Incremental Rank Updates for Moving Query Points

L. Kulik and E. Tanin

Department of Computer Science and Software Engineering
NICTA Victoria Laboratory, University of Melbourne, Victoria 3010, Australia
{lars, egemen}@csse.unimelb.edu.au

Abstract. The query for retrieving the rank of all neighbors of a moving object at any given time, a continuous rank query, is an important case of *continuous nearest neighbor* (CNN) queries. An application for ranking queries is given by an ambulance driver who needs to keep track of the closest hospitals at all times. We present a set of *incremental* algorithms that facilitate efficient rank updates for some or all neighbors of a moving query point. The proposed algorithms allow us not only to maintain the exact rank of all n neighbors at any given time but also to track the rank of a subset of all neighbors. We show that updates for these continuous rank queries can be performed in linear time for arbitrary polygonal curves in two dimensions and in logarithmic time for movements along a fixed direction. Instead of using Voronoi diagrams, our algorithms are based on small subsets of all bisectors between neighbors. We prove that it is sufficient to keep track of only $n - 1$ bisectors for all n neighbors. The algorithms for maintaining the rank only require minimal incremental updates on the bisector sets.

1 Introduction

Nearest neighbor (NN) queries are a well investigated research topic. With the ubiquitous availability of location information for mobile devices, continuous nearest neighbor (CNN) queries have become a major research focus. An important variant of CNNs are continuous ranking queries that retrieve the rank of k neighbors of a moving object. *Continuous ranking queries* have a wide application potential. For example, an ambulance driver may wish to continuously keep track of some of the hospitals in a city ranked by their distance using a mobile device. Alternatively, the driver of a car might want to track a set of gas stations or hotels with respect to their distance. In this paper, we propose a class of algorithms that facilitate efficient rank updates for a moving query point in order to continuously query the rank of the neighbors.

Algorithms retrieving the neighbors of a query point in the order of their distance relative to the query point are called *ranking* algorithms, because they rank spatial objects of a given database for that query point [6]. If only a subset of objects has to be retrieved from a database, the queries are called kNN queries, where k is the number of retrieved objects. The number of objects k can be less than or equal to the total number n of objects in the database. If the order of the neighbors is irrelevant, the query is called an *order insensitive* kNN query [7]. In this paper, we study *order sensitive* kNN queries, i.e., queries that retrieve objects in the order of their distance.

A number of algorithms have been developed to efficiently find the nearest neighbors for a static query point. One way to derive an algorithm for a CNN query is to use a classical NN query algorithm: for each predetermined time or location update interval, a new NN query for a moving query point can be executed using one of the classical NN query algorithms. However, a repeated execution of the classical algorithms will not take advantage of the steps and the results of previously run NN queries. In addition, many of the objects of a previously run NN query may not have changed their status. Continuous requerying also might miss some of the rank updates if the frequency of update queries is set too low for a given application domain.

In this paper, we present a set of incremental algorithms that track the rank of the neighbors of a moving query point continuously. The positions of the neighbors are assumed to be fixed. We show that updates for such continuous rank queries can be performed in linear time for any arbitrary polygonal curve without any prior knowledge of the curve itself. For polygonal curves that largely consist of straight line segments with a limited number of turns, a modified algorithm is presented that achieves logarithmic complexity for movements along the straight line segments.

Our algorithms use a small subset of all bisectors between neighbors of the moving query point, and only need to maintain a list of $n-1$ bisectors for all n neighbors at any given time. We show that the bisector list is sufficient for a moving query point in order to (i) detect when rank updates are needed and (ii) which ranks need to be updated once the need arises. We finally show the connection of our work with higher-order Voronoi diagrams [1,11].

We believe that incremental algorithms for continuous queries will gain significant importance with the increasing use of wireless and mobile systems as wireless applications are typically constrained by limited communication and computation resources. The remainder of the paper is organized as follows: Current approaches for CNN queries are presented in Section 2. In Section 3, we demonstrate the algorithm for continuous rank updates in the one-dimensional case in order to motivate the underlying ideas of our approach. Section 4 presents the algorithm for ranking all neighbors of a moving query point in the two-dimensional case. In Section 5, we modify our algorithm for continuous rank updates to capture travel patterns that contain only a few directional changes. Section 6 develops algorithms for two variants of continuous rank updates: how to maintain a single rank and how to maintain multiple ranks by possibly different agents. Section 7 concludes our work and outlines future work.

2 Related Work

NN queries for static query points have been investigated by many researchers [4,5,6,13]. Different types of branch-and-bound algorithms have been studied on a number of spatial data structures, in particular R-trees and quadtrees [15,16,17]. Roussopoulos et al. [13] propose a depth-first branch-and-bound algorithm on R-trees, while Hjaltason and Samet [5] take a best-first approach. These algorithms are designed to efficiently find static nearest neighbors for a static query point in various applications.

For efficient indexing and querying of moving objects Saltenis et al. [14] introduce the Time-Parameterized R-tree (TPR-tree). TPR-trees are based on the assumption that the trajectory information is available and the location of each moving object can be represented as a linear function of time. Benetis et al. [2] build on the TPR-tree and propose algorithms for NN and reverse NN queries. Reverse NN queries are used to find the objects that have a query point as their nearest neighbor. Recent approaches also use a grid-based model [3,9,22] rather than extending the R-tree concept. Our work differs from the moving object indexing research as we assume that the objects are static but the query point is in motion. In addition, many moving object indexing methods, such as the TPR-tree-based schemes, assume that the trajectories are known to a certain extent. Finally, these indexing methods commonly focus on spatial *snapshot* queries while our work focuses on continuous queries.

Tao and Papadias [19] introduce time-parameterized queries in spatio-temporal data that extend the snapshot queries with the *expiry time* concept. A time-parameterized query returns an expiry time along with a set of results. Other researchers have extended this work. For example, Iwerks et al. [8] build on this concept to allow for changes on pending update events for enabling truly continuous spatial queries. However, these papers neither focus on maintaining the rank information for the spatial objects nor on static objects with a moving query point.

Zheng and Lee [24], and Song and Roussopoulos [18] developed the first approaches to address queries where the objects are assumed to be static but the query point is in motion. Zheng and Lee use a Voronoi diagram-based approach to answer 1-NN queries. Song and Roussopoulos address kNN queries, where k can be greater than one, and use sampling-based methods to avoid brute-force resubmissions of new queries for a moving query point. A sampling-based approach, however, may miss certain updates or become computationally expensive for frequent updates.

