time and pruned space.

The rest of the paper is structured as follows. Related work is discussed in Section 2, while Section 3 introduces, at an abstract level, the set of *k*-NN algorithms over moving object trajectories, as well as the metrics that support our search ordering and pruning strategies. Sections 4, 5 and 6 constitute the core of the paper describing in detail the query processing algorithms to perform NN search over historical trajectory information (Sections 4 and 5) together with their continuous counterparts (Section 6). Section 7 presents the results of our experimental study and Section 8 provides the conclusions of the paper and some interesting research directions.

2. Related Work

In this section we will firstly deal with R-tree-like structures indexing historical trajectory information, and subsequently we will examine the related work performed in the domain of nearest neighbor query processing with stationary or moving query objects over stationary or moving datasets.

2.1 Indexing trajectories

A variety of spatiotemporal access methods for the past positions of moving objects have been proposed during the last years, most of them based on the R-tree [Gut84, MNPT05], which is an extension of B-tree in multidimensional spaces. Like B-tree, R-tree, is a height-balanced tree with the index records in its leaf nodes containing pointers to the actual data objects and guarantees that the space utilization is at least 50%. Leaf node entries are in the form (*id*, *MBB*), where *id* is an identifier that points to the actual object and *MBB* (Minimum Bounding Box) is a n-dimensional interval. Non-leaf node entries are of the form (*ptr*, *MBB*), where *ptr* is a pointer to a child node, and *MBB* the bounding box that covers all child nodes. A node in the tree corresponds to a disk page and contains between *m* and *M* entries. The 3D R-tree [TVS96] is a straightforward extension of the R-tree in the 3D space constituting by 2+1 (spatial and temporal, respectively) dimensions. It treats time as an extra spatial dimension and is capable of answering range and timeslice queries.

In fact, the first index proposed to support trajectory-based queries in historical MODs was the Trajectory Bundle tree (TB-tree) [PJT00], following an approach fundamentally different from other spatiotemporal access methods mainly because of its insertion and split strategy. It is a height-balanced tree with the index records in its leaf nodes; leaf nodes contain entries of the same trajectories, and are of the form (*MBB, Orientation*), where *MBB* is the 3D bounding box of the 3D line segment belonging to an object's trajectory (handling time as the third dimension) and *Orientation* is a flag used to reconstruct the actual 3D line segment inside the MBB among four different alternatives that exist. Since each leaf node contains entries of the same trajectory, object *id* can be stored once in the leaf node header.

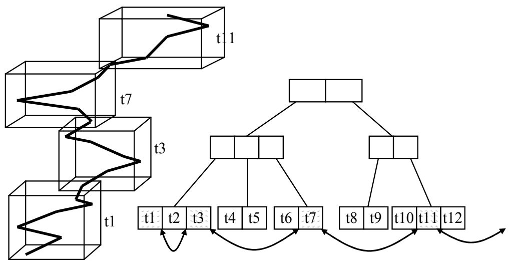


Figure 2: The TB-structure

The TB-tree insertion algorithm is not based upon the spatial and temporal relations of moving objects but it relies only on the moving object identifier (*id*). When new line segments are inserted, the algorithm searches for the leaf node containing the last entry of the same trajectory, and simply inserts the new entry in it, thus forming with leaf nodes that contain line segments from a single trajectory. If the leaf node is full, then a new one is created and is inserted in the right-end of the tree. For each trajectory, a double linked list connects leaf nodes together (Figure 2), resulting in a structure that can efficiently answer trajectory-based queries.

2.2 Nearest Neighbor Search in spatiotemporal databases

In the last decade, NN queries have fueled the spatial and spatiotemporal database community with a series of interesting noteworthy research issues. An affluence of methods for the efficient processing of NN queries for static query points already exist, the most influential probably being the branch-and-bound R-tree traversal algorithm proposed by Roussopoulos et al. [RKV95] for finding the nearest neighbor of a single stationary point. The algorithm utilizes two metrics, MINDIST and MINMAXDIST, in order to implement tree pruning and ordering. Specifically, starting from the root of the tree, the algorithm identifies the entry with the minimum distance from the query point (with the use of the above metrics). The process is recursively repeated until the leaf level is reached, where the first candidate nearest neighbor is found. Returning from this recursion, only the entries with a minimum distance less than the distance of the nearest neighbor already found are visited. The above process was generalized to support *k*-NN queries. Later, Cheung and Fu [CF98] proved that, given the MINDIST-based ordering, the pruning obtained by [RKV95] can be preserved without the use of MINMAXDIST metric (the calculation of which is computationally expensive).

Hjaltason and Samet [HS99] presented a general incremental NN algorithm, which employs a best-first traversal of the R-tree structure. When deciding what node of the tree to traverse next, the proposed algorithm picks the node with the least distance in the set of all nodes that have yet to be visited. In order to achieve this, the algorithm utilizes a priority queue where the tree nodes are stored in increasing order of their distance from the query object. This best-first algorithm outperforms Roussopoulos et al. algorithm in terms of pruned space. Additionally, once the nearest neighbor has been found, the *k*-NN can be retrieved with virtually no additional work, since the algorithm is incremental. The basic drawback of this best-first algorithm is that its performance depends on the size of the priority queue. In case the priority queue becomes very large, the execution time of the algorithm increases rapidly.

The first algorithm for *k* nearest neighbor search over a moving query point was proposed in [SR01]. The algorithm assumes that sites (landmark points) are static and their locations (known in advance) are stored in an R-tree-like structure. A discrete time dimension is assumed, thus a periodical sampling technique is applied on the trace of the moving query point. The location of the query point that lies between two consecutive sampled locations is estimated using linear or polynomial splines. Executing a Point Nearest Neighbor (PNN) query for every sample point of the query trace is highly inefficient, so the proposed algorithm adopts a progressive approach, based on the observation that when two query points are close, the results of the *k*-NN search at these locations have to be related. Therefore, when computing the result set for a sample location, the algorithm tries to exploit information provided by the result sets of the previous samples. The basic drawback of this approach is that the accuracy of the results depends on the sampling rate. Moreover, there is a significant computational overhead.

A technique that avoids the drawbacks of sampling relies on the concept of time-parameterized (TP) queries [TP02]. TP queries retrieve the current result at the time the query is issued, the validity period of the result and the change (i.e. the set of objects) that causes the expiration of the result. Given the current result and the set of objects that affect its validity, the next result can be incrementally computed. The significance of TP queries is two-fold: i) as stand-alone methods, they are suitable for applications involving dynamic environments, where any result is valid for a certain period of time, and ii) they lie at the core of more complex query mechanisms, such as the Continuous NN (CNN) queries. The main disadvantage of using TP queries for the processing of a CNN query is that several NN queries are required to be performed. Thus, the cost of the method is prohibitive for large datasets.

Using the TPR-tree (Time Parameterized Tree) structure [SJLL00], Benetis et al. [BJKS02] presented efficient solutions for NN and RNN (Reverse Nearest Neighbor) queries for moving objects. (An RNN query returns all the objects that the query object is the nearest neighbor of.) The proposed algorithm was the first to address continuous RNN queries, since previous existing RNN algorithms were developed under the assumption that the query point is stationary. The algorithms for both NN and RNN queries in [BJKS02] refer to future (estimated) locations of the query and data points, which are assumed to be continuously moving on the plane. In the same paper, an algorithm for answering CNN queries is also proposed.

Tao et al. [TPS02] also studied CNN queries and proposed an R-tree based algorithm (for moving query points and static data points) that avoids the pitfalls of previous ones (false misses and high processing cost). The proposed tree pruning heuristics exploit the MINDIST metric presented in [RKV95]. At each leaf entry, the algorithm focuses on the accurate calculation of the split points (the points of the query segment that demonstrate a change of neighborhood). A theoretical analysis of the optimal performance for CNN algorithms was presented and cost models for node accesses were proposed. Furthermore, the CNN algorithm was extended for the case of *k* neighbors and trajectory inputs.

Based on the TP queries presented in [TP02], Iwerks et al. [ISS03] described a technique that focuses on the maintenance of CNN queries (for future predicted locations) in the presence of updates on moving points, where the motion of the points is represented as a function of time. A new approach was also presented, which filters the number of objects to be taken into account when maintaining a future CNN query.

Recently, under the same field, Xiong et al. [XMA05], proposed a method for scalable processing of CNN queries in spatiotemporal databases. They propose a general framework for processing large numbers of simultaneous k-CNN queries with static or moving queries over static or (currently) moving datasets without making any assumptions about the object trajectories. Unlike other proposals, their solution in order to support high update rates is not based on the R-tree but on a simple grid structure maintained on the disk. A similar method was also proposed by Yu et al. [YKP05] for monitoring k-CNN queries over (currently) moving objects without making any assumptions about the object trajectories. The method also uses (main memory) grid indices indexing moving objects and queries and is shown to outperform R-tree-based solutions. Mouratidis et al. [MHP05] also relax the assumption that moving object's trajectories are fully predictable by their motion parameters, and propose a comprehensive technique for the efficient monitoring of continuous NN queries. The proposed method, named conceptual partitioning monitoring method (CPM), uses also a grid structure and achieves low running time by handling moving object's location updates only from objects falling in the vicinity of some query. The experimental results presented in [MHP05] show that the CPM method outperforms the techniques presented in [XMA05] and [YPK05].

Shahabi et al. [SKS03] presented the first algorithm for processing the *k*-NN queries for moving objects in road networks. Their proposed algorithm, which utilizes the network distance between two locations instead of the Euclidean, is based on transforming the road network into a higher dimensional space, in which simpler distance functions can be applied. Using this embedding space, efficient techniques are proposed for finding the shortest path between two points in the road network. The above procedure, which is utilized in the case of static query points, is slightly modified in order to support the case of moving query points.

Acknowledging the advantages of the above fundamental techniques, in this paper we present the first complete treatment of historical NN queries over moving object trajectories, handling both stationary and moving query objects.

3. Problem Statement and Metrics

We first define the NN queries that are considered in this paper. Subsequently, we present the heuristics utilized by our algorithms to implement the metrics needed to formulate our ordering and pruning strategy.

3.1 Problem statement

Let D be a database of N moving objects with objects ids $\{O_1, O_2, \ldots, O_N\}$. The trajectory T_i of a moving object O_i consists of M_i 3D-line segments $\{L_{i1}, L_{i2}, \ldots, L_{iM_i}\}$. Each 3D line segment L_j is of the form $((x_{j\text{-}start}, y_{j\text{-}start}, t_{j\text{-}start}), (x_{j\text{-}end}, y_{j\text{-}end}, t_{j\text{-}end}))$, where $t_0 \leq t_{j\text{-}start} < t_{j\text{-}end} \leq now$. Obviously, as we treat only historical moving object trajectories, each partial linear movement is temporally restricted between t_0, the beginning of the calendar, and *now*, the current time point.

We have already stated that NN queries search for the closest trajectories to a query object Q. In our case, we distinguish two types of query objects: Q_p, a point (x, y) that remains stationary during the time period of the query $Q_{per}[t_{start}, t_{end}]$, and Q_T, a moving object with trajectory T. Furthermore, the MOD is indexed by an R-tree like structure such as the 3D R-tree [TVS96], the STR-tree or the TB-tree [PJT00]. Having in mind the previous discussion, we define the following two types of NN queries:

- NN_Q_p (D, Q_p, Q_{per}) query searches database D for the NN over a point Q_p that remains stationary during a time period Q_{per}, and returns the closest to Q_p point p_c from which a moving object O_i passed during the time period Q_{per}, as well as the implied minimum distance.
- NN_Q_T (D, Q_T, Q_{per}) query is similar to the previous with the difference being upon the query object Q which in the current case is a moving object with trajectory T.

