# Continuous K-Nearest Neighbor Search for Moving Objects [*]

Yifan Li, Jiong Yang, Jiawei Han
Department of Computer Science
University of Illinois
Urbana, IL 61801
{yifanli, jioyang, hanj}@cs.uiuc.edu

## Abstract

*The paper describes a new method of continuously monitoring the $k$ nearest neighbors of a given object in the mobile environment. Instead of monitoring all $k$ nearest neighbors, we choose to monitor the $k$-th (nearest) neighbor since the necessary condition of changes in the KNN is the change of the $k$-th neighbor. In addition, rather than in the original space, we consider the moving objects in a transformed time-distance (TD) space, where each object is represented by a curve. A beach-line algorithm is developed to monitor the change of the $k$-th neighbor, which enables us to maintain the KNN incrementally. An extensive empirical study shows that the beach-line algorithm outperforms the most efficient existing algorithm by a wide margin, especially when $k$ or $n$ (the total number of objects) is large.*

## 1. Introduction

The proliferation of mobile objects and highly advanced positioning techniques make some e-services to large number of mobile objects possible and desirable, which also poses new challenges to exploit the knowledge of objects' changing locations, one of which involves providing the information of the $k$ nearest neighbors of a given object. Informally, the KNN problem is to find a set of $k$ nearest mobile objects to a given location at a given moment.

The KNN problem on moving objects is receiving increasing attention in the research community. Saltenis et al. [3] propose an R*-tree [1] based indexing structure (TPR-tree) to answer range queries on moving objects. However, the performance tends to degrade with time due to the deterioration of the TPR-tree. Tao and Papadias provide a general framework for handling the queries on moving objects in [4]. However, their approach cannot work well if there are updates of the moving objects, which occur frequently in real applications. In the most recent work [2], Iwerks et al. propose an efficient continuous windowing algorithm, reducing the KNN query to the less expensive range query. Their work provides the best known result regarding the KNN query on moving objects with updates. However, their algorithm gives sub-optimal performance when the rebuilds are frequent, or when $k$ is large. Our approach is found to perform better in comparison with theirs, as shown in our experiments.

Our proposal is based on the following observation: *the necessary condition for the change of KNN is that the order of the $k$-th nearest neighbor and the $(k+1)$-th nearest neighbor switches*[1], that is, when the distance from the query object of the $k$-th nearest neighbor becomes larger than that of the $(k+1)$-th one. In consequence, rather than monitoring all of the $k$ nearest neighbors, one may keep track of only the $k$-th and the $(k+1)$-th nearest neighbors. This will greatly reduce the number of objects that one needs to monitor. However, the $k$-th and the $(k+1)$-th may be physically located at different spatial regions. Thus, instead of considering objects in their original space, we transform the space into the time-distance space. Figure 1 (a) shows the object movements in the original space while the objects in the time-distance space are illustrated in Figure 1 (b). In the time-distance space, the $x$-axis is the time while the $y$-axis is the distance from the query object (point) at a given time. Although the $k$-th and the $(k+1)$-th nearest neighbors may be located far away in the original space, but they should be located close in the transformed space by definition. Therefore, we only need to monitor the region around the $k$-th nearest neighbor in the time-distance space.

However, one side effect of this approach is that the trace

[*] The work was supported in part by U.S. National Science Foundation NSF IIS-03-08215, University of Illinois, and an IBM Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

[1] Note that we delay the consideration of updates here, which will be handled later.

of an object in the time-distance space is a curve. It is quite difficult and complicated to index or search curves. In this paper, we decompose a curve into a set of segments, each of which is approximated by a bounding rectangle (BR). An R-tree is utilized to index these BRs.

When an object changes its location and velocity, we need to update its curve, which brings out the update of the R-tree. This could be an expensive operation. Fortunately, we do not need to update the R-tree for every object update. A *delayed update* approach is suggested to minimize the cost.

## 2. Problem definition and notations

We assume objects move in a piecewise linear manner. In this scenario, an object moves along a straight line with some constant speed till it changes the direction and/or speed. We use a vector $\vec{x} = (x_1, x_2, \ldots, x_n)$ to denote the location of a moving object. It is a function of time $t$ and can be written as $\vec{x}(t) = \vec{x}(t_0) + \vec{v}t$, where $\vec{x}(t_0)$ is the initial location of the object at some referential instance $t_0$, and $\vec{v} \in \Re^n$ is the velocity vector. We only consider the 2-$D$ case in this paper, which can be easily generalized to $n$-$D$ (for arbitrary $n$).

In the moving object framework, the KNN problem is defined as: *Given a set $S$ of moving objects and an object $o$, at some time instant $t$, find $k$ objects which are the closest to $o$ at time $t$.* We choose Euclidean distance here for simplicity and clarity, although any other eligible metric will work under our framework.

