

# **Nearest Neighbor Search on Moving Object Trajectories**

Elias Frentzos, Kostas Gratsias, Nikos Pelekis, Yannis Theodoridis

Laboratory of Information Systems

Department of Informatics

University of Piraeus

Hellas



**Technical Report Series**

---

**UNIPi-ISL-TR-2005-2**

**April 2005**

# Nearest Neighbor Search on Moving Object Trajectories

Elias Frentzos<sup>†</sup>, Kostas Gratsias, Nikos Pelekis, Yannis Theodoridis

Department of Informatics  
University of Piraeus  
Piraeus, Hellas  
URL: <http://isl.cs.unipi.gr/db>  
E-mail: {efrentzo, gratsias, npelekis, ythead}@unipi.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 branch-and-bound algorithms vary with respect to the type of the query object (stationary or moving point) as well as the type of the query result (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 a member of the R-tree family for trajectory data (the TB-tree), we demonstrate their scalability and efficiency through an extensive experimental study using 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, spatio-temporal queries are continuous in nature. In contrast to snapshot queries, which are invoked only once, continuous

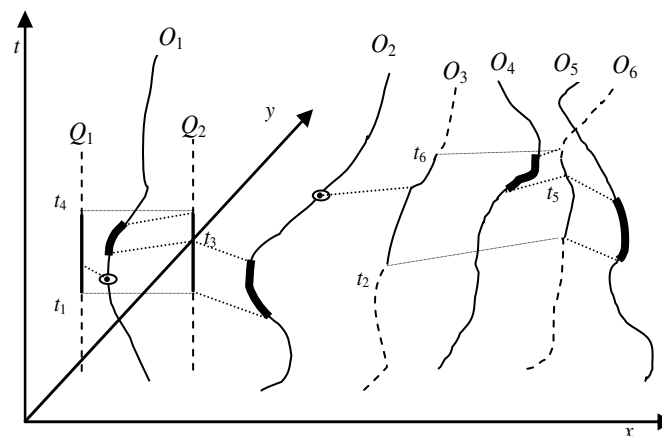
---

<sup>†</sup> Contact author's address: 80 Karaoli-Dimitriou St., GR-18534 Piraeus, Greece. Tel: +30-2104142449, Fax: +30-2104142264

queries require continuous evaluation as the query result becomes invalid after a short period of time. Secondly, we typically have to deal with vast 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 positions of a set of continuously moving points ([BJKS02], [TP02], [ISS03]). 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 will 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 users of MODs 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 (the nearest distance depends on time) along with a collection of trajectory ids and the appropriate time intervals for which each moving object is valid  $\{i_1[t_1, t_2], i_2[t_2, t_3], \dots\}$ .



**Figure 1:** 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  $[t_5, t_6]$ .

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 spatio-temporal 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 continuous  $k$ -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 a set of four query processing algorithms to perform NN search on R-tree-like structures storing historical information about moving objects. The description of our branch-and-bound traversal algorithms for different queries depends on the type of the query object as well as on whether the query itself is continuous or not. The 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.
- We conduct a comprehensive set of experiments over synthetic and real datasets demonstrating that the algorithms are highly scalable and efficient in terms of node accesses 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 and 5 constitute the core of the paper describing in detail the query processing algorithms to perform NN search over historical trajectory information (Section 4) together with their continuous counterparts (Section 5). Section 6 presents the results of our experimental study and Section 7 provides the conclusions of the paper and some interesting research directions.

## 2. Related Work

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 computational expensive).

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 process 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. Finally, 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.

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  $\{O_1, O_2, \dots, O_N\}$ . The trajectory  $T_i$  of a moving object  $O_i$  consists of  $M_i$  3D-line segments  $\{L_{i1}, L_{i2}, \dots, L_{iM_i}\}$ . Each 3D line segment  $L_j$  is of the form  $((x_{j-start}, y_{j-start}, t_{j-start}), (x_{j-end}, y_{j-end}, t_{j-end}))$ , where  $t_0 \leq t_{j-start} < t_{j-end} \leq now$ . 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 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 CNN queries:

- $CNN_{Q_p}(D, Q_p, Q_{per})$  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  $[t_{i-start}, t_{i-end})$  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  $[t_{i-start}, t_{i-end})$  are mutually disjoint and their union forms  $Q_{per}$ .
- Similarly,  $CNN_{Q_T}(D, Q_T, Q_{per})$  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  $[t_{i-start}, t_{i-end})$  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 1-st, 2-nd, ...,  $k$ -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 continuous queries are generalized to produce  $k$ -CNN requesting to provide with  $k$  lists of  $\{R_i, [t_{i-start}, t_{i-end}), O_i\}$  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 (non-continuous):  $O_1$  (1<sup>st</sup> NN) and  $O_2$  (2<sup>nd</sup> NN)
- Query 2 (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 (non-continuous):  $O_2$  (1<sup>st</sup> NN) and  $O_4$  (2<sup>nd</sup> NN)
- Query 4 (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  $[t_5, t_6]$ .

