Noname manuscript No.
(will be inserted by the editor)

Analysis and Evaluation of V*-kNN: An Efficient Algorithm for Moving kNN Queries

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

the date of receipt and acceptance should be inserted later

Abstract The *moving k nearest neighbor (MkNN) query* continuously finds the k nearest neighbors of a moving query point. MkNN queries can be efficiently processed through the use of safe regions. In general, a safe region is a region within which the query point can move without changing the query answer. This paper presents an incremental safe-region-based technique for answering MkNN queries, called the *V*-Diagram*, as well as analysis and evaluation of its associated algorithm, V*-kNN. Traditional safe-region approaches compute a safe region based on the data objects but independent of the query location. Our approach exploits the knowledge of the query location and the boundary of the search space in addition to the data objects. As a result, V*-kNN has much smaller I/O and computation costs than existing methods. We further provide cost models to estimate the number of data accesses for V*-kNN and a competitive technique, RIS-kNN. The V*-Diagram and V*-kNN are also applicable to the domain of spatial networks and we present algorithms to construct a spatial-network V*-Diagram. Our experimental results show that V*-kNN significantly outperforms the competitive technique. The results also verify the accuracy of the cost models.

Keywords Spatial databases, Nearest neighbor search.

1 Introduction

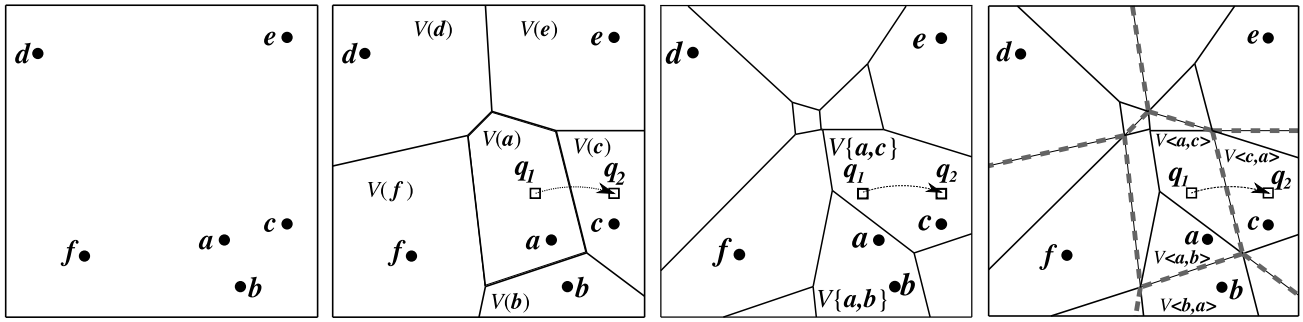
Current location-based services provide accurate position information with a high degree of temporal precision. Consider the following two scenarios. A driver in a GPS-equipped car issues a continuous query to find the nearest gas station while driving in a city. A tourist uses a location-aware mobile device to issue a continuous query for the nearest restaurant. 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* [33] and the scenarios above are typical examples of *moving k nearest neighbor queries* (MkNN).

A straightforward way to process a MkNN query is using a *sampling-based* method, which processes the MkNN query as a k nearest neighbor (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* [20]. 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 $\mathcal{S}$ of points $\boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_n}$ is defined as a set of Voronoi cells where each cell $V(\boldsymbol{p_i})$ is a set $\mathcal{V}$ of locations such that $\boldsymbol{p_i}$ is the nearest object to any location in $\mathcal{V}$.

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 Fig-

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



(a) Point set S, $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}\}$ (b) Voronoi Diagram of S (c) Second-order Voronoi Diagram of S (d) Ordered second-order Voronoi Diagram of S

Fig. 1 The Voronoi diagram and its generalizations

ure 1(b), for example, $\boldsymbol{q_1}$ falls in $V(\boldsymbol{a})$ and therefore $\boldsymbol{a}$ is the nearest neighbor (NN) 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.