The extensions of the above queries to their historical continuous counterparts vary in the output of the algorithms. In the continuous case, each query returns a time-varying real number, as the nearest distance depends on time. We introduce the following two types of historical CNN queries:

- $HCNN\_Q_p$ (D, Q_p, Q_{per}) query over a point Q_p that remains stationary during a time period Q_{per} returns a list of triplets consisting of the time-varying real value R_i along with a moving object O_i (belonging in database D) and the corresponding time period $[t_{i\text{-}start}, t_{i\text{-}end})$ for which the nearest distance between Q_p and O_i stands. These time-varying real values R_i are, in any time instance of their lifetime, smaller or equal to the distance between any moving object O_j in D and the query point Q_p. The time periods $[t_{i\text{-}start}, t_{i\text{-}end})$ are mutually disjoint and their union forms Q_{per}.

- Similarly, *HCNN_Q*$_T$ (D, Q_T, Q_{per}) differs, compared to the previous, upon the query object Q which in the current case is a moving object with trajectory T. These time-varying real values R_i are, in any time instance of their lifetime, smaller or equal to the distance between any moving object O_j and the query trajectory Q_T. The time periods $[t_{i\text{-}start}, t_{i\text{-}end})$ are mutually disjoint and their union forms Q_{per}.

The above four queries are generalized to produce the corresponding *k*-NN queries. The generalization of the first two queries is straightforward by simply requesting the 1st, 2nd, …, *k*-th nearest point – with respect to a query point or a query trajectory – from which a moving object O_i passed during the time period Q_{per}, excluding at the same time points belonging to a moving object already marked as the *j*-th nearest ($1 \leq j < k$). The historical continuous queries are generalized to produce *k*-HCNN requesting to provide with *k* lists of $\{R_i, [t_{i\text{-}start}, t_{i\text{-}end}), O_i\}$ triplets. Then, for any time during the time period Q_{per}, the *i*-th list ($1 \leq i \leq k$) will contain the *i*-order NN moving object (with respect to the query point or the query trajectory) at this time instance.

To exemplify the proposed k-NN extensions, let us recall Figure 1. Searching for the 2-NN versions of the four queries (Query 1, 2, 3 and 4) presented in Section 1, we will have the following results:

- Query 1 (historical non-continuous): O_1 (1st NN) and O_2 (2nd NN)
- Query 2 (historical continuous): 1-NN list includes O_2 for the interval $[t_1,t_3)$ and O_1 for the interval $[t_3,t_4]$; 2-NN list includes O_1 for the interval $[t_1,t_3)$ and O_2 for the interval $[t_3,t_4]$
- Query 3 (historical non-continuous): O_2 (1st NN) and O_4 (2nd NN)
- Query 4 (historical continuous): 1-NN list includes O_5 for the interval $[t_2,t_5)$ and O_4 for the interval $[t_5,t_6]$; 2-NN list includes O_4 for the interval $[t_2,t_5)$ and O_5 for the interval $[t_5,t_6]$.

3.2 Metrics

We exploit on the definition of the minimum distance metric (MINDIST) presented in [RKV95] between points and rectangles, in order to calculate, the minimum distance between line segments and rectangles and, the minimum distance between trajectories and rectangles that are needed to implement the above discussed algorithms.

Initially, in [RKV95], Roussopoulos et al. defined the Minimum Distance (MINDIST) between a point *P* and a rectangle *R* in the n-dimensional space as the square of the Euclidean distance between *P* and the nearest edge of *R*, if *P* is outside *R* (or zero, if *P* is inside *R*). Then, Tao et al. [TPS02] proposed a method to calculate the MINDIST between a 2D line segment *L* and a rectangle M (Figure 3).

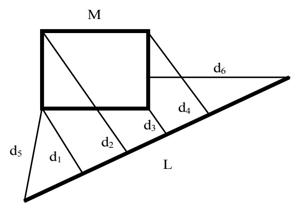
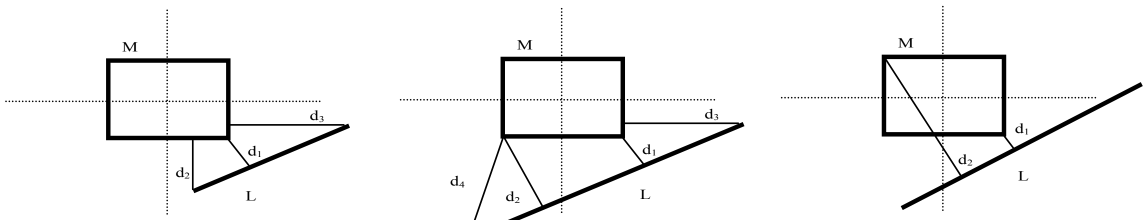


Figure 3: Calculating MINDIST between a line segment and a rectangle [TPS02]

MINDIST calculation method in [TPS02] initially determines whether *L* intersects *M*; if so, MINDIST is set to zero. Otherwise, they choose the shortest among six distances, namely the four distances between each corner point of *M* and *L* (d_1, d_2, d_3, d_4) and the two minimum distances from the start and end point of *L* to *M*

(d_5, d_6). Therefore, the calculation of MINDIST between a line segment and a rectangle involves an intersection check, four segment-to-point MINDIST calculations and two point-to-rectangle MINDIST calculations.

In this paper, we propose a more efficient method to calculate MINDIST between a line segment *L* and a rectangle *M* (Figure 4). As before, if *L* intersects *M*, then MINDIST is obviously zero. Otherwise, we decompose the space in four quadrants using the two axes passing through the center of *M* and we determine the quadrants *Qs* and *Qe* in which the start (*L.start*) and the end (*L.end*) point of *L* lie in, respectively.



Case 1: *L.start* and *L.end* belong to the same quadrant

Case 2: *L.start* and *L.end* belong to adjacent quadrants

Case 3: *L.start* and *L.end* belong to non adjacent quadrants

Figure 4 The proposed calculation method of MINDIST between a line segment and a rectangle

Then, MINDIST is the minimum among:

- **Case 1** (the two end points of the line segment belong to the same quadrant (*Qs*)): (i) MINDIST between the corner of *M* in *Qs* and *L*, (ii) MINDIST between *L.start* and *M* or (iii) MINDIST between *L.end* and *M*.
- **Case 2** (*L.start* and *L.end* belong to adjacent quadrants *Qs* and *Qe*, respectively): (i) MINDIST between the corner of *M* in *Qs* and *L*, (ii) MINDIST between the corner of *M* in *Qe* and *L*, (iii) MINDIST between *L.start* and *M* or (iv) MINDIST between *L.end* and *M*.
- **Case 3** (*L.start* and *L.end* belong to non adjacent quadrants *Qs* and *Qe*, respectively): two MINDIST between the two corners of *M*, that do not belong in either *Qs* or *Qe*, and *L*.

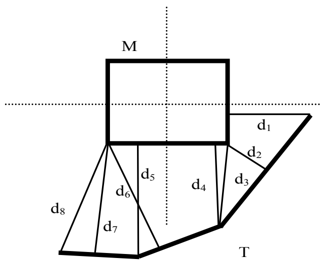


Figure 5: The proposed calculation method of MINDIST between a route (projection of a trajectory on the plane) and a rectangle

This method utilizes a smaller number of (point-to-segment and point-to-rectangle) distance calculations compared to the corresponding algorithm in [TPS02]. The worst-case scenario of the proposed MINDIST calculation includes the determination of the quadrant in which the starting and ending points of the line segment belong, i.e., two point-to-segment and two point-to-rectangle distance calculations, while the corresponding algorithm of [TPS02] employs four point-to-segment and two point-to-rectangle calculations. Therefore, the proposed MINDIST calculation, in its worst case, determines the quadrant of the starting and ending point instead of performing two additional point-to-segment distance calculations.

Finally, we extend the above algorithm in order to calculate MINDIST metric between the projection of a trajectory *T* on the plane (usually called *route*) and a rectangle *M* (Figure 5). Since a route can be viewed as a collection of 2D line segments, the MINDIST between a route of a trajectory and a rectangle can be computed as the minimum of all MINDIST between the rectangle and each line segment composing the route. The efficiency of this calculation can be enhanced by simply not computing twice, with respect to the query rectangle, the quadrant and the MINDIST of the end and the start of adjacent line segments. The efficiency of the proposed improvement over the MINDIST computation for line segments and trajectories will be shown in the experimental section.

4. Non-incremental (Depth-First) NN Algorithms over Trajectories

In this section we describe in details the non-incremental algorithms answering the first two (historical non-continuous) types of NN queries presented in Section 3.1 and, then, we generalize them in order to support the respective *k*-NN queries.

4.1. Non-incremental NN algorithm for stationary query objects (points)

The non-incremental NN algorithm for stationary query objects (`PointNNSearch`) algorithm, illustrated in Figure 6, provides the ability to answer NN queries for a static query object Q_p, during a certain query time period $Q_{per}[t_{start}, t_{end}]$. The algorithm uses the same heuristics as in [RKV95] and [CF98], pruning the search space according to Q_{per}.

```
Algorithm PointNNSearch(node N, 2D point Q, time period $Q_{per}$, struct Nearest)
 1.  IF N Is Leaf
     // Iterate through leaf entries computing Euclidean distance from point Q
 2.    FOR EACH Entry E in N
         // If entry is (fully or partially) inside the period
 3.      IF $Q_{per}$ Overlaps (E.$T_S$, E.$T_E$)
         // Compute entry's spatial extent inside the period
 4.        nE = Interpolate(E, Max($Q_{per}$.$T_S$, E.$T_S$), Min($Q_{per}$.$T_E$, E.$T_E$))
          // Compute Entry's actual distance from Q. Update Nearest if necessary
 5.        Dist = Euclidean_Dist_2D(Q, nE)
 6.        IF Dist < Nearest.Dist THEN Update Nearest with nE, Dist
 7.      END IF
 8.    NEXT
 9.  ELSE
       // Generate Node's branch list with entries overlapping the query period
10.    BranchList = GenBranchList(Q, N, $Q_{per}$)
       // Sort active branch List by MinDist
11.    SortBranchList(BranchList)
       // Iterate through active branch List
12.    FOR EACH Entry E in BranchList
         // Visit Child Nodes
13.      PointNNSearch(E.ChildNode, Q, $Q_{per}$, Nearest)
         // Apply MinDist heuristic to do pruning
14.      PruneBranchList(BranchList)
15.    NEXT
16.  END IF
```

Figure 6: Historical NN search algorithm for stationary query points (`PointNNSearch` algorithm)

The algorithm accesses the tree structure (which indexes the trajectories of the moving objects) in a depth-first way pruning the tree nodes according to Q_{per} rejecting those being fully outside it. At leaf level, the algorithm iterates through the leaf entries checking whether the lifetime of an entry overlaps Q_{per} (Line 3); if the temporal component of the entry is fully inside Q_{per}, the algorithm calculates the actual Euclidean distance between Q and the (spatial component of the) entry; otherwise, if the temporal component of the entry is only partially inside Q_{per}, a linear interpolation is applied so as to compute the entry's portion being inside Q_{per} (Line 4) and calculate the Euclidean distance between Q and the portion of that entry. When a candidate *nearest* is selected, the algorithm, backtracking to the upper level, prunes the nodes in the active branch list (Line 14) applying the MINDIST heuristic [RKV95] [CF98].