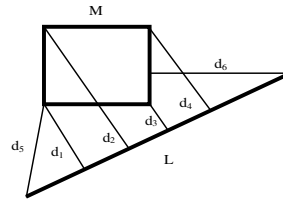
### 3.2 Metrics

We exploit on the definition of the minimum distance metric (MINDIST) presented in [RKV95] between points and rectangles, in order to calculate, on the one hand, the minimum distance between line segments and rectangles and, on the other hand, 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  $P$  in the  $n$ -dimensional space and a rectangle  $R$  in the same 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  $R$ ).

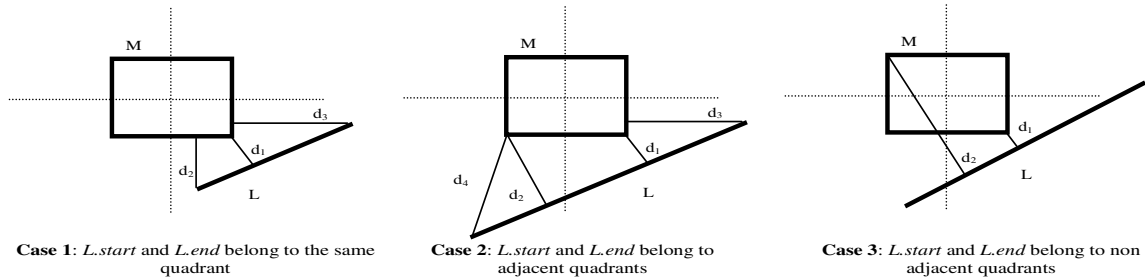
In the sequel, Tao et al. [TPS02] proposed a method to calculate the MINDIST between a 2D line segment  $L$  and a rectangle  $M$  (Figure 2). They initially determine whether  $L$  intersects  $M$ ; if so, MINDIST is set to zero. Otherwise, they choose the shortest among six distances, namely the four 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.



**Figure 2:** Calculating MINDIST between a line segment and a rectangle [TPS02]

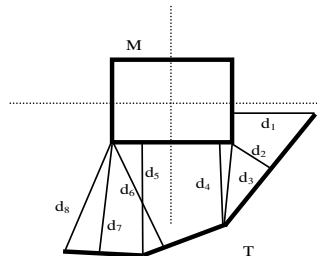
In this paper, we propose a more efficient method to calculate MINDIST between a line segment  $L$  and a rectangle  $M$  (Figure 3). As before, if  $L$  intersects  $M$ , then MINDIST is obviously zero. Otherwise, we decompose the space in four quadrants using the two axes passing through the center of  $M$  and we determine the quadrants  $Q_s$  and  $Q_e$  in which the start ( $L.start$ ) and the end ( $L.end$ ) point of  $L$  lie in, respectively.



**Figure 3** 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 ( $Q_s$ )): (i) MINDIST between the corner of  $M$  in  $Q_s$  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  $Q_s$  and  $Q_e$ , respectively): (i) MINDIST between the corner of  $M$  in  $Q_s$  and  $L$ , (ii) MINDIST between the corner of  $M$  in  $Q_e$  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  $Q_s$  and  $Q_e$ , respectively): two MINDIST between the two corners of  $M$ , that do not belong in either  $Q_s$  or  $Q_e$ , and  $L$ .



**Figure 4:** 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]. Finally, we extend the above algorithm in order to calculate MINDIST metric between a the projection of a trajectory  $T$  on the plane (usually called *route*) and a



rectangle  $M$  (Figure 4). 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.

## 4. NN Algorithms over trajectories

In this section we describe in details the algorithms answering the four 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. NN algorithm for stationary query objects (points)

The NN algorithm for stationary query objects (PointNNSearch algorithm, illustrated in Figure 5, 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 actual Euclidean distance from
    point  $Q$ 
2.  FOR i = 1 to N.EntriesCount
3.    E = N.Entry(i)
    // If entry is (fully or partially) inside the period
4.    IF  $Q_{per}$  Overlaps (E. $T_S$ , E. $T_E$ )
    // Compute entry's spatial extent inside the period
5.    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
6.    Dist = Euclidean_Dist_2D(Q, nE)
7.    IF Dist < Nearest.Dist
8.      Nearest.Entry = nE
9.      Nearest.Dist = Dist
10.   END IF
11.  END IF
12.  NEXT
13.  ELSE
    // Generate Node's branch list with entries overlapping the query period
14.  BranchList = GenBranchList(Q, N,  $Q_{per}$ )
    // Sort active branch List by MinDist
15.  SortBranchList(BranchList)
    // Iterate through active branch List
16.  FOR i = 1 TO BranchList.Count
17.    E = N.Entry(i)
    // Visit Child Nodes
18.    NN = E.ChildNode
19.    PointNNSearch(NN, Q,  $Q_{per}$ , Nearest)
    // Apply MinDist heuristic to do pruning
20.    PruneBranchList(BranchList)
21.  NEXT
22.  END IF

```

---

**Figure 5:** 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 4); 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 5) 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 20) applying the MINDIST heuristic [RKV95] [CF98].

## 4.2. 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 horizontal Euclidean Distance between each leaf entry and each query trajectory segment using the `Min_Horizontal_Dist` function (Line 10) computing the minimum horizontal Euclidean Distance between two 3D line segments. The formulization of the horizontal Euclidean Distance Function between two 3D line segments and the calculation of its minimum value needed for the algorithm can be found in Appendix A. In addition, for each segment of trajectory  $Q$  and before calculating its distance from the current entry we first check whether its temporal extent overlaps the temporal extent of the bounding rectangle of node  $N$ .

---