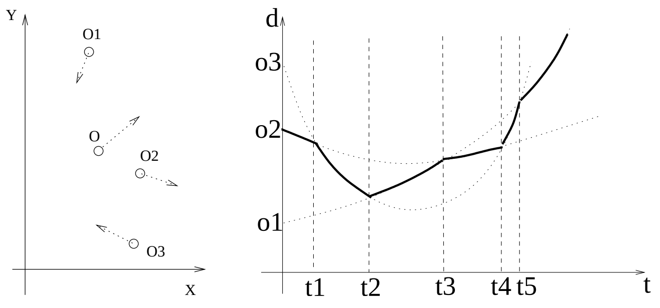
Given a set $S$ of moving objects, the KNN of a query object $o$ at time $t$ is represented by $S_k^o(t)$. The $k$-th neighbor is denoted as $o_k^o(t)$. We use *KNN* and *KNN set* interchangeably. We omit the term $o$ and $t$ when there is no confusion.

In order to represent moving objects, we denote the location of an object $o$ at time $t$ as $loc_o^t$. In 2-$D$ space, $loc_o^t = (x_o^t, y_o^t)$, where $x_o^t$ (or $y_o^t$) is the $X$ (or $Y$) coordinate of object $o$ at time $t$. The term $t$ is omitted when it is clear from the context. Let $\vec{v}_o^{(t_1,t_2)}$ denote the velocity vector of object $o$ during time slot $(t_1, t_2)$. The term $(t_1, t_2)$ is omitted when it is clear from the context. The distance between two objects $o_1, o_2$ at time $t$ is defined as $d_t(o_1, o_2)$. The term $t$ is omitted when there is no confusion.

## 3. Our algorithm

Upon the above assumptions, the distance between two moving objects is a polynomial function of time $t$, *i.e.*, $d_t^2(o_1, o_2) = at^2 + bt + c$, where $a$, $b$, and $c$ are parameters dependent on the velocities of objects and the initial distance between them.

Given a query object, we are going to continuously monitor its KNN. Figure 1 (a) shows a *snapshot* of moving objects in the original space at some time $t$. The query object is $o$. The arrows indicate the moving directions of the objects. Clearly, $d(o, o_1) \approx d(o, o_3)$, though $loc_{o_1}$ is far from $loc_{o_3}$. For this reason, it is more meaningful to map the objects onto a time-distance space, where each object $o_i$ other than $o$ is denoted by a curve $d(o, o_i)$, where $o$ is the query object.



(a)The original space. (b)The time-distance space.

**Figure 1. Space transformation.**

| Time slot | 2NN | 2nd neighbor |
|---|---|---|
| $(-\infty, t_1]$ | $o_1, o_2$ | $o_2$ |
| $(t_1, t_2]$ | $o_1, o_3$ | $o_3$ |
| $(t_2, t_3]$ | $o_1, o_3$ | $o_1$ |
| $(t_3, t_4]$ | $o_2, o_3$ | $o_2$ |
| $(t_4, t_5]$ | $o_2, o_3$ | $o_3$ |
| $(t_5, +\infty)$ | $o_1, o_2$ | $o_1$ |

**Table 1. The 2NN and the 2nd neighbor of Figure 1 (b)**

Assume there are $n$ moving objects in MOD, so there will be $n-1$ such curves on the time-distance space. For the time being, we assume that there is no update, which will be discussed later. As shown in Figure 1 (b), in time-distance space, $o_i$ is represented by functions $d(o, o_i)$ (dotted line) ($i = 1, 2, 3$). Suppose we want to find the 2NN for object $o$. The solid line indicates the 2nd neighbor within that time duration. We call such a line *beach line*, because it is typically composed of a sequence of parabolic-like arcs. From the figure, we can find out the 2NN and 2nd neighbor easily, which is summarized in Table 1.

In general, the beach line represents the $k$-th neighbor over time. Namely, at any time, objects that are below the beach line constitute the KNN. Hence, it suffices to maintain the beach line to keep track of the KNN over time.

Intuitively, the beach line is composed of parts of distance curves, each of which represents the $k$-th neighbor during the corresponding time slot. For example, during time period $(t_2, t_3)$, the 2nd neighbor is $o_1$ (Figure 1 (b)). Note that the beach line is not necessarily continuous when there are updates. The time instances when the $k$-th neighbor changes are called $events$, which are the t-coordinate of the intersections. We also use $event$ to represent the intersection when there is no confusion. The set of events is defined as $E$. The time instances when the $k$-NN changes are called *critical events*, whose set is denoted as $CE$. In general, $CE \subseteq E$. For example, in Figure 1 (b), $E = \{t_1, t_2, t_3, t_4, t_5\}$, and $CE = \{t_1, t_3, t_5\}$. We have the following property.