4.2. Non-incremental NN algorithm for moving query objects (trajectories)

`PointNNSearch` algorithm can be modified in order to support the second type of NN query where the query object is a trajectory of a moving point (`TrajectoryNNSearch` algorithm, illustrated in Figure 7). At the leaf level, the algorithm calculates the minimum Euclidean distance between the projections on the 2D (x-,y-) plane of each leaf entry rectangle and each query trajectory segment by using the `Min_Horizontal_Dist` function (Line 9), which computes the minimum ("horizontal") Euclidean distance between the projections on the 2D plane of two 3D line segments. The formulization of the `Min_Horizontal_Dist` function and the calculation of its minimum value required by the `TrajectoryNNSearch` algorithm can be found in Appendix A. In addition, for each query trajectory segment *QE* and before calculating its distance from the current leaf entry we first interpolate in order to produce a tuple of entry - query segment with identical temporal extent (Lines 7, 8). In order to decrease the number of temporal overlap evaluations between leaf entries and trajectory segments, our algorithm utilizes a plane sweep method, which scans leaf entries and trajectory segments in their temporal dimension in a single pass (Lines 4, 5, 6). This requires that the leaf entries are previously sorted according to their temporal extent (Line 3), unless the underlying tree structure (such as the TB-tree) stores them in temporal order anyway.

```
Algorithm TrajectoryNNSearch(node N, trajectory Q, time period $Q_{per}$, struct
Nearest)
 1.  Q = Interpolate(Q, Max(Q.$T_S$, $Q_{per}$.$T_S$), Min(Q.$T_E$, $Q_{per}$.$T_E$))
 2.  IF N Is Leaf
 3.    Sort(N, $T_S$) // Sort A-Z Entries in Node N by their $T_{start}$
 4.    FOR EACH Entry E in N
 5.      FIND next query trajectory entry QS with QS.$T_e$<N.$T_S$: QE=QS
 6.      DO UNTIL QE.$T_s$ > E.$T_e$
 7.        nE = Interpolate(E, Max(QE.$T_S$, E.$T_S$), Min(QE.$T_E$, E.$T_E$))
 8.        nQE = Interpolate(QE, Max(QE.$T_S$, E.$T_S$), Min(QE.$T_E$, E.$T_E$))
 9.        Dist = Min_Horizontal_Dist(nQE, nE)
10.        IF Dist < Nearest.Dist THEN Update Nearest with nE, Dist
11.      NEXT query entry QE
12.      Return QE in the query entry QS
13.    NEXT
14.  ELSE
15.    BranchList = GenTrajectoryBranchList(Q, N)
16.    SortBranchList(BranchList)
17.    FOR EACH Entry E in BranchList
18.      TrajectoryNNSearch(E.ChildNode, E.Trajectory, Nearest)
19.      PruneBranchList(BranchList)
20.    NEXT
21.  END IF
```

Figure 7: Historical NN search algorithm for moving query points (`TrajectoryNNSearch` algorithm)

At the non-leaf levels, the algorithm utilizes the `GenTrajectoryBranchList` function (pseudo-code in Figure 8) instead of `GenBranchList`. The `GenTrajectoryBranchList` function utilizes the `MinDist_Trajectory_Rectangle` metric introduced in Section 3.2 in order to calculate MINDIST between the query trajectory and the rectangle of each entry of node *N*. Here, we have to point out that we do not need to calculate `MinDist_Trajectory_Rectangle` against the actual query trajectory *Q*, but against the part of *Q* being inside the temporal extent of the bounding rectangle of *N*, and in order to do this (if it is necessary) we interpolate to produce the new query trajectory *nQ* (Line 3). The interpolated trajectory *nQ* is also stored inside the *Branchlist* along with the respective node entry and the calculated distance (Line 5). Since all the nodes in the sub-tree of *N* are spatially and temporally contained inside *N*, the interpolated trajectory *nQ* can be used as the query trajectory for the nodes of the next level inside the sub-tree, allowing us to avoid unnecessary calculations.

```
Algorithm genTrajectoryBranchList(node N, trajectory Q)
 1.  FOR EACH Entry E in N
       // If entry is (fully or partially) inside the trajectory lifetime
 2.    IF (Q.$T_S$, Q.$T_E$) Overlaps (E.$T_S$, E.$T_E$)
         // Compute trajectory's spatial extent inside E's lifetime
 3.      nQ = Interpolate(Q, Max(Q.$T_S$, E.$T_S$), Min(Q.$T_E$, E.$T_E$))
         // Compute MinDist between the resulted trajectory and the rectangle
 4.      Dist=MinDist_Trajectory_Rectangle(nQ, E)
         // Add the rectangle along with its calculated distance and the
            interpolated trajectory in the list
 5.      List.Add(E, Dist, nQ)
 6.    END IF
 7.  NEXT
 8.  RETURN List
```

Figure 8: Generating Branch List of Node N against Trajectory Q

4.3. Extending to non-incremental *k*-NN algorithms

In the same fashion as in [RKV95], we generalize the above two algorithms to searching the *k*-nearest neighbors by considering the following:

- Using a buffer of at most *k* (current) nearest objects sorted by their actual distance from the query object (point or trajectory)
- Pruning according to the distance of the (currently) furthest nearest object in the buffer.
- Updating the distance of each moving object inside the buffer when visiting a node that contains an entry of the same object closer to the query object.

5. Incremental (Best-First) NN Algorithms over Trajectories

In this section, we present best-first algorithms that process the same NN queries with the ones described in Section 4 and, then, we generalize them in order to support the respective *k*-NN queries.

5.1. Incremental NN algorithm for stationary query objects (points)

The proposed algorithm, which is based on the NN algorithm for static objects presented in [HS99], traverses the tree structure in a best-first way. The algorithm uses a priority queue, in which the entries of the tree nodes are stored in increasing order of their distance from the query object.

```
Algorithm IncPointNNSearch(R-tree R, 2D point Q, time period Q_per)
 1.  ENQUEUE Queue, R.RootNode, 0
 2.  DO WHILE Queue.Count > 0
 3.    Element = DEQUEUE(Queue)
 4.    IF Element Is MovingObjectEntry
 5.      RETURN Element as the next nearest object
 6.    ELSEIF Element Is Leaf
       // Iterate through leaf entries computing Euclidean distance from Q
 7.      FOR EACH Entry E in leaf node Element
           // If entry is (fully or partially) inside the period
 8.        IF Q_per Overlaps (E.T_S, E.T_E)
             // Compute entry's spatial extent inside the period
 9.          nE = Interpolate(E, Max(Q_per.T_S, E.T_S), Min(Q_per.T_E, E.T_E))
             // Compute Entry's actual distance from Q.
10.          Dist = Euclidean_Dist_2D(Q, nE)
11.          EnQueue Queue, nE, Dist
12.        ENDIF
13.      NEXT
14.    ELSE // Element is a non leaf node
         // Iterate through node entries computing their minimum distance from Q
15.      FOR EACH Entry E in node Element
           // If entry is (fully or partially) inside the period
16.        IF Q_per Overlaps (E.T_S, E.T_E)
             // Compute Entry's MinDist from Q.
17.          Dist = MinDist(Q, E)
18.          EnQueue Queue, E, Dist
19.        ENDIF
20.      NEXT
21.    ENDIF
22.  LOOP
```

Figure 9: Historical Incremental NN search algorithm for stationary query points (`IncPointNNSearch` algorithm)

Figure 9 illustrates the `IncPointNNSearch` algorithm. In Line 1, the priority queue is initialized. In Line 5, the next nearest object is reported. As in the respective depth-first algorithm described in Section 4.1, at leaf level the algorithm iterates through the leaf entries checking whether the lifetime of an entry overlaps the time period of the query Q_{per} (Line 8); if the temporal component of the entry is fully inside Q_{per}, the algorithm calculates the actual Euclidean distance between Q and the (spatial component of the) entry; otherwise, if the temporal component of the entry is only partially inside Q_{per}, a linear interpolation is applied so as to compute the entry's portion being inside Q_{per} (Line 9) and calculate the Euclidean distance between Q and the portion of that entry (Line 10). In Line 11, the leaf entry is enqueued along with its real distance from the query object. At the non leaf levels (Lines 15-21), the algorithm simply calculates MINDIST between the query object and each node's entry overlapping the query period Q_{per}, and in the sequel enqueues this entry along with its MINDIST value.

5.2. Incremental NN algorithm for moving query objects (trajectories)

The `IncPointNNSearch` algorithm proposed above can be slightly modified in order to support the second type of NN query where the query object is a trajectory of a moving point, thus resulting in `IncTrajectoryNNSearch` algorithm, illustrated in Figure 10. The changes to be made are the following three: firstly, as in the respective depth-first algorithm (Section 4.2), at the leaf level, the algorithm calculates the minimum "horizontal" Euclidean distance between each leaf entry and each segment of the query trajectory *Q*, using the `Min_Horizontal_Dist` function (Line 13). We also utilize the same plane sweep algorithm, so as to determine which leaf entries and segments of *Q* overlap in their temporal dimension, and then we calculate the distance between those who do overlap (Lines 8-10).

```
Algorithm IncTrajectoryNNSearch(R-tree R, trajectory Q, time period $Q_{per}$)
 1.  EnQueue Queue, R.RootNode, Q, 0
 2.  Do While Queue.Count > 0
 3.    DeQueue(Queue, Element, Q)
 4.    IF Element Is MovingObjectEntry
 5.      Return Element as the next nearest object
 6.    ELSEIF Element Is Leaf
 7.      Sort(Element, $T_S$) // Sort A-Z Entries in Node Element by their $T_{start}$
 8.      FOR EACH Entry E in leaf node Element
 9.        FIND next query trajectory entry QS with QS.$T_e$<N.$T_S$: QE=QS
10.        DO UNTIL QE.$T_s$ > E.$T_e$
11.          nE = Interpolate(E, Max(QE.$T_S$, E.$T_S$), Min(QE.$T_E$, E.$T_E$))
12.          nQE = Interpolate(QE, Max(QE.$T_S$, E.$T_S$), Min(QE.$T_E$, E.$T_E$))
13.          Dist = Min_Horizontal_Dist(nQE, nE)
14.          EnQueue Queue, nE, Dist
15.        NEXT query entry QE
16.        Return QE in the query entry QS
17.      NEXT
18.    ELSE
19.      FOR EACH Entry E in node Element
20.        IF (Q.$T_S$, Q.$T_E$) Overlaps (E.$T_S$, E.$T_E$)
21.          nQ = Interpolate(Q, Max(Q.$T_S$, E.$T_S$), Min(Q.$T_E$, E.$T_E$))
22.          Dist = MinDist_Trajectory_Rectangle(nQ, E)
23.          EnQueue Queue, E, Dist, nQ
24.        ENDIF
25.      NEXT
26.    ENDIF
27.  LOOP
```

Figure 10: Historical Incremental NN search algorithm for moving query points (`IncTrajectoryNNSearch` algorithm)

At the non-leaf levels, the algorithm utilizes the `MinDist_Trajectory_Rectangle` metric in order to calculate the MINDIST between the query trajectory and the rectangle of each entry of the node (Line 22). Just like `TrajectoryNNSearch` algorithm, if necessary, we interpolate in order to produce *nQ,* which is the part of *Q* being inside the temporal extent of the bounding rectangle of each node's entry (Line 21), and then we store it inside the *Queue* along with the respective node entry and the calculated distance (Line 23). Since all the nodes in the *N*'s sub-tree are spatially and temporally contained inside *N*, then, the interpolated trajectory *nQ* can be further used as the query trajectory for the nodes of the next level inside the sub-tree, allowing us to avoid unnecessary calculations.

5.3. Extending to incremental *k*-NN algorithms

The algorithms described in Sections 5.2 and 5.3 are incremental in the sense that the *k*-th NN can be obtained with very little additional work once the (*k*-1)-th NN has been found. Recall for example `IncTrajectoryNNSearch` illustrated in Figure 10. After having found the 1st NN, the next time the condition of Line 4 is true, the 2nd NN will have been found.

Here, we have to point out that the two different strategies used for the historical non-continuous NN algorithms appear to have both advantages and drawbacks. As already mentioned, while the best-first approach results always in fewer actually visited nodes, and fewer distance evaluations, its performance heavily depends on the size of the priority queue; as it will be clearly shown in the experiments, this drawback can cause the incremental algorithms to perform worse than the depth-first algorithms in terms of execution time, even though they require fewer nodes to be visited and less distances to be evaluated. On the other hand, the incremental algorithms have a serious advantage over the depth-first ones, which is the ability of retrieving each of the *k* nearest neighbors incrementally, while the depth-first approach requires the a priory knowledge of the parameter *k*.

6. HCNN Algorithms over Trajectories

In this section we describe the historical continuous counterparts of the algorithms of Section 4. In particular, we will address the third type of NN query (searching for NN with respect to a stationary query point at any time during a given time period) and the fourth type of NN query (where the query object is the trajectory of a moving point) and then we will extend them towards *k*-NN search.