```

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

```

---

**Figure 6:** Generating Branch List of Node  $N$  against Trajectory  $Q$

At the non-leaf levels, the algorithm utilizes the `GenTrajectoryBranchList` function (pseudo-code in Figure 6) instead of `GenBranchList`. `GenTrajectoryBranchList(node N, Trajectory Q)` utilizes the `MinDist_Trajectory_Rectangle` metric introduced in Section 3.2 in order to calculate the MINDIST between the query trajectory and the rectangle of each entry of the node. Here, we have to point that we do not calculate the `MinDist_Trajectory_Rectangle` against the original query trajectory  $Q$ , but against the part of  $Q$  being inside the temporal extent of the bounding rectangle of  $N$ , and therefore (if necessary) we have to interpolate to produce the new query trajectory  $nQ$ .

---

```

Algorithm TrajectoryNNSearch(node N, trajectory Q, time period Qper, struct
Nearest)
1. Q = Interpolate(Q, Max(Q.TS, Qper.TS), Min(Q.TE, Qper.TE))
2. IF N Is Leaf
3.   FOR j = 1 to Q.Entries
4.     QE=Q.Entry(j)
5.     IF (QE.TS, QE.Te) Overlaps (N.TS, N.TE)
6.       FOR i = 1 to N.EntriesCount
7.         E = N.Entry(i)
8.         IF (QE.TS, QE.Te) Overlaps (E.TS, E.TE)
9.           nE = Interpolate(E, Max(QE.TS, E.TS), Min(QE.TE, E.TE))
10.          Dist = Min_Horizontal_Dist(QE, nE)
11.          IF Dist < Nearest.Dist
12.            Nearest.Entry = nE
13.            Nearest.Dist = Dist
14.          END IF
15.        END IF
16.      NEXT
17.    END IF
18.  NEXT
19. ELSE
20.   BranchList = GenTrajectoryBranchList(Q, N)
21.   SortBranchList(BranchList)
22.   FOR i = 1 TO BranchList.Count
23.     E = N.Entry(i)
24.     NN = E.ChildNode
25.     nQ = Interpolate(Q, Max(Q.TS NN.TS), Min(Q.TE NN.TE))
26.     TrajectoryNNSearch(NN, nQ, Nearest)
27.     PruneBranchList(BranchList)
28.   NEXT
29. END IF

```

---

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

### 4.3. Extending to $k$ -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. CNN Algorithms over trajectories

The continuous counterparts of the previously described algorithms are also of branch-and-bound type.

### 5.1. CNN 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. ContPointNNSearch algorithm used to process this type of query is illustrated in Figure 8.

---

```

Algorithm ContPointNNSearch(node N, 2D point Q, Period  $Q_{per}$ , List Nearests, Roof)
1. IF N Is Leaf
2.   FOR i = 1 to N.EntriesCount
3.     E = N.Entry(i)
4.     IF  $Q_{per}$  Overlaps (E. $T_S$ , E. $T_E$ )
5.       nE = Interpolate(E, Max( $Q_{per} \cdot T_S$ , E. $T_S$ ), Min( $Q_{per} \cdot T_E$ , E. $T_E$ ))
6.       MovingDist = ConstructMovingDistance(nE, Q)
7.       IF MovingDist. $D_{min}$  < Roof
8.         UpdateNearests(Nearests, MovingDist, Roof)
9.       END IF
10.    END IF
11.  NEXT
12. ELSE
13.   BranchList = GenBranchList(Q, N,  $Q_{per}$ )
14.   SortBranchList(BranchList)
15.   PruneContBranchList(BranchList, Nearests, Roof)
16.   FOR i = 1 TO BranchList.Count
17.     E = N.Entry(i)
18.     NN = E.ChildNode
19.     ContPointNNSearch(NN, Q,  $Q_{per}$ , Nearests, Roof)
20.     PruneContBranchList(BranchList, Nearests, Roof)
21.   NEXT
22. END IF

```

---

**Figure 8:** Historical CNN search algorithm for stationary query points (ContPointNNSearch algorithm)

All the continuous algorithms use a `MovingDist` structure (Figure 8, Line 6), 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 8, 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 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 A, we present in detail how we maintain the `Nearests` list.

When at non-leaf levels, the `ContPointNNSearch` algorithm in its backtracking applies the pruning algorithm `PruneContBranchList`, 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.

## 5.2. CNN algorithm for moving query objects (trajectories)

The fourth type of NN query is the continuous version of the NN query where the query object is the trajectory of a moving point. The algorithm `ContTrajectoryNNSearch`, used to process this type of query is illustrated in Figure 9.

---

