# The V*-Diagram: A Query-Dependent Approach to Moving KNN Queries

Sarana Nutanong[†‡] Rui Zhang[†] Egemen Tanin[†‡] Lars Kulik[†‡]
{sarana,rui,egemen,lars}@csse.unimelb.edu.au

[†]Department of Computer Science and Software Engineering
University of Melbourne
Victoria, Australia

[‡]NICTA Victoria Laboratory
Australia

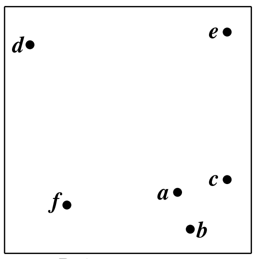
## ABSTRACT

The moving $k$ *nearest neighbor* (M$k$NN) query finds the $k$ nearest neighbors of a moving query point continuously. The high potential of reducing the query processing cost as well as the large spectrum of associated applications have attracted considerable attention to this query type from the database community. This paper presents an incremental safe-region-based technique for answering M$k$NN queries, called the *V*-Diagram*. In general, a safe region is a set of points where the query point can move without changing the query answer. Traditional safe-region approaches compute a safe region based on the data objects but independent of the query location. Our approach exploits the current knowledge of the query point and the search space in addition to the data objects. As a result, the V*-Diagram has much smaller IO and computation costs than existing methods. The experimental results show that the V*-Diagram outperforms the best existing technique by two orders of magnitude.

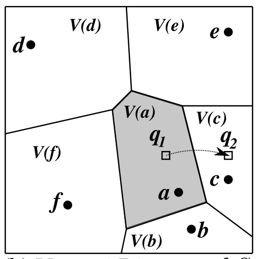
## 1. INTRODUCTION

Current location-based services provide accurate position information with a high degree of temporal precision. Consider the following two scenarios. A driver in a GPS-equipped car issues a continuous query to find the nearest gas station while driving in a city. A tourist uses a location-aware mobile device to issue a continuous query for the nearest restaurant while walking to a museum. The queries are sent to a server that processes the queries and returns the answers. In these scenarios, the server has to continuously maintain the answer set which may change depending on the location of the query point. These queries are *location-based continuous spatial queries* [23] and the scenarios above are typical examples of *moving $k$ nearest neighbor queries* (M$k$NN). A straightforward way to process a M$k$NN query is using a *sampling-based* method, which processes the M$k$NN query as a kNN query at sampled locations. This method does not provide answers between sampled locations. In order to provide an (almost) continuous answer to the query, a high sampling rate is required, which makes the method inefficient due to the frequent processing of kNN queries. The concept of the *safe region* provides a more effective way to achieve continuous answers to location-based spatial queries. In a safe-region-based method, an answer is returned with a safe region. As long as the query point stays in the safe region, the answer remains the same. When the query point moves out of the safe region, another answer with its associated region is returned. Therefore, a safe-region-based method always (that is, continuously) provides accurate answers without the need for sampling. This approach also requires much less frequent communication between the mobile client and the server.

A classic example of safe-region-based techniques is the *Voronoi Diagram* [14]. The Voronoi Diagram is a well known space decomposition determined by distances to a given discrete set of objects, typically, a set of points. Specifically, the Voronoi Diagram of a set of points $P = \{\boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_n}\}$ is defined as a set of cells where each cell $V(\boldsymbol{p_i})$ is a region of space that consists of all points of the data space that are closer to $\boldsymbol{p_i}$ than any other point in $P$. An example is given in Figure 1. Figure 1(a) is a set of points in a 2-dimensional (2D) space and Figure 1(b) is the Voronoi Diagram.



(a) Point set $S$, $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}\}$



(b) Voronoi Diagram of $S$

**Figure 1: The Voronoi Diagram**

Processing a 1NN query using the Voronoi Diagram involves: (i) locating which Voronoi cell the query point falls into; and (ii) identifying the associated object. In the above example, $\boldsymbol{q_1}$ falls in $V(\boldsymbol{a})$ (the grey region) and therefore $\boldsymbol{a}$ is the nearest neighbor of $\boldsymbol{q_1}$. The answer remains valid as long as the query point stays in $V(\boldsymbol{a})$. When the query point moves across the boundary of $V(\boldsymbol{a})$ to $V(\boldsymbol{c})$, $\boldsymbol{c}$ becomes the NN.

The Voronoi Diagram can be generalized to the $k^{th}$*-order Voronoi Diagram ($k$VD)*. In a $k$VD, each region is associated with the set of $k$ nearest neighbors, termed *$k$NN set* or *$k$ NNs*, rather than only the nearest one. The $k$VD can handle M$k$NN queries in the same manner as the basic first order VD handles 1NN queries.

Another useful generalization of the Voronoi Diagram is order sensitivity. The *ordered* $k$VD partitions the space into cells where each cell is associated to a particular ordering of the $k$NN set. The ordered $k$VD can be used to answer M$k$NN queries that require the ranking of the $k$ NNs by their distances to the query point.

The $k$VD has the following shortcomings:

1. **Expensive precomputation.** The $k$VD requires precomputing all the $k$VD cells and access to all the data points. Both computation and storage costs are high.
2. **No support for dynamically changing $k$ values.** The $k$VD can only accommodate $k$NN queries with a specific $k$ value; the ordered $k$VD can only accommodate $k$NN queries with $k$ values no larger than the order of the diagram. As a result, the technique is not suitable for situations where the value of $k$ is unknown in advance or can change dynamically.
3. **Inefficient update operations.** Many cells have to be recomputed for each insertion or deletion on the dataset.

The expensive precomputation is especially not justified if the query point is confined to a small region of the whole data space. For example, if a car is moving in a small part of a city, then it is unnecessary to compute the $k$VD for the entire city. Furthermore, one may require different $k$VDs for different needs. For example, a driver may need to find a gas station with a restroom facility while another driver needs one with a special type of fuel. Precomputing $k$VDs for all possible scenarios may be prohibitive.

In [23], Zhang et al. proposed an algorithm to locally compute a $k$VD cell, which mitigates the precomputation and update problems (shortcomings 1 and 3). However, the algorithm is still relatively expensive and does not address the problem of dynamically changing $k$ values (shortcoming 2).

In this paper, a technique called the *V*-Diagram* for M$k$NN queries is proposed. The V*-Diagram has the following key advantages:

1. It requires no precomputation.
2. It incrementally computes answers and therefore efficiently adapts to changes – such as insertions and deletions of objects, as well as, dynamically changing values of $k$.

The V*-Diagram is based on the safe-region concept, but differs from any previous technique for M$k$NN queries in the following aspect: previous safe-region-based techniques compute safe regions purely based on the data (for example, you can compute the $k$VD without referring to the query point); **the V*-Diagram computes safe regions based on not only the data objects, but also the query point and the current knowledge of the search space.** This is one of the main novelties of the technique. By doing so, both computation and data retrieval of the V*-Diagram are more economical than those of the other techniques.

The contributions of this paper are summarized as follows:

- We propose the V*-Diagram technique and the associated algorithm, called V*-$k$NN, to support efficient processing of M$k$NN queries.
- We show how the V*-Diagram technique can be applied to the domain of spatial networks.
- We perform an extensive experimental study with the results showing that the V*-Diagram outperforms the best existing technique [23] by two orders of magnitude.

The rest of the paper is organized as follows: Section 2 describes the problem setup. Related work is discussed in Section 3. We formulate the V*-Diagram in Section 4 and present the algorithm for M$k$NN queries based on the V*-Diagram in Section 5. In Section 6, we show how the V*-Diagram technique can be applied to the domain of spatial networks. A performance analysis of the V*-Diagram is given in Section 7. Section 8 presents experimental results and Section 9 concludes the paper.

## 2. PROBLEM SETUP

Let $D$ be a set of data objects (represented by points) in a $d$-dimensional space and $dist(\ .\ )$ be the distance function. The $k$ nearest neighbor ($k$NN) query is defined as follows: *given $D$ and a static query point $\boldsymbol{q}$, find a set $H$ that consists of $k$ objects from $D$ such that for any $\boldsymbol{p_1} \in H$ and any $\boldsymbol{p_2} \in D - H$, $dist(\boldsymbol{p_1}, \boldsymbol{q}) \leq dist(\boldsymbol{p_2}, \boldsymbol{q})$.*

The moving $k$ nearest neighbor query (M$k$NN) is defined as follows: *given $D$ and a moving query point $\boldsymbol{q}$, find the $k$ NNs of $\boldsymbol{q}$ for every position of $\boldsymbol{q}$.*

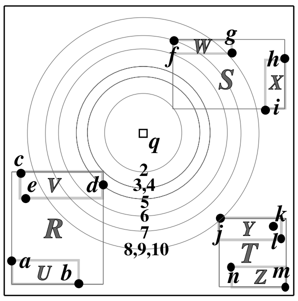
Due to the nature of location-based applications, M$k$NN queries are discussed in the context of two settings: (i) **Centralized processing paradigm**. Both the query issuer and the processor are on the same machine. Then the main performance measure is the query processing cost. (ii) **Client-server paradigm**. The query is issued by a client to a server through a wireless network (such as a mobile phone network). The performance measure involves both communication and query processing costs on the server side, and the former one is more important in delay-sensitive applications.

We assume an unknown trajectory which means that the location of $\boldsymbol{q}$ gets updated in a periodic manner. We also assume that no $k$VD is maintained but there is a generic spatial index such as the R-tree [4] built on the data objects, since it can be used for various query types and is efficient to maintain. This is also argued as a valid assumption in previous work [23].

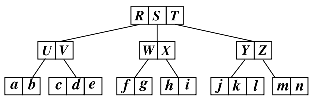
## 3. RELATED WORK

### 3.1 KNN algorithms

Many $k$NN search algorithms have been proposed based on spatial hierarchical structures. One of their common features is the application of the branch-and-bound strategy on the tree structure. The tree can be traversed in a depth-first (*DF-kNN*) [16] or a best-first (*BF-kNN*) [5] manner. BF-$k$NN can retrieve more nearest neighbors incrementally if $k$ increases.



(a)



| Step | Priority Queue *pq* | Retrieved NN |
|---|---|---|
| 1 | *<S,R,T>* | <> |
| 2 | *<R,W,T,X>* | <> |
| 3 | *<V,W,T,X,U>* | <> |
| 4 | *<d,W,T,X,c,e,U>* | <> |
| 5 | *<W,T,X,c,e,U>* | *<d>* |
| 6 | *<f,T,g,X,c,e,U>* | *<d>* |
| 7 | *<T,g,X,c,e,U>* | *<d,f>* |
| 8 | *<Y,g,X,c,e,U,Z>* | *<d,f>* |
| 9 | *<j,g,X,c,e,U,k,Z,l>* | *<d,f>* |
| 10 | *<g,X,c,e,U,k,Z,l>* | *<d,f,j>* |

(b)

**Figure 2: R-tree and BF-$k$NN**

Figure 2 shows an example R-tree and the BF-$k$NN algorithm. Figure 2(a) shows a set of points and how they can be grouped in an R-tree, where objects are indexed in a hierarchy of *minimum bounding rectangles* (MBRs). The corresponding R-tree is shown in the upper part of Figure 2(b). Starting from the root, BF-$k$NN traverses all the entries in increasing order of their *mindist*, where the *mindist* of an entry is the minimum distance between the entry and the query point $\boldsymbol{q}$. To do so, a priority queue is maintained to keep all the retrieved data points and *active* nodes. A node is active if its parent has been accessed but itself has not. The traversal stops if the first $k$ elements retrieved from the priority queue are all data points.