6.1. HCNN algorithm for stationary query objects (points)

We begin the description of the algorithms with the third type of NN query, which searches for the nearest moving objects to a stationary query point at any time during a given time period, The `HContPointNNSearch` algorithm proposed for this type of query is illustrated in Figure 11.

```
Algorithm HContPointNNSearch(node N, 2D point Q, Period $Q_{per}$, List Nearests, Roof)
 1.  IF N Is Leaf
 2.    FOR EACH Entry E in N
 3.      IF $Q_{per}$ Overlaps (E.$T_S$, E.$T_E$)
 4.        nE = Interpolate(E, Max($Q_{per}$.$T_S$, E.$T_S$), Min($Q_{per}$.$T_E$, E.$T_E$))
 5.        MovingDist = ConstructMovingDistance(nE, Q)
 6.        IF MovingDist.$D_{min}$ < Roof THEN UpdateNearests(Nearests,MovingDist,Roof)
 7.      END IF
 8.    NEXT
 9.  ELSE
10.    BranchList = GenBranchList(Q, N, $Q_{per}$)
11.    SortBranchList(BranchList)
12.    PruneHContBranchList(BranchList, Nearests, Roof)
13.    FOR EACH Entry E in BranchList
14.      HContPointNNSearch(E.ChildNode, Q, $Q_{per}$, Nearests, Roof)
15.      PruneHContBranchList(BranchList, Nearests, Roof)
16.    NEXT
17.  END IF
```

Figure 11: Historical CNN search algorithm for stationary query points (`HContPointNNSearch` algorithm)

All the historical continuous algorithms use a `MovingDist` structure (Figure 11, Line 5), storing the parameters of the distance function (calculated using the methodology described in Appendix A), along with the entry's temporal extent and the associated minimum and maximum of the function during its lifetime. We also store the actual entry inside the structure in order to be able to return it as the query result. The

`ConstructMovingDistance` function simply calculates this structure (e.g. the parameters of the distance function *a*, *b*, *c*, and the minimum D_{min} and maximum D_{max} of the function inside the lifetime of the entry).

An interesting point of the algorithm is exposed in Line 6, where the `Nearests` structure is introduced. `Nearests` is a list of adjacent "*Moving Distances*" temporally covering the period Q_{Per}. `Roof` is the maximum of all moving distances stored inside the *Nearests* list and is used as a threshold to quickly reject those entries (and prune those branches at the non-leaf level) having their minimum distance greater than `Roof` (consequently, greater than all moving distances stored inside the *Nearests* list). In Appendix B, we present in detail how we maintain the *Nearests* list.

When at non-leaf levels, the `HContPointNNSearch` algorithm in its backtracking applies the pruning algorithm `PruneHContBranchList` (Line 15), which prunes the branch list using the MINDIST heuristic: First, it compares the MINDIST of each entry with `Roof` and then it calculates the maximum distance inside the *Nearests* list during the entry's lifetime. Then, it prunes all entries having MINDIST greater than the one calculated.

```
Algorithm HContTrajectoryNNSearch (node N, Trajectory Q, time period $Q_{per}$, List
Nearests, Roof)
 1.  Q = Interpolate(Q, Max(Q.$T_S$, $Q_{per}$.$T_S$), Min(Q.$T_E$, $Q_{per}$.$T_E$))
 2.  IF N Is Leaf
 3.    Sort(N, $T_S$)
 4.    FOR EACH Entry E in N
 5.      FIND next query trajectory entry QS with QS.$T_e$<N.$T_S$: QE=QS
 6.      DO UNTIL QE.$T_s$ > E.$T_e$
 7.        nE = Interpolate(E, Max(QE.$T_S$, E.$T_S$), Min(QE.$T_E$, E.$T_E$))
 8.        nQE = Interpolate(QE, Max(QE.$T_S$, E.$T_S$), Min(QE.$T_E$, E.$T_E$))
 9.        MovingDist = ConstructMovingDistance(nE, nQE)
 10.       IF MovingDist.$D_{min}$ < Roof THEN UpdateNearests(Nearests,MovingDist,Roof)
 11      NEXT query entry QE
 12      Return QE in the query entry QS
 13.   NEXT
 14. ELSE
 15.   BranchList = GenTrajectoryBranchList(Q, N)
 16.   SortBranchList(BranchList)
 17.   PruneHContBranchList(BranchList, Nearests, Roof)
 18.   FOR EACH Entry E in BranchList
 19.     HContTrajectoryNNSearch(E.ChildNode, E.Trajectory, Nearests, Roof)
 20.     PruneHContBranchList(BranchList, Nearests, Roof)
 21.   NEXT
 22. END IF
```

Figure 12: Historical CNN search algorithm for moving query points (`HContTrajectoryNNSearch` algorithm)

6.2. HCNN algorithm for moving query objects (trajectories)

The fourth type of NN query is the historical continuous version of the NN query where the query object is the trajectory of a moving point. The `HContTrajectoryNNSearch` algorithm, used to process this type of query is illustrated in Figure 12.

`HContTrajectoryNNSearch` differs from `HContPointNNSearch` at two points only: The first is that, at leaf level, the `ConstructMovingDistance` function calculates the "moving distance" between two moving points, instead of one moving and one stationary (Line 9). Secondly, at the non-leaf levels, `GenBranchList` is replaced by the `GenTrajectoryBranchList` function introduced in the description of the `TrajectoryNNSearch` algorithm (Line 15). Moreover, as in `TrajectoryNNSearch`, for each query trajectory segment *QE* and before calculating the moving distance from the current leaf entry we first interpolate in order to produce a tuple of entry - query segment with identical temporal extent (Lines 7, 8). We also use the same plane sweep method, in order to reduce the number of distance calculations between the segments of *Q* and the leaf entries (Lines 4-6).

6.3. Extending to *k*-HCNN algorithms

The two historical continuous algorithms proposed above can be also generalized to searching the *k*- nearest neighbors by considering the following:

- Using a buffer of at most *k* current `Nearests` Lists
- Pruning according to the distance of the furthest `Nearests` Lists in the buffer – therefore `Roof` is calculated as the maximum distance of the furthest `Nearests` List
- Processing each entry against the *i*-th list (with *i* increasing, from 1 to *k*) checking whether it qualifies to be in a list
- When a moving distance is replaced by a new entry in the *i*-th list, testing it against the (*i*+1)-th list to find whether it qualifies to be in that list.

7. Performance Study

The above illustrated algorithms can be implemented in any R-tree-like structure storing historical moving object information such as the 3D R-tree [TVS96], the STR-tree [PJT00] and the TB-tree [PJT00]. Among them, we have chosen to implement the algorithms using the 3D R-tree and the TB-tree, which are considered state-of-the-art techniques in particular settings each: as shown in [PJT00], regarding range queries, the 3D R-tree usually outperforms the TB-tree as the cardinality of the dataset grows, while the TB-tree is more efficient in retrieving historical trajectory information with combined and other trajectory-based queries. In our implementation, both TB- and 3D R-tree leaf entries are modified as shown in [PJT00] storing the actual trajectory entries and not each entry's MBR. Both trees along with all the proposed algorithms were implemented using Visual Basic. We used a page size of 4096 bytes and a (variable size) buffer fitting the 10% of the index size, with a maximum capacity of 1000 pages. The experiments were performed in a PC running Microsoft Windows XP with AMD Athlon 64 3GHz processor, 512 MB RAM and several GB of disk space.

7.1. Datasets

While several real spatial datasets are around for experimental purposes, this is not true for the moving object domain. Nevertheless, in this paper, we have experimented with two real datasets from a fleet of trucks and a fleet of school buses (illustrated in Figure 13(a) and (b), respectively) [The03]. The two real datasets consist of 276 (112203) and 145 (66096) trajectories (entries in the index), respectively, thus building indices of up to 5 Mbytes size (the case of 3D R-tree index for the Trucks dataset). The performance study was not limited to real data. We have also used synthetic datasets generated by the GSTD data generator [TSN99] in order to achieve scalability in the volumes of the datasets. A snapshot of the generated data using GSTD is illustrated in Figure

13(c). The synthetic trajectories generated by GSTD correspond to 100, 250, 500, 1000 and 2000 moving objects resulting in datasets of 500K, 1250K, 2500K, 5000K, and 10000K entries (the position of each object was sampled approximately 5000 times), thus building indices of up to 500 Mbytes size (the case of 3D R-tree index for the GSTD 2000 dataset). Regarding the rest parameters of the GSTD generator, the initial distribution of points was Gaussian while their movement was ruled by a random distribution. Table 1 illustrates summary information about the number of pages occupied by both indexes for each dataset.

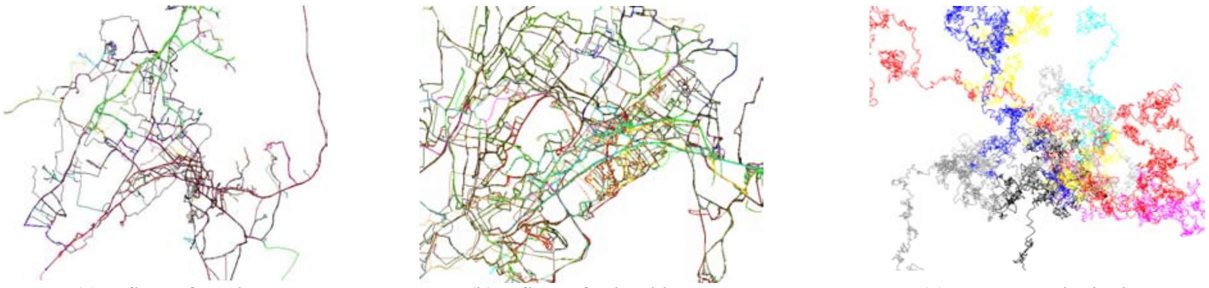


Figure 13: Snapshots of real and synthetic spatiotemporal data

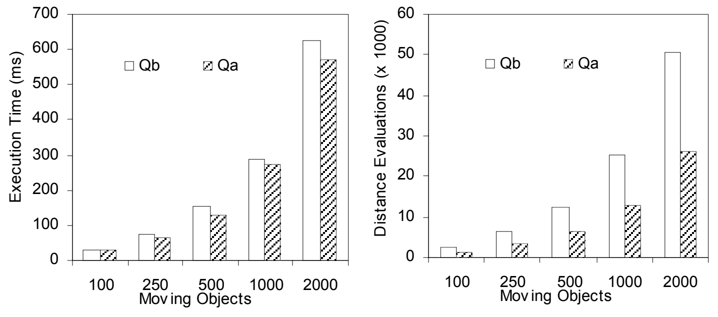
	# trajectories	*# entries*	*index size in pages (of 4096 bytes each)*	
			3D R-tree	***TB-tree***
Real Data (Trucks)	276	112203	1288	835
Real Data (Buses)	145	66096	805	466
GSTD 100	100	485017	6253	3054
GSTD 250	250	1213195	15471	7649
GSTD 500	500	2426345	30937	15301
GSTD 1000	1000	4850750	61864	30587
GSTD 2000	2000	9701500	122703	61171

Table 1: Summary dataset information

7.2. Results on the calculation of the MINDIST metric

In order to demonstrate the efficiency of the proposed MINDIST calculation over the one presented in [TPS02], we conducted a set of experiments executing 500 queries over the GSTD datasets indexed by the TB-tree using the `TrajectoryNNSearch` algorithm. The queries were initially executed with the the proposed MINDIST calculation, forming the Q_a query set, and then with the MINDIST calculation proposed in [TPS02], forming the Q_b query set. The set of 500 query objects (trajectories) were produced using GSTD also employing a Gaussian initial distribution and a random movement distribution. Then, a random 1% part of each trajectory was used as the query trajectory. Each query performance was measured in terms of execution time and actual distance evaluations between point and point, point and line and point and MBR.

Figure 14(a) illustrates the average execution time for query sets Q_a and Q_b. Clearly, the `TrajectoryNNSearch` algorithm with the proposed improvement over the MINDIST computation is always superior over the corresponding computation as proposed in [TPS02], in all datasets. The improvement over the execution time varies between 8% (in the GSTD 100 dataset) and 17% (in the GSTD 250 dataset). The efficiency of the proposed improvement over the MINDIST computation can be further established by Figure 14(b), illustrating the actual distance evaluations made from each alternative computation; Figure 14(b) shows that the proposed MINDIST computation requires in all settings almost half of the distance evaluations made by the analogous computation proposed in [TPS02].



