GeoInformatica 7:2, 113–137, 2003

Fast Nearest-Neighbor Query Processing in Moving-Object Databases

K. RAPTOPOULOU, A.N. PAPADOPOULOS AND Y. MANOLOPOULOS
Department of Informatics, Aristotle University, Thessaloniki 54006, Greece
E-mail: {katerina, apostol, manolopo}@delab.csd.auth.gr

Abstract

A desirable feature in spatio-temporal databases is the ability to answer future queries, based on the current data characteristics (reference position and velocity vector). Given a moving query and a set of moving objects, a future query asks for the set of objects that satisfy the query in a given time interval. The difficulty in such a case is that both the query and the data objects change positions continuously, and therefore we can not rely on a given fixed reference position to determine the answer. Existing techniques are either based on sampling, or on repetitive application of time-parameterized queries in order to provide the answer. In this paper we develop an efficient method in order to process nearest-neighbor queries in moving-object databases. The basic advantage of the proposed approach is that only one query is issued per time interval. The time-parameterized R-tree structure is used to index the moving objects. An extensive performance evaluation, based on CPU and I/O time, shows that significant improvements are achieved compared to existing techniques.

Keywords: spatio-temporal databases, moving objects, nearest-neighbors, continuous queries

1. Introduction

Spatio-temporal database systems aim at combining the spatial and temporal characteristics of data. There are many applications that benefit from efficient processing of spatio-temporal queries such as: mobile communication systems, traffic control systems (e.g., air-traffic monitoring), geographical information systems, multimedia applications. The common basis of the above applications is the requirement to handle both the space and time characteristics of the underlying data [22], [30], [31]. These applications pose high requirements concerning the data and the operations that need to be supported, and therefore new techniques and tools are needed towards increased processing efficiency.

Many research efforts have focused on indexing schemes and efficient processing techniques for moving-object datasets [1], [5], [11], [21], [24], [29]. A moving dataset is composed of objects whose positions change with respect to time (e.g., moving vehicles). Examples of basic queries that could be posed to such a dataset include:

- *Window query*: given a rectangle R that changes position and size with respect to time, determine the objects that are covered by R from time point t_s to t_e.

- *Nearest-neighbor query*: given a moving point P determine the k nearest-neighbors of P within the time interval $[t_s, t_e]$.
- *Join query*: given two moving datasets S_1 and S_2, determine the pairs of objects (s_1, s_2) with $s_1 \in S_1$ and $s_2 \in S_2$ such that s_1 and s_2 overlap at some point in $[t_s, t_e]$.

Queries that require an answer for a specific time point (time-slice queries) are special cases of the above examples, and generally are more easily processed. Queries that must be evaluated for a time interval $[t_s, t_e]$ are characterized as continuous [23], [27]. In some cases, the query must be evaluated continuously as time advances. The basic characteristic of continuous queries is that there is a change in the answer at specific time points, which must be identified in order to produce correct results.

Among the plethora of spatio-temporal queries we focus on k nearest-neighbors queries (NN for short). Existing methods are either computationally intensive performing repetitive queries to the database, or are restrictive with respect to the application settings (i.e., are applied only for static datasets, or are applicable for special cases that limit the space dimensionality or the requested number of NNs). The objective of this work is twofold:

- to study efficient algorithms for NN query processing on moving object datasets,
- to compare the proposed algorithms with existing methods through an extensive experimental evaluation, by considering several parameters that affect query processing performance.

The rest of the article is organized as follows: In the next section we give the appropriate background and related work to keep the paper self-contained. In Section 3, the proposed approach is studied in detail and the application to TPR-trees is presented. In Section 4, a performance evaluation of all methods is conducted and the results are interpreted. Finally, Section 5 concludes and provides ideas for future work in the area.

2. Background

2.1. *Organizing moving objects*

The research conducted in access methods and query processing techniques for moving-object databases are generally categorized in the following areas:

- query processing techniques for past positions of objects, where past positions of moving objects are archived and queried, using multi-version access methods or specialized access methods for object trajectories [12], [14], [16], [17], [25], [26], [33],
- query processing techniques for present and future positions of objects, where each moving object is represented as a function of time, giving the ability to determine its future positions according to the current characteristics of the object movement (reference position, velocity vector) [1], [8]–[11], [13], [15], [18], [21], [32].

We focus on the second category, where it is assumed that the dataset consists of moving point objects, which are organized by means of a time-parameterized R-tree (TPR-tree) [21]. The TPR-tree is an extension of the well known R^*-tree [2], designed to handle object movement. Objects are organized in such a way that a set of moving objects is bounded by a moving rectangle, in order to maintain a hierarchical organization of the underlying dataset. The TPR-tree differs from the R-tree [4] and its variations in several aspects:

- bounding rectangles in the TPR-tree internal nodes although are conservative, they are not minimum in general,
- the TPR-tree is efficient for a time interval $[t_0, H)$, where H (horizon) is the time point which suggests a reorganization, due to extensive overlapping of bounding rectangles,
- all metrics used for insertion, reinsertion and node splitting in TPR-trees are based on integrals which calculate overlap, enlargement and margin for the time interval $[t_0, H)$,
- TPR-trees answer time-parameterized queries (range, NN, joins) for a given time interval $[t_s, t_e]$, or for a specific time point.

2.2. *Nearest-neighbor queries*

Allowing the query and the objects to move, an NN query takes the following forms:

- Given a query point reference position q_{pos}, a query velocity vector $\mathbf{q}_v$, a time point t_x and an integer k, determine the k NNs of $\mathbf{q}$ at t_x (time-slice NN query).
- Given a query point reference position $\mathbf{q}$, a query velocity vector $\mathbf{q}_v$, an integer k and a time interval $[t_1, t_2)$, determine the k NNs of $\mathbf{q}$ according to the movement of the query and the movement of the objects from t_1 to t_2 (continuous or time-interval NN query).

The second query type is more difficult to answer, since it requires knowledge of specific time points which indicate that there is a change in the answer set (split points).

Figure 1 shows an example database of four moving objects. Assume that the $k = 2$

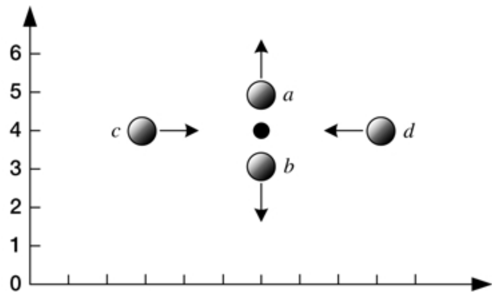


Figure 1. A query example.

NNs are requested for the time interval [0,5]. Assume also that the query point is static (black circle). By observing the movement of the objects with respect to the query, it is evident that for the time interval [0,2) the NNs of q are b and a, whereas for the time interval [2,5) the NNs are c and d. In the sequel we briefly describe research results towards solving NN queries in moving datasets.

Kollios et al. [10] propose a method able to answer NN queries for moving objects in 1D space. The method is based on the dual transformation where a line segment in the native space corresponds to a point in the transformed space, and vice-versa. The method determines the object that comes closer to the query between $[t_s, t_e]$ and not the NNs for every time instance.

Zheng and Lee [34] proposed a method for computing a single NN $(k = 1)$ of a moving query, applied to static points indexed by an R-tree. The method is based on Voronoi diagrams and it seems quite difficult to be extended for other values of k and higher space dimensions.

In Song and Roussopoulos [23] a method is presented to answer such queries on moving-query, static-objects cases. Objects are indexed by an R-tree, and sampling is used to query the R-tree at specific points. However, due to the nature of sampling, the method may return incorrect results if a split point is missed. A low sampling rate yields more efficient performance, but increases the probability of incorrect results, whereas a high sampling rate poses unnecessary computational overhead, but decreases the probability of incorrect results.

Benetis et al. [3] propose an algorithm capable of answering NN queries and reverse NN queries in moving-object datasets. The proposed method is restricted in answering only one NN per query.