```

Algorithm ContTrajectoryNNSearch (node N, Trajectory Q, time period Qper, List
Nearests, Roof)
1. Q = Interpolate(Q, Max(Q.TS, Qper.TS), Min(Q.TE, Qper.TE))
2. IF N Is Leaf
3.   FOR j = 1 to Q.Entries
4.     QE=Q.Entry(j)
5.     IF (QE.TS, QE.TE) Overlaps (N.TS, N.TE)
6.       FOR i = 1 to N.EntriesCount
7.         E = N.Entry(i)
8.         IF (QE.TS, QE.TE) Overlaps (E.TS, E.TE)
9.           nE = Interpolate(E, Max(QE.TS, E.TS), Min(QE.TE, E.TE))
10.          MovingDist = ConstructMovingDistance(nE, QE)
11.          IF MovingDist.Dmin < Roof
12.            UpdateNearests(Nearests, MovingDist, Roof)
13.          END IF
14.        END IF
15.      NEXT
16.    END IF
17.  NEXT
18. ELSE
19.   BranchList = GenTrajectoryBranchList(Q, N)
20.   SortBranchList(BranchList)
21.   PruneContBranchList(BranchList, Nearests, Roof)
22.   FOR i = 1 TO BranchList.Count
23.     E = N.Entry(i)
24.     NN = E.ChildNode
25.     nQ = Interpolate(Q, Max(Q.TS, NN.TS), Min(Q.TE, NN.TE))
26.     ContTrajectoryNNSearch(NN, nQ, Nearests, Roof)
27.     PruneContBranchList(BranchList, Nearests, Roof)
28.   NEXT
29. END IF

```

---

**Figure 9:** Historical CNN search algorithm for moving query points (ContTrajectoryNNSearch algorithm)

It differs from ContPointNNSearch algorithm at two points only: Firstly, at leaf level, the algorithm ConstructMovingDistance calculates the “Moving distance” between two moving points, instead of one moving and one stationary in the non-continuous case (Line 10). As in TrajectoryNNSearch, we perform a loop through all the 3D line segments of the query trajectory  $Q$  and, for each segment of  $Q$  and before processing the leaf entries, we first check whether the lifetime of  $Q$  overlaps the temporal extent of the bounding rectangle of  $N$  (Line 8). Secondly, at the non-leaf level, the GenBranchList is replaced by the GenTrajectoryBranchList introduced in the description of TrajectoryNNSearch algorithm (Line 19).

### 5.3. Extending to $k$ -CNN algorithms

The two continuous algorithms 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

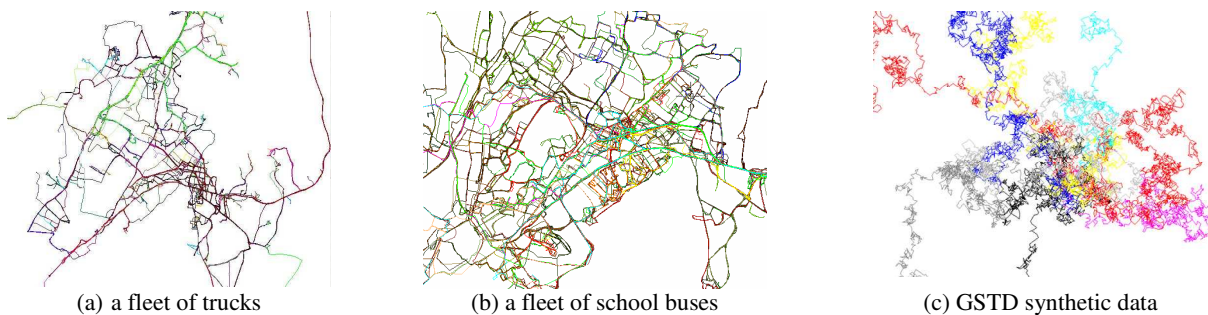
- Processing each entry against the  $i$ -th list (with  $i$  increasing, from 1 to  $k$ ) checking whether it qualifies to be in a list
- Testing each moving distance, replaced by a new entry in the  $i$ -th list, against the  $(i+1)$ -th list to find whether it qualifies to be in a list.

## 6. 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 TB-tree due to its proven efficiency regarding historical trajectory information, as demonstrated in [PJT00]. In our implementation, we set a page size of 4096 bytes and a (variable size) buffer fitting the 10% of the index size, thus leading to a maximum 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 size.

### 6.1. 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 provide a set of real-world datasets for experimentation purposes. In particular, in our experiments we have used two real datasets from a fleet of trucks and a fleet of school buses (illustrated in Figure (a) and (b), respectively). The two real datasets consist of 276 (112203) and 145 (66096) trajectories (entries in the index), respectively. 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 a scalability in the volumes of the datasets. A snapshot of the generated data using GSTD is illustrated in Figure (c). The synthetic trajectories generated by GSTD correspond to 20, 50, 100, 250, 500 and 1000 moving objects resulting in datasets of 30K, 75K, 150K, 375K, 750K and 1500K entries (the position of each object was sampled approximately 1500 times). Regarding the rest parameters of the GSTD generator, the initial distribution of points was Gaussian while their movement was ruled by a random distribution.



**Figure 10:** Snapshots of real and synthetic spatiotemporal data

Table 1 illustrates summary information about the number of pages occupied by the index for each dataset and will be used in the following sections so as to produce a percentage of the pruning done in the search space.