(a) (b)

Figure 14: (a) Execution Time and (b) actual Distance Evaluations for query sets Q_a and Q_b increasing the number of moving objects

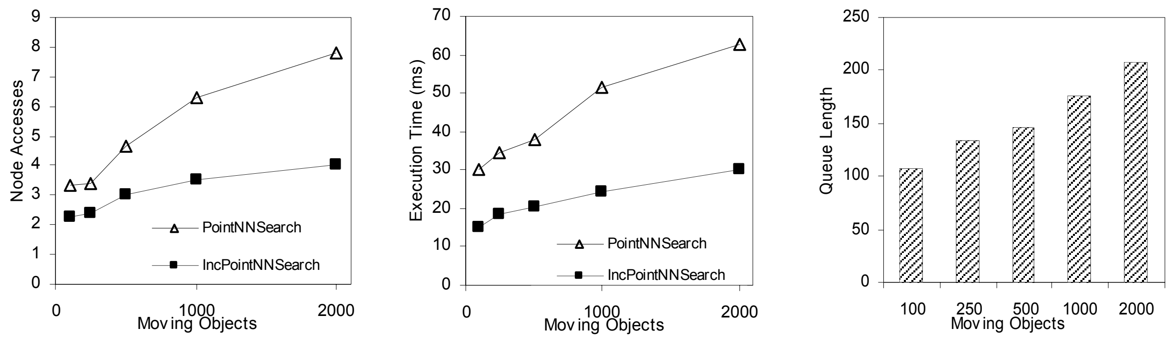
7.3. Results on the search cost of the historical non-continuous algorithms

The performance of the proposed algorithms was measured in terms of node accesses and execution time. Several queries were used in order to evaluate the performance of the proposed algorithms over the synthetic and real data. In particular, we have used the following query sets:

- Q1: the `PointNNSearch` and the `IncPointNNSearch` algorithms were evaluated with one set of 500 NN queries increasing the number of moving objects over the GSTD datasets indexed by both TB- and 3D R-tree. The queries used a random point in the 2D space and a time period of 1% of the temporal dimension for Q1.
- Q2: the `TrajectoryNNSearch` and the `IncPointNNSearch` algorithms were evaluated with one set of 500 NN queries increasing the number of moving objects over the GSTD datasets indexed by both TB and 3D R-tree. The set of 500 query objects (trajectories) was produced using GSTD employing also a Gaussian initial distribution and a random movement distribution. Then, in Q2 we used a random 1% part of each trajectory as the query trajectory.
- Q3, Q4: two sets of 500 *k*-NN queries over the real Trucks dataset increasing the number of *k* with fixed time and increasing the size of the time interval (with fixed *k*=1) respectively. For the `PointNNSearch` algorithm we used a random point in the 2D space with a 1% of time as query period, while for `TrajectoryNNSearch` algorithm we used a random part of a random trajectory belonging to the Buses dataset, temporally covering 1% of the time.

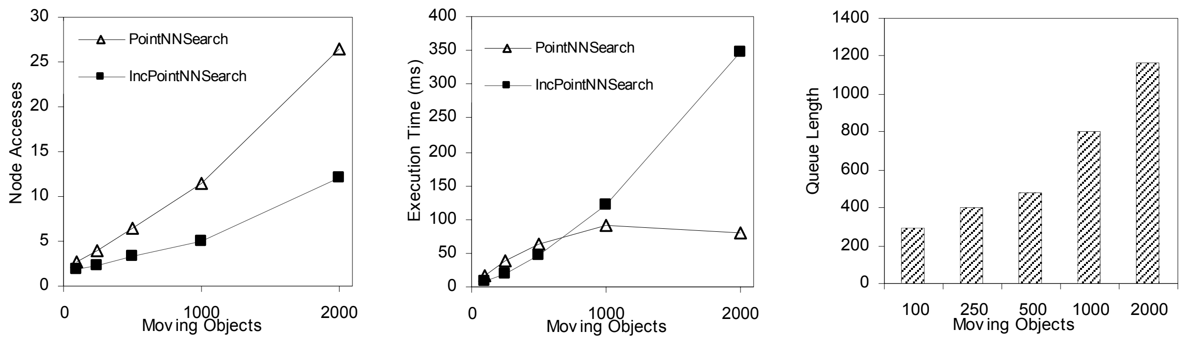
Figure 15 illustrates the results for the Q1 query set evaluating `PointNNSearch` and `IncPointNNSearch` algorithms over the 3D R-tree, in terms of (a) average number of node accesses and (b) average execution time per query. As it is clearly illustrated, the performance of both algorithms depends sub-linearly on the dataset cardinality, downgrading (more pages are accessed) as the cardinality grows. Another conclusion drawn from the same charts is that `IncPointNNSearch` algorithm outperforms the `PointNNSearch` algorithm in all datasets, in terms of both node accesses and execution time. Figure 15(c) illustrates the average length (in nodes) of the queue utilized by the `IncPointNNSearch` in order to answer the queries, increasing linearly with the cardinality of the dataset.

The Q1 query set evaluating `PointNNSearch` and `IncPointNNSearch` was also executed against the TB-tree, leading to the results presented in Figure 16. Although, just as reported for the 3D R-tree, the `IncPointNNSearch` outperforms `PointNNSearch` in terms of average node accesses per query in all datasets (Figure 16(a)), the actual average time required for each query execution (Figure 16(b)) by the `IncPointNNSearch`, increases faster than the respective execution time of the `PointNNSearch`, leading to a superiority of the non-incremental algorithm as the cardinality of the dataset grows.



(a) (b) (c)

Figure 15: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q1 executing point NN search over the 3D R-tree indexing the GSTD datasets increasing the number of moving objects



(a) (b) (c)

Figure 16: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q1 executing point NN search over the TB-tree indexing the GSTD datasets increasing the number of moving objects

Exactly the same trend as the one presented for the execution time of the `IncPointNNSearch` is presented in Figure 16(c) illustrating the length of the queue utilized by the respective algorithm. More specifically, `PointNNSearch` outperforms its incremental counterpart when the average length of the respective queue exceeds a certain number of nodes (approximately 400 nodes in the GSTD 500 dataset). The above conclusion can be also verified from the results of the 3D R-tree, where the length of the queue is always less than 400, leading to a superiority of the incremental algorithm. Regarding the comparison between the performance of the TB and the 3D T-tree, the latter outperforms the former as the dataset cardinality grows, like what was reported in [PJT00] for the simple range queries.

Figure 17 illustrates the results for the Q2 query set evaluating `TrajectoryNNSearch` and `IncTrajectoryNNSearch` algorithms over the 3D R-tree, in terms of average number of node accesses (a) and average execution time per query (b). The performance of both algorithms depends linearly on the dataset cardinality, downgrading as the dataset cardinality grows. Although `IncTrajectoryNNSearch` outperforms

TrajectoryNNSearch in all datasets in terms of node accesses, the average execution time of the incremental algorithm becomes greater than the respective time of the non-incremental one, as the dataset cardinality grows. The average queue length utilized by the IncTrajectoryNNSearch, is also illustrated in Figure 17(c); following the results for the execution time of the incremental algorithm, the queue length increases linearly with the cardinality of the dataset. This enlargement of the queue length is also responsible for the behavior showed regarding the comparison of the execution time between the TrajectoryNNSearch and the IncTrajectoryNNSearch algorithm; as the queue length increases, each update becomes a more expensive operation leading to the downgrade of the performance of the respective algorithm.

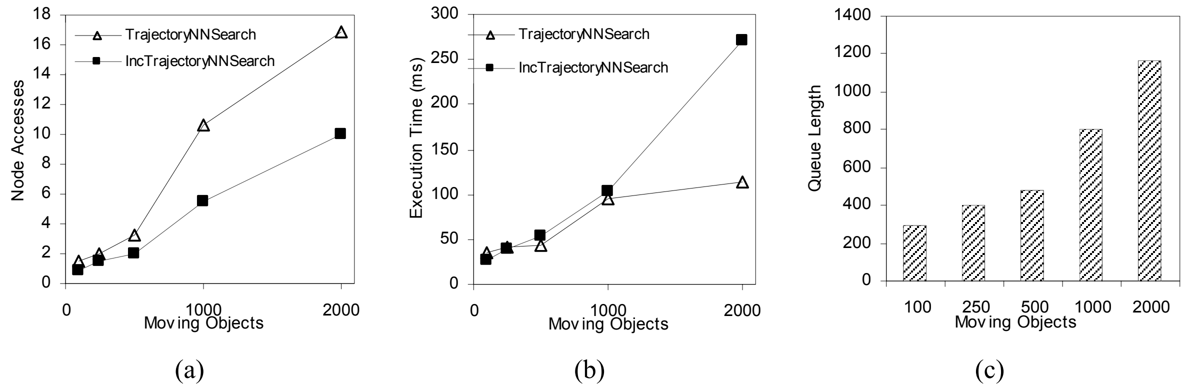


Figure 17: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q2 executing trajectory NN search over the 3D R-tree indexing the GSTD datasets increasing the number of moving objects

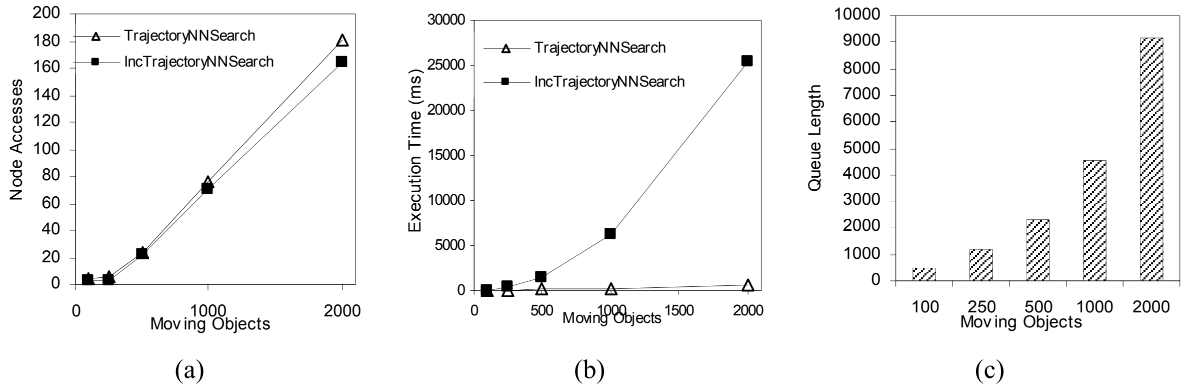
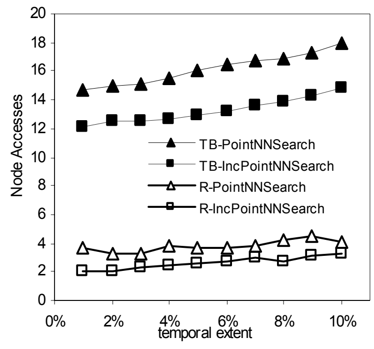


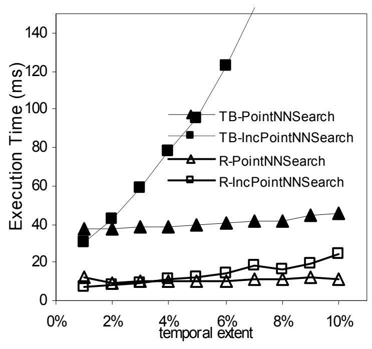
Figure 18: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q2 executing trajectory NN search over the TB-tree indexing the GSTD datasets increasing the number of moving objects

Regarding a comparison of the performance of the incremental algorithms illustrated in Figure 15 and Figure 17 leads to the observation that while in the first case, fewer node accesses leads to smaller execution time (than the non-incremental one), in the second case the execution time of the incremental algorithm becomes greater than the respective of its non incremental counterpart. This fact can be explained by observing the respective queue lengths: in the first case the queue length in not more than 200 objects (e.g. less than a typical BranchList), while in the second case, the queue length includes 1000's of objects resulting in a decrease of the algorithm's performance.