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

Another useful generalization of the Voronoi Diagram is order sensitivity. As shown in Figure 1(d), the *ordered* kVD partitions the space into cells where each cell is associated to a particular ordering of the k NNs. The ordered kVD can be used to answer MkNN queries that require the ranking of the k NNs by their distances to the query point.

Although a kVD for any value of k can be constructed to help process MkNN queries, the technique has the following shortcomings:

(i) **Expensive precomputation.** The kVD requires precomputing all kVD cells and access to all data points. Both computation and storage costs are high.
(ii) **No support for dynamically changing k values.** The kVD can only accommodate kNN queries with a specific k value; the ordered kVD can only accommodate kNN 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.
(iii) **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 kVD for the entire city. Furthermore, one may require different kVDs 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 kVDs for all possible scenarios is considered prohibitive.

Zhang et al. [33] proposed an algorithm called *Retrieve-Influence-Set kNN* (RIS-kNN) to locally compute a kVD 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). This algorithm is used as our competitive method.

In this paper, we present a technique called the *V*-Diagram*, which is an incremental safe-region technique for processing MkNN queries. Our technique has the following key advantages.

(i) It requires no precomputation.
(ii) It gracefully handles dynamically changing k values.
(iii) It efficiently adapts to changes in the dataset, i.e., insertions and deletions of objects.

The V*-Diagram is based on the safe-region concept, but differs from any previous technique for MkNN queries in the following aspect: previous safe-region-based techniques compute safe regions purely based on the data (for example, one can compute the kVD 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.

(i) We present the V*-Diagram technique and the associated algorithm, V*-kNN, to process MkNN queries.
(ii) We provide detailed algorithms to construct the V*-Diagram in spatial networks and conduct experimental studies on a spatial-network implementation of the V*-kNN algorithm.

(iii) We derive cost models for V*-kNN and a competitive method, RIS-kNN, and verify that the cost models are accurate.
(iv) We conduct an extensive experimental study. The experimental results show that V*-kNN significantly outperforms RIS-kNN.

This paper is an extended version of our previous paper [19]. In the previous paper, we introduced the basic concept of the V*-Diagram, formulated the V*-kNN algorithm, and gave a complexity analysis on V*-kNN and RIS-kNN. We also showed an example of how the V*-Diagram can be applied to the domain of spatial networks. In this paper, we provide cost models which can accurately capture the number of data accesses for V*-kNN and RIS-kNN. We formally present algorithms to construct the V*-Diagram in spatial networks. Furthermore, we extend the experimental study to cover a large set of parameters as well as the V*-Diagram in spatial networks, and to verify the cost models.

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. We present V*-kNN, which is an algorithm for MkNN queries based on the V*-Diagram, in Section 5. Cost models and complexity analyses of the V*-kNN and the competitive technique RIS-kNN are given in Section 6. In Section 7, we show how the V*-Diagram technique can be applied to the domain of spatial networks. Section 8 presents experimental results. Section 9 concludes the paper.

2 Problem Setup

We assume a metric space, i.e., for any points $\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3}$ and the distance function $\textsc{Dist}(.)$, all of the following conditions are satisfied.

(i) $\textsc{Dist}(\boldsymbol{p_1}, \boldsymbol{p_2}) \geq 0$.
(ii) $\textsc{Dist}(\boldsymbol{p_1}, \boldsymbol{p_2}) = \textsc{Dist}(\boldsymbol{p_2}, \boldsymbol{p_1})$.
(iii) $\textsc{Dist}(\boldsymbol{p_1}, \boldsymbol{p_2}) = 0$ *iff* $\boldsymbol{p_1} = \boldsymbol{p_2}$.
(iv) $\textsc{Dist}(\boldsymbol{p_1}, \boldsymbol{p_2}) \leq \textsc{Dist}(\boldsymbol{p_1}, \boldsymbol{p_3}) + \textsc{Dist}(\boldsymbol{p_3}, \boldsymbol{p_2})$.