An example run of BF-$k$NN is shown in the lower part of Figure 2(b). At step 1, all entries in the root are inserted into the priority queue $pq$. Then the entries are dequeued and the corresponding nodes are retrieved in order. The first dequeued item is $S$ and its two child entries $X$ and $W$ are put back into $pq$. Then $R$ is dequeued. Nodes $U$ and $V$ are put in $pq$ and so on. At step 4, data point $\boldsymbol{d}$ is the head of $pq$. Data point $\boldsymbol{d}$ must be the first NN because all other entries has distances to $\boldsymbol{q}$ (which are bounded by $mindist$ of their nodes) larger than $dist(\boldsymbol{q}, \boldsymbol{d})$. $\boldsymbol{d}$ can be retrieved as the first NN. If another NN is needed, the process continues until another data point is the head of $pq$. At step 6, $\boldsymbol{f}$ is discovered as the second NN. By this means, an arbitrary number of NNs can be incrementally obtained. If the value of $k$ is fixed, an aggressive pruning can be performed on the nodes in $pq$ to reduce the queue size, though the page access cost cannot be further reduced due to the search-space optimality of the algorithm [6]. Other techniques such as iDistance [10] has superior performance than these hierarchical structure based algorithms in high dimensional space. However, it cannot retrieve $k$ NNs in an incremental manner.

## 3.2 Techniques for processing M$k$NN queries

We have discussed two approaches for M$k$NN queries in Section 1: one is sampling based and the other one is safe-region based.

**SR-$k$NN.** Song and Roussopoulos [18] introduced a method which will be referred to as *SR-kNN* in this paper. SR-$k$NN reduces the access costs in the sampling-based approach by retrieving redundant data entries and caching. It greatly reduces the cost of query reevaluation, but does not solve the problem of inaccurate answers between sampled locations. Thus, SR-$k$NN does not provide *truly continuous* answers.

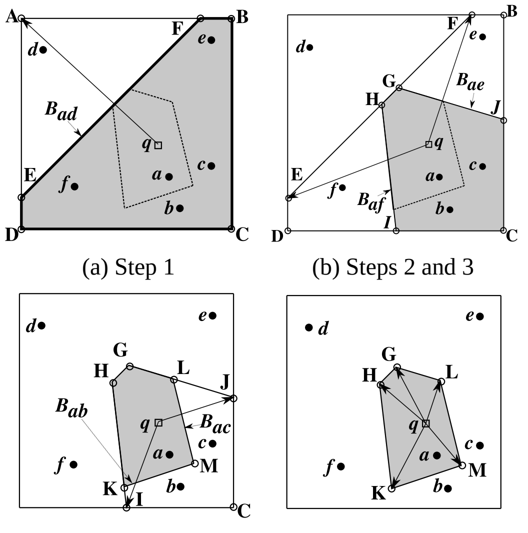
**The $k^{th}$-order Voronoi Diagram ($k$VD).** Safe-region based techniques produce continuous answers and reduce processing and communication costs. The Voronoi Diagram [14] is a classic example and can be used to process M1NN queries as described in Section 1. For M$k$NN queries, the $k$VD is used. However, the $k$VD has shortcomings described in Section 1.

**TP$k$NN and C$k$NN.** Tao and Papadias [19] proposed the *time-parameterized kNN* (TP$k$NN) query. Assuming a linear trajectory of the query point, a TP$k$NN query finds (i) the current $k$NN set, (ii) a position on the trajectory where the $k$NN set changes and (iii) the objects that cause the change. This is done by finding the earliest point on the trajectory that has a different $k$NN set from the current one. This point is also known as the *influence point* (or equivalently *influence time* when the speed is known), which can be considered as the boundary of a safe region.

Tao and Papadias [20] also considered the *Continuous kNN* (C$k$NN) query, which finds the $k$NN for every single point on a predefined linear trajectory. This is achieved by identifying all influence points on the trajectory. The main difference between C$k$NN and TP$k$NN is that C$k$NN obtains all the influence points on the trajectory but TP$k$NN finds only the first one. Both TP$k$NN and C$k$NN are limited to known linear trajectories.

**RIS-$k$NN.** Zhang et al. [23] proposed an algorithm called *Retrieve-Influence-Set kNN* (*RIS-kNN*) to locally compute $k$VD cells using a spatial index. RIS-$k$NN uses the *TPkNN query* [19] to find each edge of a $k$VD cell, 360 degrees around the query point.

Figure 3 shows how RIS-$k$NN discovers all edges of $V(\boldsymbol{a})$ from the example in Figure 1(a). Initially, $\boldsymbol{a}$ is found to be the NN of $\boldsymbol{q}$; the cell is initialized to the whole data space, that is, rectangle $\boldsymbol{ABCD}$. At step 1, a TPNN query is executed with the trajectory from $\boldsymbol{q}$ to the top left corner of the space ($\overrightarrow{\boldsymbol{qA}}$) and $\boldsymbol{d}$ is returned as the object that changes the NN result along $\overrightarrow{\boldsymbol{qA}}$. The perpendicular bisector of $\boldsymbol{a}$ and $\boldsymbol{d}$, $B_{\boldsymbol{ad}}$, contributes one edge to the cell. The cell is updated to the polygon $\boldsymbol{BCDEF}$. At steps 2 and 3, two TPNN queries are executed with the trajectories from $\boldsymbol{q}$ to the new corners ($\overrightarrow{\boldsymbol{qE}}$ and $\overrightarrow{\boldsymbol{qF}}$). Two new edges are found according to $B_{\boldsymbol{af}}$ and $B_{\boldsymbol{ae}}$. Since $k$VD cells for data points are convex polygons, this process continues until all corners of the cell have been checked



(a) Step 1 (b) Steps 2 and 3

(c) Steps 4 and 5 (d) Final steps

**Figure 3: Computing a Voronoi cell locally**

and they all have the same NN as $\boldsymbol{q}$.

RIS-$k$NN mitigates the precomputation problem (shortcoming 1) of the $k$VD because it only accesses local data, but this algorithm is still expensive because it performs multiple (on average 12 [23]) TP$k$NN queries, where each TP$k$NN query involves a costly tree traversal. RIS-$k$NN does not solve the problem of dynamically changing $k$ values (shortcoming 2); changing $k$ to a larger value incurs recalculation of the $k$VD cell. The computation of the previous $k$VD cell cannot be reused and hence this algorithm is not *incremental*.

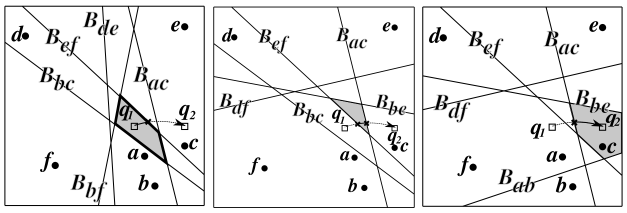
Due to the fact that only one $k$VD cell is maintained at a time, RIS-$k$NN handles dataset updates (insertions and deletions of objects) more efficiently than the traditional $k$VD technique. However, in a case where an update effects the $k$NN answers, the current $k$VD cell has to be recalculated.

**IRU.** Kulik and Tanin [13] introduced an algorithm called *incremental rank updates* (IRU) to compute regions where the ranking of all the objects (based on their distances) is the same. This is equivalent to computing the ordered $k$VD cell with $k = n$, where $n$ is the total number of objects. Rather than computing the whole $n$VD, IRU incrementally computes a neighboring $n$VD cell from the current cell. In [13], an $n$VD cell is termed a *fixed-rank region* (FRR) since for any point in the region, the ranking of all the objects based on their distances is fixed. Based on the observation that only rank-adjacent objects can swap their ranks[1], defining the FRR of $n$ objects requires at most $n - 1$ bisectors[2] of the $n - 1$ pairs of rank-adjacent objects. Continuous monitoring of the ranking of the objects is done by maintaining a rank-sorted list of objects and its corresponding list of bisectors of pairs of rank-adjacent objects (*rank-adjacent bisectors*). Each time a bisector is crossed by the query point, the ranks of the two corresponding objects are swapped and the list of rank-adjacent bisectors are updated.

An example is given in Figure 4, where the grey region is the current FRR that $\boldsymbol{q_1}$ is in. Let us assume that $\boldsymbol{q}$ is the location of a moving query point which starts at $\boldsymbol{q_1}$ and stops at $\boldsymbol{q_2}$. In Figure 4(a), $\boldsymbol{q}$ is at $\boldsymbol{q_1}$ and the ranking is initially $\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f}, \boldsymbol{e}, \boldsymbol{d} \rangle$ and the corre-

[1]In this paper, the rank of an object means the object's position in a list of objects sorted by their distances to some other object. We use "ranking" and "ordering" interchangeably due to the usage of both in the literature.

[2]The *bisector* of two objects $\boldsymbol{a}$ and $\boldsymbol{b}$ is the set of points where each point is equidistant to $\boldsymbol{a}$ and $\boldsymbol{b}$.



(a) Initial step (b) First update (c) Second update

**Figure 4: Incremental rank update**

sponding list of bisectors is $\langle\, B_{ac},\ B_{cb},\ B_{bf},\ B_{fe},\ B_{ed}\rangle$. Then $q$ crosses $B_{ef}$ in Figure 4(b). This causes $e$ and $f$ to swap their ranks. Therefore $B_{bf}$ and $B_{de}$ are replaced by $B_{be}$ and $B_{df}$, respectively. In Figure 4(c), $B_{ac}$ is crossed. This causes $a$ and $c$ to swap their ranks and $B_{bc}$ is replaced by $B_{ab}$. It is shown in [13] that only $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2)$ bisectors are maintained during the iterations in the IRU algorithm. However, IRU still accesses all data objects to obtain the sorted list and checks $n-1$ bisectors every time $q$ moves.

**Related spatial-network queries.** Several $k$NN techniques for static query points were proposed in [8, 9, 15, 17]. There are also M$k$NN techniques specific to the domain of spatial networks. Kolahdouzan et al. [12] proposed an algorithm that utilizes the *Voronoi Network Nearest Neighbor* (VN$^3$) [11]. Cho et al. [3] proposed a technique that issues static $k$NN queries at the intersection points on the query path.

**Related moving-object queries.** Hu et al. [7] proposed a safe-region-based technique for static window and $k$NN queries on moving objects. Each moving object maintains its own safe region and only report if its new location changes the results of any of the queries. Yu et al. [22] presented a query-indexing technique for monitoring $k$NN queries for moving objects and a given set of queries, where each query object maintain its own *critical region* to keep track of the $k$NN set. The similarity between [7, 22] and the M$k$NN techniques [13, 14, 18–20, 23] including the V*-Diagram is the use of the safe-region concept to reduce the access cost.

Benetis et al. [2] presented algorithms to process NN and reverse NN queries for moving objects with known trajectories. Result validity is thus expressed as a function of time. Similar to our work, Benetis et al. included methods to handle insertions and deletions of data points.

These techniques [2, 7, 22] focus on monitoring changes caused by location updates of data objects. The emphasis of the M$k$NN techniques, on the other hand, is on the changes caused by location updates of the query point.

**Summary.** The M$k$NN techniques are summarized in Table 1 on five features: providing **continuous** answer, **incremental** evaluation, **accessing** only **local** data (instead of all data), working on **unknown** query **path**, and providing **order-sensitive** $k$ NNs. Only our proposed algorithm, V*-$k$NN, has all these features.