	<i># trajectories</i>	<i># entries</i>	<i>index size in pages (of 4096 bytes each)</i>
<i>Real Data (Trucks)</i>	276	112203	835
<i>Real Data (Buses)</i>	145	66096	466
<i>GSTD 20</i>	20	30277	205
<i>GSTD 50</i>	50	75717	507
<i>GSTD 100</i>	100	151482	1010
<i>GSTD 250</i>	250	378803	2521
<i>GSTD 500</i>	500	757360	5040
<i>GSTD 1000</i>	1000	1514844	10073

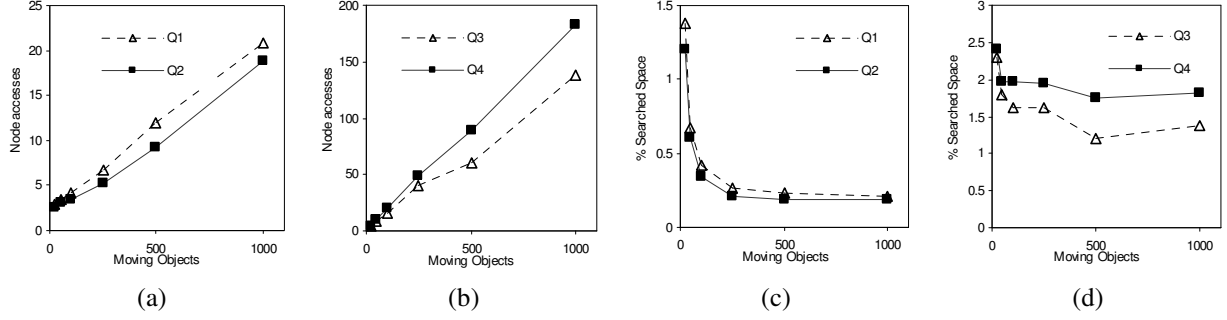
**Table 1:** Summary dataset information

## 6.2. Results on the Search Cost of the non-continuous algorithms

The performance of the proposed algorithms was measured in terms of node accesses. 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:

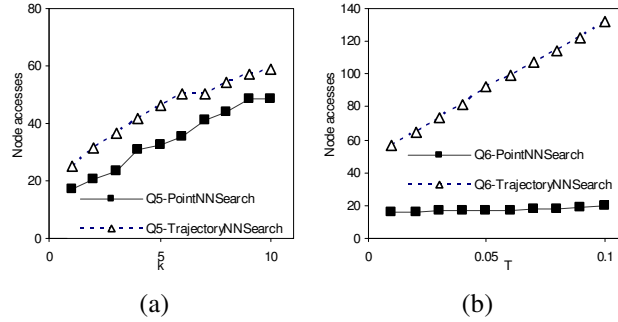
- Q1, Q2: the `PointNNSearch` algorithm was evaluated with two sets of 500 NN queries increasing the number of moving objects over the GSTD datasets. The queries used a random point in the 2D space and a time period of 1% (5%) of the temporal dimension for Q1 (Q2).
- Q3, Q4: the `TrajectoryNNSearch` algorithm was evaluated with two sets of 500 NN queries increasing the number of moving objects over the GSTD datasets. The 500 query objects (trajectories) were produced using GSTD also employing a Gaussian initial distribution and a random movement distribution. Then, in Q3 (Q4) we used a random 1% (5%) part of each trajectory as the query trajectory.
- Q5, Q6: 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 5% of time as query period, while for `TrajectoryNNSearch` algorithm we used a random part of a random trajectory belonging to Buses dataset, temporally covering 1% of time.

Figure 9 illustrates the average number of node accesses per query for the query sets Q1-Q4 evaluating `PointNNSearch` and `TrajectoryNNSearch` algorithms. In particular, Figure 9(a) shows the average number of node accesses per query using the point query sets Q1 and Q2, while Figure 9(b) shows the average number of node accesses per query using the trajectory query sets Q3 and Q4. As it is clearly illustrated, the performance of the algorithm depends linearly on the dataset cardinality and degrades (more pages are accessed) as the cardinality grows. It is worth to point out that comparing query sets Q1 and Q2, the algorithm accesses more pages in query set Q1, although the lifetime of Q2 is greater than that of Q1 (5% against 1% of the total time). This observation can be explained bearing in mind that decreasing the query temporal extent, the expected nearest distance increases, resulting in fewer pruned nodes in the backtracking procedure of the algorithm. As expected, `TrajectoryNNSearch` tends to be much more expensive than the `PointNNSearch`.



**Figure 9:** Node Accesses and % searched space in queries Q1-Q4 increasing the number of moving objects

The results in Figure 9(c) and (d) demonstrate the percentage of the indexed space actually used for searching. As illustrated, in all cases, increasing the index size, the percentage of the space to be searched decreases, resulting (for over 1000 moving objects) in a 0.20% of the whole index space for point NN queries and in a 1.2% - 2% for trajectory NN queries. To make the results more readable, we have to point out that a range search over the index with zero spatial and 1% temporal extent would lead to a searching among the 10% of the whole indexed space – showing that the pruning performed by our algorithms is much more efficient than a sequential search. The conclusion gathered from the previous observations is that the algorithms presented show high pruning ability, well bounding the space to be searched in order to answer NN queries.