We focus our discussion on the L_2 normed vector space (the Euclidean space) and the spatial-network model. The kNN query in other metric spaces, e.g., L_1 and L_∞, is generally discussed in the context of similarity search in high dimensional data spaces [3, 4]. Application of the MkNN query and the V*-Diagram in such a problem is beyond the scope of this paper and will be a subject of further investigation.

Let $\mathcal{D}$ be a metric data space and $\mathcal{S}$ be a finite set of point objects. The k nearest neighbor (kNN) query is defined as follows.

Definition 1 (k Nearest Neighbor (kNN) Query) *Given a set $\mathcal{S}$ of objects and a query point $\boldsymbol{q}$, the kNN query finds a set $\mathcal{A}$ of objects such that: (i) $\mathcal{A}$ contains k objects from $\mathcal{S}$; (ii) for any object $\boldsymbol{x} \in \mathcal{A}$ and object $\boldsymbol{y} \in (\mathcal{S} - \mathcal{A})$, $\textsc{Dist}(\boldsymbol{q}, \boldsymbol{x}) \leq \textsc{Dist}(\boldsymbol{q}, \boldsymbol{y})$.*

The *moving kNN (MkNN) query* is defined as follows.

Definition 2 (Moving kNN (MkNN) Query) *Given a set $\mathcal{S}$ of objects and a moving query point $\boldsymbol{q}$, the MkNN query finds a kNN result for every position of $\boldsymbol{q}$.*

Due to the nature of location-based applications, MkNN 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 total 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 the communication cost and the query processing cost on the server side. In mobile applications, the former one is generally more important than the latter, since such settings usually have a high latency.

We assume an unknown trajectory which means that the location of $\boldsymbol{q}$ gets updated in a periodic manner. We also assume that no kVD is maintained but there is a generic spatial index such as the R-tree [7] 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 related work [33].

3 Related Work

3.1 kNN algorithms

Many kNN 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 structure. A tree can be traversed in a depth-first (*DF-kNN*) [22] or a best-first (*BF-kNN*) [9] manner. BF-kNN can incrementally retrieve more nearest neighbors if k increases.

Figure 2 shows an example R-tree and the BF-kNN algorithm. An R-tree [7] consists of a hierarchy of *minimum bounding rectangles* (MBRs), where each corresponds to a tree node and bounds all the MBRs in its sub-tree. Data objects are stored in leaf nodes and they are partitioned based on a heuristic that aims to minimize the I/O cost. The BF-kNN algorithm starts processing a kNN query from the root node and traverses all the entries in increasing order according to the MINDIST metric [23]. A priority queue is used to maintain the search order of all retrieved data points and nodes. The traversal stops if the first k elements retrieved from the priority queue are all data points.

An example run of BF-kNN is shown in Table 1. For ease of exposition, the priority queue is illustrated as a list in this example. At Step 1, all child nodes of the root are inserted into the priority queue. Then the entries are dequeued and the corresponding nodes are retrieved in order. The first