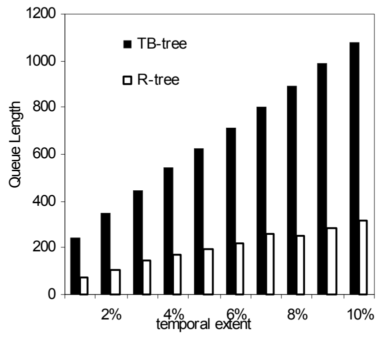
The results of the Q2 query set over the TB-tree are presented in Figure 18. While IncTrajectoryNNSearch always outperforms TrajectoryNNSearch in terms of average node accesses (Figure 18(a)), their disparity is not as significant as it was reported for the 3D R-tree. Moreover, the actual execution time of the incremental algorithm (Figure 18(b)) is always by far longer than the respective execution time of the non-incremental one. These results can be explained by two facts. The first one is that the actual execution time of the incremental algorithm depends heavily on the respective queue length which, as shown in Figure 18(c), exceeds 1000 nodes for the GSTD 250 dataset reaching the 9000 nodes in the GSTD 2000. The second is that TB-tree groups entries belonging to the same trajectory together, exploiting only the temporal order in which the entry insertion occurs ignoring at the same time any spatial proximity. This insertion strategy leads to nodes with high spatial (and low temporal) overlap, meaning that internal nodes will often cross the query trajectory, and the respective MINDIST will be equal to zero. Then, the internal nodes need to be visited since their MINDIST equals to zero and they are leading inside the queue, resulting to the loss of the advantage of the incremental algorithm. The same reasons also affect the comparison of the performance between the TB- and the 3D R-tree, with the latter one outperforming the former as the dataset cardinality grows.



(a)

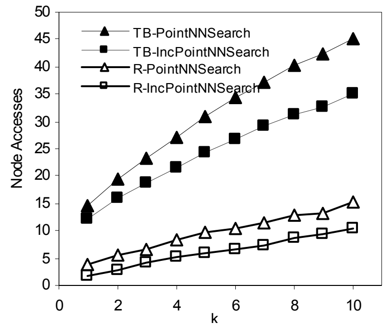


(b)

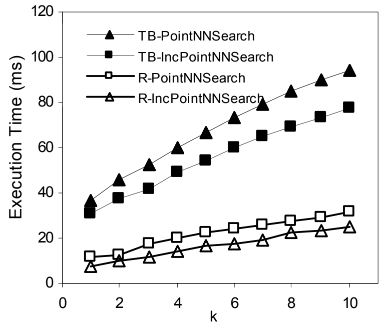


(c)

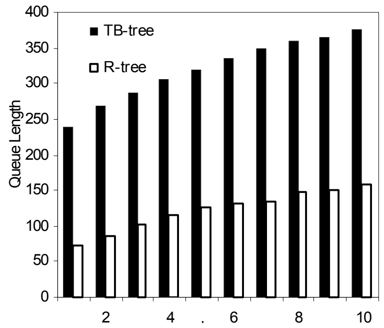
Figure 19: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q3 executing point NN search over the 3D R- and the TB-tree indexing the Trucks dataset increasing the query temporal extent



(a)



(b)

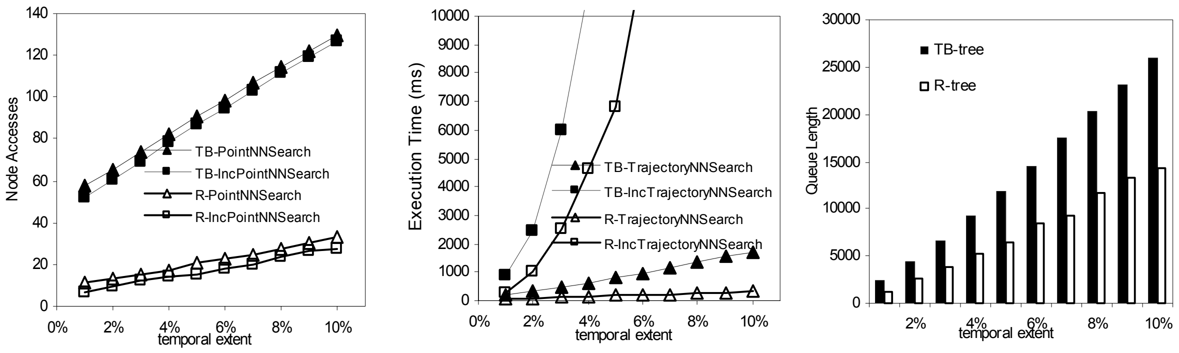


(c)

Figure 20: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q3 executing point NN search over the 3D R- and the TB-tree indexing the Trucks dataset increasing the number of k

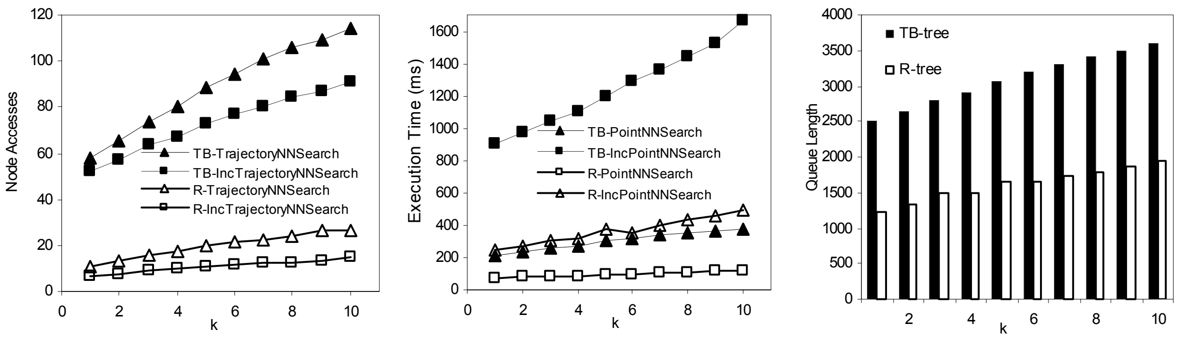
The performance of the historical non-continuous point NN algorithms increasing the query temporal extent, in terms of average node access and average execution time per query, is shown in Figure 19 against the 3D R-tree and the TB-tree, both indexing the Trucks dataset. Clearly, under both indexes, the number of node accesses needed for the processing of a NN query, increases linearly with the query temporal extent, with the `IncPointNNSearch` being always below the `PointNNSearch`. In terms of execution time, both indexes show the same behavior having a breakeven point where the superlinearly increasing execution time of the `IncPointNNSearch` (a consequence of the increasing queue length (Figure 19 (c))) becomes even with the linearly increasing execution time of the `PointNNSearch` algorithm. Regarding the TB-tree, the breakeven point is around the 1.5% of the temporal extent while in the 3D R-tree increases around 3.5%.

Figure 20 illustrates the average number of node accesses and execution time per historical non-continuous point query increasing the number of *k* against the Trucks dataset indexed by the 3D R-tree and TB-tree. Under both indexes it is clear that the incremental algorithm outperforms the `PointNNSearch` in terms of both average node accesses and execution time. Using the 3D R-tree, the performance of both algorithms decreases linearly with the number of k, whereas when using the TB-tree the reduction is sub-linear.



(a) (b) (c)

Figure 21: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q4 executing trajectory NN search over the 3D R- and the TB-tree indexing the Trucks dataset increasing the query temporal extent



(a) (b) (c)

Figure 22: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q4 executing trajectory NN search over the 3D R-tree indexing the Trucks dataset increasing the number of k

The results for the historical non-continuous trajectory NN algorithms increasing the query temporal extent against the 3D R-tree and TB-tree indexing the Trucks dataset are illustrated in Figure 21. Once again, the number of node accesses required for the processing of a NN query with both algorithms under both indexes, increases linearly with the query temporal extent. However, regarding the execution time, the performance of the incremental algorithm grows superlinearly with the temporal extent as a consequence of the excessive queue length (Figure 21(c)).

The performance of the historical non-continuous trajectory query increasing the number of *k* against the Trucks dataset is shown in Figure 22 where the `TrajectoryNNSearch` algorithm outperforms its incremental counterpart in terms of execution time, with the respective queue containing in any case more than 1000 nodes.

7.4. Results on the search cost of the historical continuous algorithms

In coincidence with the experiments conducted for the historical non-continuous algorithms, the historical continuous NN search algorithms were evaluated, also in terms of node accesses and execution time, with the following query sets:

- Q5: the `HContPointNNSearch` algorithm was evaluated with one set of 500 NN queries increasing the number of moving objects over the GSTD datasets indexed by both TB- and 3D R-tree like what was done for query set Q1.
- Q6: the `HContTrajectoryNNSearch` algorithm was evaluated with one set of 500 NN queries increasing the number of moving objects over the GSTD datasets indexed by both TB- and 3D R-tree like what was done for query set Q2
- Q7, Q8: two sets of 500 *k*-NN queries over the real Buses dataset increasing the number of *k* with fixed time and increasing the size of the time interval (with fixed *k* = 1), respectively. For the `HContPointNNSearch` algorithm we used a random point in the 2D space with a 1% of time as query period, while for `HContTrajectoryNNSearch` algorithm we used a random part of a random trajectory belonging to the Trucks dataset, temporally covering 1% of the time.

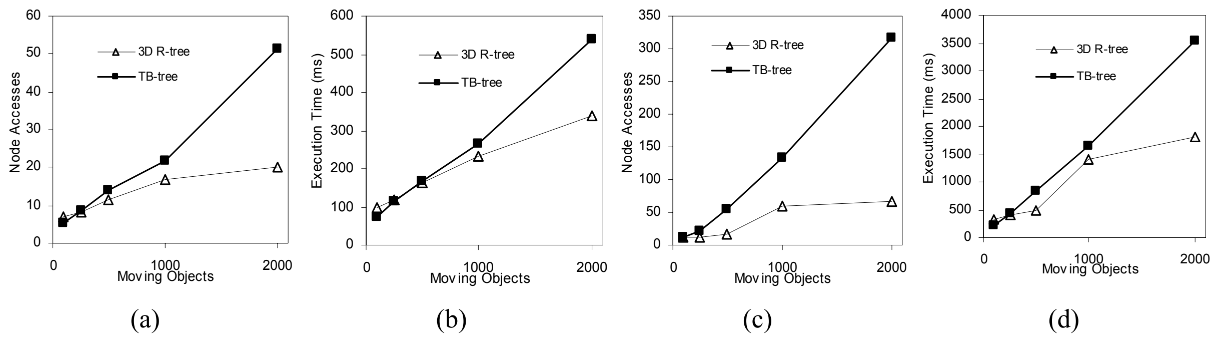
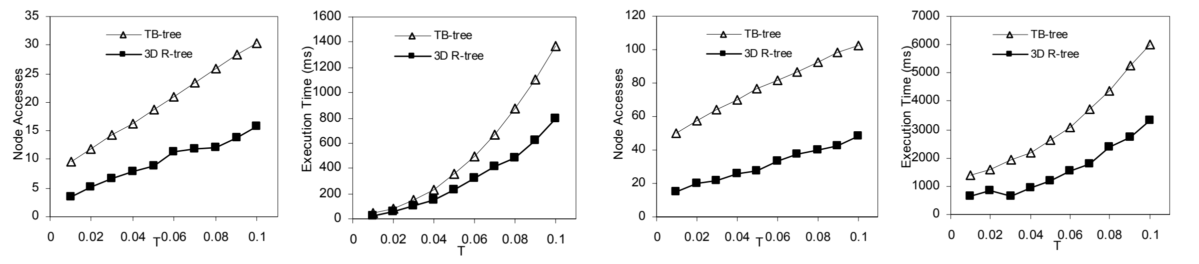


Figure 23: Node Accesses and Execution Time in queries Q5 (a, b) and Q6 (c, d) over the 3D R-tree and the TB-tree indexes increasing the number of moving objects

Figure 23 (a, b) illustrates the results of the `HContPointNNSearch` algorithm over the GSTD datasets by increasing the number of moving objects in terms of (a) average node accesses and (b) average execution time per query. Just as in its historical non-continuous counterpart, the performance of the algorithm depends linearly on the dataset cardinality downgrading as the cardinality grows, while the average execution time for both indexes follows the same trend as the average number of visited nodes. Another result gathered is that, as the cardinality grows, the 3D R-tree outperforms the TB-tree following the same trend illustrated in [PJT00] for simple range queries. Similar results are illustrated in Figure 23 (c, d) where the `HContTrajectoryNNSearch` algorithm is executed against the TB- and the 3D R-tree indexing the GSTD datasets.