**Figure 10:** Node Accesses in queries (a) Q5 increasing the number of  $k$  and (b) Q6 increasing the query temporal extent

The performance of the two non-continuous NN algorithms increasing the number of  $k$  is shown in Figure 10(a) against Buses dataset. Clearly, the number of node accesses needed for the processing of a  $k$ -NN query increases linearly with  $k$ . Figure 10(b) illustrates the average number of node accesses per non-continuous point and trajectory query increasing the temporal extent against the real “trucks” dataset. It is clear that the cost of TrajectoryNNSearch tends to increase with greater rate than the increase of PointNNSearch. This observation can be easily explained since when increasing the temporal interval, the spatial extent of the query trajectory also increases leading to a greater spatial space to be searched.

### 6.3. Results on the Search Cost of the continuous algorithms

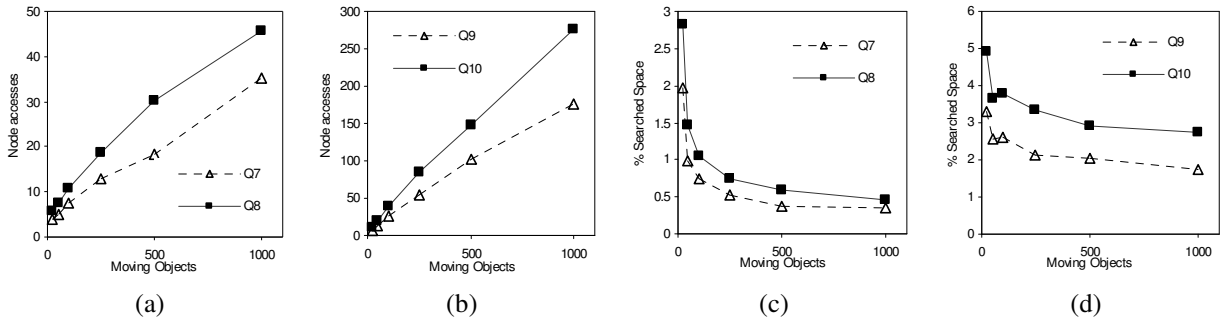
In coincidence with the experiments conducted for the non-continuous algorithms, the continuous NN search algorithms were evaluated with the following query sets:

- Q7, Q8: the ContPointNNSearch algorithm was evaluated with two sets of 500 NN queries increasing the number of moving objects over the GSTD datasets like what was done for query sets Q1 and Q2.



- Q9, Q10: the `ContTrajectoryNNSearch` algorithm was evaluated with two sets of 500 NN queries increasing the number of moving objects over the GSTD datasets like what was done for query sets Q3 and Q4.
- Q11, Q12: two sets of 500  $k$ -CNN queries over the real dataset of buses increasing the number of  $k$  with fixed time and increasing the size of the time interval (with fixed  $k=1$ ) respectively. For the `ContPointNNSearch` algorithm we used a random point in 2D space with a 5% of time as query period, while for `ContTrajectoryNNSearch` algorithm we used a random part of a random trajectory belonging to the buses dataset, temporally covering 1% of time.

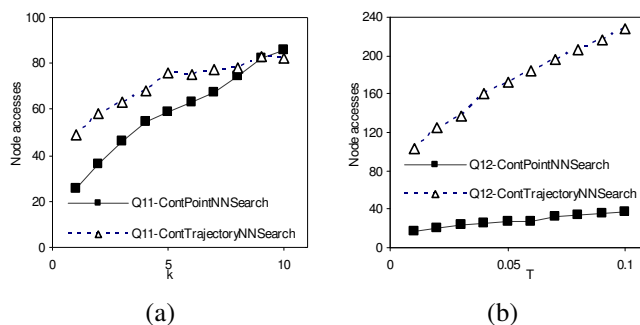
Figure 11 illustrates similar results as in Figure 9, regarding the continuous counterpart of the NN algorithms, thus, illustrating the average number of node accesses per query for the queries sets Q7- Q10. In particular, Figure 11(a) presents the average number of node accesses per query using the `ContPointNNSearch` against query sets Q7 and Q8 while Figure 11(b) presents the average number of node accesses per query using the `ContTrajectoryNNSearch` against query sets Q9 and Q10. Again, the performance of the algorithms linearly depends on the dataset cardinality and degrades (more pages are accessed) as the cardinality grows. Figure 11(c) and (d) show the accessed index part as a percentage of the indexed space, illustrating that in all cases, increasing the index size the percentage of the space to be searched decreases, resulting (for over 1000 moving objects) in a 0.50% of the whole index space for point CNN search and in a 2.5% - 3 % for trajectory CNN search.



**Figure 11:** Node Accesses and % pruned space in queries Q7-Q10 increasing the number of moving objects

A comparison between the non-continuous NN algorithms with their continuous counterpart (e.g. Figure 9 vs. Figure 11), shows that the continuous algorithms are much more expensive than the non-continuous ones. This conclusion was expected since the 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 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 performance of the continuous NN algorithms increasing the number of  $k$  is illustrated in Figure 12(a) for the real