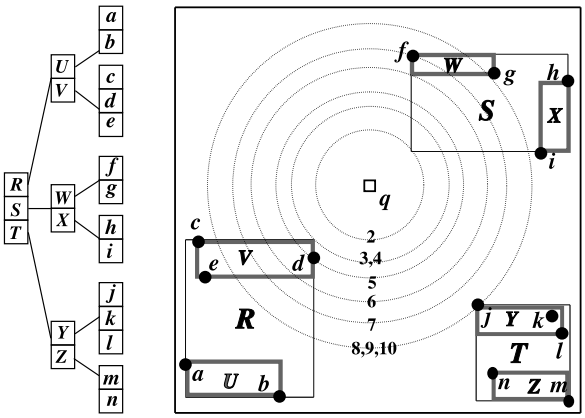


Fig. 2 R-tree and BF-kNN

Table 1 Example run of BF-kNN

Step	Priority Queue	Retrieved NN(s)
1	$\langle S, R, T\rangle$	$\langle\rangle$
2	$\langle R, W, T, X\rangle$	$\langle\rangle$
3	$\langle V, W, T, X, U\rangle$	$\langle\rangle$
4	$\langle \boldsymbol{d}, W, T, X, \boldsymbol{c}, \boldsymbol{e}, U\rangle$	$\langle\rangle$
5	$\langle W, T, X, \boldsymbol{c}, \boldsymbol{e}, U\rangle$	$\langle \boldsymbol{d}\rangle$
6	$\langle \boldsymbol{f}, T, \boldsymbol{g}, X, \boldsymbol{c}, \boldsymbol{e}, U\rangle$	$\langle \boldsymbol{d}\rangle$
7	$\langle T, \boldsymbol{g}, X, \boldsymbol{c}, \boldsymbol{e}, U\rangle$	$\langle \boldsymbol{d}, \boldsymbol{f}\rangle$
8	$\langle Y, \boldsymbol{g}, X, \boldsymbol{c}, \boldsymbol{e}, U, Z\rangle$	$\langle \boldsymbol{d}, \boldsymbol{f}\rangle$
9	$\langle \boldsymbol{j}, \boldsymbol{g}, X, \boldsymbol{c}, \boldsymbol{e}, U, \boldsymbol{k}, Z, \boldsymbol{l}\rangle$	$\langle \boldsymbol{d}, \boldsymbol{f}\rangle$
10	$\langle \boldsymbol{g}, X, \boldsymbol{c}, \boldsymbol{e}, U, \boldsymbol{k}, Z, \boldsymbol{l}\rangle$	$\langle \boldsymbol{d}, \boldsymbol{f}, \boldsymbol{j}\rangle$

dequeued item is S. The two child entries X and W of S are inserted into the priority queue. Then R is dequeued. Nodes U and V are inserted into the priority queue and so on. At Step 4, $\boldsymbol{d}$ is the head of the priority queue. Data point $\boldsymbol{d}$ is dequeued and inserted into the list of retrieved NNs as the first NN. If another NN is needed, the process continues until another data point is the head of the priority queue. 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 entries in the priority queue to reduce the queue size, though the page access cost cannot be further reduced due to the search-space optimality of the algorithm [10].

3.2 Techniques for processing MkNN queries

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

SR-kNN. Song and Roussopoulos [25] introduced a method which will be referred to as *SR-kNN* in this paper. SR-kNN 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-kNN does not provide *truly continuous* answers.

The k^{th}-order Voronoi Diagram (kVD). Safe-region based techniques produce continuous answers and reduce processing and communication costs. The Voronoi Diagram [20] is a classic example and can be used to process M1NN queries as described in Section 1. For MkNN queries, the kVD is used. There is a difference between the order-sensitive kVD and the order-insensitive kVD, which produce order-sensitive and order-insensitive k NNs, respectively [20]. However, they both have the shortcomings described in Section 1.