In Tao et al. [27] the authors propose an NN query processing algorithm for moving-query moving-objects, based on the concept of time-parameterized queries. Each query result is composed of the following components: (i) R, is the current result set of the query, (ii) T, is the time point in which the result becomes invalid, and (iii) C, the set of objects that influence the result at time T. Therefore, by continuously calculating the next set of objects that will influence the result, we determine the NNs of the query from t_1 to t_2. A TPR-tree index is used to organize the moving objects.

The main drawback of the aforementioned method is that the TPR-tree is searched several times in order to determine the next object that influences the current result. This implies additional overhead in CPU and I/O time, which is more severe as the number of requested NNs increases. In Tao and Papadias [28] the same authors present a method which is applicable for static datasets, in order to overcome the problems of repetitive NN queries. By assuming that the dataset is indexed by an R-tree structure, a single query is performed and therefore each participating tree node is accessed only once. Performance results demonstrate that NN queries are answered much more efficiently concerning query response time. However, the proposed techniques can only be applied for static datasets.

Table 1 presents a categorization of NN queries with respect to the characteristics of queries and datasets. There are four different versions of the problem which are formulated by considering queries and datasets as static or moving. The table also summarizes the previously mentioned related work for each problem.

Table 1. NN queries for different query and data characteristics.

Query	*Data*	*Related Work*
Static	Static	Conventional techniques
Static	Moving	Handled by MQMD
Moving	Static	Roussopoulos et al. [19], Song and Roussopoulos [23] Zheng and Lee [34] Tao and Papadias [28]
Moving	Moving	Tao et al. [27] Kollios et al. [10] Benetis et al. [3]

2.3. *Motivation*

To the best of the authors knowledge, there is no method based on the TPR-tree to answer NN queries for moving-query moving-objects other than the repetitive approach proposed in Tao et al. [27]. Therefore, motivated by the extensive overhead of the existing method and taking into account that the continuous algorithm reported in Tao and Papadias [28] can not handle moving-object datasets, we provide efficient methods for NN query processing for moving-query moving-object databases, with the following characteristics:

- the method is applied for any number of requested NNs,
- the method can be applied for any number of space dimensions, since only relative distances are computed during query processing,
- different tree pruning algorithms may be applied during tree traversal,
- each tree node is accessed only once, therefore reducing the consumption of system resources,
- the method not only reports the time points when there is a change in the result, but also the time points when there is a change in the order of the NNs in the current result.

3. NN query processing

The challenge is to determine the k NNs of q, given a moving query q, a query velocity vector **vq** and a time interval $[t_s, t_e]$. We want to answer such a query, by performing only one search, thus avoiding posing repetitive queries to the database. The answer to the query is a set of mutually exclusive time intervals, and a sorted list of object IDs for each time interval, which are the k NNs of q for the respective interval.

By assuming that the distance between two points is given by the Euclidean distance, the distance $D_{q,o}(t)$ between query q and object o as a function of time is given by the following equation:

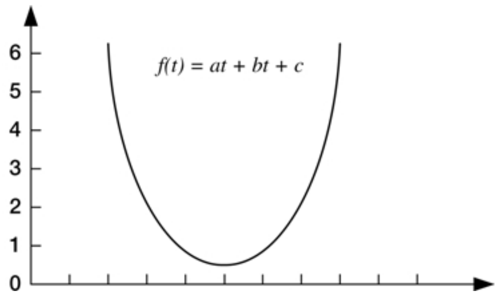


Figure 2. Visualization of the distance between a moving object and a moving query.

$$D_{q,o}(t) = \sqrt{c_1 \cdot t^2 + c_2 \cdot t + c_3}, \tag{1}$$

where c_1, c_2, c_3 are constants given by:

$$c_1 = (vo_x - vq_x)^2 + (vo_y - vq_y)^2$$
$$c_2 = 2 \cdot [(o_x - q_x) \cdot (vo_x - vq_x) + (o_y - q_y) \cdot (vo_y - vq_y)]$$
$$c_3 = (o_x - q_x)^2 + (o_y - q_y)^2$$

vo_x, vo_y are the velocities of object o, vq_x, vq_y are the velocities of the query in each dimension, and (o_x, o_y), (q_x, q_y) are the reference positions of the object o and the query q respectively. In the sequel, we assume that the distance is given by $(D_{q,o}(t))^2$ in order to perform simpler calculations.

The movement of an object with respect to the query is visualized by plotting the function $(D_{q,o}(t))^2$, as it is illustrated in figure 2. For NN query processing the distance from the query point contains all the necessary information, since the exact position of the object is irrelevant. Note that since $c_1 \geq 0$ the plot of the function always has the shape of a ‘‘valley’’.

Assume that we have a set of moving objects $\mathcal{O}$ and a moving query q. The objects and the query are represented as points in a multi-dimensional space. Although the proposed method can be applied to any number of dimensions, the presentation is restricted to 2-D space for clarity and convenience. Moving queries and objects are characterized by their reference positions and velocity vectors. Therefore, we have all the necessary information to define the distance $(D_{q,o}(t))^2$ for every object $o \in \mathcal{O}$. By visualizing the relative movement of the objects during $[t_s, t_e]$ a graphical representation is derived, such as the one depicted in figure 3.

By inspecting figure 3 we obtain the k NNs of the moving query during the time interval $[t_s, t_e]$. For example, for $k = 2$ the NNs of q for the time interval are contained in the shaded area of figure 3. The NNs of q for various values of k along with the corresponding time

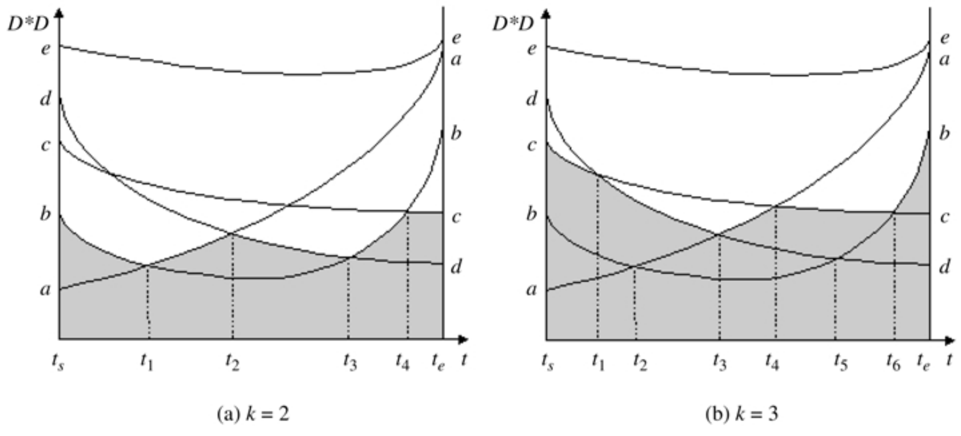


Figure 3. Relative distance of objects with respect to a moving query.

intervals are depicted in figure 4. The pair of objects above each time point t_x declare the objects that have an intersection at t_x. These time points where a modification of the result is performed, are called split points. Note that not all intersection points are split points. For example, the intersection of objects a and c in figure 3 is not considered as a split point for $k = 2$, whereas it is a split point for $k = 3$.

The previous example demonstrates that the k NNs of a moving query can be determined by using the functions that represent the distance of each moving object with respect to the moving query. Based on the previous discussion, the next section presents the design of an algorithm for NN query processing (*NNS*) which operates on moving objects.

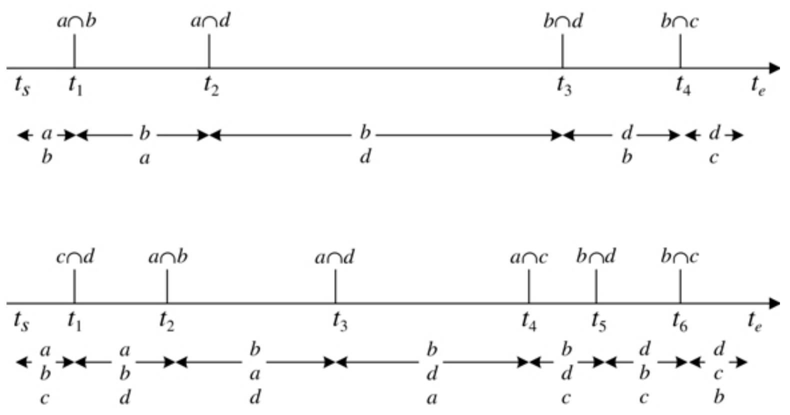


Figure 4. NNs of the moving query for $k = 2$ (top) and $k = 3$ (bottom).