Tao et al. [20,21] introduce CNN queries for moving objects with *a priori* known trajectories. Given a line segment, their result set contains a set of (*object*, *interval*) tuples. Each tuple represents the NN for a given interval on the line segment. A kNN version of the algorithm is also available. This work can be extended to trajectories consisting of several line segments.

Zhang et al. [23] introduce a location-based querying approach that is closely related to our work. They define *validity regions* for which the result of a query remains the same. Hence, clients do not need to resubmit queries that will not change the result set. In contrast to our work, they do not consider order sensitive ranking queries. They acknowledge the necessity of incremental algorithms that reduce redundant computations and communication, a requirement which we directly address in our approach.

Extensions to compute multiple CNN queries more efficiently are given in [7,10]. In [7] *safe regions* (similar to validity regions) are defined, where the output of multiple, registered CNN queries cannot change for a database of moving objects. They use the concept of safe regions to reduce the communication overhead in wireless applications. In [10] *conceptual partitioning* is used to ignore updates from objects that are far away from a query to reduce runtime costs. Such techniques can be used in tandem with our work to handle multiple CNN queries.

3 One-Dimensional Queries

We first present our approach in the case where all bisectors can be ordered along one axis to motivate the basic ideas. As a simplified notation, we call this case the one-dimensional case and present our approach on the x-axis. Many properties carry over to CNN queries on higher dimensions.

3.1 Notation and Definitions

Assume n sites $\{S_1, S_2, \ldots, S_n\}$ are ordered on a straight line, i.e., a moving query point Q will first hit S_1, then S_2, and so forth until Q finally hits the last site S_n. A perpendicular bisector between two sites S_i and S_j, is called $B(S_i, S_j) =_{\text{def}} B_{ij}$ and is defined as the straight line that is equidistant from the sites S_i and S_j:

$$B(S_i, S_j) =_{\text{def}} \{P \mid d(P, S_i) = d(P, S_j)\}$$

where d is the usual Euclidean distance. Since $B_{ij} = B_{ji}$ by definition, we can assume without loss of generality that $i < j$ for B_{ij} throughout the rest of this paper. Let d_i denote the distance from the query point Q to a site S_i, i.e., $d_i =_{\text{def}} d(Q, S_i)$. In addition, we introduce a precedence relation $\prec$ for bisectors: $B_{ij} \prec B_{kl}$, if $d(Q, B_{ij}) < d(Q, B_{kl})$. Given these definitions, a set of basic properties that we use for our algorithms is summarized in the following observation.

Observation 1. *If the sites $\{S_1, S_2, \ldots, S_n\}$ are all on a single straight line with $d_i < d_{i+1}$ $\forall i \in \{1, \ldots, n-1\}$, then*

1. $\forall i, j, k : i < j \Rightarrow B_{ik} \prec B_{jk}$
2. $\forall i, j, k : j < k \Rightarrow B_{ij} \prec B_{ik}$
3. $\forall i, j, k, l : i < k \wedge j < l \Rightarrow B_{ij} \prec B_{kl}$

These properties follow from the fact that $d_i < d_j$ if $i < j$ and the triangle inequality for distances.

3.2 A Base Algorithm

We explain the algorithm in the one-dimensional case with an example. At each step we maintain two sorted lists $\mathcal{S}$ and $\mathcal{B}$. $\mathcal{S}$ contains the sites ordered in terms of their distance to the query point, i.e., their rank order. We also maintain $\mathcal{B}$, a list of bisectors ordered by increasing x coordinates. We use the bisector list to detect when the next rank update will occur. It suffices to keep only those $n-1$ bisectors in the list $\mathcal{B}$ that are of the form B_{ii+1}. This means we only keep those bisectors between sites that are neighbored in $\mathcal{S}$. We show in Section 4.1 for the two-dimensional case, why we do not need to track all bisectors (i.e., $\mathcal{O}(n^2)$ bisectors).

Let $\{\ldots, S_i, S_j, S_k, S_l, \ldots\}$ be an ordered list for the sites. If the query point crosses a bisector B_{jk} then S_k is now closer to the query point than the site S_j. This also means that the list $\mathcal{S}$ changes to $\{\ldots, S_i, S_k, S_j, S_l, \ldots\}$. Since we only keep track of the bisectors between neighboring sites, our list $\{\ldots, B_{ij}, B_{jk}, B_{kl}, \ldots\}$ changes to $\{\ldots, B_{ik}, B_{jk}, B_{jl}, \ldots\}$. More precisely, we have to delete at most two bisectors from $\mathcal{B}$ and correspondingly insert at most two new ones. Note that these bisectors need not to be located next to each other on the straight line. They will be neighbors to each other if they are kept under the $\prec$ order rather than the order given by the x-axis. For each rank update, we need to update the list of neighboring bisectors in order to detect the next rank update. In our example of a query point that moves from S_1 to S_5 (Figure 1), the algorithm will update the two lists as indicated below. At each step, the position of the query point with respect to the bisectors is shown with a '*'. We also show which bisectors are introduced and removed from $\mathcal{B}$.

$$\begin{aligned} S &= \{S_1, S_2, S_3, S_4, S_5\},\\ B &= \{*, B_{12}, B_{23}, B_{34}, B_{45}\}, \end{aligned} \tag{1}$$

$$\begin{aligned} S &= \{S_2, S_1, S_3, S_4, S_5\},\\ B &= \{B_{12}, *, B_{13}, B_{34}, B_{45}\}, &&\text{in}: B_{13}, &&\text{out}: B_{23} \end{aligned} \tag{2}$$

$$\begin{aligned} S &= \{S_2, S_3, S_1, S_4, S_5\},\\ B &= \{B_{13}, *, B_{14}, B_{23}, B_{45}\}, &&\text{in}: B_{23}, B_{14}, &&\text{out}: B_{12}, B_{34} \end{aligned} \tag{3}$$

$$\begin{aligned} S &= \{S_2, S_3, S_4, S_1, S_5\},\\ B &= \{B_{14}, *, B_{23}, B_{15}, B_{34}\}, &&\text{in}: B_{34}, B_{15}, &&\text{out}: B_{13}, B_{45} \end{aligned} \tag{4}$$

$$\begin{aligned} S &= \{S_3, S_2, S_4, S_1, S_5\},\\ B &= \{B_{14}, B_{23}, *, B_{15}, B_{24}\}, &&\text{in}: B_{24}, &&\text{out}: B_{34} \end{aligned} \tag{5}$$

$$\begin{aligned} S &= \{S_3, S_2, S_4, S_5, S_1\},\\ B &= \{B_{23}, B_{15}, *, B_{24}, B_{45}\}, &&\text{in}: B_{45}, &&\text{out}: B_{14} \end{aligned} \tag{6}$$