**Property 1** *Given the event $e$, the $k$-th neighbor before $e$ ($o_k^{e-\Delta t}$), and KNN before $e$ ($S_k^{e-\Delta t}$), $e$ arises when curve $d(o, o_k^{e-\Delta t})$ intersects with some other curve $d(o, o')$, after which $o_k^{e+\Delta t} = o'$. Particularly, if $o' \notin S_k^{e-\Delta t}$, $S_k^{e+\Delta t} = S_k^{e-\Delta t} - \{o_k^{e-\Delta t}\} + \{o_k^{e+\Delta t}\}$. Namely, $e$ is also a critical event. Otherwise, if $o' \in S_k^{e-\Delta t}$, then $S_k^{e+\Delta t} = S_k^{e-\Delta t}$, where $\Delta t$ is a small amount of time.*

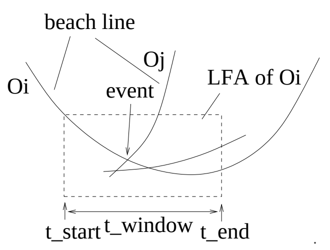
The property shows how to obtain/identify events and critical events, and incrementally maintain the $k$-th neighbor and KNN. Consequently, the KNN problem is reduced to the *problem of efficient construction of the beach line*.

## 3.1. Beach line construction

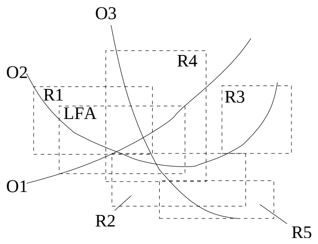
Initially, $o_k$ and KNN are obtained by scanning the database, which is done only once. After that, the beach line is dynamically maintained by finding the next event. In principle, we can scan the database to find the curves that intersect with the current curve $d(o, o_k)$. The next event will be the minimum t-coordinate of those intersection points, if there is any.

As the number of objects $n$ is usually huge, the above brutal-force method appears less desirable due to the expensive scanning operation. The key observation is that curves that are close together are candidates for intersection. Therefore, it is sufficient to pay attention to the surrounding area of the curve to find the next intersection. Typically the number of curves that are $near$ $d(o, o_k)$ is much less than $n$, which is both intuitively correct and empirically proved in the experiments.

As a matter of fact, given the curve $d(o, o_k)$, the start time $t\_start$, and time window $t\_window$, we could define a rectangular *looking-forward area* $LFA_{o_k}$ $s.t.$ either the event $e$ takes place within the area, or there is no event within the area, which means that the curve will intersect nothing during the time slot $(t\_start, t\_start + t\_window)$ (Figure 2 (a)). The $k$-th neighbor switches upon an event. In Figure 2 (a), $o_k^{e-\Delta t} = o_i, o_k^{e+\Delta t} = o_j$. In this way, the beach line can be constructed by *following* the looking-forward area of the current $k$-th neighbor. We use $t\_end$ to represent the end time of $LFA_{o_k}$, i.e., $t\_end = t\_start + t\_window$. We use simplified notation $LFA$ when the $k$-th neighbor is clear.



(a) The looking forward area.



(b) The bounding rectangles.

**Figure 2. The construction of the beach line.**

However, the previous question remains: though we are now able to bound the area where an event could take place, do we still have to scan the whole object set to find the event? The answer is negative because we need only to consider the adjacent curves. Furthermore, we are interested in only the adjoining parts of those curves. In order to efficiently find those parts, we decompose each curve $o_i$ into bounding rectangles ($BR_{o_i}$s). Consequently, the problem is reduced to *spatial query problem* that is to be solved by existing spatial indexing structures.

In Figure 2 (b), $BR_{o_2} = \{R_1, R_2, R_3\}, BR_{o_3} = \{R_4, R_5\}$. By some indexing structure (e.g., R-tree), the close rectangles that overlap the $LFA$, which are $R_1, R_2$ and $R_4$ in the figure, can be efficiently fetched. The event is obtained by computing the intersections between the curve of the $k$-th neighbor and the parts of curves inside the overlapping bounding rectangles.

Note that $BR_{o_i}$s constitute a partition of curve $o_i$ such that each point of the curve $o_i$ belongs to some $BR_{o_i}$, and there is no overlapping between different $BR_{o_i}$s. The $BR$s of all curves are stored as leaves of an R-tree, called *NN-Tree*. As a consequence, the event can be computed with small cost, considering that the number of nearby $BR$s is usually very small.

## 3.2. Updates

In most applications, moving objects could stop or change their velocities, and new moving objects could join the system. These correspond to the *deletion*, *modification*, and *insertion* of MOD, which is the *update* of the database. Our approach provides sound support for such updates.