3.1. The NNS algorithm

The *NNS* algorithm consists of two parts, which are described separately:

- *NNS-a* algorithm: given a set of moving objects, a moving query and a time interval, the algorithm returns the k NNs for the given interval, and
- *NNS-b* algorithm: given the k NNs, the corresponding time intervals, and a new moving object, the algorithm computes the new result.

3.1.1. Algorithm NNS-a.

We are given a moving query q, a set $\mathcal{O}$ of N moving objects, a time interval $[t_s, t_e]$ and the k NNs of q are requested. The target is to partition the time interval into one or more sub-intervals, in which the list of NNs remains unchanged. Each time sub-interval is defined by two time split points, declaring the beginning and the end of the sub-interval. During the calculation, the set $\mathcal{O}$ is partitioned into three sub-sets:

- the set $\mathcal{K}$, which always contains k objects that are currently the NNs of q,
- the set $\mathcal{C}$, which contain objects that are possible candidates for subsequent time points, and
- the set $\mathcal{R}$, which contains rejected objects whose contribution to the answer is impossible for the given time interval $[t_s, t_e]$.

Initially, $\mathcal{K} = \emptyset$, $\mathcal{C} = \mathcal{O}$, and $\mathcal{R} = \emptyset$. The first step is to determine the k NNs for time point t_s. By inspecting figure 3 for $k = 2$ we get that these objects are a and b. Therefore, $\mathcal{K} = \{a, b\}, \mathcal{C} = \{c, d, e\}$ and $\mathcal{R} = \emptyset$. Next, for each $o \in \mathcal{K}$ the intersections with objects in $\mathcal{K} + \mathcal{C}$ are determined. If there are any objects in $\mathcal{C}$ that do not intersect any objects in $\mathcal{K}$, they are removed from $\mathcal{C}$ and are put in $\mathcal{R}$, meaning that they will not be considered again (Proposition 1). In our example, object e is removed from $\mathcal{C}$ and we have $\mathcal{K} = \{a, b\}, \mathcal{C} = \{c, d\}$ and $\mathcal{R} = \{e\}$. The currently determined intersections are kept in an ordered list, in increasing time order. Each intersection is represented as $(t_x, \{u, v\})$, where t_x is the time point of the intersection and $\{u, v\}$ is the objects that intersect at t_x.

Proposition 1: Moving objects that do not intersect the k nearest neighbors of the query at time t_s, can be rejected.

Proof: An intersection between o_1 and o_2 denotes a change in the result. Therefore, if none of the k nearest-neighbor objects intersect any other object between $[t_s, t_e]$, there will be no change in the result. This means that we do not have to consider other objects for determining the nearest-neighbors. $\square$

Each intersection is defined by two objects[1] u and v. The currently determined intersection points comprise the current list of time split points. According to the example, the split point list has as follows: $(t_1, \{a, b\})$, $(t_2, \{a, d\})$, $(t_x, \{a, c\})$, $(t_3, \{b, d\})$, $(t_4, \{b, c\})$. For each intersection we distinguish between two cases:

- $u \in \mathscr{K}$ and $v \in \mathscr{K}$
- $u \in \mathscr{K}$ and $v \in \mathscr{C}$ (or $u \in \mathscr{C}$ and $v \in \mathscr{K}$)

In the first case, the current set of NNs does not change. However, the order of the currently determined objects changes, since two objects in $\mathscr{K}$ intersect, and therefore they exchange their position in the ordered list of NNs. Therefore, objects u and v exchange their position. In the second case, object v is inserted into $\mathscr{K}$ and therefore the list of NNs must be updated accordingly (Proposition 2).

Proposition 2: Let us consider a split point at time t_x, at which objects o_1 and o_2 intersect. If $o_1 \in \mathscr{K}$ and $o_2 \in \mathscr{C}$ then at t_x, o_1 is the k-th nearest-neighbor of the query.

Proof: Assume that o_1 is not the k-th nearest-neighbor at the time of the interscection. However, o_1 belongs to the result (is among the k nearest-neighbors) at time t_x. The intersection at time t_x denotes that objects o_1 and o_2 are consequtive in the result. This implies that o_2 is already contained in the current result (set $\mathscr{K}$) which contradicts our assumption that o_2 is not contained in the result set. Therefore, object o_1 must be the k-th nearest-neighbor of the query. $\square$

According to the currently determined split points, the first split point is t_1, where objects a and b intersect. Since both objects are contained in $\mathscr{K}$, no new objects are inserted into $\mathscr{K}$, and simply objects a and b exchange their position. Up to this point concerning the sub-interval $[t_s, t_1)$ the nearest neighbors of q are a and b. We are ready now to check the next split point, which is t_2 where objects a and d intersect. Since $a \in \mathscr{K}$ and $d \in \mathscr{C}$ object a is removed from $\mathscr{K}$ and it is inserted into $\mathscr{C}$. On the other hand, object d is removed from $\mathscr{C}$ and it is inserted into $\mathscr{K}$ taking the position of a. Up to this point, another part of the answer has been determined, since in the sub-interval $[t_1, t_2)$ the NNs of q are b and a. Moving to the next intersection, t_x, we see that this intersection is caused by objects a and c. However, neither of these objects is contained in $\mathscr{K}$. Therefore, we ignore t_x and remove it from the list of time split points. Since a new object d has been inserted into $\mathscr{K}$, we check for new intersections between d and objects in $\mathscr{K}$ and $\mathscr{C}$. No new intersections are discovered, and therefore we move to the next split point t_3. Currently, for the time sub-interval $[t_2, t_3)$ the NNs of q are b and d. At t_3 objects b and d intersect, and this causes a position exchange. We move to the next split point t_4 where objects b and c intersect. Therefore, object b is removed from $\mathscr{K}$ and it is inserted into $\mathscr{C}$, whereas object c is removed from $\mathscr{C}$ and it is inserted into $\mathscr{K}$. Since c does not have any other intersections with objects in $\mathscr{K}$ and $\mathscr{C}$, the algorithm terminates. The final result is depicted in figure 4, along with the corresponding result for $k = 3$. The outline of the method is illustrated in figure 5.

Each object $o \in \mathscr{K}$ is responsible for a number of potential time split points, which are defined by the intersections of o and the objects contained in $\mathscr{C}$. Therefore, each time an object is inserted into $\mathscr{K}$ intersection checks must be performed with the objects in $\mathscr{C}$. In order to reduce the number of intersection tests, if an object was previously inserted into $\mathscr{K}$ and now it is reinserted, it is not necessary to recompute the intersections. Moreover,

```
**Algorithm** *NNS-a*
**Input:** a set of moving objects $\mathscr{O}$, a moving query $q$,
time interval $[t_s, t_e]$, the number $k$ of requested NNs
**Output:** a list of elements of the form $([t_1, t_2], o_1, o_2, ..., o_k)$
where $o_1, ..., o_k$ are the NNs of $q$ from $t_1$ to $t_2$ (CNN-list),
split-list containing the split points
**Local:** $k$-list containing the current NNs
 1.  initialize $\mathscr{K} = \emptyset$, $\mathscr{C} = \mathscr{O}$, and $\mathscr{R} = \emptyset$
 2.  initialize split-list with split points $t_s$ and $t_e$
 3.  find the $k$ NNs of $q$ at time point $t_s$
 4.  update $k$-list
 5.  foreach $u \in \mathscr{K}$ do
 6.     find intersections with $v \in \mathscr{K}$
 7.     find intersections with $v \in \mathscr{C}$
 8.     update split list
 9.     move irrelevant objects from $\mathscr{C}$ to $\mathscr{R}$
10.  endfor
11.  while more split-points are available do
12.     check next time split point $t_x$ (intersection)
13.     if ($u \in \mathscr{K}$) and ($v \in \mathscr{K}$) then
14.        update CNN-list
15.        exchange positions in $k$-list
16.     endif
17.     if ($u \in \mathscr{K}$) and ($v \in \mathscr{C}$) then
18.        move $u$ from $\mathscr{K}$ to $\mathscr{C}$
19.        move $v$ from $\mathscr{C}$ to $\mathscr{K}$
20.        update $k$-list
21.        update CNN-list
22.        if ($v$ participates for the first time in $k$-list) then
23.           determine intersections of $v$ with objects in $\mathscr{C}$
24.           update split-list
25.        endif
26.     endif
27.     if ($u \in \mathscr{C}$) and ($v \in \mathscr{C}$) then
28.        ignore split point $t_x$
29.     endif
30.  endwhile
31.  return CNN-list, split-list
```

Figure 5. The *NNS-a* algorithm.

according to Proposition 3, intersections at time points prior to the currently examined split point can be safely ignored.