**Table 1: Comparison of M$k$NN techniques**

| Technique | Continuous | Incremental | Local access | Unknown path | Order-sensitive |
|---|---|---|---|---|---|
| SB-$k$NN [18] | × | ✓ | ✓ | ✓ | ✓ |
| $k$VD [14] | ✓ | × | × | ✓ | × |
| Ordered $k$VD [14] | ✓ | × | × | ✓ | ✓ |
| TP$k$NN [19] | ✓ | × | ✓ | × | × |
| C$k$NN [20] | ✓ | × | ✓ | × | × |
| RIS-$k$NN [23] | ✓ | × | ✓ | ✓ | × |
| IRU [13] | ✓ | ✓ | × | ✓ | ✓ |
| V*-$k$NN | ✓ | ✓ | ✓ | ✓ | ✓ |

## 4. THE V*-DIAGRAM

We formulate the V*-Diagram in this section. The V*-Diagram is a safe-region-based technique. Previous techniques compute safe regions purely based on the data. The V*-Diagram computes safe regions based on not only the data, but also the query point and the current knowledge of the search space.

The V*-Diagram assumes a metric space and a spatial hierarchical index on the dataset. Hence the BF-kNN algorithm can be used to retrieve NNs incrementally as discussed in Section 3.1.

In the V*-Diagram, the $(k+x)$ NNs of the moving query point $q$ are maintained, where $x$ is the number of auxiliary objects to help the V*-Diagram work effectively (more analysis on $x$ is in Section 5.1). The V*-Diagram comprises two types of regions, which are discussed in Section 4.1 and Section 4.2. These regions are then put together to form a combined safe region, discussed in Section 4.3. Commonly used symbols are summarized in Table 2.

**Table 2: Symbols**

| Symbol | Meaning |
|---|---|
| $n$ | The number of objects in the database. |
| $k$ | The number of requested nearest neighbors. |
| $x$ | The number of auxiliary objects. |
| $q$ | The moving $k$NN query point. |
| $q_b$ | The position where the latest BF-$k$NN call is made. |
| $q'$ | The current position of the query point. |
| $p$ | A data object. |
| $p_k$ | The current $k^{th}$ NN of $q$. |
| $z$ | The $(k+x)^{th}$ NN of $q_b$ when $q$ is at $q_b$. |
| $S_k$ | The safe region with regard to $p_k$. |
| $B_{p_i p_j}$ | The bisector of two objects $p_i$ and $p_j$. |

### 4.1 Safe region with regard to a data object

To help explain the concept of a *safe region with regard to a data object*, the notions of *search sphere*, *known region* and *reliable region* are first introduced. Recall the BF-$k$NN algorithm in Section 3.1. Each object/node retrieved from the priority queue corresponds to an implicit *search sphere* (centered at the query point), which delimits the current search coverage, and the sphere expands gradually as more nodes/objects are accessed. Numbers are assigned to those spheres in Figure 2(a) according to the steps in Figure 2(b) that access the corresponding nodes. For example, sphere 2 corresponds to step 2 where node $S$ is retrieved; sphere 5 corresponds to step 5 where object $d$ is retrieved. Intuitively, the search sphere denotes the region we have full knowledge of, because all the objects in the sphere are already retrieved.

In the V*-Diagram, BF-$k$NN is called repeatedly to help maintain $(k+x)$ NNs. Let $q_b$ be the position of $q$ where the latest BF-$k$NN call is made. Let $z$ be the $(k+x)^{th}$ NN to $q_b$. The search sphere corresponding to $z$ centered at $q_b$ is the latest one (since BF-$k$NN stops when $z$ is obtained), and we call this search sphere the *known region*, denoted by $W(q_b, z)$. We highlight $z$ because it determines the boundary of the known region. Figure 5 gives an example. $W(q_b, z)$ is actually a sphere centered at $q_b$ with the radius $dist(q_b, z)$. Point $p$ is one of the $(k+x-1)$ NNs of $q_b$ and other objects in $W(q_b, z)$ are not shown.

Next, we formulate a region within which $q$ can move while $p$ remains one of the $(k+x)$ NNs of $q$. Let $q'$ denote a later position of $q$ after $q_b$. Suppose $q'$ is at the position as shown in Figure 5. We extend the line segment $\overline{q_b q'}$ and it exits $W(q_b, z)$ at $\chi$. Let $sph(v, l)$ denote a sphere with center $v$ and radius $l$. As long as $p$ is in $sph(q', dist(q', \chi))$, it is one of the $(k+x)$ NNs of $q'$. This is because any object outside $sph(q', dist(q', \chi))$ must be farther to $q'$ than $p$ to $q'$ and there are at most $(k+x)$ objects inside $sph(q', dist(q', \chi))$. Since any object in $sph(q', dist(q', \chi))$ remains one of the $(k+x)$ NNs of $q'$, we call $sph(q', dist(q', \chi))$

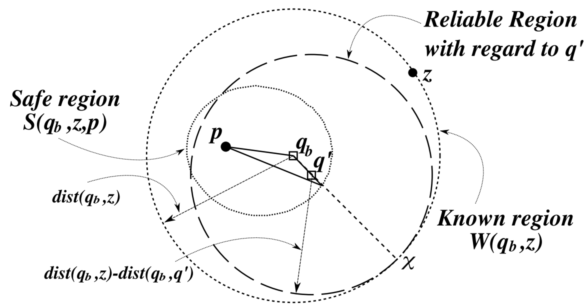


**Figure 5: The known, reliable, and safe regions**

the *reliable region* with regard to $q$ [3] and any object in the reliable region a *reliable object*. If $p$ is a reliable object, $p$ is said to be *reliable*; otherwise, it is said to be *unreliable*. Mathematically, $p$ being in the reliable region with regard to $q'$ is expressed as

$$dist(q', p) \leq dist(q_b, z) - dist(q_b, q'), \tag{1}$$

where $dist(q_b, z) - dist(q_b, q')$ is the length of $\overline{q'\chi}$.

If we view $q'$ as a variable, then Equation (1) actually describes all the possible positions of $q'$ that guarantee $p$ remaining among the $(k + x)$ NNs. Consequently, we can formulate the safe region with regard to $p$ as follows.

DEFINITION 1 (SAFE REGION WITH REGARD TO A POINT). *Given a known region $W(q_b, z)$ and a data point $p$ within $W(q_b, z)$, the safe region with regard to $p$ is:*

$$S(q_b, z, p) = \{q' : dist(q', p) + dist(q_b, q') \leq dist(q_b, z)\}.$$

For Euclidean distance in a 2D space, the boundary of $S(q_b, z, p)$ is an ellipse as illustrated in Figure 5. The two foci of the ellipse are $q_b$ and $p$; the sum of the distances from $q_b$ and $p$ to any point on the ellipse is $dist(q_b, z)$. We further have the following results.

COROLLARY 1. *In Euclidean space, the safe region with regard to z, $S(q_b, z, z)$, is the line segment $\overline{q_b z}$.*

PROOF. For any $q'$ in $S(q_b, z, z)$, $dist(q', z) + dist(q_b, q') \leq dist(q_b, z)$. By the triangular inequality, we have $dist(q', z) + dist(q_b, q') \geq dist(q_b, z)$. The set of points that satisfies both inequalities is $\{q' : dist(q', z) + dist(q_b, q') = dist(q_b, z)\}$. In Euclidean space, this is the line segment $\overline{q_b z}$. $\square$

COROLLARY 2. *$p$ is unreliable iff $q'$ is outside of $S(q_b, z, p)$.*

The proof is straightforward and we omit it.

## 4.2 Fixed-rank region

As discussed in Section 3.2, the fixed-rank region (FRR) was introduced in [13] to denote a set of possible query-point locations that share a specific ranking of all objects. For the V*-Diagram, the FRR is applied to a subset of all objects, specifically, the $(k+x)$NN. The IRU algorithm is used to incrementally maintain the FRR.

Given a list $L$ of objects, $\langle p_1, p_2, ..., p_m\rangle$, the FRR of $L$ is the set of all points such that for any point $v$ in the set, $p_1, p_2, ..., p_m$ are already in sorted (ascending) order by their distances to $v$. Let $H_{p_i p_j}$ be defined as $\{v \in DS : dist(v, p_i) \leq dist(v, p_j)\}$, where $DS$ is the data space. An FRR is a function of a list and is formulated as follows.

DEFINITION 2 (FIXED-RANK REGION).

$$F\langle p_1, p_2, ..., p_m\rangle = \bigcap_{i=1}^{m-1} H_{p_i p_{i+1}}.$$

[3]We omit "with regard to $q$" when the context is clear.

$F\langle p_1, p_2, ..., p_m\rangle$ may be written in the compact format of $F(L)$.

In Figure 4(a), the ranking of the objects according to their distances to $q_1$ is $L = \langle a, c, b, f, e, d\rangle$. The FRR $F(L)$ is defined as $H_{ac} \cap H_{cb} \cap H_{bf} \cap H_{fe} \cap H_{ed}$. The boundary of $F(L)$ is defined by five bisectors, $B_{ac}$, $B_{cb}$, $B_{bf}$, $B_{fe}$ and $B_{ed}$.

We use the IRU algorithm described in Section 3.2 to incrementally compute $F(L)$ in which $q$ currently resides for the $(k + x)$ maintained objects. An FRR is represented as (i) a list $B$ of the $(k+x-1)$ rank-adjacent bisectors, $B_{p_i p_{i+1}}$, for $i = 1, 2, ..., k+x-1$, and (ii) a reference point (which could be any point in $F(L)$, $q_b$ in our case). The FRR is incrementally maintained by (i) checking whether $q$ crosses a bisector in $B$, and (ii) if yes, performing updates accordingly.

The purpose of maintaining the FRR using the IRU algorithm is to keep the $(k + x)$ objects sorted according to their distances to $q$. One alternative solution to IRU is to computing distances between the $(k + x)$ objects and $q$ for every position of $q$, which is sampling-based. Although both methods have the same complexity, the FRR is important to the formulation of a region where the (order-sensitive) $k$NN does not change.

## 4.3 Integrated safe region

We are now ready to formulate the safe region for the M$k$NN query, called the *integrated safe region* (ISR). The ISR is the intersection of the current FRR of the $(k + x)$ maintained objects and the safe regions with regard to the $k$ nearest objects. We will first define the ISR formally and then prove that ISR satisfies the M$k$NN safe-region requirements.

Let $O$ denote the $(k + x)$NN set of $q_b$, $L$ be the list of these $(k + x)$ objects sorted by their distances to $q$, and $z$ still be the farthest retrieved object to $q_b$, which is $p_{k+x}$. The ISR is then formulated as:

$$F(L) \cap (\bigcap_{i=1}^{k} S(q_b, z, p_i)) \tag{2}$$

The computation of the ISR can be greatly reduced based on Lemma 1 and Theorem 3 below.

LEMMA 1. *$F\langle p_i, p_j\rangle \cap S(q_b, z, p_j) \cap S(q_b, z, p_i) = F\langle p_i, p_j\rangle \cap S(q_b, z, p_j)$.*

PROOF. For any point $v \in F\langle p_i, p_j\rangle$, $v$ satisfies

$$dist(v, p_i) \leq dist(v, p_j). \tag{3}$$

For any point $v \in S(q_b, z, p_j)$, by definition $v$ satisfies

$$dist(v, p_j) + dist(q_b, v) \leq dist(q_b, z). \tag{4}$$

For any $v \in F\langle p_i, p_j\rangle \cap S(q_b, z, p_i)$, it satisfies inequalities (3) and (4). By adding the two inequalities,

$$dist(v, p_i) + dist(q_b, v) \leq dist(q_b, z). \tag{5}$$

Inequality (5) shows that $v \in S(q_b, z, p_i)$, given that $v \in F\langle p_i, p_j\rangle \cap S(q_b, z, p_j)$. It can be concluded that: $F\langle p_i, p_j\rangle \cap S(q_b, z, p_j) \subseteq S(q_b, z, p_i)$ and hence $S(q_b, z, p_i)$ can be discarded in $F\langle p_i, p_j\rangle \cap S(q_b, z, p_j) \cap S(q_b, z, p_i)$. $\square$

An example is given in Figure 6. The grey region $F\langle a, c, b, f\rangle \cap S(q_1, f, c) \cap S(q_1, f, a)$ is exactly the same as $F\langle a, c, b, f\rangle \cap S(q_1, f, c)$.