$$\begin{aligned} S &= \{S_3, S_4, S_2, S_5, S_1\},\\ B &= \{B_{15}, B_{24}, *, B_{25}, B_{34}\}, &&\text{in}: B_{34}, B_{25}, &&\text{out}: B_{23}, B_{45} \end{aligned} \tag{7}$$

$$\begin{aligned} S &= \{S_3, S_4, S_5, S_2, S_1\},\\ B &= \{B_{12}, B_{25}, *, B_{34}, B_{45}\}, &&\text{in}: B_{45}, B_{12}, &&\text{out}: B_{24}, B_{15} \end{aligned} \tag{8}$$

$$\begin{aligned} S &= \{S_4, S_3, S_5, S_2, S_1\},\\ B &= \{B_{12}, B_{25}, B_{34}, *, B_{35}\}, &&\text{in}: B_{35}, &&\text{out}: B_{45} \end{aligned} \tag{9}$$

$$\begin{aligned} S &= \{S_4, S_5, S_3, S_2, S_1\},\\ B &= \{B_{12}, B_{23}, B_{35}, *, B_{45}\}, &&\text{in}: B_{45}, B_{23}, &&\text{out}: B_{34}, B_{25} \end{aligned} \tag{10}$$

$$\begin{aligned} S &= \{S_5, S_4, S_3, S_2, S_1\},\\ B &= \{B_{12}, B_{23}, B_{34}, B_{45}, *\}, &&\text{in}: B_{34}, &&\text{out}: B_{35} \end{aligned} \tag{11}$$

After an initial $\mathcal{O}(n \log n)$ sort to rank the sites, by using a doubly-linked list, in tandem with an array of the sites for direct access in their natural order, one can decrease the cost of each rank update on the sorted site list to $\mathcal{O}(1)$. Unfortunately, for the bisector list, we can show that for every bisector crossed, we have to perform $\mathcal{O}(\log n)$ steps for n sites. We can use a balanced tree over the bisector order $\prec$ to achieve this.

The existence of a natural order between the sites in one dimension significantly improves the runtime behavior of the basic algorithm stated above. We will see that the moving query point will take $\mathcal{O}(n)$ steps in the two-dimensional case to locate the next bisector to be intersected.

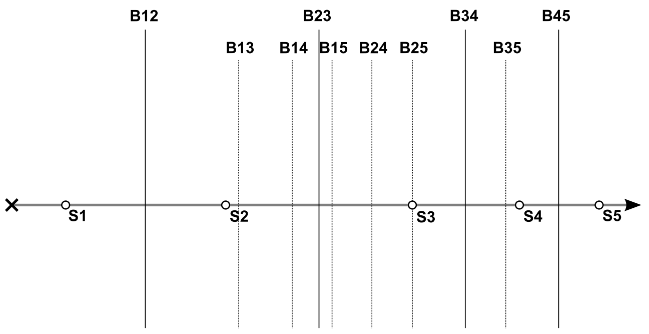

Fig. 1. A straight line, with five sites and a moving query point (marked with a cross). The long bisectors form the original bisector list when the query point is on the left of the site S_1. The shorter bisectors are the new bisectors that are introduced when the moving query point crosses an existing bisector.

This simple one-dimensional version of our algorithm shows: first, we only need to make incremental updates to keep our site ranks at all times; second, the updates are only required when the moving query point crosses a bisector, which avoids unnecessary query resubmissions at any other time.

4 Multi-dimensional Queries

For higher dimensional spaces, the main idea of the base algorithm does not change. In this section, we present the algorithm in the two-dimensional space but the approach can be applied to higher dimensional spaces. The main difference between the one-dimensional case and the higher dimensional case is the lack of a natural order for an arbitrary set of lines, i.e., the bisectors in our case (or hyper-planes for higher dimensions).

As in the one-dimensional case, we maintain for n sites $n-1$ bisectors in two dimensions. Using these bisectors, we define a region where the ranks of the sites do not change. In fact, this region maps to a cell in the *ordered order-k Voronoi diagram* [11] of these n sites where $k = n$. We only keep track of one cell at any time and only $n-1$ bisectors, and thus, we do not require the precomputation of this Voronoi diagram. As long as the query point stays in this cell, the ranks of the sites do not change, which renders a query resubmission redundant. Once the query point crosses a bisector, a rank update is required.

Crossing a bisector implies that the order for the two corresponding sites of this bisector needs to be altered. As in one dimension, we alter the rank of these sites first. As a side effect, we also have to update the $n-1$ bisectors. Similar to the one-dimensional case only a small subset of bisectors needs to be updated when a rank update occurs. The updated bisector list then defines a new ordered order-k Voronoi cell ($k = n$), which we can use to detect the next bisector crossing.

We first introduce the basic mathematical concepts for our algorithm and state the relevance of our work to ordered order-k Voronoi diagrams. We then present the final algorithm and its runtime complexity.

4.1 Rank Maintenance Using Regions

At the heart of our algorithm lies the observation that we only need $n - 1$ neighboring bisectors for the n sites in order to determine the next rank update. We prove this observation in this subsection.

Three straight lines that are not pairwise parallel, generally have three distinct intersection points. Bisectors, however, have an important property that does not hold for *general line arrangements*: *concurrency*. Three lines are *concurrent* if they have a common intersection point P. Because two straight lines can only intersect in at most one point, the point P is the only point in which the three bisectors intersect. Given two points Q and R, we use the notation $\overline{QR}$ for the segment connecting those two points. The following theorem states that there are always triplets of bisectors for sites in *general position* (i.e., no three points are collinear) that are concurrent.

Theorem 1. *Let S_i, S_j, and S_k be sites in general position and B_{ij}, B_{jk}, B_{ik} their bisectors. If P is an intersection point of B_{ij} and B_{jk}, i.e., $B_{ij} \cap B_{jk} = \{P\}$, then B_{ik} also intersects B_{ij} and B_{jk} in P, i.e., $P \in B_{ik}$.*

Proof. Let P be an intersection of B_{ij} and B_{jk} (e.g., as in Figure 2). Because B_{ij} is a bisector, it follows $|\overline{S_iP_1}| = |\overline{S_jP_1}|$. Applying Pythagoras' Theorem we get $|\overline{S_iP}| = |\overline{S_jP}|$. Similarly, we obtain $|\overline{S_jP}| = |\overline{S_kP}|$ due to $|\overline{S_jP_2}| = |\overline{S_kP_2}|$. Hence, we get $|\overline{S_iP}| = |\overline{S_kP}|$, which means that $P \in B_{ik}$.