Proposition 3: If there is a split point at time t_x, where $o_1 \in \mathscr{K}$ and $o_2 \in \mathscr{C}$ intersect, all intersections of o_2 with the other objects in $\mathscr{K}$ that occur at a time before t_x are not considered as split points.

Proof: This is evident, since the nearest-neighbors of the query object up to time t_x have been already determined and therefore the intersections at time points prior to t_x do not denote a change in the result. □

Evidently, in order to determine if two objects u and v intersect at some time point between t_s and t_e, we have to solve an equation. Let the square of the Euclidean distance between q and the objects be described by the functions $D_{u,q}(t)^2 = u_1 \cdot t^2 + u_2 \cdot t + u_3$ and $D_{v,q}(t)^2 = v_1 \cdot t^2 + v_2 \cdot t + v_3$ respectively. In order for the two object to have an intersection in $[t_s, t_e]$ there must be at least one value t_x, $t_s \le t_x \le t_e$ such that:

$$(u_1 - v_1) \cdot t_x^2 + (u_2 - v_2) \cdot t_x + (u_3 - v_3) = 0.$$

From elementary calculus it is known that this equation can be satisfied by none, one, or two values of t_x. If $(u_2 - v_2)^2 - 4 \cdot (u_1 - v_1) \cdot (u_3 - v_3) < 0$, then there is no intersection between u and v. If $(u_2 - v_2)^2 - 4 \cdot (u_1 - v_1) \cdot (u_3 - v_3) = 0$ then the two objects intersect at $t_x = -(u_2 - v_2)/2 \cdot (u_1 - v_1)$. Otherwise the objects intersect at two points t_x and t_y given by:

$$t_x = \frac{-(u_2 - v_2) + \sqrt{(u_2 - v_2)^2 - 4 \cdot (u_1 - v_1) \cdot (u_3 - v_3)}}{2 \cdot (u_1 - v_1)}$$
$$t_y = \frac{-(u_2 - v_2) - \sqrt{(u_2 - v_2)^2 - 4 \cdot (u_1 - v_1) \cdot (u_3 - v_3)}}{2 \cdot (u_1 - v_1)}.$$

3.1.2. Algorithm NNS-b. After the execution of *NNS-a*, the *CNN-list* is formulated, which contains elements of the form: $([t_1, t_2], o_1, o_2, \ldots, o_k)$ where $o_1, \ldots, o_k$ are the NNs of q from t_1 to t_2, in increasing distance order. Let $\mathscr{S}$ be the set containing the NNs of q at any given time between t_s and t_e. Clearly, $k \le |\mathscr{S}| \le |\mathscr{O}|$. Assume now that we have to consider another object w, which was not known during the execution of *NNS-a*. We distinguish among the following cases, which describe the relation of w to the current answer:

Case 1: w does not intersect any of the objects in $\mathscr{S}$ between t_s and t_e, and it is ''above'' the area of relevance. In this case, w is ignored, since it can not contribute to the NNs. The number of split points remains the same.

Case 2: w does not intersect any of the objects in $\mathscr{S}$ between t_s and t_e, and it is completely ''inside'' the area of relevance. In this case w must be taken into account, since it affects the answer from t_s to t_e (Proposition 4). The number of split points may be reduced.

Case 3: w intersects at least one object $v \in \mathscr{S}$ at time $t_s \le t_x \le t_e$, but at time t_x v is not contained in the set of NNs. In this case, again w is ignored, since this intersection can not be considered as a split point because the answer is not affected. Therefore, no new split points are generated.

Case 4: w intersects at least one object $v \in \mathscr{S}$ at time $t_s \le t_x \le t_e$, and object v is contained in the set of NNs at time t_x. In this case w must be considered because at least one new split point is generated. We note, however, that some of the old split points may be discarded.

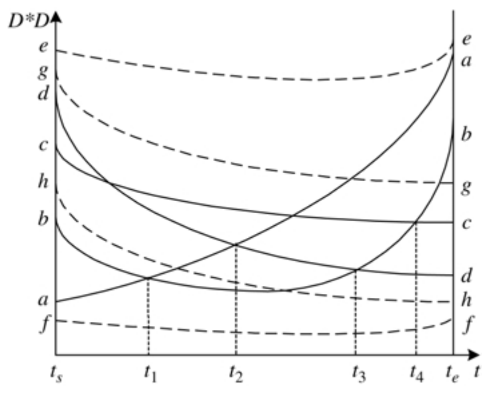


Figure 6. The four different cases that show the relation of a new object to the current NNs.

Proposition 4: Assume that a new object w does not intersect any of the NNs from t_s to t_e. If at time t_s its position among the k NNs is pos_w, then it maintains this position throughout the query duration.

Proof: Assume that there is a change in the result at some point t_x, where object w changes its position among the nearest-neighbors. This implies that there is an intersection at time t_x, since only an intersection denotes a result change. This contradicts our assumption that there are no intersections of w with other objects in the result. □

The aforementioned cases are depicted in figure 6. Object e corresponds to *Case 1*, since it is above the area of interest. Object f corresponds to *Case 2*, because it is completely covered by the relevant area. Object g although intersects some objects, the time of these intersections are irrelevant to the answer, and therefore the situation corresponds to *Case 3*. Finally, object h intersects a number of objects at time points that are critical to the answer and therefore corresponds to *Case 4*.

The outline of the *NNS-b* algorithm is presented in figure 7. Note that in lines 14 and 20 a call to the procedure *modify-CNN-list* is performed. This procedure, takes into consideration the *CNN-list* and the new *split-list* that is generated. It scans the *split-list* in increasing time order and performs the necessary modifications to the *CNN-list* and the *split-list*. Some of the split-points may be discarded during the process. The steps of the procedure are illustrated in figure 8.

3.2. Query processing with TPR-trees

Having described in detail the query processing algorithms in the previous section we are ready now to elaborate in the way these methods are combined with the TPR-tree. Let T be a TPR-tree which is built to index the underlying data. Starting from the root node of T the

```
Algorithm NNS-b
Input: a list of elements of the form $([t_1, t_2], o_1, o_2, \ldots, o_k)$
where $o_1, ..., o_k$ are the NNs of $q$ from $t_1$ to $t_2$ (CNN list}),
a new object $w$, the split-list
Output: an updated list of the form $([t_1, t_2], o_1, o_2, \ldots, o_k)$
where $o_1, ..., o_k$ are the NNs of $q$ from $t_1$ to $t_2$ (CNN list)
Local: $k$-list current list of NNs,
split-list, the current list of split points
 1.  initialize $\mathscr{S}$ = union of NNs from $t_s$ to $t_e$
 2.  intersectionFlag = FALSE
 3.  foreach $s \in \mathscr{S}$ do
 4.     check intersection between $s$ and $w$
 5.     if ($s$ and $w$ intersect) then // handle cases 3 and 4
 6.        intersectionFlag = TRUE
 7.        collect all $t_j, s$ // $t_j$ is where $w$ and $s$ intersect
 8.        if (at $t_j$ object $s$ contributes to the NNs) then
 9.           update split-list
10           endif
11.       endif
12.  endfor
13.  if (intersectionFlag == TRUE) then
14.     call modify-CNN-list
15.  else // handle cases 1 and 2
16.     calculate $D_{q,w}(t)^2$ at time point $t_s$
17.     if $(D_{q,w}(t_s))^2 \geq D^2_{kNN})$ then
18.        ignore $w$
19.     else
20.        call modify-CNN-list
21.     endif
22.  endif
23.  return CNN-list, split-list
```

Figure 7. The *NNS-b* algorithm.

tree is searched in a depth-first-search manner (DFS).[2] The first phase of the algorithm is completed when $m \geq k$ objects have been collected from the dataset. Tree branches are selected for descendant according to the *mindist* metric [19] (Definition 1) between the moving query and bounding rectangles at time t_s. These m moving objects are used as input to the *NNS-a* algorithm in order to determine the result from t_s to t_e. Therefore, up to now we have a first version of the *split-list* and the *CNN-list*. However, other relevant objects may reside in leaf nodes of T that are not yet examined.