THEOREM 3. *For $k \geq 2$,*

$$F\langle p_1, p_2, ..., p_k\rangle \cap (\bigcap_{i=1}^{k} S(q_b, z, p_i)) =$$

$$F\langle p_1, p_2, ..., p_k\rangle \cap S(q_b, z, p_k)$$

PROOF. Lemma 1 shows the case of $k = 2$, that is, $F\langle \boldsymbol{p_1}, \boldsymbol{p_2}\rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_2}) \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_1}) = F\langle \boldsymbol{p_1}, \boldsymbol{p_2}\rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_2})$.

If the theorem holds for $k = l$, that is,

$$F\langle \boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_l}\rangle \cap (\bigcap_{i=1}^{l} S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i})) = F\langle \boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_l}\rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_l}),$$

then the theorem can be verified for $k = l + 1$ as follows.

$$\begin{aligned} & F\langle \boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_{l+1}}\rangle \cap (\bigcap_{i=1}^{l+1} S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i})) \\ = \; & F\langle \boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_l}\rangle \cap F\langle \boldsymbol{p_l}, \boldsymbol{p_{l+1}}\rangle \\ & \cap (\bigcap_{i=1}^{l} S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i})) \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_{l+1}}) \\ = \; & F\langle \boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_l}\rangle \cap F\langle \boldsymbol{p_l}, \boldsymbol{p_{l+1}}\rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_l}) \\ & \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_{l+1}}). \end{aligned}$$

By applying Lemma 1, $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_l})$ can be removed from the above expression. Hence, we obtain a final result of

$$\begin{aligned} & F\langle \boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_l}\rangle \cap F\langle \boldsymbol{p_l}, \boldsymbol{p_{l+1}}\rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_{l+1}}) \\ = \; & F\langle \boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_{l+1}}\rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_{l+1}}). \end{aligned}$$

The theorem therefore holds for any integer value of $k$ greater than or equal to 2. □

Since $F(L) = F\langle \boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_k}\rangle \cap F\langle \boldsymbol{p_k}, \boldsymbol{p_{k+1}} ..., \boldsymbol{p_{k+x}}\rangle$, based on Theorem 3, expression (2) can be reduced to $F(L) \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k})$. Therefore, the ISR can be defined as follows.

DEFINITION 3 (INTEGRATED SAFE REGION (ISR)). *Let $O$ be the $(k + x)$NN set of $\boldsymbol{q_b}$, $L$ be the list of these $(k + x)$ objects sorted by their distances to $\boldsymbol{q}$, $\boldsymbol{z}$ be the farthest retrieved object to $\boldsymbol{q_b}$, and $\boldsymbol{p_k}$ be the $k^{th}$ object in L. The integrated safe region with respect to $\boldsymbol{q_b}$, $\boldsymbol{z}$, $\boldsymbol{p_k}$ and $L$ is defined as:*

$$I(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k}, L) = F(L) \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k}) \quad (6)$$

Next, we prove that the ISR defined above satisfies the requirements of being a safe region for the M$k$NN query, that is, the $k$ NNs as well as their order do not change when $\boldsymbol{q}$ remains in the ISR.

THEOREM 4. *If the ISR $I(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k}, L)$ is not an empty set, every point $\boldsymbol{q'}$ in $I(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k}, L)$ has the same order-sensitive $k$ NNs.*

PROOF. According to Definition 3, (1) since $I \subseteq F(L)$ (parameters of $I$ omitted), the ranking of the $(k + x)$ objects is fixed for all points in $I$, which satisfies the order-sensitivity requirement; (2) every point $\boldsymbol{q'}$ in $I$ is also in the safe regions with regard to the first $k$ objects in $L$. As a result, there can be no object outside $W(\boldsymbol{q_b}, \boldsymbol{z})$ nearer to $\boldsymbol{q'}$ than any of the first $k$ objects in $L$. Therefore, for any $\boldsymbol{q'}$ in $I$, $\boldsymbol{q'}$ has the same order-sensitive $k$ NNs as $\boldsymbol{q_b}$. □

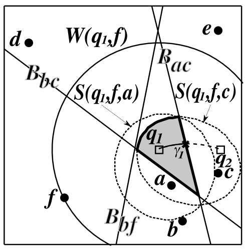


**Figure 6: Integrated safe region example ($k = 2$, $x = 2$)**

As exemplified in Figure 6, four objects retrieved by a 4NN query ($k = 2$ and $x = 2$) at $\boldsymbol{q_1}$ are $\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f}\rangle$. Point $\boldsymbol{q_1}$ is the most recent point where $(k+x)$ NNs are retrieved from the database. As long as $\boldsymbol{q'}$ remains in $F\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f}\rangle \cap S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{c})$ (the grey region), then: (i) no object outside $W(\boldsymbol{q_1}, \boldsymbol{f})$ is nearer to the two objects: $\boldsymbol{a}$ and $\boldsymbol{c}$; and (ii) the ranking of $\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f}\rangle$ is unchanged.

The diagram that contains the information used in computing the ISR is called the **V*-Diagram**. It consists of: (i) the bisectors of the rank-adjacent pairs in $L$; and (ii) the boundary of the safe region with regard to $\boldsymbol{p_k}$. We may also use the V*-Diagram to refer generally to the whole technique based on it, including the algorithms. Although the V*-Diagram in the example of Figure 6 only computes a single ISR, it actually allows incremental computation of new ISRs, which is further discussed in Section 5.

## 5. ALGORITHMS

In this section, we present V*-$k$NN, an algorithm for M$k$NN queries based on the V*-Diagram, followed by a discussion on the effect of $x$, the number of auxiliary objects. We also present the algorithms to handle insertions/deletions and dynamically changing $k$ values.

The V*-$k$NN algorithm uses the following data structures and variables to compute and maintain the ISR.

1. $L$: a list of $(k + x)$ objects always sorted in ascending order by their distances to $\boldsymbol{q}$; these objects are the $(k + x)$ NNs retrieved at $\boldsymbol{q_b}$.
2. $\boldsymbol{z}$: the farthest retrieved object in the known region when $\boldsymbol{q}$ is at $\boldsymbol{q_b}$.
3. $\boldsymbol{p_k}$: the $k^{th}$ object in $L$.
4. $S_k$: the safe region with regard to $\boldsymbol{p_k}$.
5. $B$: a list of bisectors of pairs of rank-adjacent objects (*rank-adjacent bisectors*) in the order corresponding to $L$.

We do not explicitly maintain $F(L)$ because it is represented by $B$, and checking whether $\boldsymbol{q}$ moves out of the current $F(L)$ is also done by checking whether $\boldsymbol{q}$ crosses any bisector in $B$.

V*-$k$NN produces answers continuously as shown in Algorithm 1. It has an initialization part (lines 1 to 3) and a continuous processing part (lines 4 to 19). The initialization part calls the algorithm *Compute-V** (Algorithm 2) to compute the initial ISR using the starting point $\boldsymbol{q_0}$ of the trajectory as $\boldsymbol{q_b}$. Then the continuous processing part starts.

**Algorithm 1**: V*-$k$NN($\boldsymbol{q_0}$, $k$, $x$)

```
1  $q_b \leftarrow q_0$
2  ($L$, $z$, $S_k$, $B$, ISR) ← Compute-V*($q_b$, $k$, $x$)
3  ReportResult($L$.Head($k$))
4  while (Event ← GetEvent()) do
5      $q \leftarrow$ Event.Position
6      switch Event.Type do
7          case RankUpdate
8              Bisector ← Event.Bisector
9              $L$.OrderSwap(Bisector.Index)
10             $B$.Update($L$,Bisector.Index)
11             if Bisector.Index $\leq k$ then
12                 ReportResult($L$.Head($k$))
13             if Bisector.Index $\in [k-1, k]$ then
14                 $p_k \leftarrow L$.Item($k$)
15                 $S_k \leftarrow S(q_b, z, p_k)$
16             ISR ← ConstructISR($S_k$,$B$,$q$)
17         case ReliabilityUpdate
18             $q_b \leftarrow q$
19             ($L$, $z$, $S_k$, $B$, ISR) ← Compute-V*($q_b$, $k$, $x$)
```

Algorithm Compute-V* (Algorithm 2) runs as follows. It first calls the BF-$k$NN algorithm to retrieve $(k + x)$ objects with $\boldsymbol{q_b}$ as the query point; with the retrieved objects, it sets $\boldsymbol{z}$ and $\boldsymbol{p_k}$ accordingly; then, it computes $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k})$ and rank-adjacent bisectors based on $L$ and assigns them to $S_k$ and $B$, respectively; finally, the current ISR is computed. To determine the correct half plane for each bisector in $B$, $\boldsymbol{q_b}$ is used as the reference point. Readers may notice that symbol $\boldsymbol{z}$ is explained differently here from Table 2. Object $\boldsymbol{z}$ is used to determine the known region and it is the $(k+x)^{th}$ NN of $\boldsymbol{q_b}$ when $\boldsymbol{q}$ is at $\boldsymbol{q_b}$. When deletion is taken into account, if the $(k + x)^{th}$ NN of $\boldsymbol{q_b}$ gets deleted, we still use the deleted object to determine the known region. Therefore, we make it more accurate here than in Table 2 when we did not consider deletions.

The continuous processing part of V*-$k$NN is event driven. It basically maintains the ISR as $\boldsymbol{q}$ moves. An event is triggered when $\boldsymbol{q}$ exits the current ISR. There are two types of events with this regard, *RankUpdate* and *ReliabilityUpdate*. These events are generated by a separate inexpensive process that constantly checks the current position of $\boldsymbol{q}$ against the ISR. When an event is generated, it is associated with a timestamp and the corresponding query position.

**Algorithm 2**: Compute-V*($\boldsymbol{q_b}$, $k$, $x$)

```
1 $L \leftarrow$ BF-$k$NN($\boldsymbol{q_b}$, $k + x$)
2 $\boldsymbol{z} \leftarrow L$.Item($k + x$)
3 $\boldsymbol{p_k} \leftarrow L$.Item($k$)
4 $S_k \leftarrow S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k})$
5 $B \leftarrow$ CreateBisectorList($L$)
6 ISR $\leftarrow$ ConstructISR($S_k$,$B$,$\boldsymbol{q_b}$)
7 return ($L$, $\boldsymbol{z}$, $S_k$, $B$, ISR)
```

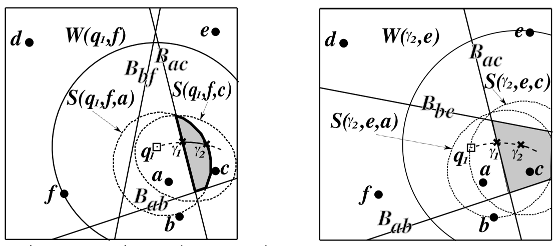
Given that the query trajectory is unknown, the query positions are updated discretely and checking for new events has to be done based on these discrete updates. To provide accurate answers, the checking should be performed at a high frequency, which is acceptable because of the low cost. Note that this is different from *processing the query* based on sampling, which requires frequent tree searches instead of event checking. The answer we provide is continuous, not based on sampled locations. Since the query positions are updated discretely, the events could happen anytime between two consecutive query updates. To compute the exact time (position) the events happen, we assume a linear trajectory between two consecutive query positions.

We describe the two event types, *RankUpdate* and *ReliabilityUpdate* below and discuss how to handle them with reference to Algorithm 1 (in the algorithm, $\boldsymbol{q}$ is used to denote the position of the query point when an event happens).