Therefore, the set of all bisectors does not constitute a *simple arrangement* [12] of lines (a line arrangement is simple if there are no concurrent intersection points). Theorem 1 allows us to introduce a precedence relation on bisectors. In order to motivate the general definition, we will first introduce the precedence relation for two bisectors.

Let $H(S_i, S_j) =_{\text{def}} H_{ij} =_{\text{def}} \{P \,|\, d(P, S_i) \leq d(P, S_j)\}$ denote the *closed half plane* that includes S_i but not S_j. This half plane is also called the *dominance region* of S_i over S_j. The intersection of two half planes is a *closed sector*. We obtain from Theorem 1 that three bisectors generate exactly six sectors (see Figure 2). A segment that connects a point in a sector with a point outside of this sector always first intersects its bounding bisectors before it intersects the third bisector. We record this observation in the following lemma. We use the notation $\beta(P, Q, R)$ if the point Q is between the points P and R, i.e.:

$$\beta(P, Q, R) \Leftrightarrow_{\text{def}} \exists \lambda \in [0, 1] : Q = \lambda P + (1 - \lambda) R.$$

It is important to note that this notation also captures the case where concurrent intersection points occur.

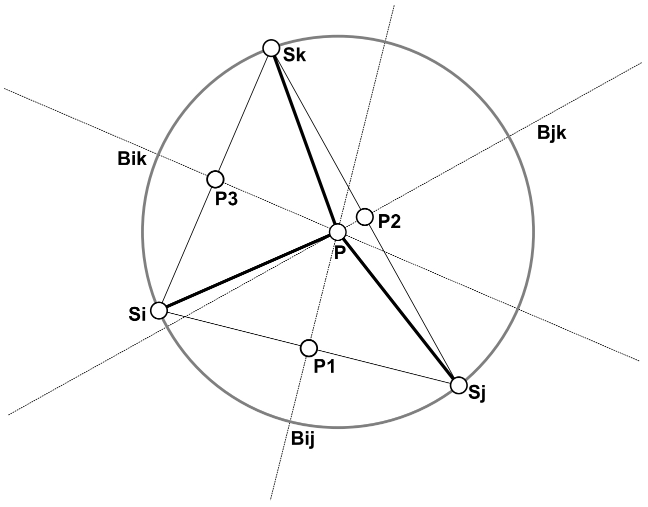

Fig. 2. Three bisectors for three sites intersecting at one point

Lemma 1. *Let S_i, S_j, and S_k be sites in general position, and B_{ij}, B_{jk}, and B_{ik} be their bisectors, then:*

$$\begin{aligned} &Q \in H_{ij} \cap H_{jk} : \\ &\quad \forall Q', R\, [R \in \overline{QQ'} \wedge R \in B_{ik} \Rightarrow \\ &\qquad \exists R'\, [\beta(Q, R', R)] \wedge (R' \in B_{ij} \vee R' \in B_{jk})]. \end{aligned}$$

On the basis of Lemma 1, we can formally define a precedence relation for bisectors:

Definition 1. *Let S_{i_1}, S_{i_2}, $\ldots$, S_{i_k} be k sites in general position, $\mathcal{B}$ be the set $\{B_{i_1 i_2}, B_{i_2 i_3}, \ldots, B_{i_{k-1} i_k}\}$ of the $k-1$ corresponding bisectors, and B_{lm} another bisector not in $\mathcal{B}$. We say that the bisector set $\mathcal{B}$* precedes *the bisector B_{lm} with respect to an arbitrary set $\mathcal{A}$, written as $\mathcal{B} \preccurlyeq_{\mathcal{A}} B_{lm}$, if and only if:*

$$\forall Q \in \mathcal{A} : \forall Q', R \left[R \in \overline{QQ'} \wedge R \in B_{lm} \Rightarrow \exists R'\, \beta(Q, R', R) \wedge \left(\bigvee_{j=1}^{k-1} R' \in B_{i_j i_{j+1}} \right) \right].$$

An immediate consequence of the definition of the precedence relation is the following corollary.

Corollary 1. *For every bisector B_{lm} holds:*

$$\mathcal{B} \preccurlyeq_{\mathcal{A}} B_{lm} \wedge \mathcal{B}' \preccurlyeq_{\mathcal{A}'} B_{lm} \Rightarrow \mathcal{B} \cup \mathcal{B}' \preccurlyeq_{\mathcal{A} \cap \mathcal{A}'} B_{lm}.$$

Proof. Since $\mathcal{A} \cap \mathcal{A}' \subset \mathcal{A}, \mathcal{A}'$ and $\mathcal{B}, \mathcal{B}' \subset \mathcal{B} \cup \mathcal{B}'$, the corollary follows from the definition of the precedence relation (if the disjunction is true for the bisectors in the set $\mathcal{B}$ and $\mathcal{B}'$ then it is true a fortiori for $\mathcal{B} \cup \mathcal{B}'$).

Our goal is to rank the sites for a moving query point at all times. We previously stated that two sites change their rank for a query point if the query point crosses their bisector. On the other hand, as long as a query point is in one half plane, or more generally in the intersection of a set of half planes, the ranks of the sites do not change. This motivates the concept of a *fixed-rank* region. A fixed-rank region is related to the ordered order-k Voronoi cell concept [11]. The fixed-rank region $H^{i_1 i_2 \cdots i_k}$ of order k is the set of all points such that the sites $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ are ranked by their distance[1]. More formally:

Definition 2

$$H^{i_1 i_2 \cdots i_k} =_{\text{def}} \bigcap_{j=1}^{k-1} H_{i_j i_{j+1}}$$

If all indices are consecutive then we use the following notation:

$$H^{l,m} =_{\text{def}} \bigcap_{j=l}^{m-1} H_{jj+1}.$$

The triangle in Figure 3 is the fixed-rank region $H^{1,4}$. The figure illustrates that the set of bisectors B_{12}, B_{23}, and B_{34} precede the bisectors B_{14} and B_{24} with respect to the gray triangle. Note that the bisector B_{24} has to be concurrent with the bisectors B_{23} and B_{34} according to Theorem 1 (which means that B_{24} intersects the boundary of the gray triangle at a single point). In addition, using B_{45} will introduce another rank for the site 5, i.e., $H^{1,5}$.

Using the precedence definition, we can rewrite Lemma 1 as $\{B_{ij}, B_{jk}\} \preccurlyeq_{H^{ijk}} B_{ik}$. If a set of bisectors $\mathcal{B}$ precedes another bisector B_{lm}, we also say that B_{lm} succeeds $\mathcal{B}$. The next theorem states that every bisector B_{lm} succeeds a set of neighboring bisectors with respect to the set $H^{l,m}$.