Definition 1: Given a point p at $(p_1, p_2, \ldots, p_n)$ and a rectangle r whose lower-left and upper-right corners are $(s_1, s_2, \ldots, s_n)$ and $(t_1, t_2, \ldots, t_n)$, the distance *mindist*(p, r) is defined as follows:

$$mindist(p, r) = \sqrt{\sum_{j=1}^{n} |p_j - r_j|^2},$$

```
Procedure modify-CNN-list
Input: a list of elements $([t_1, t_2], o_1, o_2, ..., o_k)$
where $o_1, ..., o_k$ are the NNs of $q$ from $t_1$ to $t_2$ (CNN list),
a new object $w$, the split-list
Output: an updated list of elements $([t_1, t_2], o_1, o_2, ..., o_k)$
where $o_1, ..., o_k$ are the NNs of $q$ from $t_1$ to $t_2$ (CNN list)
Local: k-list current list of NNs
 1.  calculate $D_{q,w}(t)^2$ at time point $t_s$
 2.  consult CNN-list and update the current k-list
 3.  while more split-points are available do
 4.     check next split-point $(t_x, \{u, v\})$
 5.     update CNN-list
 6.     if $(u \notin k\text{-}list)$ and $(v \notin k\text{-}list)$ then
 7.        remove split-point $(t_x, \{u, v\})$
 8.     elseif $(u \notin k\text{-}list)$ and $(v \notin k\text{-}list)$ then
 9.        remove $u$ from k-list
10         insert $v$ in k-list
11.        update k-list
12.     elseif $(v \notin k\text{-}list)$ and $(u \notin k\text{-}list)$ then
13.        remove $v$ from k-list
14.        insert $u$ in k-list
15.        update k-list
16.     else
17.        exchange positions between $u$ and $v$
18.        update k-list
19.     endif
20.  endwhile
```

Figure 8. The *modify-CNN-list* procedure.

where

$$r_j = \begin{cases} s_j, & p_j < s_j \\ t_j, & p_j > t_j \\ p_j, & \text{otherwise.} \end{cases}$$

In the second phase of the algorithm, the DFS continues to search the tree, by selecting possibly relevant tree branches and discarding non-relevant ones. Every time a possibly relevant moving object is reached, algorithm *NNS-b* is called in order to update the *split-list* and the *CNN-list* of the result. The algorithm terminates when there are no relevant branches to examine.

In order to complete the description of the algorithm, the terms possibly relevant tree branches and possibly relevant moving objects must be clarified. In other words, the pruning strategy must be described in detail. Figure 9 illustrates two possible pruning techniques that can be used to determine relevant and non-relevant tree branches and moving objects:

Pruning technique 1 (PT1): In this technique we keep track of the maximum distance $D_{\max}$ between the query and the current set of NNs. In figure 9(a) this distance is defined

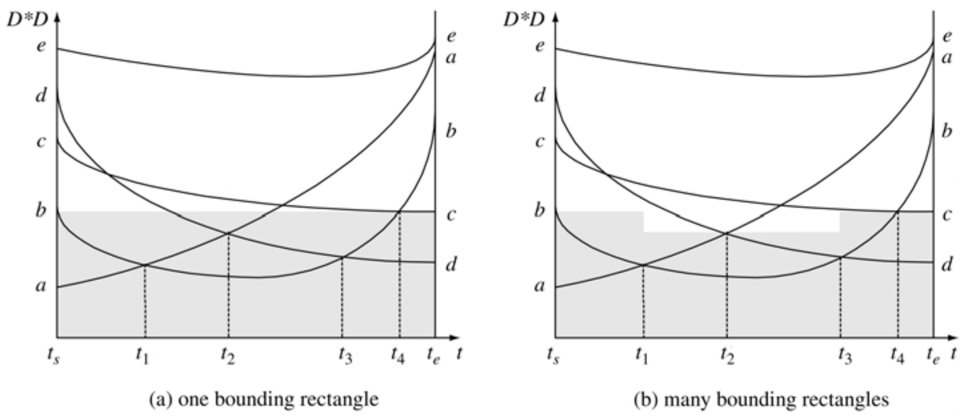


Figure 9. Pruning techniques.

between the query and object b at time t_{start}. We formulate a moving bounding rectangle R centered at q with extends $D_{\max}$ in each dimension and moving with the same velocity vector as $\mathbf{q}$. If R intersects a bounding rectangle E in an internal node, the corresponding tree branch may contain objects that contribute to the answer and therefore must be examined further. Otherwise, it can be safely rejected since it is impossible to contain relevant objects. In the same manner, if a moving object o_x found in a leaf node intersects R it may contribute to the answer, otherwise it is rejected.

Pruning technique 2 (PT2): This technique differs from the previous one in the level of granularity that moving bounding rectangles are formulated. Instead of using only one bounding rectangle, a set of bounding rectangles is defined according to the currently determined split points. Note that it is not necessary to consider all split points, but only these that are defined by the k-th nearest-neighbor in each time interval. An example set of moving bounding rectangles is illustrated in figure 9(b). Each internal bounding rectangle and moving object is checked for intersection with the whole set of moving bounding rectangles and it is considered relevant only if it intersects at least one of them.

Other pruning techniques can also be determined by grouping split points in order to keep the balance between the number of generated bounding rectangles and the existing empty space. Several pruning techniques can be combined in a single search by selecting the preferred technique according to some criteria (e.g., current number of split-points, existing empty space).

It is anticipated that PT1 will be more efficient with respect to CPU time, but less efficient concerning I/O time, because the empty space will cause unnecessary disk accesses. On the other hand, PT2 seems to incur more CPU overhead due to the increased number of intersection computations, but also less I/O time owing to the detailed pruning performed. Based on the above discussion, we define the *NNS-CON* algorithm which operates on TPR-trees and can be used with either of the two pruning techniques. The outline of the algorithm is illustrated in figure 10.

```
Algorithm NNS-CON
Input: the TPR-tree root,
    a moving query $q$,
    the number $k$ of NNs
Output: the $k$ NNs in $[t_s, t_e]$
Local: a set $\mathcal{C}$ of collected objects,
    Flag is FALSE if NNS-a has not yet been called
number col of collected objects
 1. if (node is LEAF) then
 2.    if ($|\mathcal{C}| < k$) then
 3.       add each entry of node to $\mathcal{C}$
 4.       update $|\mathcal{C}|$
 5.    endif
 6.    if ($|\mathcal{C}| \geq k$) and (Flag == FALSE) then
 7.       call NNS-a
 8.       set Flag = TRUE
 9.    elseif ($|\mathcal{C}| \geq k$) and (Flag == TRUE) then
10.       apply pruning technique
11.       for each entry of node call NNS-b
12.    endif
13. elseif (node is INTERNAL) then
14.    apply pruning technique
15.    sort entries of node wrt mindist at $t_s$
16.    call NNS-CON recursively
17. endif
```

Figure 10. The *NNS-CON* algorithm.