It is clear that each update could result in the corresponding change in the NN-Tree. A *delayed update approach* is proposed so as to avoid the expensive cost resulted from frequent updates: We delay the operation of all updates, saving them into an *update list* in memory. When we compute the event, we need to check the update list to see whether the object $o$ inside the $BR$ has been updated. If yes, we need to use the updated parameters to compute. Simultaneously, we need to examine if the curves in the update list would intersect the current $k$-th neighbor. If so, we also consider them when computing the next event. Since the number of adjacent curves is usually very small, and we have the observation that most such updates do not affect the current KNN set, as shown in the experiments, the approach is proved to be well worth the additional cost. We batch update until the update list reaches its size limitation.

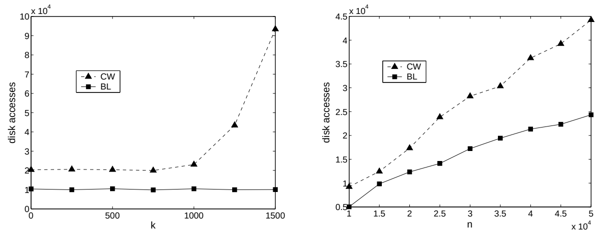
## 4. Experimental evaluation

We compare our result with the CW algorithm proposed by Iwerks et al. in [2]. We implement our algorithm on a Pentium 4 PC under Linux 2.4.20-8 with C++.

Since each moving object is represented by multiple BRs in our BL (Beach Line) algorithm, we expect the data structure size of ours is larger than that of the CW algorithm, although they are of the same asymptotic complexity $O(n)$, where $n$ is the size of the dataset. As a result, our rebuild cost will be higher. However, the CW algorithm leads to significantly more rebuilds. During the time interval of length $TimeLen$, there is no rebuild with BL algorithm, while there usually are multiple rebuilds with CW algorithm, due to the breaks of the constraints. The BL algorithm excels in terms of the overall performance, since the rebuild cost is the dominant factor. Note that in the experiment, we do not consider the initial build, as did in [2].

We also set up a similar experimental environment as follows: Our experiments are conducted in a 2-D world with size 1000x1000, where objects are uniformly distributed. The speed of an object is a random variable that satisfies the uniform distribution on $(0, 3)$. We decompose each curve into 4 bounding rectangles. The window of LFA is set to be 2 time units. The time between two consecutive updates for an object is uniformly distributed on $(0, 120)$ with the average 60. We use $k = 10$ and 50000 objects unless otherwise specified. For the CW algorithm, distance $d$ is set to 50 ([2]). We select the query object randomly, and the query object does not change the object during the experiments.

In Figure 3 (a), we compare the disk accesses of BL algorithm with CW algorithm under different $k$ values. From the figure, it is easy to observe that when $k$ is not very large, both algorithms scale well. However, the number of disk accesses of CW begins to increase quickly for some larger $k$, while BL is insensitive to different $k$ values. This is due to the fact that CW needs to keep at least $k$ objects in memory, whereas BL only pays attention to the $k$-th neighbor. As a result, BL successfully avoids the *thrashing* phenomenon that CW will inevitably meet with large $k$ value.



(a) (b)

**Figure 3. Varying $k$ and $n$**

We examine the scalability of BL and CW with respect to the object set size. The result is shown in Figure 3 (b), from which we can see that CK is linearly scalable with respect to the number of objects, and BL demonstrates the sub-linear scalability. This confirms our expectation, since CW scans the whole database to rebuild when it fails to have the appropriate number of objects in memory. In contrast, the query/update operation on NN-Tree only involves sub-linear ($O(\log n)$) disk accesses in that the NN-Tree is well balanced.

## 5. Conclusion

In this paper, we propose an algorithm on KNN query over moving objects, which is based on the following observation: rather than keep track of the whole k nearest neighbors, it is sufficient to monitor the $k$-th neighbor. Due to the possible large distance between the $k$-th neighbor and those *adjacent* neighbors (e.g., the $(k + 1)$-th neighbor) in the original space, we consider them in the time-distance space, where they are guaranteed to be *close* to each other. Each object is represented by a function of time in the new space. With a proposed technique to index curves, we are able to monitor the $k$-th neighbor dynamically, and maintain the KNN incrementally. The approach is shown to outperform the best known algorithm significantly in our experiments.

## References

[1] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. ”The $R^*-$tree: an effi cient and robust access method for points and rectangles”, *SIGMOD*, 1990.

[2] G.S. Iwerks, H. Samet and K. Smith. ”Continuous k-nearest neighbor queries for continuously moving points with updates”, *VLDB*, 2003.

[3] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez. ”Indexing the positions of continuously moving objects”, *SIGMOD*, 2000.

[4] Y. Tao and D. Papadias. ”Time-parameterized queries in spatio-temporal databases”, *SIGMOD*, 2002.