Theorem 2. *For every bisector B_{lm} holds:*

$$\bigcup_{i=l}^{m-1} \{B_{ii+1}\} \preccurlyeq_{H^{l,m}} B_{lm}.$$

Proof. We prove this theorem via induction over $k = m - l$. If $k = 1$, i.e., $m = l + 1$, then $H^{l,l+1} = H_{ll+1}$, and $B_{ll+1} \preccurlyeq_{H^{l,l+1}} B_{ll+1}$ is trivially true. Assume now the theorem holds for $k-1$, i.e., $k-1 = m-l-1$, and $m > l+1$. We can apply Lemma 1 and get:

$$\{B_{ll+1}, B_{l+1m}\} \preccurlyeq_{H^{l,m}} B_{lm}.$$

According to the induction assumption the theorem is true for $k - 1$ so that we can apply it to B_{l+1m} and obtain:

$$\bigcup_{i=l+1}^{m-1} \{B_{ii+1}\} \preccurlyeq_{H^{l+1,m}} B_{l+1m}.$$

[1] It is also possible to define a partial-fixed-rank region, i.e., a more general concept where only a subset of the ranks are relevant, but we omit the details for this concept here since they are not relevant to our discussion.

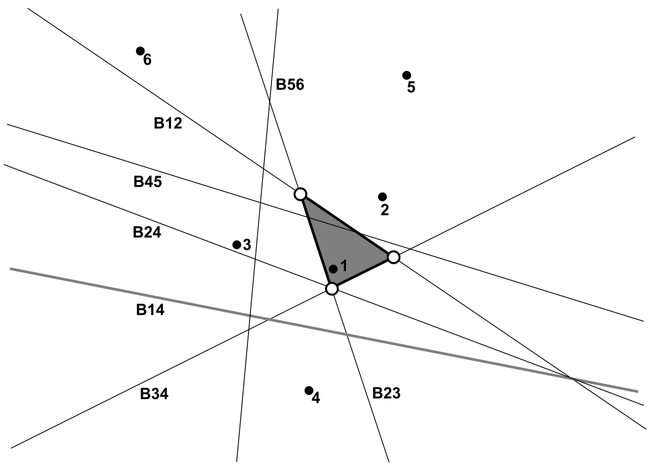

Fig. 3. A set of sites and some bisectors and a fixed-rank region. The ranks of the sites are used for naming the sites and the bisectors. The moving query point is initially located in the grey triangle.

Due to Corollary 1 and $H^{l,m} \subset H^{l+1,m}$ and two precedence relations above, the theorem follows.

Since $H^{1,n} \subset H^{l,m}$, the following corollary is an immediate consequence of Corollary 1 and Theorem 2.

Corollary 2. *For every bisector B_{lm} holds:*

$$\bigcup_{i=1}^{n-1} \{B_{ii+1}\} \preccurlyeq_{H^{1,n}} B_{lm}.$$

4.2 Incremental Rank Maintenance

According to Corollary 2, the set $\mathcal{B}$ of the $n-1$ bisectors $B_{ii+1}, i = 1..n-1$, precedes all other bisectors. Therefore, it is sufficient that the algorithm only keeps a list of the bisectors in $\mathcal{B}$. Our approach is to partition the set $\mathcal{B}$ of bisectors and keep two sub-lists of them: (1) a minimal set $\mathcal{M}$ of those k bisectors that enclose a query point (this list is cyclicly sorted), and (2) an unsorted list $\mathcal{R}$ of the remaining $n-k-1$ bisectors. This partition improves the performance of the spatial tests on the fixed-rank region condition, i.e., whether or not the moving query point has exited the current region. Other improvements, such as keeping a simple cache for the bisector toward which the motion of the query point is currently directed, can also be used to improve the complexity of the tests but are not the focus of this paper.

As it was the case in one dimension, the algorithm keeps track of the ranking of the sites in a sorted list, i.e., $\mathcal{S}$. $\mathcal{S}$ sorts the sites with respect to their distance. Once the query point crosses one of the bisectors of $\mathcal{M}$, the two corresponding sites change their ranks in $\mathcal{S}$, i.e., the site further away is now closer and vice versa. Hence, the $\mathcal{S}$ list is updated. Updating this list implies there are some bisectors that are now redundant in our lists. These have to be deleted. Then, new bisectors need to be inserted. These steps are again similar to the one-dimensional case. As we alter the ranks of only two sites, and we only have to maintain the neighboring bisectors $\mathcal{B}$, it follows that the number of bisectors that need to be deleted is at most two. Similarly, the number of bisectors that need to be added is also at most two.

Assume $\mathcal{S} = \{S_{i_1}, S_{i_2}, \ldots, S_{i_n}\}$, then a rank update consists of the following two events:

1. If the query point crosses the bisector $B_{i_1 i_2}$ ($B_{i_{n-1} i_n}$), then we have to delete the bisector $B_{i_2 i_3}$ ($B_{i_{n-2} i_{n-1}}$) and insert the bisector $B_{i_1 i_3}$ ($B_{i_{n-2} i_n}$). If the query point crosses any other bisector, for example $B_{i_j i_{j+1}}$, we have to delete two bisectors, $B_{i_{j-1} i_j}$ and $B_{i_{j+1} i_{j+2}}$, and then insert the two new bisectors $B_{i_{j-1} i_{j+1}}$ and $B_{i_j i_{j+2}}$.
2. Finally, we have to check if any of the bisectors in $\mathcal{R}$ are now in $\mathcal{M}$. This is required as the new fixed-rank region could now be defined by the inclusion of some of the bisectors that were not in the previous version of the list $\mathcal{M}$.

In the case that the query point crosses two (or in general m) bisectors at the same time, i.e., the the crossing point is the intersection of at least two bisectors, then we have to update the ranks and lists simultaneously. The algorithm, however, will work without any modifications, if each bisector crossing is treated independently. The aggregated update can be decomposed into m separate updates that can be treated independently of each other and without any modifications required from our algorithm. Obviously, for scenarios in which many bisectors are crossed simultaneously, more efficient methods, which update multiple bisectors at the same time, could be introduced. We do not focus on this improvement in this paper but leave it for future work.