4. Performance evaluation

4.1. Preliminaries

In the sequel a description of the performance evaluation procedure is given, aiming at providing a comparison study among the different processing methods. The methods under consideration are (i) the *NNS-CON* algorithm enabled by pruning technique 1 described in the previous section, and (ii) the *NNS-REP* algorithm which operates by posing repetitive NN queries to the TPR-tree [28]. Both algorithms as well as the TPR-tree access method have been implemented in the C programming language.

There are several parameters that contribute to the method performance. These parameters, along with their respective values assigned during the experimentation are summarized in Table 2.

The datasets used for the experimentation are synthetically generated using the uniform or the gauss distribution. The dataspace extends are 1,000,000 $\times$ 1,000,000 meters and the velocity vectors of the moving objects are uniformly generated, having speed values between 0 and 30 m/sec. Based on these objects, a TPR-tree is constructed. The page size of the TPR-tree is fixed at 2 Kbytes.

The query workload is composed of 500 uniformly distributed queries having the same characteristics (length, velocity). The comparison study is performed by using several performance indices, such as: (i) the number of disk accesses, (ii) the CPU-time, (iii) the

Table 2. Parameters and corresponding values.

Parameter	*Value*
Database size, N	10 K, 50 K, 100 K, 1 M
Space dimensions, d	1, 2, 3
Data distribution, D	Uniform, gaussian
Number of NNs, k	1–100
Travel time, t_{travel}	26–1048 sec
LRU buffer size, B	0.1–20% of tree pages

I/O time and (iv) the total running time. In order to accurately estimate the I/O time for each method a disk model is used to model the disk, instead of assigning a constant value for each disk access [20]. Since the usage of a buffer plays a very important role for the query performance we assume the existence of an LRU buffer having its size vary between 0.1% and 20% of the database size.

The results presented here correspond to uniformly distributed datasets. Results performed for gaussian distributions of data and queries demonstrated similar performance and therefore are omitted. The main difference between the two distributions is that in the case of the gaussian distribution, the algorithms require more resources since the data density increases and therefore more split-points and distance computations are needed to evaluate the queries.

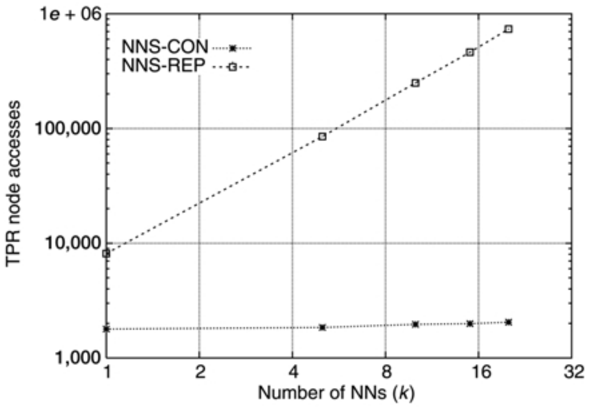
4.2. Performance results

Several experimental series have been conducted in order to test the performance of the different methods. The experimental series are summarized in Table 3.

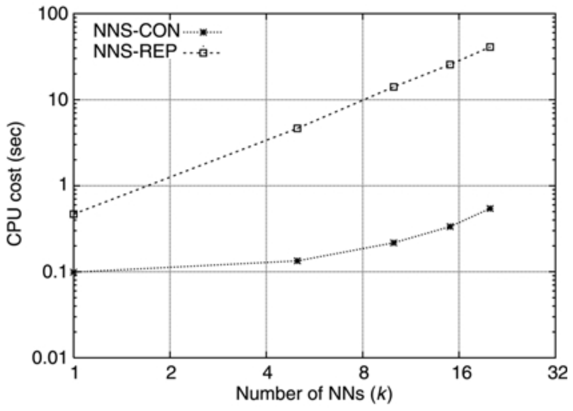
The purpose of the first experiment (EXP1) is to investigate the behavior of the methods for various values of the requested NNs. The corresponding results are depicted in figure 11. By increasing k, more split points are introduced for the *NNS-CON* method, whereas

Table 3. Experiments conducted.

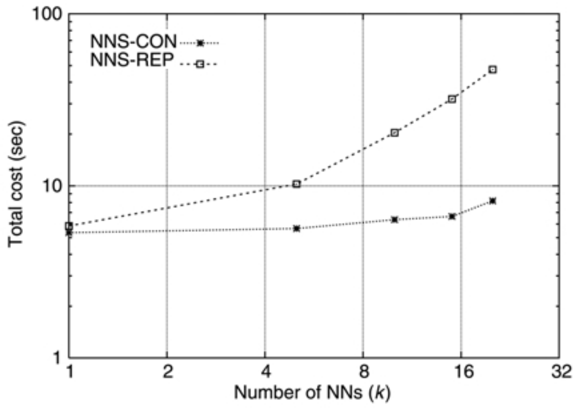
Experiment	*Varying Parameter*	*Fixed Parameters*
EXP1	NNs, k	$N = 1\,\text{M}, B = 10\%, t_{\text{travel}} = 110\,\text{sec}$ $d = 2, D = \text{uniform}$
EXP2	Buffer size, B	$N = 1\,\text{M}, k = 5, t_{\text{travel}} = 110\,\text{sec}$ $d = 2, D = \text{uniform}$
EXP3	Travel time, t_{travel}	$N = 1\,\text{M}, k = 5, B = 10\%,$ $d = 2, D = \text{uniform}$
EXP4	Space dimensions, d NNs, k	$N = 1\,\text{M}, B = 10\%, t_{\text{travel}} = 110\,\text{sec}$ $D = \text{uniform}$
EXP5	Database size, N NNs, k	$B = 500$ pages, $t_{\text{travel}} = 110\,\text{sec}$ $d = 2, D = \text{uniform}$



(a) Tree node accesses



(b) CPU cost (sec)



(c) Total cost (sec)

Figure 11. Results for different values of the number of NNs.

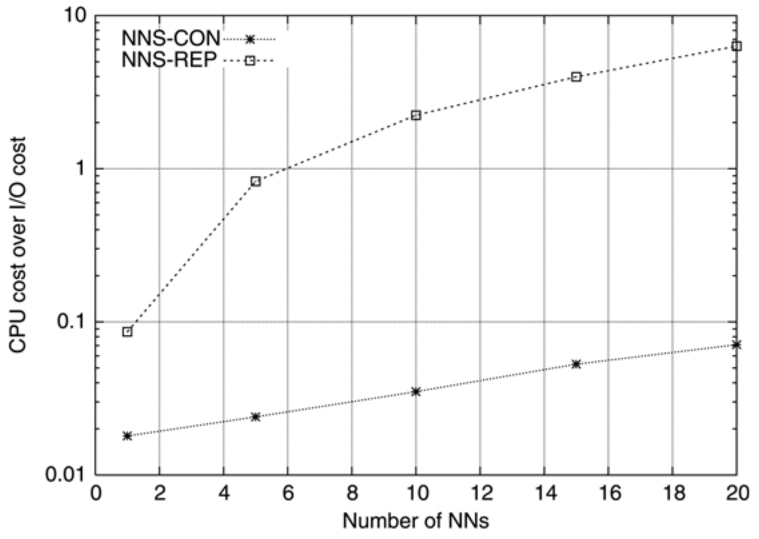


Figure 12. CPU cost over I/O cost.

more influence calculations are needed by the *NNS-REP* method. It is evident that *NNS-CON* outperforms significantly the *NNS-REP* method. Although both methods are highly affected by *k*, the performance of *NNS-REP* degrades more rapidly. As figure 11(a) illustrates, *NNS-REP* requires a large number of node accesses. However, since there is a high locality in the page references performed by a query, the page faults are limited. As a result, the performance difference occurs due to the increased CPU cost required by *NNS-REP* (figure 12). Another interesting observation derived from figure 12 is that the CPU cost becomes more significant than the I/O cost by increasing the number of nearest-neighbors.

The next experiment (EXP2) illustrates the impact of the buffer capacity (figure 13). Evidently, the more buffer space is available the less disk accesses are required by both methods. It is interesting that although the number of node accesses required by *NNS-REP* is very large, (see figure 11(a)) the buffer manages to reduce the number of disk accesses significantly due to buffer hits. However, even if the buffer capacity is limited, *NNS-CON* demonstrates excellent performance.