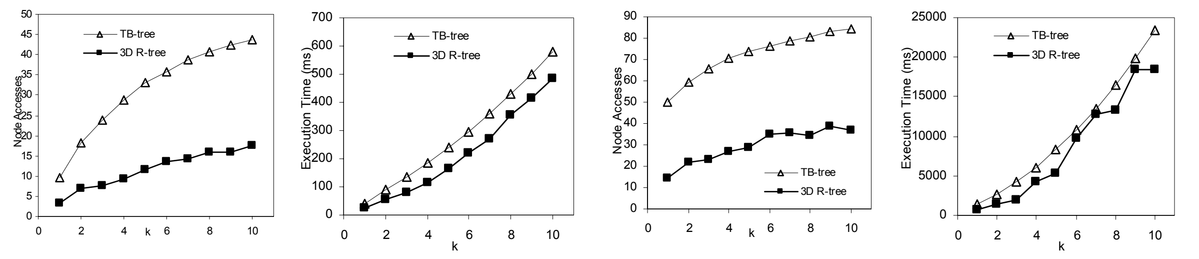
A comparison between the historical non-continuous NN algorithms with their continuous counterpart (e.g. Figure 15 and Figure 16 vs. Figure 23 (a,b), and Figure 17 and Figure 18 vs. Figure 23 (c,d)), shows that the historical continuous algorithms are much more expensive than the non-continuous ones. This conclusion was expected since the historical continuous algorithms do not utilize a single distance to prune the search space; instead they use a list of moving distances, which in general stores greater distances than the minimum. Actually, the historical non-continuous algorithms prune the search space with the minimum possible distance stored inside the *Nearests* list, therefore performing pruning much more efficiently than their continuous counterpart.

The scaling of the historical continuous algorithms with the query temporal extent is presented in Figure 24. Both algorithms (`HContPointNNSearch` and `HContTrajectoryNNSearch`) were executed over the real Buses dataset indexed by the TB- and the 3D R-tree. From Figure 24 (a) and (c) it is clear that the performance of both algorithms in terms of node accesses is sub-linear with respect to the query temporal extent. Nevertheless, the actual execution time needed by each query increases superlinearly with the query extent, as a consequence of the increasing length of the query output (the *Nearests* list). The performance of the historical continuous NN algorithms increasing the number of *k* against the Buses dataset indexed by the TB and the 3D R-tree is illustrated in Figure 25. As drawn from Figure 25 (a) and (c), the average number of node accesses required for the processing of a *k*-HCNN point or trajectory query increases sub-linearly with *k*. However, the actual execution time presented in Figure 25 (b) and (d) increases superlinearly with the *k*, similarly with the temporal extent, as a consequence of the increasing size of the query output (the *k Nearests* lists).



(a) (b) (c) (d)

Figure 24: Node Accesses and Execution Time in queries Q7 (a, b) and Q8 (c, d) over the 3D R-tree and the TB-tree indexes increasing the query temporal extent



(a) (b) (c) (d)

Figure 25: Node Accesses and Execution Time in queries Q7 (a, b) and Q8 (c, d) over the 3D R-tree and the TB-tree indexes increasing the number of *k*

7.5. Summary of the Experiments

In order to measure the performance of our algorithms we conducted the above experimental study based on synthetic and real datasets. Regarding the historical non-continuous algorithms, it has been shown that while the incremental (best-first) approach is always less expensive than the non-incremental (depth-first) in terms of node accesses, its actual execution time heavily depends on the used queue length. In general, the best first approach outperforms its competitor only for point NN queries under small temporal extent (less than 2-4% depending on the index used and under any *k*), while in all other cases the depth first approach takes less time to be executed. This drawback of the incremental algorithms is mainly due to the queue length which may become huge, especially in the case of the TB-tree. Regarding a comparison between the two used indexes, the 3D R-tree outperforms the TB-tree in terms of both node accesses and execution time. Moreover, we demonstrated that our improvement over the MINDIST computation can sufficiently increase the performance of the proposed algorithms.

Most of the presented algorithms, in terms of node accesses, are linear or sub-linear with the main parameters of our experimental study: the dataset cardinality, the query temporal extent and the number of *k*. However, the execution time of the `IncPointNNSearch` and `IncTrajectoryNNSearch` algorithms seems to grow super-linearly with the query temporal extent as a result of the increasing queue length, similarly with the execution time of `HContPointNNSearch` and `HContTrajectoryNNSearch`, which have the same trend with respect to the temporal extend and the number of *k*, as a consequence of the increasing *Nearests* list length.

Algorithm	*TB-tree*	*3D R-tree*
`PointNNSearch`	0.022%	0.006%
`IncPointNNSearch`	0.010%	0.003%
`TrajectoryNNSearch`	0.148%	0.014%
`IncTrajectoryNNSearch`	0.134%	0.008%
`HContPointNNSearch`	0.042%	0.016%
`HContTrajectoryNNSearch`	0.259%	0.053%

Table 2: Actual indexed space accessed by each NN algorithm for the GSTD 2000 dataset

Table 2 summarizes the pruning power of our algorithms presenting the percentage of the indexed space accessed in order to execute all the proposed algorithms with *k*=1 and temporal extent the 1% of the indexed time. As it can be concluded our algorithms show high pruning ability, well bounding the space to be searched in order to answer NN and HCNN queries.

8. Conclusion and Future Work

NN queries have been in the core of the spatial and spatiotemporal database research during the last decade. The majority of the algorithms processing such queries so far mainly deals with either stationary or moving query points over static datasets or future (predicted) locations over a set of continuously moving points. In this work, acknowledging the contribution of related work, we presented the first complete treatment of historical NN queries over moving object trajectories stored on R-tree like structures. Based on our proposed novel metrics, which support our searching and pruning strategies, we presented algorithms answering the NN and HCNN queries for stationary query points or trajectories and generalized them to search for the *k* nearest neighbors. The algorithms are applicable to R-tree variations for trajectory data, among which, we used both 3D R-tree and TB-tree for our performance study. Under various synthetic datasets (which were generated by the GSTD data generator) and two real trajectory datasets, we illustrated that our algorithms show high pruning ability, well bounding the space to be searched in order to answer NN and HCNN queries. The pruning power of our algorithms is also verified in the case of the *k*-NN and *k*-HCNN queries (for various values of *k*).

Future work includes the development of algorithms to support distance join queries ("*find pairs of objects passed nearest to each other (or within distance d from each other) during a certain time interval and/or under a certain space constraint*"). A second research direction includes the development of selectivity estimation formulae for query optimization purposes investing on the work presented in [TSP03] for predictive spatiotemporal queries.

Acknowledgements

Research partially supported by the GeoPKDD ("Geographic Privacy-aware Knowledge Discovery and Delivery") project funded by the European Community under FP6-014915 contract. Research grants by the Archimedes and Pythagoras EPEAEK II Programmes jointly funded by the European Community and the Greek Ministry of National Education and Religious Affairs are also acknowledged. We also thank the anonymous reviewers for providing valuable comments which improved the quality of the paper.

References

[BJKS02] Benetis, R., Jensen, C., Karciauskas, G., and Saltenis, S., Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. *Proceedings of IDEAS*, 2002.

[BW01] Babu, S., and Widom, J., Continuous Queries over Data Streams, *SIGMOD Record*, vol.30(3), pp. 109-120, September 2001.

[CF98] Cheung, K.L., and Fu, A.,W., Enhanced Nearest Neighbour Search on the R-tree. *SIGMOD Record*, vol. 27(3), pp. 16-21, September 1998.

[FGPT05] Frentzos, E., Gratsias, K., Pelekis, N., and Theodoridis, Y., Nearest Neighbor Search on Moving Object Trajectories, *Proceedings of SSTD,* 2005.

[Gut84] Guttman, A., Rtrees: a Dynamic Index Structure for Spatial Searching, *Proceedings of ACM SIGMOD*, 1984.

[HS99] Hjaltason, G., and Samet, H., Distance Browsing in Spatial Databases, *ACM Transactions in Database Systems*, vol. 24(2), pp. 265-318, 1999.

[HXL05] Hu, H., Xu, J., and Lee, D.L., A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects, *Proceedings of ACM SIGMOD*, 2005.

[ISS03] Iwerks, G.S., Samet, H., and Smith, K., Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates, *Proceedings of VLDB*, 2003.

[MNPT05] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A. N., and Theodoridis, Y., *R-trees: Theory and Applications*, Springer-Verlag, 2005.

[MHP05] Mouratidis K., Hadjieleftheriou M., Papadias, D., Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring, *Proceedings of ACM SIGMOD*, 2005.

[MXA04] Mokbel, M.F., Xiong, X., and Aref, W.G., SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases, *Proceedings of ACM SIGMOD*, 2004.

[PJT00] Pfoser D., Jensen C. S., and Theodoridis, Y., Novel Approaches to the Indexing of Moving Object Trajectories, *Proceedings of VLDB*, 2000.

[RKV95] Roussopoulos, N., Kelley, S., and Vincent, F., Nearest Neighbor Queries, *Proceedings of ACM SIGMOD,* 1995.

[SJLL00] Saltenis, S., Jensen, C. S., Leutenegger, S. and Lopez, M., Indexing the Positions of Continuously Moving Objects, *Proceedings of ACM SIGMOD*, 2000.

[SKS03] Shahabi, C., Kolahdouzan, M., and Sharifzadeh, M., A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases, *GeoInformatica*, vol. 7(3), pp. 255-273, 2003.

[SR01] Song, Z., and Roussopoulos, N., K-Nearest Neighbor Search for Moving Query Point, *Proceedings of SSTD*, 2001.

[TP02] Tao, Y., and Papadias, D., Time Parameterized Queries in Spatio-Temporal Databases, *Proceedings of ACM SIGMOD*, 2001.

[TPS02] Tao, Y., Papadias, D., and Shen, Q., Continuous Nearest Neighbor Search, *Proceedings of VLDB*, 2002.

[The03] Theodoridis, Y., The R-tree Portal. URL: www.rtreeportal.org (accessed 13 December 2005).

[TSN99] Theodoridis, Y., Silva, J. R. O., and Nascimento, M. A., On the Generation of Spatio-temporal Datasets, *Proceedings of SSD,* 1999.

[TSP03] Tao, Y., Sun, J., and Papadias, D., Analysis of predictive spatio-temporal queries, *ACM Transactions on Database Systems* vol. 28(4), pp. 295-336, December 2003.

[TVS96] Theodoridis, Y., Vazirgiannis, M., and Sellis, T., Spatio-temporal Indexing for Large Multimedia Applications. *Proceedings of ICMCS*, 1996.

[YPK05] Yu, X., Pu, K., and Koudas, N., Monitoring k-Nearest Neighbor Queries Over Moving Objects. *Proceedings of ICDE*, 2005.

[XMA05] Xiong, X., Mokbel, M., and Aref, W., SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. *Proceedings of ICDE*, 2005.