***RankUpdate*** This event is triggered when $\boldsymbol{q}$ exits the current $F(L)$, that is, crossing a rank-adjacent bisector. Besides the timestamp and query position, a *RankUpdate* event also contains the information of the bisector crossed by $\boldsymbol{q}$ (line 8). For this event, the ranks of the two objects corresponding to the bisector are swapped (line 9) and the bisector list $B$ is updated accordingly (line 10) as explained in the IRU algorithm (Section 3.2). If the event affects the rank of any of the $k$ NNs (line 11), then the new $k$ NNs are reported. Moveover, if the rank update changes $\boldsymbol{p_k}$ (line 13), $S_k$ also needs to be updated (lines 14 to 15). Changes are reported to the user if at least one of the $k$ nearest objects is affected.

***ReliabilityUpdate*** This event is triggered when $\boldsymbol{q}$ is leaving $S_k$ (that is, on the boundary of $S_k$). It means that the $k^{th}$ NN is about to become unreliable and hence the number of reliable objects is about to become less than $k$. Therefore, the *ReliabilityUpdate* event calls Compute-V* using the event position, $\boldsymbol{q}$, to obtain $x$ new auxiliary objects so that all the $(k+x)$ maintained objects are reliable again. The new ISR is constructed accordingly. This event does not cause result update because neither the $k$NN set nor their ordering changes. If another object really becomes nearer than the $k^{th}$ NN, a *RankUpdate* will be triggered, which will update the result.

Next, we give a running example of the algorithm. Recall the example in Figure 6. At the starting point $\boldsymbol{q_1}$, 4 NNs are retrieved in the order of $\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f} \rangle$. The ISR is $F\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f} \rangle \cap S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{c})$. On the trajectory of $\boldsymbol{q}$ (starting at $\boldsymbol{q_1}$), two events happen when $\boldsymbol{q}$ crosses $B_{\boldsymbol{ac}}$ at $\boldsymbol{\gamma_1}$ and exits $S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{a})$ at $\boldsymbol{\gamma_2}$. Figure 7 shows the effects of these two events.



(a) $F\langle \boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{f} \rangle \cap S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{a})$ (b) $F\langle \boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{e} \rangle \cap S(\boldsymbol{\gamma_2}, \boldsymbol{e}, \boldsymbol{a})$

**Figure 7: Example for Algorithm 1, ($k = 2$, $x = 2$)**

Figure 7(a) shows how the ISR changes after $\boldsymbol{q}$ crosses $B_{\boldsymbol{ac}}$. At the instant that $\boldsymbol{q}$ is crossing $B_{\boldsymbol{ac}}$ at $\boldsymbol{\gamma_1}$, a *RankUpdate* event is triggered, which causes $\boldsymbol{a}$ and $\boldsymbol{c}$ to swap their ranks. The list $L$ becomes $\langle \boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{f} \rangle$, and this causes both $F(L)$ and $S_k$ to change. Now $\boldsymbol{a}$ becomes $\boldsymbol{p_k}$ ($2^{nd}$NN), and hence the ISR becomes $F\langle \boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{f} \rangle \cap S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{a})$ (the grey region). The current 2 NNs, $\boldsymbol{c}$ and $\boldsymbol{a}$, are reported to the user in that order.

Figure 7(b) shows how the ISR changes after $\boldsymbol{q}$ exits $S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{a})$. At the instant that $\boldsymbol{q}$ is exiting $S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{a})$ at $\boldsymbol{\gamma_2}$, a *ReliabilityUpdate* event is triggered, which calls *Compute-V** to retrieve more objects. The new $(k+x)$ NNs are $\langle \boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{e} \rangle$, with the corresponding ISR, $F\langle \boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{e} \rangle \cap S(\boldsymbol{\gamma_2}, \boldsymbol{e}, \boldsymbol{a})$ (the grey region).

## 5.1 On the number of auxiliary objects

Auxiliary objects are an important part of the V*-Diagram technique. They allow $\boldsymbol{q}$ to move away from $\boldsymbol{q_b}$ while retaining the current $k$ NNs by providing the knowledge beyond the coverage of the search sphere of the original $k$ NNs. This makes it possible to continuously evaluate the M$k$NN query. In this subsection, we discuss possible values of $x$, the number of auxiliary objects. Generally, we find that $x$ should not assume the values of 0 and 1, which are explained below.

Having $x$ equal to 0 implies that $\boldsymbol{z}$ and the $k^{th}$ object in $L$, $\boldsymbol{p_k}$, are the same object. According to Corollary 1, in Euclidean space, the safe region with regard to $\boldsymbol{z}$ (which is $S_k$) is the line segment $\overline{\boldsymbol{q_b}\boldsymbol{z}}$. Unless $\boldsymbol{q}$ moves along $\overline{\boldsymbol{q_b}\boldsymbol{z}}$, $\boldsymbol{q}$ exits $S_k$ as it starts moving. Probabilistically, it is highly unlikely that $\boldsymbol{q}$ moves along $\overline{\boldsymbol{q_b}\boldsymbol{z}}$ since $\overline{\boldsymbol{q_b}\boldsymbol{z}}$ is just one direction among the infinite possible directions $\boldsymbol{q}$ can move towards. Therefore, it is probable that $\boldsymbol{q}$ always exits $S_k$ as it moves and it triggers the *ReliabilityUpdate* event infinitely. As a result, $x$ should not be set to 0 for Euclidean space.

When $x$ is a positive integer, the problem of infinitely triggering the *ReliabilityUpdate* event does not happen except under the coincidence described in the next paragraph. Therefore, any integer greater than 1 is a valid value for $x$. The effect of the value of $x$ on performance is further investigated in Sections 7 and 8.

In theory, the problem of $S_k$ being a line segment may happen with any value of $x$ when the last $(x + 1)$ objects in $L$ have the same distance to $\boldsymbol{q_b}$, which is $dist(\boldsymbol{q_b}, \boldsymbol{z})$. In this case, $dist(\boldsymbol{q_b}, \boldsymbol{p_k})$ equals to $dist(\boldsymbol{q_b}, \boldsymbol{z})$. By definition, $S_k = \{\boldsymbol{q'} : dist(\boldsymbol{q'}, \boldsymbol{p_k}) + dist(\boldsymbol{q_b}, \boldsymbol{q'}) \leq dist(\boldsymbol{q_b}, \boldsymbol{z})\}$. If we replace $dist(\boldsymbol{q_b}, \boldsymbol{z})$ by $dist(\boldsymbol{q_b}, \boldsymbol{p_k})$ in the inequality of the definition, we get $S_k = \{\boldsymbol{q'} : dist(\boldsymbol{q'}, \boldsymbol{p_k}) + dist(\boldsymbol{q_b}, \boldsymbol{q'}) \le dist(\boldsymbol{q_b}, \boldsymbol{p_k})\}$, which is a line segment in Euclidean space. To completely avoid this problem, we can check whether $dist(\boldsymbol{q_b}, \boldsymbol{p_k})$ is equal to $dist(\boldsymbol{q_b}, \boldsymbol{z})$ after we retrieve $(k + x)$ objects by a BF-$k$NN call. If they are equal, then we increase the value of $x$ until $dist(\boldsymbol{q_b}, \boldsymbol{p_k})$ is different from $dist(\boldsymbol{q_b}, \boldsymbol{z})$.

In general, a larger value of $x$ provides a larger $S_k$, and hence the less frequent we need to retrieve new objects from the database. Having the value of $x$ too small will result in highly frequent BF-$k$NN calls. On the other hand, a too large $x$ value also incurs the overhead of retrieving more objects in every BF-$k$NN call and more computation for maintaining them.

## 5.2 Insertions and deletions of objects

**Algorithm 3**: DatasetUpdate($\boldsymbol{q}$,$\boldsymbol{p}$,*Operation*)

```
1  if $\boldsymbol{p} \in W(\boldsymbol{q_b}, \boldsymbol{z})$ then
2      if Operation = Insertion then
3          $L \leftarrow$ Insert($L$,$\boldsymbol{p}$,$\boldsymbol{q}$)
4      else
5          $L \leftarrow$ Delete($L$,$\boldsymbol{p}$)
6      $B$.Update($L$)
7      if $k > L$.Length() then
8          $\boldsymbol{q_b} \leftarrow \boldsymbol{q}$
9          ($L$, $\boldsymbol{z}$, $S_k$, $B$, ISR) $\leftarrow$ Compute-V*($\boldsymbol{q_b}$, $k$, $x$)
10         ReportResult($L$.Head($k$))
11     else if $dist(\boldsymbol{q}, \boldsymbol{p}) \le dist(\boldsymbol{q}, \boldsymbol{p_k})$ then
12         $\boldsymbol{p_k} \leftarrow L$.Item($k$)
13         $S_k \leftarrow S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k})$
14         if $\boldsymbol{q} \notin S_k$ then
15             $\boldsymbol{q_b} \leftarrow \boldsymbol{q}$
16             ($L$, $\boldsymbol{z}$, $S_k$, $B$, ISR) $\leftarrow$ Compute-V*($\boldsymbol{q_b}$, $k$, $x$)
17         else
18             ISR $\leftarrow$ ConstructISR($S_k$,$B$,$\boldsymbol{q}$)
19         ReportResult($L$.Head($k$))
20     else
21         ISR $\leftarrow$ ConstructISR($S_k$,$B$,$\boldsymbol{q}$)
```

In this subsection, we describe the algorithm to perform updates (that is, insertions and deletions) to the dataset for V*-$k$NN. The algorithm is called *DatasetUpdate* and is presented in Algorithm 3. In this algorithm, $\boldsymbol{q}$ denotes the position of the query point when the update happens and it is passed in as an input. Let $\boldsymbol{p}$ be the object to be inserted or deleted. First, the algorithm checks whether the $\boldsymbol{p}$ is in $W(\boldsymbol{q_b}, \boldsymbol{z})$. If not, the update can be safely ignored because it cannot affect the ISR. Otherwise, an insertion/deletion of $\boldsymbol{p}$ into/from $L$ is performed and $B$ is updated accordingly (lines 2 to 6): insertion of $\boldsymbol{p}$ needs $\boldsymbol{q}$ for computing distances from $\boldsymbol{q}$ and the maintained objects to find the correct insertion slot in $L$; deleting $\boldsymbol{p}$ from $L$ requires only a simple lookup operation. After the bisector update, the ISR and $L$ could be in one of the following three cases:

(i) *The length of L becomes smaller than k as a result of a deletion (line 7).* In this case, $\boldsymbol{q_b}$ is set to $\boldsymbol{q}$ and Compute-V* is called to retrieve more objects and compute the new ISR accordingly (lines 8 and 9). The new result is reported (line 10).

(ii) *The length of L is still greater than k but the update affects the kNN set (line 11).* We update $\boldsymbol{p_k}$ and $S_k$ (lines 12-13) and check if $\boldsymbol{q}$ is inside the new $S_k$ (line 14). If $\boldsymbol{q}$ is not inside the new $S_k$, then Compute-V* is called (line 16). Otherwise, the ISR is updated to reflect the changes in $B$ and $S_k$. Since the $k$ NNs have changed, the new result is reported to the user (line 19).

(iii) *The update has no effects to the kNN set (line 20).* Only the ISR is updated to reflect the change in $B$.

## 5.3 M$k$NN with dynamically changing $k$ values

The ability to gracefully handle changes to the value of $k$ is crucial for the *distance browsing* functionality [6]. For static $k$NN queries, distance browsing is a feature that allows NNs to be incrementally retrieved without having to specify the value of $k$ in advance. In this paper, we allow the value of $k$ to be changed without incurring heavy computation.