Finally, we illustrate the algorithm in an example below (Figure 4). We show in the example how the sets $\mathcal{S}$, $\mathcal{M}$, and $\mathcal{R}$ have to updated in the illustrated configuration.

$$
\begin{aligned}
S &= \{S_1, S_2, S_3, S_4, S_5, S_6\}, & & \\
\mathcal{M} &= \{B_{23}, B_{34}, B_{12}, B_{45}\}, & \mathcal{R} &= \{B_{56}\}, \\
S &= \{S_2, S_1, S_3, S_4, S_5, S_6\}, & &\text{in} : B_{13},\ \text{out} : B_{23} \\
\mathcal{M} &= \{B_{12}, B_{34}, B_{45}\}, & \mathcal{R} &= \{B_{13}, B_{56}\}, \\
S &= \{S_2, S_1, S_4, S_3, S_5, S_6\}, & &\text{in} : B_{14}, B_{35},\ \text{out} : B_{13}, B_{45} \\
\mathcal{M} &= \{B_{34}, B_{12}, B_{14}, B_{35}\}, & \mathcal{R} &= \{B_{56}\}, \\
S &= \{S_2, S_4, S_1, S_3, S_5, S_6\}, & &\text{in} : B_{13}, B_{24},\ \text{out} : B_{12}, B_{34} \\
\mathcal{M} &= \{B_{14}, B_{24}, B_{56}, B_{35}\}, & \mathcal{R} &= \{B_{13}\}, \\
S &= \{S_4, S_2, S_1, S_3, S_5, S_6\}, & &\text{in} : B_{12},\ \text{out} : B_{14} \\
\mathcal{M} &= \{B_{24}, B_{12}, B_{13}, B_{56}\}, & \mathcal{R} &= \{B_{35}\}, \\
S &= \{S_4, S_2, S_1, S_3, S_6, S_5\}, & &\text{in} : B_{36},\ \text{out} : B_{35} \\
\mathcal{M} &= \{B_{56}, B_{13}, B_{36}, B_{24}\}, & \mathcal{R} &= \{B_{12}\}
\end{aligned}
$$

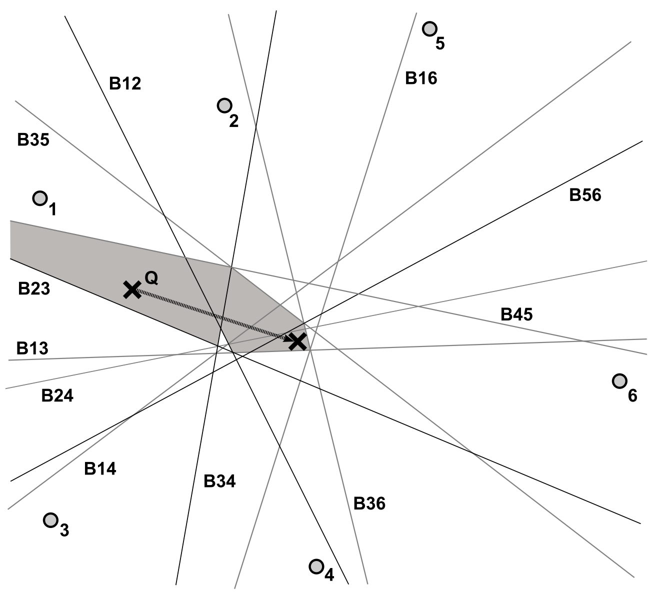

Fig. 4. A section of the trajectory of a moving query point. Sites are labeled with their ranks (i.e., when the query point was at its starting position). The grey areas are the fixed rank regions.

4.3 Complexity

As in the one-dimensional case, the initial ranking process can be computed in $\mathcal{O}(n \log n)$ time for n sites. For each crossed bisector we need to make an $\mathcal{O}(n)$ pass on the bisector lists to keep them up-to-date, because, in contrast to the one-dimensional case, there is no natural order of bisectors in two or higher dimensions. Thus, we make a full pass over the $\mathcal{R}$ to construct the new $\mathcal{M}$. In Section 5, we will see that given a direction of travel (an optimistic approach that assumes that this direction will be followed by the moving query point for a long time period), we can reduce the update complexity significantly. On the other hand, for trajectories that are arbitrary polygonal curves (note that in any case we do not have prior knowledge about the trajectory itself), it is an open question whether the bisectors can be maintained in $\mathcal{O}(\log n)$ time. This is a future research direction: the neighboring bisectors do not necessarily form an arbitrary set of lines, but may involve relationships similar to the ones presented previously in this paper.

We now show that the worst case complexity of the algorithm is $\mathcal{O}(n^3)$ for a movement of the query point that crosses a field of sites in its entirety.

Remark 1. If n sites are in general position and a query point Q moves on an arbitrary polygonal curve, then each line segment s of the polygonal curve can intersect up to $n(n-1)/2$ bisectors. The total number of updates on that line segment for our algorithm is then $\mathcal{O}(n^3)$.

Proof. Note that two sites can change their rank with respect to Q if and only if Q crosses their corresponding bisector. Each bisector can be written as a two-subset of the n sites. Thus, there are at most $\binom{n}{2} = n(n-1)/2$ bisectors and each bisector corresponds to exactly one rank update. Each rank update is $\mathcal{O}(n)$, because the list $\mathcal{R}$ maintained in the algorithm is an unsorted list. Therefore, the total number of updates is at most $\mathcal{O}(n^3)$. This bound is tight (see Figure 5 for an example where $n = 6$). If all sites lie on a half-circle above the diameter of the circle aligned with the x-axis and Q moves along a line segment that is parallel to the x-axis and the line segment is above the intersection point of all bisectors and below the intersection points of B_{12} and B_{n-1n} with the circle, then Q has to intersect all the bisectors during its travel over the segment. It is important to note that along the path of the moving query point the bisectors are always introduced in front.

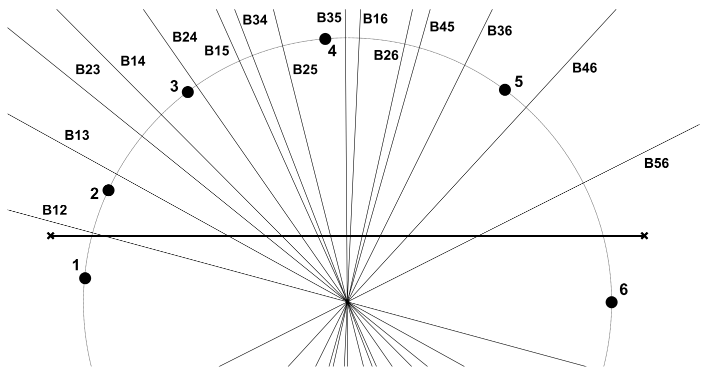

Fig. 5. All sites lie on a circle. A trajectory of a query point that will cross all the bisectors.