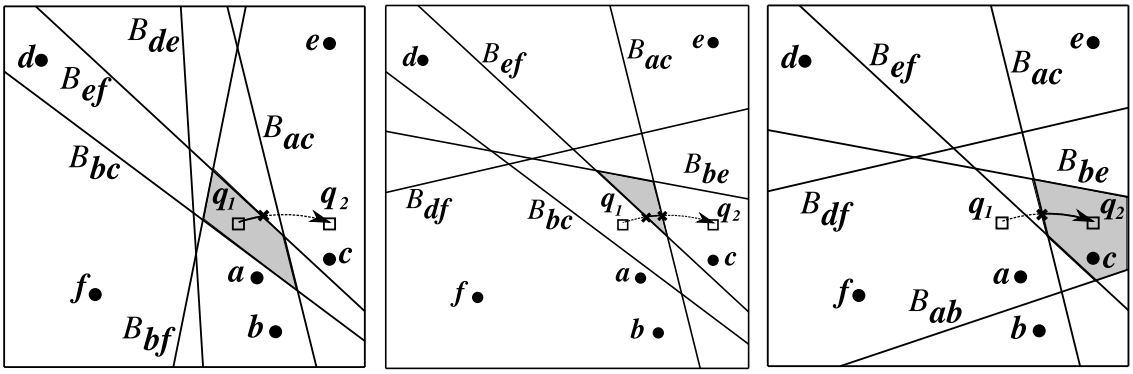
TPkNN and CkNN. Tao and Papadias [27] proposed the *time-parameterized kNN (TPkNN) query*. Assuming a linear trajectory of the query point, a TPkNN query finds (i) the current kNN set, (ii) a position on the trajectory where the kNN set changes and (iii) the objects that cause the change. This is done by finding the earliest point on the trajectory that has a different kNN 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 [29] also considered the *Continuous kNN* (CkNN) query, which finds the kNN for every single point on a predefined linear trajectory. This is achieved by identifying all influence points on the trajectory. The main difference between CkNN and TPkNN is that CkNN obtains all the influence points on the trajectory but TPkNN finds only the first one. Both TPkNN and CkNN are limited to known trajectories.

RIS-kNN. Zhang et al. [33] proposed an algorithm called *Retrieve-Influence-Set kNN* (*RIS-kNN*) to locally compute kVD cells using a spatial index. RIS-kNN uses the *TPkNN query* [27] to find each edge of a kVD cell, 360 degrees around the query point.

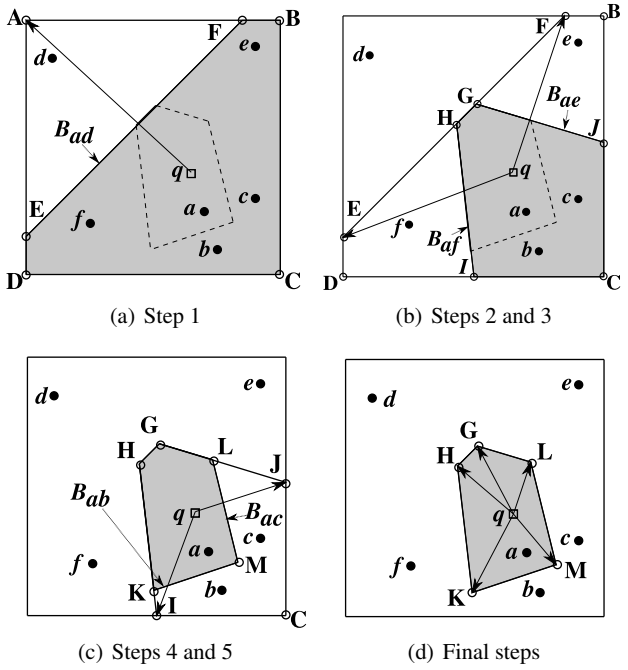
Figure 4 shows how RIS-kNN discovers all edges of $V(\boldsymbol{a})$ from the example in Figure 1(a). Initially, $\boldsymbol{a}$ is found to be the NN of $\boldsymbol{q}$; the cell is initialized to the whole data space, i.e., rectangle $\boldsymbol{ABCD}$ in this example. At Step 1, a TPkNN query 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 bisector[1] 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 TPkNN 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 kVD cells for data points are convex polygons, this process continues until all corners of the cell have been checked and they all have the same k NNs as $\boldsymbol{q}$. The influence set, which the set of (non-kNN) objects that determines the discovered bisectors, is $\{\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}\}$.

[1] The *bisector* of two objects $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ is the set of points $\boldsymbol{v}$ such that $\boldsymbol{v}$ is equidistant to $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$.