Algorithm *KUpdate* (Algorithm 4) shows how V*-$k$NN handles dynamically changing $k$ values. The algorithm has two inputs: the current location $\boldsymbol{q}$ of the query point and the new $k$ value. We first check if the new $k$ is greater than the length of $L$. If yes, $\boldsymbol{q_b}$ is set to $\boldsymbol{q}$ and Compute-V* is called. Otherwise, $\boldsymbol{p_k}$ and $S_k$ are updated for the new $k$ (lines 5 and 6). If $\boldsymbol{q}$ is not inside the new $S_k$, $\boldsymbol{q_b}$ is set to $\boldsymbol{q}$ and Compute-V* is called. Otherwise, only the ISR has to be updated to incorporate the new $S_k$ (line 11). Finally, the new $k$ NNs are reported (line 12). As we can see, the V*-$k$NN algorithm can easily accommodate dynamically changing $k$ values due to its incremental nature.

**Algorithm 4**: KUpdate($\boldsymbol{q}$,$k$)

```
1  if $k > L$.Length() then
2      $\boldsymbol{q_b} \leftarrow \boldsymbol{q}$
3      ($L$, $\boldsymbol{z}$, $S_k$, $B$, ISR) $\leftarrow$ Compute-V*($\boldsymbol{q_b}$, $k$, $x$)
4  else
5      $\boldsymbol{p_k} \leftarrow L$.Item($k$)
6      $S_k \leftarrow S(\boldsymbol{q}, \boldsymbol{z}, \boldsymbol{p_k})$
7      if $\boldsymbol{q} \notin S_k$ then
8          $\boldsymbol{q_b} \leftarrow \boldsymbol{q}$
9          ($L$, $\boldsymbol{z}$, $S_k$, $B$, ISR) $\leftarrow$ Compute-V*($\boldsymbol{q_b}$, $k$, $x$)
10     else
11         ISR $\leftarrow$ ConstructISR($S_k$,$B$,$\boldsymbol{q}$)
12 ReportResult($L$.Head($k$))
```

## 6. THE V*-DIAGRAM IN SPATIAL NETWORKS

When the movement of $\boldsymbol{q}$ is constrained by network connectivity, NN problems should be solved based on the network distance. For example, a car is travelling on a road and it keeps track of the $k$ nearest gas stations based on the road network distance.

In this section, we show how the V*-Diagram technique can be applied to the domain of spatial networks, which also satisfies our metric space assumption. Due to the space limitation, we could not fully elaborate the application of V*-Diagram in the spatial-network model, but only present the essential components of the technique.

The essence of the V*-Diagram is the ISR, which consists of two key components: safe regions with regard to data objects and the fixed-rank region (FRR). Hence, we focus our discussions on how to determine these two components in spatial networks, while the algorithms to compute the ISR and process M$k$NN queries can be reused. To avoid an exhaustive discussion, we use an example to illustrate the main idea.

A spatial network is usually represented as a set of vertices and a set of edges, where an edge is defined by two vertices. Given points $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ in the network, $dist(\boldsymbol{p_1}, \boldsymbol{p_2})$ is the length of the shortest path between $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$. Figure 8 shows a spatial network of eight nodes, $\boldsymbol{a}$ to $\boldsymbol{h}$. Distance $dist(\boldsymbol{q_b}, \boldsymbol{p})$ is 2 using the path via $\boldsymbol{b}$.

We use $pnt(\boldsymbol{p_1}, \boldsymbol{p_2}, l)$ to denote a point on the edge $(\boldsymbol{p_1}, \boldsymbol{p_2})$ with distance $l$ to $\boldsymbol{p_1}$, and $seg(\boldsymbol{p_1}, \boldsymbol{p_2}, l)$ to denote a section on the edge $(\boldsymbol{p_1}, \boldsymbol{p_2})$ with the starting point $\boldsymbol{p_1}$ and length $l$. For example, $\boldsymbol{z}$ is at $pnt(\boldsymbol{a}, \boldsymbol{e}, 3)$, and all points between $\boldsymbol{a}$ and $\boldsymbol{z}$ can be represented as $seg(\boldsymbol{a}, \boldsymbol{e}, 3)$.

Consider an M$k$NN query with $k = 1$. Suppose $x$ is 2 for the V*-Diagram and $\boldsymbol{q_b}$ is at $pnt(\boldsymbol{a}, \boldsymbol{b}, 3)$. The 3 NNs for $\boldsymbol{q_b}$ are: $\boldsymbol{p}$, $\boldsymbol{r}$ and $\boldsymbol{z}$. We can use any $k$NN technique for spatial networks (such as those described in [8, 9, 15, 17]) to retrieve the $(k + x)$ NNs at $\boldsymbol{q_b}$. The algorithm also returns the edges in the range of $dist(\boldsymbol{q_b}, \boldsymbol{z})$ to $\boldsymbol{q_b}$ (that is, $W(\boldsymbol{q_b}, \boldsymbol{z})$), $(\boldsymbol{a}, \boldsymbol{b})$, $(\boldsymbol{a}, \boldsymbol{c})$, $(\boldsymbol{a}, \boldsymbol{e})$, $(\boldsymbol{b}, \boldsymbol{d})$, $(\boldsymbol{b}, \boldsymbol{f})$, $(\boldsymbol{d}, \boldsymbol{c})$, $(\boldsymbol{d}, \boldsymbol{g})$, and the distances from the vertices of these edges to $\boldsymbol{q_b}$.

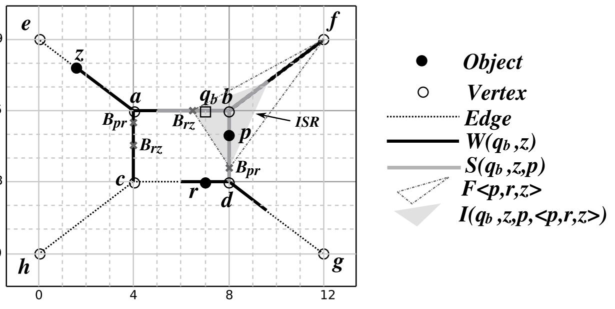


**Figure 8: V*-Diagram in a spatial network ($k = 1$ and $x = 2$)**

Next, we determine the safe region for $\boldsymbol{p_k}$ (which is $\boldsymbol{p}$ in this example). The safe region $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$ is the region where for any point $\boldsymbol{q'}$ in the region, the sum of $dist(\boldsymbol{q_b}, \boldsymbol{q'})$ and $dist(\boldsymbol{p}, \boldsymbol{q'})$ is less than or equal to $dist(\boldsymbol{q_b}, \boldsymbol{z})$, which is 6. We need to explore all the edges in $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$ to identify its boundary. Since $W(\boldsymbol{q_b}, \boldsymbol{z})$ encloses $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$, and we already know the edges that are in $W(\boldsymbol{q_b}, \boldsymbol{z})$ via the $k$NN query executed at $\boldsymbol{q_b}$, we only need to consider those edges. We use edge $(\boldsymbol{b}, \boldsymbol{f})$ as an example. For a point $\boldsymbol{q'}$ on $(\boldsymbol{b}, \boldsymbol{f})$ to be in $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$, $dist(\boldsymbol{q_b}, \boldsymbol{q'}) + dist(\boldsymbol{p}, \boldsymbol{q'})$ has to be less than 6 according to the definition. Since $dist(\boldsymbol{q_b}, \boldsymbol{q'}) + dist(\boldsymbol{p}, \boldsymbol{q'})$ is equal to $dist(\boldsymbol{q_b}, \boldsymbol{b}) + dist(\boldsymbol{p}, \boldsymbol{b}) + 2dist(\boldsymbol{b}, \boldsymbol{q'})$, and both $dist(\boldsymbol{q_b}, \boldsymbol{b})$ and $dist(\boldsymbol{p}, \boldsymbol{b})$ are 1, $\boldsymbol{q'}$ is in $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$ when $dist(\boldsymbol{b}, \boldsymbol{q'})$ is less than or equal to 2. Consequently, $pnt(\boldsymbol{b}, \boldsymbol{f}, 2)$ forms part of the boundary of $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$. The boundary points on other edges can be computed in a similar manner. After obtaining all boundary points, we get $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$, which consists of edge $(\boldsymbol{b}, \boldsymbol{d})$, $seg(\boldsymbol{b}, \boldsymbol{a}, 3)$ and $seg(\boldsymbol{b}, \boldsymbol{f}, 2)$, plotted as grey thick segments in the figure.

As discussed in Section 4.2, the FRR is determined by the rank-adjacent bisectors the maintained objects. In a spatial network, a bisector reduces to points on edges. For example, $B_{\boldsymbol{pr}}$ has two bisecting points, one on $pnt(\boldsymbol{b}, \boldsymbol{d}, 2.5)$ and the other on $pnt(\boldsymbol{a}, \boldsymbol{c}, 0.5)$. The FRR $F\langle \boldsymbol{p}, \boldsymbol{r}, \boldsymbol{z}\rangle$ is the region with $B_{\boldsymbol{pr}}$ and $B_{\boldsymbol{rz}}$ as the boundary and containing $\boldsymbol{q_b}$. It consists of $(\boldsymbol{b}, \boldsymbol{f})$, $seg(\boldsymbol{b}, \boldsymbol{a}, 1.5)$ and $seg(\boldsymbol{b}, \boldsymbol{d}, 2.5)$, shown as the segments in the dashed triangle.

As a result, the ISR ($S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p}) \cap F\langle \boldsymbol{p}, \boldsymbol{r}, \boldsymbol{z}\rangle$) consists of $seg(\boldsymbol{b}, \boldsymbol{a}, 1.5)$, $seg(\boldsymbol{b}, \boldsymbol{d}, 2.5)$ and $seg(\boldsymbol{b}, \boldsymbol{f}, 2)$. It is shown as the segments in the grey region. According to Algorithm 1, exiting the ISR via $pnt(\boldsymbol{b}, \boldsymbol{a}, 1.5)$ ( $B_{\boldsymbol{rz}}$) or $seg(\boldsymbol{b}, \boldsymbol{d}, 2.5)$ ( $B_{\boldsymbol{pr}}$) triggers a *RankUpdate* event, and exiting the ISR via $pnt(\boldsymbol{b}, \boldsymbol{f}, 2)$ triggers a *ReliabilityUpdate* event.

## 7. PERFORMANCE ANALYSIS

Among all techniques listed in Table 1, only RIS-$k$NN by Zhang et al. [23] and V*-$k$NN provide continuous answers for the M$k$NN query with unknown query trajectory and without accessing all the data. Therefore, this section focuses on a comparative performance analysis on RIS-$k$NN and V*-$k$NN in terms of tree node accesses (that is, IO cost). We assume a 2D space, although the analysis can be extended to higher dimensional spaces. We also assume that the data objects are uniformly distributed.