APPENDIX A: Calculation of the minimum "horizontal" distance between two 3D line segments

The Euclidean ("horizontal") distance function between the projections of two 3D line segments, *P* and *Q*, on the 2D (x-,y-) plane is:

$$Dist = \sqrt{(Q_x - P_x)^2 + (Q_y - P_y)^2} \tag{1}$$

where $Q_x = Q_{1x} + (Q_{2x} - Q_{1x})\Delta t$, $Q_y = Q_{1y} + (Q_{2y} - Q_{1y})\Delta t$, $P_x = P_{1x} + (P_{2x} - P_{1x})\Delta t$ and $P_y = P_{1y} + (P_{2y} - P_{1y})\Delta t$. Replacing Q_x, Q_y, P_x, P_y in (1), we get

$$Dist = \sqrt{(Q_{1x} + (Q_{2x} - Q_{1x})\Delta t - P_{1x} - (P_{2x} - P_{1x})\Delta t)^2 + (Q_{1y} + (Q_{2y} - Q_{1y})\Delta t - P_{1y} - (P_{2y} - P_{1y})\Delta t)^2}$$

In the sequel, we use the square of the Euclidean distance for sake of readiness.

$$Dist^2 = (Q_{1x} + (Q_{2x} - Q_{1x})\Delta t - P_{1x} - (P_{2x} - P_{1x})\Delta t)^2 + (Q_{1y} + (Q_{2y} - Q_{1y})\Delta t - P_{1y} - (P_{2y} - P_{1y})\Delta t)^2 =$$

$$= ((Q_{2x} - Q_{1x} - P_{2x} + P_{1x})\Delta t + (Q_{1x} - P_{1x}))^2 + ((Q_{2y} - Q_{1y} - P_{2y} + P_{1y})\Delta t + (Q_{1y} - P_{1y}))^2 =$$

$$= ((Q_{2x} - Q_{1x} - P_{2x} + P_{1x})^2 + (Q_{2y} - Q_{1y} - P_{2y} + P_{1y})^2)\Delta t^2 +$$

$$+2((Q_{2x} - Q_{1x} - P_{2x} + P_{1x})(Q_{1x} - P_{1x}) + (Q_{2y} - Q_{1y} - P_{2y} + P_{1y})(Q_{1y} - P_{1y}))\Delta t + (Q_{1x} - P_{1x})^2 + (Q_{1y} - P_{1y})^2$$

Setting

$$\mathrm{A} = (Q_{2x} - Q_{1x} - P_{2x} + P_{1x})^2 + (Q_{2y} - Q_{1y} - P_{2y} + P_{1y})^2 \tag{2}$$

$$\mathrm{B} = 2((Q_{2x} - Q_{1x} - P_{2x} + P_{1x})(Q_{1x} - P_{1x}) + (Q_{2y} - Q_{1y} - P_{2y} + P_{1y})(Q_{1y} - P_{1y})) \tag{3}$$

$$C = (Q_{1x} - P_{1x})^2 + (Q_{1y} - P_{1y})^2 \tag{4}$$

and replacing Δt according to the following formula $\Delta t = \frac{t - t_1}{t_2 - t_1}$, the Euclidean "horizontal" distance function of two 3D line segments is computed as follows:

$$Dist^2 = \frac{A}{(t_2 - t_1)^2} t^2 + (\frac{B}{t_2 - t_1} - \frac{2At_1}{(t_2 - t_1)^2})t + \frac{At_1^2}{(t_2 - t_1)^2} - \frac{Bt_1}{t_2 - t_1} + C, \tag{5}$$

where *A*, *B*, *C* are defined by formulas (2), (3), (4), respectively.

As proved before, the square of the Euclidean "horizontal" distance function between two 3D line segments has the quadratic form $P(t) = At^2 + Bt + C$, which is minimized at $P_{\min} = C - \frac{B^2}{4A}$ for $t = -\frac{B}{2A}$.

Thus, in our case

$$Dist^2_{\min} = \frac{At_1^2}{(t_2 - t_1)^2} - \frac{Bt_1}{t_2 - t_1} + C - \frac{(\frac{B}{t_2 - t_1} - \frac{2At_1}{(t_2 - t_1)^2})^2}{\frac{4A}{(t_2 - t_1)^2}} \tag{6}$$

for

$$t = \frac{\left(\frac{2At_1}{(t_2-t_1)^2} - \frac{B}{t_2-t_1}\right)}{2\frac{A}{(t_2-t_1)^2}} \tag{7}$$

where *A*, *B*, *C* are defined by formulas (2), (3), (4), respectively.

We have to note that formula (6) can be used in case where *t* calculated by formula (7) is inside the query time period $Q_{per}[t_{start}, t_{end}]$. Otherwise, we distinguish between the following two cases:

1. if $t \leq t_{start}$, then the minimum "horizontal" distance is provided by formula (5) by setting $t = t_{start}$
2. if $t \geq t_{end}$, then the minimum "horizontal" distance is provided by formula (5) by setting $t = t_{end}$.

APPENDIX B: Maintaining the *Nearests* List

The pseudo-code of the `UpdateNearests` function, which is responsible for the maintenance of the *Nearests* List, is presented in Figure 26. In particular, the algorithm iterates through the elements of the active *Nearests* list searching for those elements temporally overlapping the checked entry (*CM*). When such an element is found, the algorithm applies linear interpolation in both entries (the checked and the one already on the list) producing two new entries having the same temporal extent (*M* and *T*). Then, it compares the two distance functions in order to determine whether the entry already on the list is to be replaced or not.

```
Algorithm UpdateNearests (List Nearests, struct CM, Roof)
FOR EACH T IN Nearests
  IF (T.$T_S$, T.$T_E$)Overlaps(CM.$T_S$, CM.$T_E$)
    M=Interpolate(CM, Max(CM.$T_S$, T.$T_S$), Min(CM.$T_E$, T.$T_E$))
    T=Interpolate(T, Max(CM.$T_S$, T.$T_S$), Min(CM.$T_E$, T.$T_E$))
    IF M.$D_{Max}$ < T.$D_{Min}$
      Nearests.Replace T with M
    ELSEIF M.$D_{Max}$ < T.$D_{Max}$
      D = Discriminant(M-T)
      IF D < 0
        IF T.$D_{Min}$ > M.$D_{Min}$ THEN Nearests.Replace T with M
      ELSEIF D=0
        IF T.$D_{Max}$ > M.$D_{Max}$ THEN Nearests.Replace T with M
      ELSE
        $RR_1$=Solution1(T - M):$RR_2$=Solution2(T - M):
                $R_1$=Min($RR_1$,$RR_2$):$R_2$=Max($RR_1$,$RR_2$)
        IF $R_2$<T.$T_S$ OR $R_1$>T.$T_E$
          IF T.$D_{Max}$ > M.$D_{Max}$ THEN Nearests.Replace T with M
        ELSEIF $R_2$<T.$T_E$ AND $R_1$>T.$T_S$
          IF M.$D_{min}$ < T.$D_{min}$
            $M_1$=Part(M,,$R_1$):$M_2$=Part(M,$R_2$):$T_1$=Part(T,$R_1$,$R_2$):
            Nearests.Replace T with ($M_1$,T1,$M_2$)
          ELSE
            $T_1$=Part(T,,$R_1$):$T_2$=Part(T,$R_2$):M1=Part(M,$R_1$,$R_2$)
            Nearests.Replace T with ($T_1$,$T_2$,$M_1$)
          ENDIF
        ELSE
          IF M($R_1$ - 1)<T($R_1$ - 1)
            $M_1$=Part(M,,$R_1$):$T_1$=Part(T,$R_1$): Nearests.Replace T with ($M_1$,$T_1$)
          ELSE
            $T_1$=Part(T,,$R_1$):$M_1$=Part(M,$R_1$): Nearests.Replace T with ($T_1$,$M_1$)
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  ENDIF
  Roof=max(Roof,T.$D_{max}$)
NEXT
```

Figure 26: UpdateNearests Algorithm

Figure 27 graphically explains all the possible comparisons between the parabolas of two "*Moving Distance*" functions.

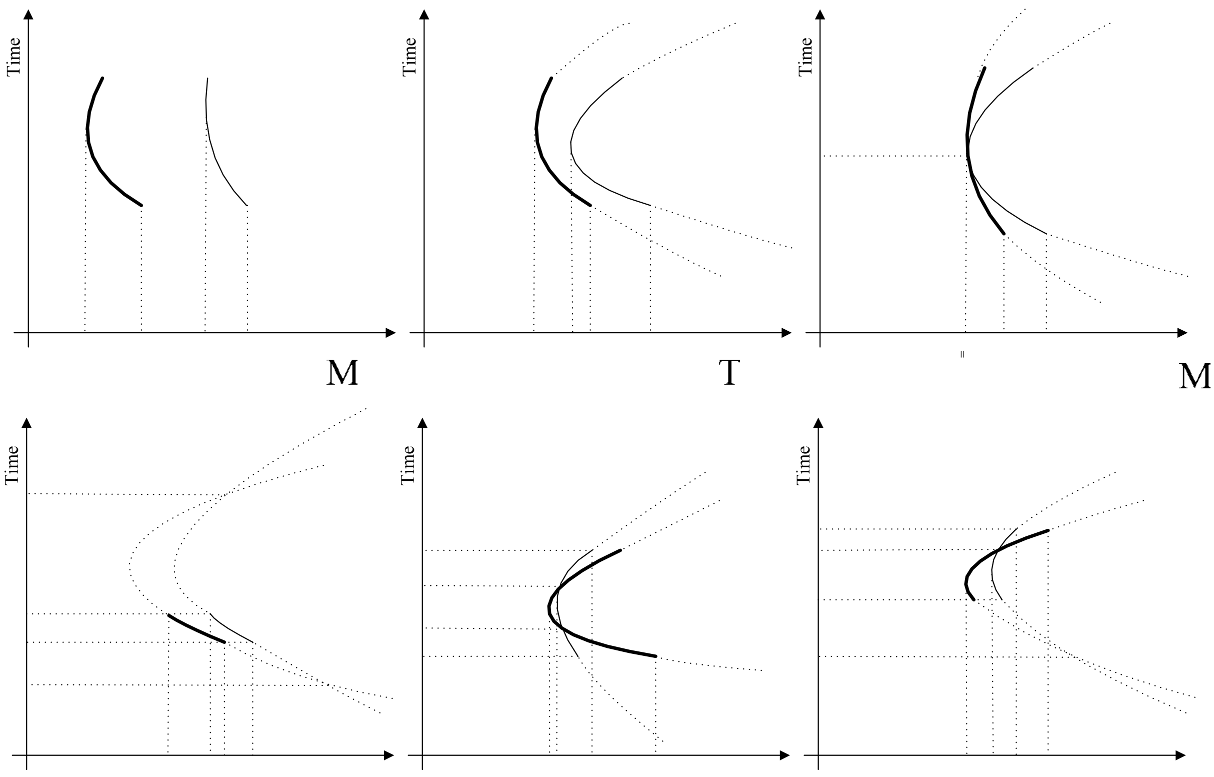


Figure 27: Graphical illustration of UpdateNearests Algorithm Comparisons

Figure 27(a) corresponds to line 5 of the algorithm presented in Figure 26, where the maximum distance of *M* is smaller than the minimum of *T*, leading to the replacement of *T* with *M*. Otherwise, after computing the discriminant of the difference between the distance functions of *M* and *T*, we have to distinguish among three different cases:

- The discriminant is less than zero, meaning that the two functions *M* and *T* are asymptotic and they do not intersect (Line 9); we only have to check their minimum in order to determine which is the global minimum (see Figure 27(b))
- The discriminant is equal to zero, meaning that the two functions osculate in their common minimum (Line 11); we only have to check their maximum in order to determine the global minimum (see Figure 27(c))
- The discriminant is greater than zero, meaning that the two functions intersect in two points (Line 13). In this case, we have to determine whether these time instances are inside the entry's lifetime. Hence, we further distinguish among three sub-cases:
  - Both solutions are outside the temporal extent of *M* (and *T*) (Line 15). We only have to check their maximum in order to determine which is the globally minimum inside the current temporal interval (see Figure 27(d))
  - Both solutions are inside the temporal extent of *M* (and *T*) (Line 17). We must break apart the entry into 3 different entries (see Figure 27(e)) and determine the part of *T* to be replaced by *M*.
  - Only one solution is inside the temporal extent of *M* (Line 25). We must break apart the entry into 2 different entries (see Figure 27(f)) and determine the part of *T* to be replaced by *M*.