(a) Initial step: $F\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f}, \boldsymbol{e}, \boldsymbol{d}\rangle$ (b) First update: $F\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{d}\rangle$ (c) Second update: $F\langle \boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{d}\rangle$

Fig. 3 Incremental rank update



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

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

Fig. 4 Using RIS-kNN to locally compute a kVD cell ($k = 1$)

RIS-kNN mitigates the precomputation problem (Shortcoming 1) of the kVD because it only accesses local data, but this algorithm is still expensive because it performs multiple (on average 12 [33]) TPkNN queries, where each TPkNN query involves a costly tree traversal. RIS-kNN does not solve the problem of dynamically changing k values (Shortcoming 2); changing k to a larger value incurs recalculation of the kVD cell. The computation of the previous kVD cell cannot be reused and hence this algorithm is not *incremental*.

Due to the fact that only one kVD cell is maintained at a time, RIS-kNN handles dataset updates (insertions and deletions of objects) more efficiently than the traditional kVD technique. However, in a case where an update affects the kNN result or the influence set, the current kVD cell has to be recalculated.

IRU. Kulik and Tanin [17] 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 kVD cell with k equal to n, where n is the total number of objects. Rather than computing the whole nVD, IRU incrementally computes a neighboring nVD cell from the current cell. In the paper [17], an order-sensitive nVD cell is termed a *fixed-rank region* (FRR) since for any point in the region, the ranking of all objects based on their distances is fixed. Based on the observation that only rank-adjacent objects can swap their ranks[2], defining the FRR of n objects requires at most $(n-1)$ bisectors of the $(n-1)$ pairs of rank-adjacent objects. Continuous monitoring of the ranking of all 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 the query point crosses a bisector, the ranks of the two corresponding objects are swapped and the list of rank-adjacent bisectors are updated.

An example is given in Figure 3, where the gray region is the current FRR that the moing query point is in. Let us assume that the query point $\boldsymbol{q}$ starts at $\boldsymbol{q_1}$ and stops at $\boldsymbol{q_2}$. In Figure 3(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 corresponding list of bisectors is $\langle B_{\boldsymbol{ac}}, B_{\boldsymbol{cb}}, B_{\boldsymbol{bf}}, B_{\boldsymbol{ef}}, B_{\boldsymbol{de}}\rangle$. Then $\boldsymbol{q}$ crosses $B_{\boldsymbol{ef}}$ in Figure 3(b). This causes $\boldsymbol{e}$ and $\boldsymbol{f}$ to swap their ranks. Therefore $B_{\boldsymbol{bf}}$ and $B_{\boldsymbol{de}}$ are replaced by $B_{\boldsymbol{be}}$ and $B_{\boldsymbol{df}}$, respectively. In Figure 3(c), $B_{\boldsymbol{ac}}$ is crossed. This causes $\boldsymbol{a}$ and $\boldsymbol{c}$ to swap their ranks and $B_{\boldsymbol{bc}}$ is replaced by $B_{\boldsymbol{ab}}$. It is shown that only $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2)$ bisectors are maintained during

[2] 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.

the iterations in the IRU algorithm [17]. However, IRU still accesses all data objects to obtain the sorted list and checks $(n-1)$ bisectors every time $\boldsymbol{q}$ moves.

Related spatial-network queries. Several kNN techniques for static query points were proposed [12, 13, 21, 24] for spatial networks. Papadias et al. [21] proposed *incremental Euclidean restriction* (IER) and *incremental network expansion* (INE) methods based on Dijkstra's shortest-path algorithm [6]. There are also MkNN techniques specific to the domain of spatial networks. Kolahdouzan et al. [16] proposed an algorithm that utilizes the *Voronoi Network Nearest Neighbor* (VN^3) [15]. Cho et al. [5] proposed a technique that issues static kNN queries at the intersection points on the query path.

Related moving-object queries. Hu et al. [11] proposed a safe-region-based technique for *static* window and kNN queries on moving objects. Each moving object maintains its own safe region and only reports if its new location changes the results of any of the queries.