Experiment EXP3 demonstrates the impact of the travel time to the performance of the methods. The corresponding results are depicted in figure 14. Small travel times are favorable for both methods, because less CPU and I/O operations are required. On the other hand, large travel times increase the number of split-points and the number of distance computations, since the probability that there is a change in the result increases.

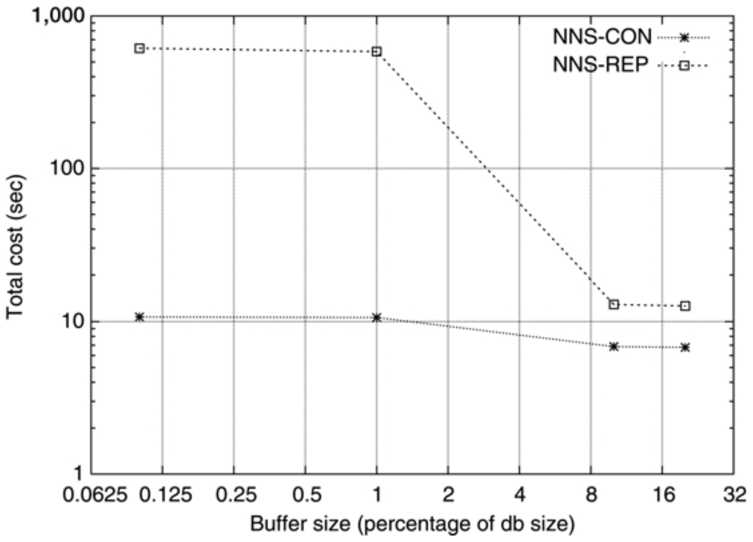
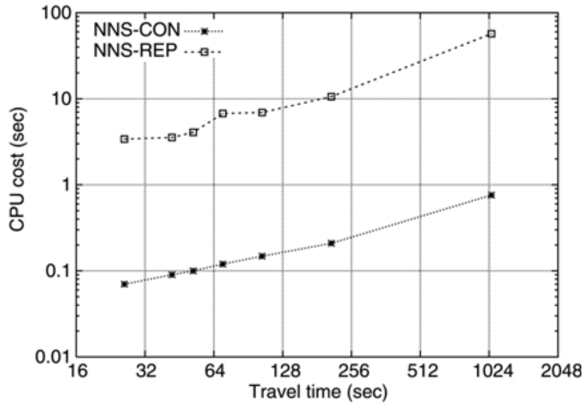


Figure 13. Results for different buffer capacities.

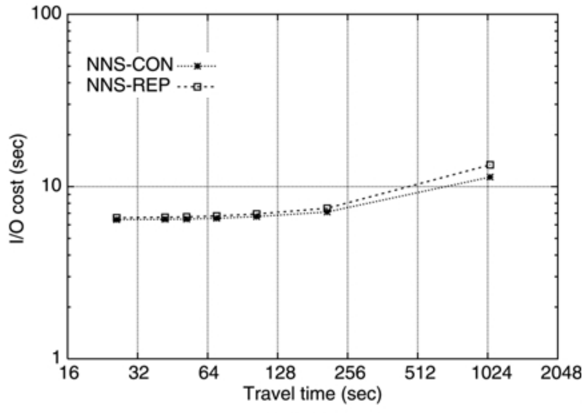
However, *NNS-CON* performs much better for large travel times in contrast to *NNS-REP* whose performance is affected significantly.

The next experiment (EXP4) demonstrates the impact of the space dimensionality. The increase in the dimensionality has the following results: (i) the database size increases due to smaller tree fanout, (ii) the TPR-tree quality degrades due to overlap increase in bounding rectangles of internal nodes, and (iii) the CPU cost increases because more computations are required for distance calculations. Both methods are affected by the dimensionality increase. However, by observing the relative performance of the methods (*NNS-REP* over *NNS-CON*) in 2-D and 3-D space illustrated in figure 15, it is realized that *NNS-REP* is affected more significantly by the number of space dimensions.

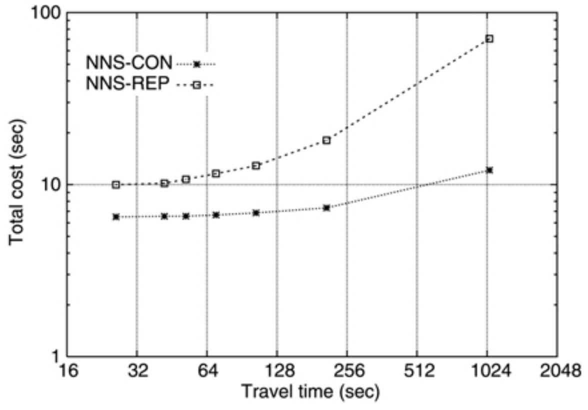
Finally, figure 16 depicts the impact of database size (EXP5). In this experiment, the buffer capacity is fixed to 500 pages, and the number of moving objects is set between 10,000 and 100,000. The number of requested NNs is varying between 1 and 15, whereas the travel time is fixed to 110 sec. By increasing the number of moving objects, more tree nodes are generated and, therefore, more time is needed to search the TPR-tree. Moreover, by keeping the buffer capacity constant, the buffer hit ratio decreases, producing more page faults. As figure 16 illustrates, the performance ratio (*NNS-REP* over *NNS-CON*) increases with the database size.



(a) Tree node accesses



(b) CPU cost (sec)



(c) Total cost (sec)

Figure 14. Results for different values of the travel time.

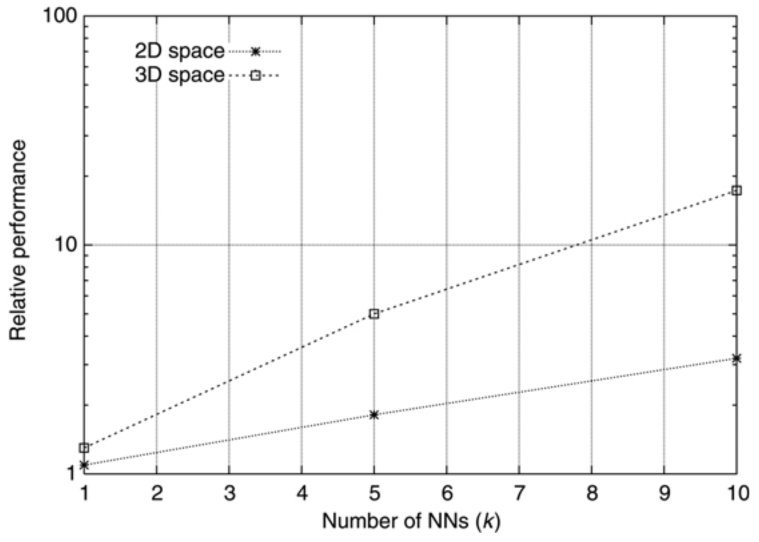


Figure 15. Results for different space dimensions.

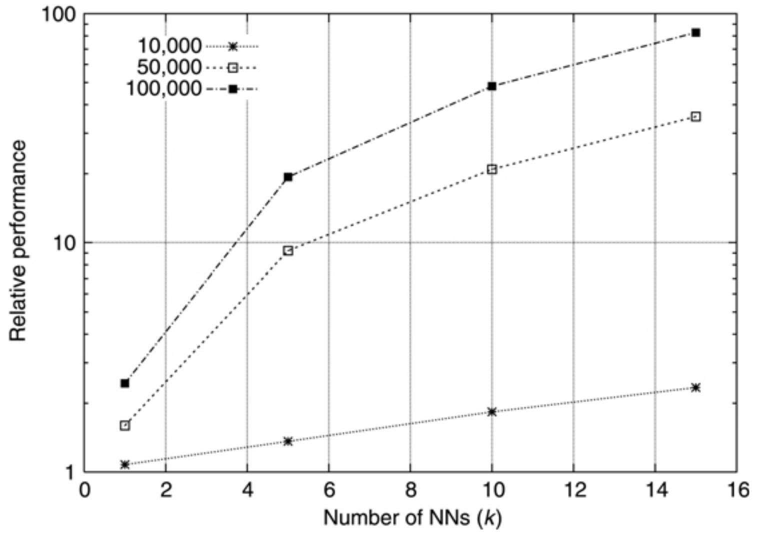


Figure 16. Results for different database size.