V*-$k$NN only has node accesses in the calls to BF-$k$NN, which are triggered by the *ReliabilityUpdate* event (line 19 of Algorithm 1). We analyze the frequency of *ReliabilityUpdate* events, $f_b$, as follows. A *ReliabilityUpdate* event happens when $\boldsymbol{q}$ exits $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k})$. We denote the point of exit as $\boldsymbol{q_e}$. The *ReliabilityUpdate* event happens only once during the process of $\boldsymbol{q}$ moving away from $\boldsymbol{q_b}$ until reaching $\boldsymbol{q_e}$. Then a BF-$k$NN is performed and $\boldsymbol{q_e}$ becomes the new $\boldsymbol{q_b}$. A *ReliabilityUpdate* event will happen again when the next time $\boldsymbol{q}$ exits $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k})$. Therefore, $f_b$ is inversely proportional to the distance $\boldsymbol{q}$ travels from $\boldsymbol{q_b}$ to $\boldsymbol{q_e}$. In the worst case, $\boldsymbol{q}$ moves in a straight line, and $f_b$ is inversely proportional to $dist(\boldsymbol{q_b}, \boldsymbol{q_e})$. The expected value of $dist(\boldsymbol{q_b}, \boldsymbol{q_e})$ is obtained as follows. When $\boldsymbol{q}$ moves to $\boldsymbol{q_e}$, $\boldsymbol{p_k}$ is on the boundary of the reliable region (with regard to $\boldsymbol{q_e}$). For a better understanding, imagine in Figure 5, $\boldsymbol{p}$ is $\boldsymbol{p_k}$, $\boldsymbol{q'}$ is $\boldsymbol{q_e}$ and it is on the boundary of $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$. We can see that $dist(\boldsymbol{q_b}, \boldsymbol{q_e})$ equals $dist(\boldsymbol{q_b}, \boldsymbol{\chi}) - dist(\boldsymbol{q_e}, \boldsymbol{\chi})$. Note that $dist(\boldsymbol{q_b}, \boldsymbol{\chi})$ is the radius of $W(\boldsymbol{q}, \boldsymbol{z})$, which is a sphere that contains $(k+x)$ points; $dist(\boldsymbol{q_e}, \boldsymbol{\chi})$ is the radius of the reliable region with regard to $\boldsymbol{q_e}$, which is a sphere that contains $k$ points.

According to [21], the distance between the query point $\boldsymbol{q_b}$ and the $k^{th}$ NN is $2/C_v(1 - \sqrt{1 - \sqrt{k/n}})$, where $C_v$ is the vicinity constant [21]. Therefore $dist(\boldsymbol{q_b}, \boldsymbol{q_e})$, which is $dist(\boldsymbol{q_b}, \boldsymbol{\chi}) - dist(\boldsymbol{q_e}, \boldsymbol{\chi})$, can be expressed as

$$\frac{2}{C_v}(\sqrt{1 - \sqrt{k/n}} - \sqrt{1 - \sqrt{(k+x)/n}}).$$

Since $f_b$ is inversely proportional to $dist(\boldsymbol{q_b}, \boldsymbol{q_e})$,

$$\mathcal{O}(f_b) \quad = \quad \mathcal{O}(1/(\sqrt{1 - \sqrt{k/n}} - \sqrt{1 - \sqrt{(k+x)/n}})).$$

The expression of $\mathcal{O}(f_b)$ can be relaxed as follows:

$$\begin{aligned} \frac{1}{\sqrt{1-\sqrt{k/n}}-\sqrt{1-\sqrt{(k+x)/n}}} &= \frac{\sqrt{1-\sqrt{k/n}}+\sqrt{1-\sqrt{(k+x)/n}}}{\sqrt{(k+x)/n}-\sqrt{k/n}} \\ \leq \frac{2\sqrt{1-\sqrt{k/n}}}{\sqrt{(k+x)/n}-\sqrt{k/n}} &\leq \frac{2}{\sqrt{(k+x)/n}-\sqrt{k/n}} \\ = 2\frac{\sqrt{(k+x)/n}+\sqrt{k/n}}{(k+x)/n-k/n} &= 2\frac{\sqrt{(k+x)/n}+\sqrt{k/n}}{x/n} \leq 4\frac{\sqrt{(k+x)/n}}{x/n}. \end{aligned}$$

Therefore $\mathcal{O}(f_b)$ is $\mathcal{O}(\sqrt{\frac{(k+x)n}{x^2}})$. Typically, $x$ is comparable to $k$ and $(k + x)$ is much smaller than $xk$. Thus, we obtain that $f_b$ is $\mathcal{O}(\sqrt{\frac{kn}{x}})$. Let $C_{nn}$ be the cost of a BF-$k$NN call. Then we obtain the total IO cost of V*-$k$NN, $C_{nn}\mathcal{O}(\sqrt{\frac{kn}{x}})$.

The RIS-$k$NN processes the M$k$NN query as follows. Every time $\boldsymbol{q}$ exits the current $k$VD cell, RIS-$k$NN is executed to obtain the new $k$VD cell and the corresponding $k$NN. The total cost depends on the frequency of crossing $k$VD cells and the cost of each RIS-$k$NN run. In the worst case, $\boldsymbol{q}$ moves along a straight line. The frequency of $\boldsymbol{q}$ crossing $k$VD cells is proportional to the average linear density[4] of the $k$VD cells. The number of the $k$VD cells is $\mathcal{O}(kn - k)$ in 2D space [14]. We assume that $k$ is much smaller than $n$. Thus, the density of $k$VD cells is $\mathcal{O}(kn)$, which corresponds to a linear density of $\mathcal{O}(\sqrt{kn})$. Each RIS-$k$NN run requires 12 TP$k$NN queries on average [23]. Let $C_{tpnn}$ be the cost of a TP$k$NN call. Then the total IO cost of RIS-$k$NN for the M$k$NN query is $12C_{tpnn}\mathcal{O}(\sqrt{kn})$.

[4]The number of $k$VD cells crossed per unit length along a straight line.

We now compare the costs of V*-$k$NN and RIS-$k$NN. For the frequency component, V*-$k$NN is smaller than RIS-$k$NN especially when $x$ is large. For the cost-per-$k$NN-algorithm-call component, the first TP$k$NN call of an RIS-$k$NN run is always more expensive than a BF-$k$NN call because TP$k$NN has to retrieve at least the $k$NN and it needs to access more nodes to obtain the influence object. In addition, there are 11 subsequent TP$k$NN calls of an RIS-$k$NN run, where each call is much more expensive than a BF-$k$NN call in practice. Consequently, the IO cost of V*-$k$NN is much lower than that of RIS-$k$NN.

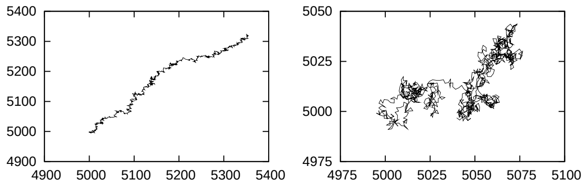
In terms of the CPU cost, V*-$k$NN maintains the rank of $(k + x)$ objects, which causes more computation than RIS-$k$NN on the client side. However, given today's mobile devices and trends (e.g., phones with multimedia and graphical functionalities), we argue that the CPU power of these devices is adequate to check $(k + x - 1)$ bisectors at a reasonably high frequency (e.g., once every second) for practical values of $k$ and $x$. In many realistic settings, the benefits from the communication-cost reduction will outweigh this overhead on the client side.

On the server side, V*-$k$NN has a lower CPU cost than RIS-$k$NN. It is because RIS-$k$NN uses the TP-$k$NN query which incurs a much higher computational cost than BF-$k$NN.

## 8. EXPERIMENTAL STUDY

This section presents the results of our experimental study. In our implementaion, the R*-tree [1] is used to index the data objects. The page size is 1 KB, which has a node capacity of 50 entries.

We used both synthetic and real datasets in our experiments. All datasets span the space of $10,000 \times 10,000$ square units. We generated synthetic datasets with uniform (U) and Zipfian (Z) distributions with the default cardinality of $25,000$ data points. The real datasets are $65,743$ and $119,897$ postal addresses from California (C) and North-Eastern USA (N), respectively.



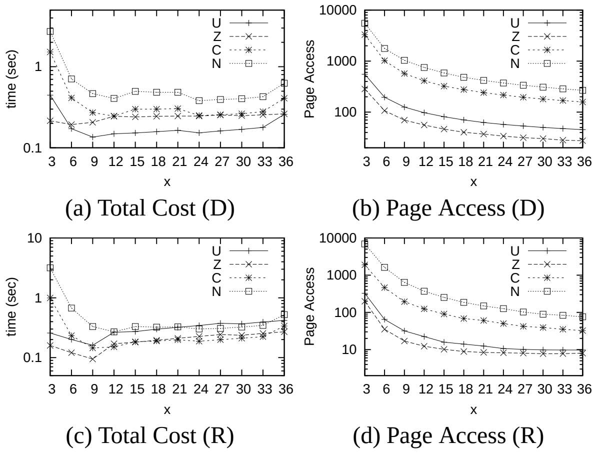
(a) Directional (D) (b) Random (R)

**Figure 9: Trajectory types**

We generated two different types of query trajectories, random (R) and directional (D), as shown in Figure 9. Each trajectory consists of $1,001$ points. Between two points, the trajectory is assumed to be a straight line segment. The length of each segment is 1 unit. For each type of trajectory, we generated 20 different trajectories. We run these 20 trajectories as a query set for each experiment and present the average result. We measured both the cumulative total response time and the cumulative number of page accesses for a whole trajectory as the performance metric.

### 8.1 Choosing the value of $x$

In the first set of experiments, we study the impact of the value of $x$ on the performance of V*-$k$NN. We varied $x$ from 3 to 36 with $k$ set to 20 and no buffer space. We did not use the values less than 3 for $x$ because too small values of $x$ do not yield a reasonable size of $S_k$ to make V*-$k$NN effective. Figure 10 shows result of the response time and number of page accesses as functions of $x$ for both query trajectory types. The response time first decreases as $x$ increases but becomes more constant as $x$ keeps increasing. There is a small increase when $x$ becomes large (around 30). The number of page accesses always decreases as $x$ increases. This is because as $x$ becomes larger, $S_k$ becomes larger and hence BF-$k$NN is called less frequently. This reduces both CPU time and IO cost. When $x$ becomes too large, the computational overhead of maintaining more objects becomes more significant and may overweigh the savings in CPU time. Therefore, the CPU time increases for a large $x$ value. The number of page accesses mainly depends on the frequency of BF-$k$NN calls and hence always decreases. These results confirms our discussion in Section 5.1 and the analysis in 7. In all these experiment, the $x$ value of 9 provides a good performance, so 9 is used as the default value of $x$ for the rest of the experiments. As we can see, some variations around the value of 9 do not affect the performance much.



(a) Total Cost (D) (b) Page Access (D)

(c) Total Cost (R) (d) Page Access (R)

**Figure 10: Effect of $x$**

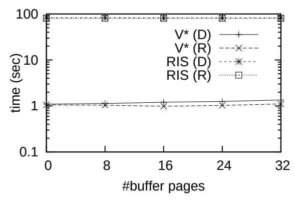
### 8.2 Comparative study: centralized

Among all the techniques discussed in the related work, RIS-$k$NN by Zhang et al. [23] is the only algorithm that is comparable to our work. Therefore, we perform a comparative experimental study on V*-$k$NN and RIS-$k$NN. The next four sets of experiments compare these two techniques using different experimental parameters.

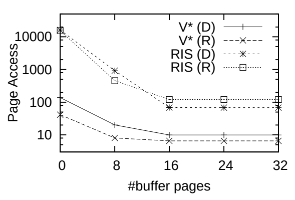
**The effect of the buffer size.** In this set of experiments, we use the buffer sizes of 0, 8, 16, 24 and 32 pages. Figure 11 shows the results for two synthetic datasets with the default dataset size and the two real datasets. V*-$k$NN outperforms the RIS-$k$NN in all settings in terms of both total response time and number of page accesses. In most cases, the improvement factor is two orders of magnitude. For both methods, the page access cost decreases as the buffer size increases as expected.

**The effect of the number of query location updates.** In this set of experiments, we vary the number of location updates in the query trajectory from 0 to 1000. Figure 12 shows the response time on the four datasets. In all experiments, V*-$k$NN outperforms RIS-$k$NN and the improvement factor is two orders of magnitude in most cases. The results of the number of page accesses have very similar behavior as those of the total response time. Therefore we do not present them for the remaining experiments due to space limitation. For both techniques, the total response time increases as the number of location updates increases.