Buses dataset. The number of node accesses required for the processing of a  $k$ -NN query increases linearly with  $k$ . Figure 12(b) illustrates the average number of node accesses per continuous point and trajectory query increasing the temporal extent for Trucks dataset. Presenting the same behavior as with the non-continuous queries, the performance of `ContTrajectoryNNSearch` tends to degrade with greater rate than that of the `ContPointNNSearch`, having the same explanation (by increasing the temporal interval, the spatial extent of the query trajectory also increases leading to a greater spatial space to be searched).



**Figure 12:** Node Accesses in queries (a) Q11 increasing the number of  $k$  and (b) Q12 increasing the query temporal extent

## 7. 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 CNN 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 the TB-tree for our performance study due to its proven efficiency regarding historical trajectory information. 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 CNN queries. The pruning power of our algorithms is also verified in the case of the  $k$ -NN and  $k$ -CNN queries (for various values of  $k$ ).

As such, 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 Archimedes and Pythagoras EPEAEK II Programmes of the Greek Ministry of National Education and Religious Affairs. We are grateful to Emphasis Telematics S.A. for providing the real Buses and Trucks datasets.

## References

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

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

## APPENDIX A

### 1. Calculation of the minimum horizontal distance between two 3D line segments

The Euclidean horizontal distance function between two 3D line segments is:

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

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.

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

Setting

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

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

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

and replacing  $\Delta t$  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:

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

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 generally has the minimum value  $P_{\min} = C - \frac{B^2}{4A}$

for  $t = -\frac{B}{2A}$ . Therefore, in our case

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

for

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

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

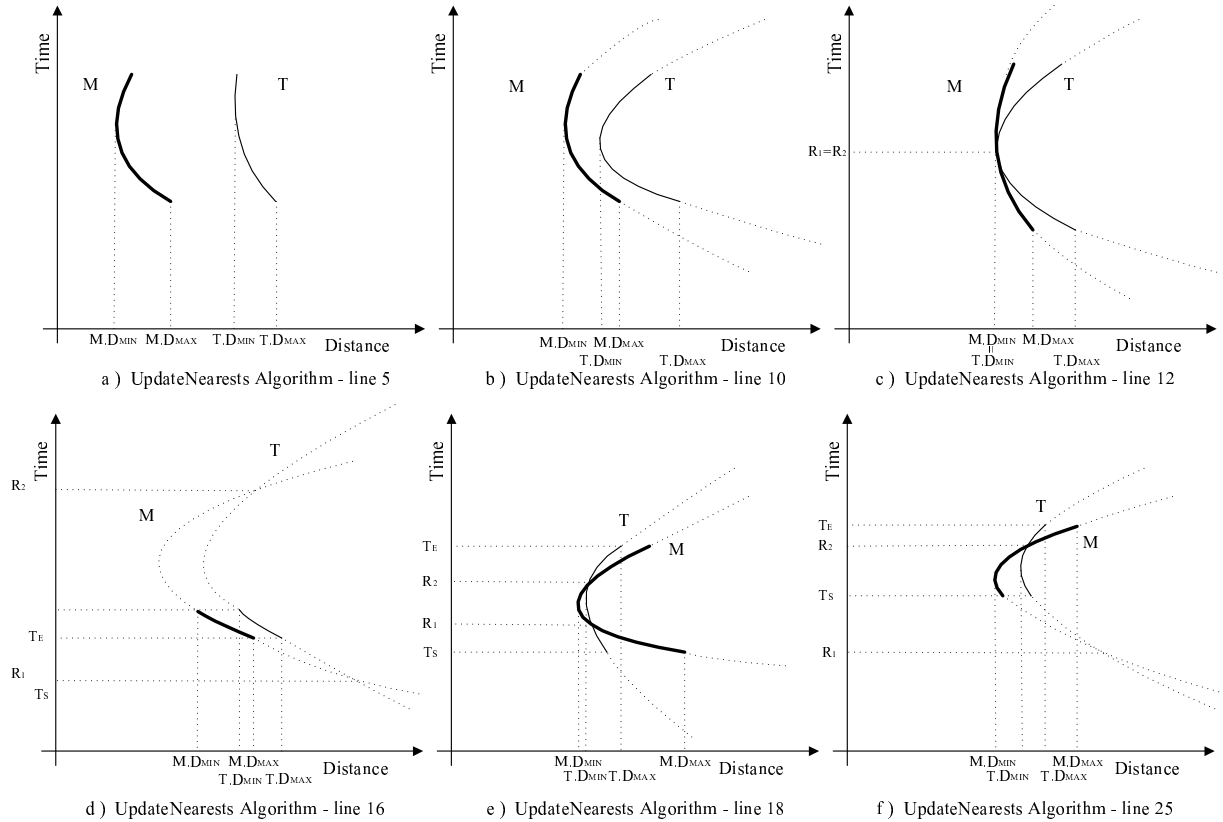
We have to note that formula (6) can be used in case where  $t$  calculated by formula (7) is inside the query time period  $Q_{per}[t_{start}, t_{end}]$ . Otherwise, we distinguish between the following two cases:

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  $t = t_{end}$ .

## 2. Update Nearests List

The pseudo-code of the `UpdateNearests` function, which is responsible for the maintenance of the *Nearests* List, is presented in Figure 14. 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. Figure 13 graphically explains all the possible comparisons between the parabolas of two “Moving Distance” functions.



**Figure 13:** Graphical illustration of UpdateNearests Algorithm Comparisons

Figure 13(a) corresponds to line 5 of the algorithm presented in Figure 14, 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 13(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 13(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 13(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 13(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 13(f)) and determine the part of  $T$  to be replaced by  $M$ .

---