5. Concluding remarks

Applications that rely on the combination of spatial and temporal characteristics of the objects demand new types of queries and efficient query processing techniques. An important query type in such a case is the k nearest-neighbor query, which requires the determination of the k closest objects to the query for a given time interval $[t_s, t_e]$. The major difficulty in such a case is that both queries and objects change positions continuously, and therefore the methods that solve the problem for the static case can not be applied directly.

In this work, a study of efficient methods for NN query processing in moving-object databases is performed, and several performance evaluation experiments are conducted to compare their efficiency. The main conclusion is that the proposed algorithm outperforms significantly the repetitive approach for different parameter values. Future research may focus on:

- extending the algorithm to work with moving rectangles (although the extension is simple, the complexity of the algorithm increases due to more distance computations),
- comparing the performance of different pruning techniques,
- studying the performance of the method to other access methods like the STAR-tree [18],
- modifying the algorithm to provide the ability for incremental computation of the NNs, as the work in Hjaltason and Samet [6], [7] suggests for static datasets,
- adapting the method to operate on access methods which store past positions of objects (trajectories), in order to answer past queries, and
- providing cost estimates concerning the number of node accesses, the number of intersection checks and the number of distance computations.

Notes

1. It is assumed that an intersection is defined by two objects. If three or more objects intersect at the same point t_x the conflict is resolved by evaluating the first derivative for each object at t_x and taking the minimum value.
2. The proposed methods can also be combined with a breadth-first-search based algorithm.

References

1. P.K. Agarwal, L. Arge, and J. Erickson. ''Indexing moving points,'' *Proceedings 19th ACM PODS Symposium*, 175–186, 2000.
2. N. Beckmann, H.P. Kriegel, and B. Seeger. ''The R*-tree: an efficient and robust method for points and rectangles,'' *Proceedings ACM SIGMOD Conference*, 322–331, 1990.
3. R. Benetis, C.S. Jensen, G. Karciauskas, and S. Saltenis. ''Nearest neighbor and reverse nearest neighbor queries for moving objects,'' *Proceedings IDEAS Conference*, 44–53, 2002.

4. A. Guttman. ''R-trees: A dynamic index structure for spatial searching,'' *Proceedings ACM SIGMOD Conference*, 47–57, 1984.
5. M. Hadjieleftheriou, G. Kollios, V.J. Tsotras, and D. Gunopoulos. ''Efficient indexing of spatio-temporal objects,'' *Proceedings 8th EDBT Conference*, 251–268, 2002.
6. G. Hjaltason and H. Samet. ''Ranking in spatial databases,'' *Proceedings 4th SSD Symposium*, 83–95, 1995.
7. G. Hjaltason and H. Samet. ''Distance browsing in spatial databases,'' *ACM Transactions on Database Systems*, Vol. 24(2):265–318, 1999.
8. Y. Ishikawa, H. Kitagawa, and T. Kawashima. ''Continual neighborhood tracking for moving objects using adaptive distances,'' *Proceedings IDEAS Conference*, 54–63, 2002.
9. D.V. Kalashnikov, S. Prabhakar, S.E. Hambrusch, and W.G. Aref. ''Efficient evaluation of continuous range queries on moving objects,'' *Proceedings 13th DEXA Conference*, 731–740, 2002.
10. G. Kollios, D. Gounopoulos, and V.J. Tsotras. ''Nearest neighbor queries in a mobile environment,'' *Proceedings Workshop on Spatio-temporal Database Management*, 119–134, 1999a.
11. G. Kollios, D. Gunopoulos, and V. Tsotras. ''On indexing mobile objects,'' *Proceedings 18th ACM PODS Symposium*, 261–272, 1999b.
12. A. Kumar, V.J. Tsotras, and C. Faloutsos. ''Designing access methods for bitemporal databases,'' *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10(1):1–20, 1998.
13. I. Lazaridis, I. Porkaew, and S. Mehrotra. ''Dynamic queries over mobile objects,'' *Proceedings 8th EDBT Conference*, 269–286, 2002.
14. D. Lomet and B. Salsberg. ''Access methods for multiversion data,'' *Proceedings ACM SIGMOD Conference*, 315–324, 1989.
15. J. Moreira, C. Ribeiro, and T. Abdessalem. ''Query operations for moving objects database systems,'' *Proceedings 8th ACM-GIS Workshop*, 108–114, 2000.
16. M.A. Nascimento and J.R.O. Silva. ''Towards historical R-trees,'' *Proceedings 13th ACM SAC Symposium*, 235–240, 1998.
17. D. Pfoser, C.S. Jensen, and Y. Theodoridis. ''Novel approaches to the indexing of moving object trajectories,'' *Proceedings 26th VLDB Conference*, 395–406, 2000.
18. C.M. Procopiuc, P.K. Agarwal, and S. Har-Peled. ''STAR-tree: An efficient self-adjusting index for moving objects,'' *Proceedings ALENEX Conference*, 178–193, 2002.
19. N. Roussopoulos, S. Kelley, and F. Vincent. ''Nearest neighbor queries,'' *Proceedings ACM SIGMOD Conference*, 71–79, 1995.
20. C. Ruemmler and J. Wilkes. ''An introduction to disk drive modeling,'' *IEEE Computer*, Vol. 27(3):17–29, 1994.
21. S. Saltenis, C.S. Jensen, S. Leutenegger, and M. Lopez. ''Indexing the positions of continuously moving objects,'' *Proceedings ACM SIGMOD Conference*, 331–342, 2000.
22. A.P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. ''Modeling and querying moving objects,'' *Proceeding 13th IEEE ICDE Conference*, 422–432, 1997.
23. Z. Song and N. Roussopoulos. ''K-NN search for moving query point,'' *Proceedings 7th SSTD Symposium*, 79–96, 2001a.
24. Z. Song and N. Roussopoulos. ''Hashing moving objects,'' *Proceedings 1st Workshop on Mobile Data Management*, 161–172, 2001b.
25. Y. Tao and D. Papadias. ''Efficient historical R-trees,'' *Proceedings 13th SSDBM Conference*, 2001a.
26. Y. Tao and D. Papadias. ''MV3R-tree – a spatio-temporal access method for timestamp and interval queries,'' *Proceedings 27th VLDB Conference*, 431–440, 2001b.
27. Y. Tao, D. Papadias, and Q. Shen. ''Continuous nearest neighbor search,'' *Proceedings 28th VLDB Conference*, 287–298, 2002a.
28. Y. Tao and D. Papadias. ''Time-parameterized queries in spatio-temporal databases,'' *Proceedings ACM SIGMOD Conference*, 334–345, 2002b.
29. Y. Theodoridis, M. Vazirgiannis, and T. Sellis. ''Spatio-temporal indexing for large multimedia applications,'' *Proceedings 3rd IEEE Conference on Multimedia Computing and Systems*, 441–448, 1996.
30. Y. Theodoridis, T. Sellis, A.N. Papadopoulos, and Y. Manolopoulos. ''Specifications for efficient indexing in spatio-temporal databases,'' *Proceedings 10th SSDBM Conference*, 123–132, 1998.

31. O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. ''Moving objects databases: Issues and solutions,'' *Proceedings 10th SSDBM Conference*, 111–122, 1998.
32. O. Wolfson, B. Xu, and S. Chamberlain. ''Location prediction and queries for tracking moving objects,'' *Proceedings 16th IEEE ICDE Conference*, 687–688, 2000.
33. X. Xu, J. Han, and W. Lu. ''RT-tree: An improved R-tree index structure for spatio-temporal databases,'' *Proceedings SDH Conference*, 1040–1049, 1990.
34. B. Zheng and D. Lee. ''Semantic caching in location-dependent query processing,'' *Proceedings 7th SSTD Symposium*, 97–116, 2001.