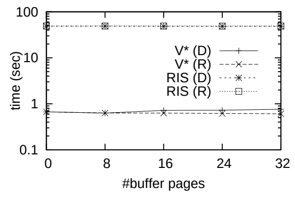
**The effect of the dataset size.** In this set of experiments, we vary the number of objects in the dataset from 5,000 to 25,000 for the synthetic datasets. Figure 13 shows the response time results. Again, V*-$k$NN outperforms RIS-$k$NN in all settings and the improvement factor is two orders of magnitude in most cases. For both techniques, the total response time increases as the number of objects increases.



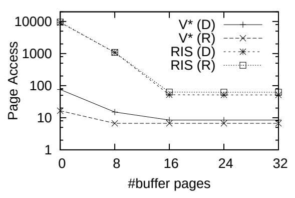
(a) Total Cost (U)



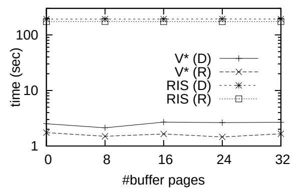
(b) Page Access (U)



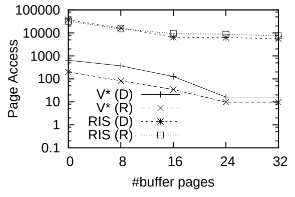
(c) Total Cost (Z)



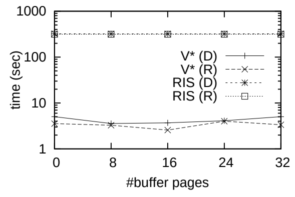
(d) Page Access (Z)



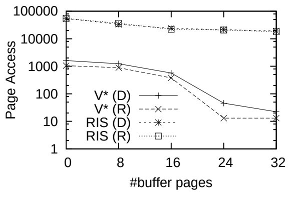
(e) Total Cost (C)



(f) Page Access (C)

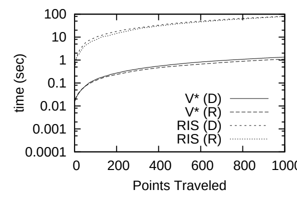


(g) Total Cost (N)

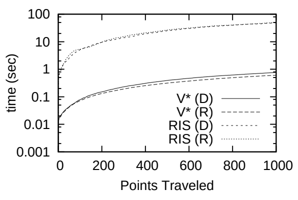


(h) Page Access (N)

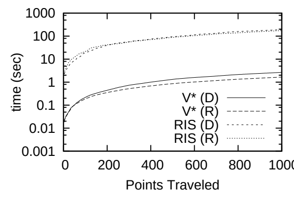
**Figure 11: Effect of buffer size**



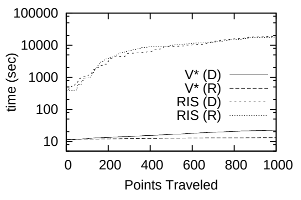
(a) Total Cost (U)



(b) Total Cost (Z)

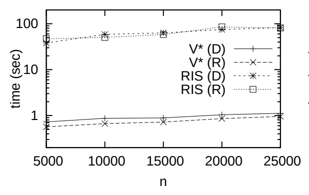


(c) Total Cost (C)

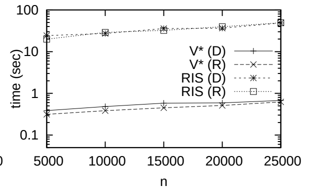


(d) Total Cost (N)

**Figure 12: Effect of the number of location updates**

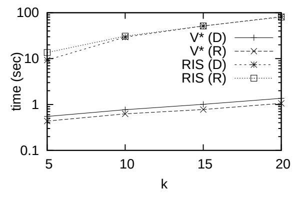


(a) Total Cost (U)

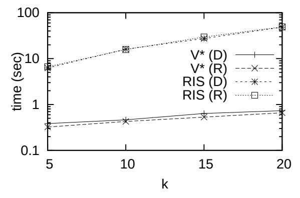


(b) Total Cost (Z)

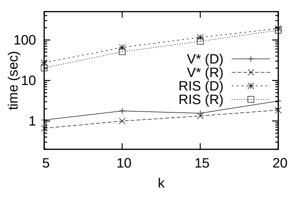
**Figure 13: Effect of dataset size**



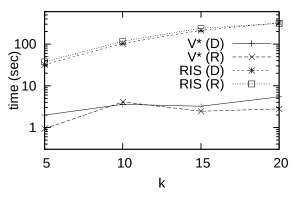
(a) Total Cost (U)



(b) Total Cost (Z)

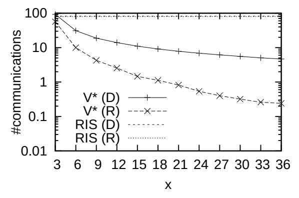


(c) Total Cost (C)

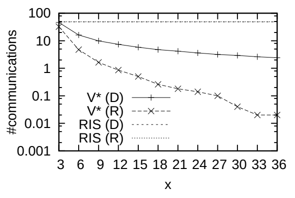


(d) Total Cost (N)

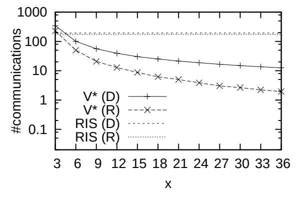
**Figure 14: Effect of $k$**



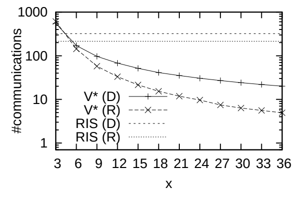
(a) Communication Cost (U)



(b) Communication Cost (Z)

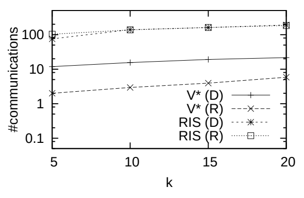


(c) Communication Cost (C)

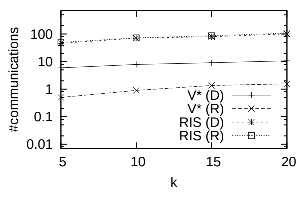


(d) Communication Cost (N)

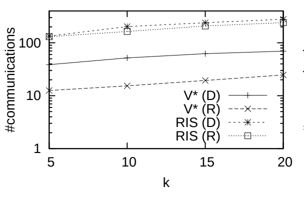
**Figure 15: Communication cost: effect of $x$**



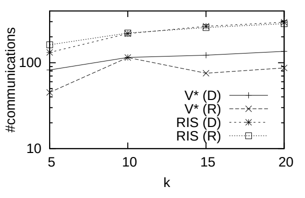
(a) Communication Cost (U)



(b) Communication Cost (Z)



(c) Communication Cost (C)



(d) Communication Cost (N)

**Figure 16: Communication cost: effect of $k$**

**The effect of $k$.** In this set of experiments, we vary the value of $k$ from 5 to 20 for the four datasets. Figure 14 shows the response time results. We observe similar results as in previous experiments. V*-$k$NN outperforms RIS-$k$NN in all settings and the improvement factor is two orders of magnitude in most cases. For both techniques, the total response time increases as the value of $k$ increases, but the total response time of V*-$k$NN increases slower than that of RIS-$k$NN.

## 8.3 Comparative study: client-server

In a high-latency client-server setting, communication costs between the mobile client and the server dominate the other costs. The following experiments compare the communication costs of V*-$k$NN and RIS-$k$NN. The number of times a client has to communicate with the server is used as the performance measure.

For V*-$k$NN, the communication cost is measured by the number of times the BF-$k$NN query is executed because other operations are local. For RIS-$k$NN, the communication cost is the number of times the query itself is executed, i.e., the number of $k$VD cells crossed.

Figure 15 shows the communication costs with the increasing $x$ values in the four datasets and two trajectory types. The default buffer size of 16 pages is used. Since the parameter $x$ does not apply to RIS-$k$NN, its communication cost is constant as $x$ changes. For all datasets, we can see that the communication cost of V*-$k$NN decreases as the value of $x$ increases. This conforms with the cost analysis in Section 7, which states that the retrieval cost of the V*-$k$NN with respect to $x$ is $x^{-1/2}$. The V*-$k$NN starts to outperform the RIS-$k$NN algorithm when $x$ is: 3 for the uniform dataset (U), 3 for the Zipfian dataset (Z), 6 for the California dataset (C), and 6 for the North-Eastern USA dataset (N).

The difference between the communication and total response time costs (Figure 10) is notable. Unlike the total response time, the communication cost measure disregards the CPU cost. Only the database-access count is considered, and thus there is no penalty for large values of $x$.

The next experiment is a study on how the communication cost changes as $k$ is varied from 5 to 20. For all datasets, $x$ is set to 9 and the buffer size is 16 pages. Since a larger value of $k$ produces denser Voronoi cells in the space, the cost of RIS-$k$NN increases as $k$ increases due to more frequent cell crossings. The communication cost of V*-$k$NN also has a positive correlation with $k$ as suggested by the analysis in Section 7. V*-$k$NN still outperforms RIS-$k$NN in all settings, as shown in Figure 16. The effect of $k$ on the communication and total costs are very similar. This is because $k$ has positive correlations with: communication, tree-traversal and computation costs.

## 8.4 Summary

The V*-$k$NN algorithm consistently outperforms the RIS-$k$NN algorithm for all settings using the measures of total response time, number of page accesses and communication costs. V*-$k$NN is also more scalable with increasing $k$ values. There is a tradeoff between CPU time and data retrieval costs for V*-$k$NN, controlled by the value of $x$. In other words, $x$ provides the ability to tune the query performance for different application domains.

## 9. CONCLUSIONS

In this paper, the V*-Diagram and the associated algorithm V*-$k$NN have been introduced to efficiently process moving $k$ nearest neighbor queries (M$k$NN). A key difference between the V*-Diagram and previous safe-region-based techniques for M$k$NN queries is that, previous techniques only utilizes the knowledge on the data while the V*-Diagram exploits also the query location and the knowledge of the current search space. As a result, the V*-Diagram is more economical in terms of both IO and CPU costs. We also showed that the V*-Diagram can be applied to other useful domains such as spatial networks. We performed an extensive experimental study and the results show that our algorithm outperforms the best existing technique by two orders of magnitude.

## 10. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments that improved our paper. This work is supported under the Australian Research Council's Discovery funding scheme (project number DP0880215).

## 11. REFERENCES

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an effi cient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[2] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *VLDB J.*, 15(3):229–249, 2006.

[3] H. J. Cho and C. W. Chung. An effi cient and scalable approach to CNN queries in a road network. In *VLDB*, pages 865–876, 2005.

[4] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[5] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Symposium on Large Spatial Databases*, pages 83–95, 1995.

[6] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[7] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD*, pages 479–490, 2005.

[8] X. Huang, C. S. Jensen, H. Lu, and S. Saltenis. S-GRID: A versatile approach to effi cient query processing in spatial networks. In *SSTD*, pages 93–111, 2007.

[9] X. Huang, C. S. Jensen, and S. Saltenis. The islands approach to nearest neighbor querying in spatial networks. In *SSTD*, pages 73–90, 2005.

[10] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B$^+$-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[11] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, pages 840–851, 2004.

[12] M. R. Kolahdouzan and C. Shahabi. Alternative solutions for continuous k nearest neighbor queries in spatial network databases. *GeoInformatica*, 9(4):321–341, 2005.

[13] L. Kulik and E. Tanin. Incremental rank updates for moving query points. In *GIScience*, pages 251–268, 2006.

[14] A. Okabe, B. Boots, and K. Sugihara. *Spatial tessellations: concepts and applications of Voronoi diagrams*. John Wiley & Sons, Inc., 1992.

[15] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.

[16] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[17] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–55, 2008.

[18] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, pages 79–96, 2001.

[19] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD*, pages 334–345, 2002.

[20] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, 2002.

[21] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An effi cient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. Knowl. Data Eng.*, 16(10):1169–1184, 2004.

[22] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.

[23] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, pages 443–454, 2003.