```

Algorithm UpdateNearests (List Nearests, struct CM, Roof)
1.  FOR EACH T IN Nearests
2.    IF (T. $T_S$ , T. $T_E$ )Overlaps(CM. $T_S$ , CM. $T_E$ )
3.      M=Interpolate(CM, Max(CM. $T_S$ , T. $T_S$ ), Min(CM. $T_E$ , T. $T_E$ ))
4.      T=Interpolate(T, Max(CM. $T_S$ , T. $T_S$ ), Min(CM. $T_E$ , T. $T_E$ ))
5.      IF M. $D_{Max}$  < T. $D_{Min}$ 
6.        Nearests.Replace T with M
7.      ELSEIF M. $D_{Max}$  < T. $D_{Max}$ 
8.        D = Discriminant(M-T)
9.        IF D < 0
10.         IF T. $D_{Min}$  > M. $D_{Min}$  THEN Nearests.Replace T with M
11.         ELSEIF D=0
12.           IF T. $D_{Max}$  > M. $D_{Max}$  THEN Nearests.Replace T with M
13.         ELSE
14.           RR1=Solution1(T - M):RR2=Solution2(T - M):
              R1=Min(RR1,RR2):R2=Max(RR1,RR2)
15.           IF R2<T. $T_S$  OR R1>T. $T_E$ 
16.             IF T. $D_{Max}$  > M. $D_{Max}$  THEN Nearests.Replace T with M
17.             ELSEIF R2<T. $T_E$  AND R1>T. $T_S$ 
18.               IF M. $D_{min}$  < T. $D_{min}$ 
19.                 M1=Part(M, R1):M2=Part(M,R2):T1=Part(T,R1,R2):
20.                 Nearests.Replace T with (M1,T1,M2)
21.               ELSE
22.                 T1=Part(T, R1):T2=Part(T,R2):M1=Part(M,R1,R2):
23.                 Nearests.Replace T with (T1,T2,M1)
24.               ENDIF
25.             ELSE
26.               IF M(R1 - 1)<T(R1 - 1)
27.                 M1=Part(M, R1):T1=Part(T,R1): Nearests.Replace T with (M1,T1)
28.               ELSE
29.                 T1=Part(T, R1):M1=Part(M,R1): Nearests.Replace T with (T1,M1)
30.               ENDIF
31.             ENDIF
32.           ENDIF
33.         ENDIF
34.       ENDIF
35.       Roof=max(Roof, T. $D_{max}$ )
36.     NEXT

```

---

**Figure 14:** UpdateNearests Algorithm