Note that the configuration in Figure 5 is not the only configuration that has the worst case complexity of $\mathcal{O}(n^3)$. Even if all sites do not lie on a circle but, for example, have slightly different positions such that the lines are not intersecting in a single point, the horizontal line in Figure 5 will still intersect the $\mathcal{O}(n^2)$ bisectors.

A simple solution to the rank maintenance problem that one can easily introduce is the precomputation of all the bisectors for n sites. In this case, for $\mathcal{O}(n^2)$ bisectors, we have to maintain our list $\mathcal{M}$ to detect the next update point. Using a similar worst case analysis, we can see that this leads to an $\mathcal{O}(n^4)$ cost, which is more expensive than our approach. This simple solution also has the disadvantage of precomputing all the bisectors even if the moving query point may lead to only a few updates in $\mathcal{S}$. Hence, this approach always performs worse than our algorithm.

Another simple solution would be to precompute the ordered order-k Voronoi diagram of n sites for $k = n$. This diagram will have $\mathcal{O}(n^4)$ faces in the worst case. The following example shows that the total number of faces is indeed $\mathcal{O}(n^4)$.

Remark 2. If $n/2$ (assuming n is even) sites are arranged parallel to the x-axis with increasing distances and the remaining $n/2$ sites are placed parallel to the y-axis (again with increasing distances), then there are

$$\underbrace{\frac{n}{2}-1}_{\text{for } S_1}+\underbrace{\frac{n}{2}-2}_{\text{for } S_2}+\ldots+\underbrace{1}_{\text{for } S_{n-1}}$$

bisectors along each axis. This generates $\mathcal{O}(n^4)$ faces.

One can link the neighboring cells of this diagram to form a data structure and enable rank maintenance by visiting sites on a trajectory of a moving query point. This approach, given a similar worst case scenario as above, will again perform poorly in comparison to our work due to the obvious precomputation overhead. In addition, it ignores the cases where only a few rank updates may be required on a trajectory, e.g., due to a short trajectory. On the other hand, for long trajectories that may include circular patterns, the precomputation of the ordered order-k Voronoi diagram may be quite beneficial. While our algorithm continues to invest $\mathcal{O}(n)$ time for maintaining the list $\mathcal{M}$ for every bisector crossing, the precomputation will only lead to an $\mathcal{O}(1)$ behavior for each update.

5 Multi-dimensional Queries with Directions

A common travel pattern that occurs in many real-life scenarios is a trajectory consisting of long straight line segments (e.g., Figure 6). In this case, an optimistic heuristic that gives a better performance than our original algorithm is to use balanced trees to store the list $\mathcal{R}$. This will reduce the cost of updating $\mathcal{M}$ from $\mathcal{R}$ to $\mathcal{O}(\log n)$ for n sites. Hence, the overall worst case behavior of the algorithm reduces to $\mathcal{O}(n^2 \log n)$ along a line segment.

This approach assumes that the number of line segments of the trajectory is much smaller than the number of bisectors to be crossed.

Given a direction we sort the n sites in $\mathcal{O}(n \log n)$ time. The list $\mathcal{R}$ of bisectors is stored in a balanced tree rather than an unsorted list. The cost of the creation of the balanced tree is also $\mathcal{O}(n \log n)$. The intersection points of the bisectors of $\mathcal{R}$ with the line segment can be used to store and find each bisector in this tree. Hence, bisector insertion and deletion operations can now be done in $\mathcal{O}(\log n)$ time as well as finding the bisector that the moving query point can intersect next after $\mathcal{M}$ is invalidated with a bisector crossing. Thus, redefinition of $\mathcal{M}$ can also be done in $\mathcal{O}(\log n)$ time.

In the worst case a moving query point crosses all $\mathcal{O}(n^2)$ bisectors along the straight line segment leading to computational complexity of $\mathcal{O}(n^2 \log n)$. Once the moving query point changes its direction, the tree that has been constructed for the old travel direction becomes invalid. The tree reconstruction requires an additional $\mathcal{O}(n \log n)$ cost for each direction change.

One marginal scenario that can occur is when many bisectors intersect at one point and a line segment from the trajectory of the moving query point also intersects with this point. In this case, many lines may fall under the same leaf node of the tree structure. For updating $\mathcal{M}$, this does not have an effect. Any of the rank updates can occur in any order. On the other hand, insertions and deletions to the tree may require an additional data structure. As we cannot differentiate these bisectors with their intersection points, we have to use their designations (e.g., k and l for B_{kl}). However, this does not change

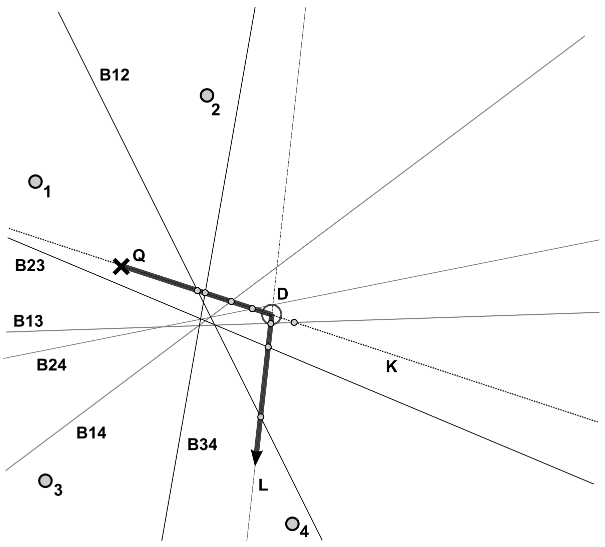

Fig. 6. A moving query point Q, four sites, and all of the bisectors for this setting. The query point moves from the starting point illustrated by a cross on K and then changes its direction at point D and moves on L. The order of the bisectors to be crossed does not change given a direction.

the space or time complexity of the algorithm as designations can also be used with a balanced tree structure.

6 General Rank Maintenance

We have introduced in Definition 2 the concept of a fixed-rank region. We developed two algorithms for incrementally maintaining the ranks of all sites. Our algorithms were based on the assumption that all sites are ranked consecutively. In this section we relax this assumption.

To motivate why non-consecutive ranks are important, we consider the following scenario: instead of a single agent ranking n sites, multiple agents are co-located and distribute their load by ranking the n sites jointly. Each agent only keeps track of a certain number of ranks. For example, if $\mathcal{S}$ consists of 9 sites, one agent keeps track of the first 3 ranks, a second agent the next 3 ranks and a third agent the last 3 ranks. Alternatively, the first agent could track the ranks 1, 5, 9, the second agent the ranks 2, 6, 8, and the last agent the ranks 3, 4, 7.

First, we present an algorithm for tracking a single rank k. The algorithm determines, for a given k, which site is of rank k for a query point at any position along its trajectory. This problem is related to the kth-nearest neighbor Voronoi diagram concept. A kth-nearest neighbor Voronoi region consists of all points for a given site S_i such that S_i is always the k-th closest site.

The algorithm for maintaining a single rank k works as follows (initially assume all sites are already ranked according to their distance). As the first step, the algorithm selects the k-th site, called S_{1_k}. Then, at every step i, the algorithm keeps, for S_{i_k}, the list of all bisectors of S_{i_k}, because the rank of S_{i_k} can only change when one of those bisectors is crossed. More precisely, at each step we keep an unsorted list $\mathcal{B}_i = \bigcup_{j=1, j\neq k}^{n} \{B_{i_j i_k}\}$, and as long as the query point does not intersect any of those bisectors of $\mathcal{B}_i$, the rank of S_{i_k} does not change. Once the moving query point intersects one of the bisectors, this bisector will determine the new site S_{i+1_k} and $\mathcal{B}_{i+1}$ will be computed. The cost to keep $\mathcal{B}_i$ at every step is $\mathcal{O}(n)$. If m sites are visited as being the k-th neighbor, the total complexity is $\mathcal{O}(mn)$.

This algorithm can be easily extended to keep track of l ranks. In the first step, the algorithm selects the l sites according to their rank and stores them as an l tuple $(S_{1_{k_1}}, \ldots, S_{1_{k_l}})$. As in the algorithm for a single rank, we keep l lists $\mathcal{B}_{i_1}, \ldots, \mathcal{B}_{i_l}$ for the l ranks. The total cost of the algorithm is $\mathcal{O}(ln)$ at each step, which leads to a total complexity of $\mathcal{O}(mln)$ for m rank updates. If j agents keep track of n ranks and the ranks form a partition of the n ranks so that $\sum_{i=1}^{j} l_i = n$, then the total complexity for all j agents is $\sum_{i=1}^{j} O(ml_i n) = O(mn^2)$.

7 Conclusions and Future Work

In this paper, we introduced the continuous rank maintenance problem for moving query points. In contrast to previous work, the CNN queries investigated in this work are order sensitive. For a moving query point, we introduced algorithms to track the ranks of sites without any prior knowledge about the trajectory of the query point. We defined a fixed-rank region that determines the set of all those points for which the ranks of the sites do not change. This reduces the need for any query reprocessing and communication because in a fixed-rank region no updates are required. When one of the bisectors of the fixed-rank region is crossed, the ranks of all the sites are updated incrementally. We showed that our algorithms only need to keep track of $n - 1$ bisectors, from a set of $n(n-1)/2$ bisectors, for all n sites. We introduced a more general concept for rank maintenance that allows us to track subsets of ranks. We developed algorithms that are able to track a single rank k as well as a subset of ranks, possibly by multiple agents. This enables a distributed approach for rank maintenance.

We introduced several extensions to our base algorithm. In the general case, in which a query point moves along an arbitrary polygonal curve, we showed that for each bisector crossing, we need an $\mathcal{O}(n)$ pass over the bisectors to keep the fixed-rank region up-to-date. In the special case, in which the trajectory consists of long straight line segments, we proved that only $\mathcal{O}(\log n)$ steps are required for updating the ranks of neighbors.

In our future work, we will investigate methods that reduce the number of updates for a moving query point that travels along an arbitrary polygonal curve in less than $\mathcal{O}(n)$ time. We plan to take advantage of the insight that the $n-1$ neighboring bisectors do not form an arbitrary arrangement of lines but may exhibit spatial relationships similar to the ones presented with this paper. Second, we will update our algorithms for situations where we can assume minimal prior knowledge about the trajectory of the query point. For example, a speed limitation may imply that the point cannot reach some segments of the bisectors given a period of time. Third, we will evaluate those cases where a query point intersects concurrent bisectors. In the long term, we aim to adapt the algoritms to cope with dynamic situations in which the sites as well as the query point are in motion.

References

1. F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.
2. R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Proceedings of the IEEE IDEAS*, pages 44–53, Edmonton, Canada, July 2002.
3. H. D. Chon, D. Agrawal, and A. El Abbadi. Range and KNN query processing for moving objects in grid model. *Mobile Networks and Applications*, 8(4):401–412, August 2003.
4. G. Hjaltason and H. Samet. Ranking in spatial databases. In *Proceedings of the SSD*, pages 83–95, Portland, ME, August 1995.
5. G. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999.
6. G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, December 2003.
7. H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *Proceedings of the ACM SIGMOD*, pages 479–490, Baltimore, MD, June 2005.
8. G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *Proceedings of the VLDB*, pages 512–523, Berlin, Germany, September 2003.
9. M. F. Mokbel. Continuous query processing in spatio-temporal databases. In *Proceedings of the EDBT (Workshops)*, pages 100–111, Heraklion, Greece, March 2004.
10. K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of the ACM SIGMOD*, pages 634–645, Baltimore, MD, June 2005.
11. A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley, Chichester, NY, second edition, 2000.
12. J. O'Rourke. *Computational Geometry in C*. Cambridge University, Cambridge, UK, 1994.
13. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD*, pages 71–79, San Jose, CA, May 1995.
14. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of ACM SIGMOD*, pages 331–342, Dallas, TX, May 2000.
15. H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.
16. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
17. H. Samet. *Foundations of Multidimensional Data Structures*. Morgan Kaufmann, San Francisco, CA, 2006.
18. Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *Proceedings of the SSTD*, pages 79–96, Redondo Beach, CA, July 2001.
19. Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *Proceedings of the ACM SIGMOD*, pages 334–345, Madison, WI, June 2002.

20. Y. Tao and D. Papadias. Spatial queries in dynamic environments. *ACM Transactions on Database Systems*, 28(2):101–139, June 2003.
21. Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proceedings of the VLDB*, pages 287–298, Hong Kong, China, August 2002.
22. X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Proceedings of the IEEE ICDE*, pages 643–654, Tokyo, Japan, April 2005.
23. J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. Lee. Location-based spatial queries. In *Proceeding of the ACM SIGMOD*, pages 443–453, San Diego, CA, June 2003.
24. B. Zheng and D. L. Lee. Semantic caching in location-dependent query processing. In *Proceedings of the SSTD*, pages 97–116, Redondo Beach, CA, July 2001.