Yu et al. [32] presented a query-indexing technique for monitoring kNN queries for moving objects and a given set of queries. Each query object maintain its own *critical region* to keep track of the kNN set.

Mouratidis et al. [18] proposed a threshold-based approach to monitoring the k NNs in a setting for moving objects. Each monitored object is associated with a range of safe distances from the query point. An object cannot influence the query answer as long as it remains within the range of safe distances.

Iwerks et al. [14] formulated an approach to processing a kNN query on moving objects by continuous evaluation of a range query. By imposing a condition that the scope of the range query must include the $(k+1)$ NNs, the k NNs can be evaluated based on only objects within the scope.

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 moving-object kNN techniques focus on monitoring changes caused by location updates of data objects. The emphasis of the MkNN techniques, on the other hand, is on the changes caused by location updates of the query point. Although the problem of moving query points was also addressed by Mouratidis et al. [18] and Iwerks et al. [14], their techniques handle query location updates through insertions and deletions of query objects, and recalculations of distances with respect to the new query locations. In contrast, the V*-Diagram and the chosen competitive technique, RIS-kNN [33], use safe regions to handle moving query points so they require no distance recalculations. In this paper, we focus our comparative study on RIS-kNN, since the method is customized for the setting of moving query points on static data objects using safe regions.

3.3 Summary

The MkNN techniques are summarized in Table 2 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*-kNN, has all these features.

Table 2 Comparison of MkNN techniques

Technique	Continuous	Incremental	Local access	Unknown path	Order-sensitive
SR-kNN [25]	×	✓	✓	✓	✓
kVD [20]	✓	×	×	✓	×
Ordered kVD [20]	✓	×	×	✓	✓
TPkNN [27]	✓	×	✓	×	×
CkNN [29]	✓	×	✓	×	×
RIS-kNN [33]	✓	×	✓	✓	×
IRU [17]	✓	✓	×	✓	✓
V*-kNN	✓	✓	✓	✓	✓

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 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 incrementally retrieve NNs as discussed in Section 3.1.

The V*-Diagram ensures the correctness of the k NNs to the moving query point $\boldsymbol{q}$ by maintaining x auxiliary objects in addition to the regular k NNs at all times. 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 kNN safe region, discussed in Section 4.3. Commonly used symbols are summarized in Table 3.

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-kNN 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 according to the steps in Table 1.

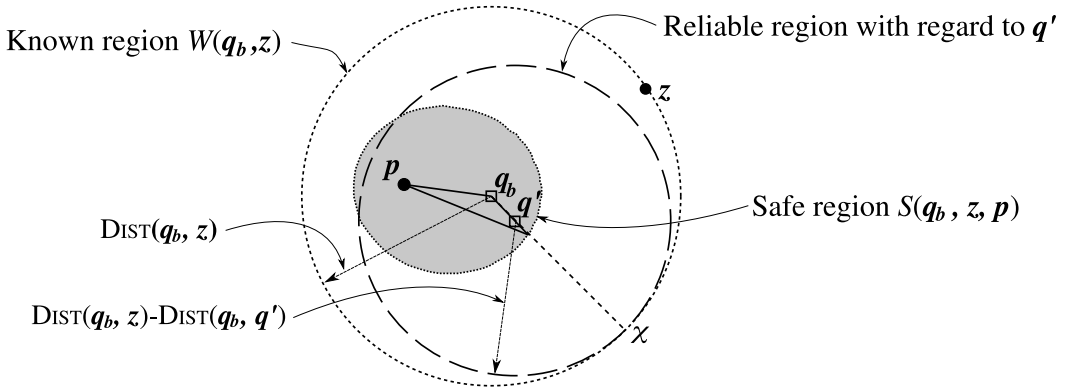


Fig. 5 The known, reliable, and safe regions

Table 3 Symbols

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

For example, sphere 2 corresponds to Step 2 where node S is retrieved; sphere 5 corresponds to Step 5 where object $\boldsymbol{d}$ 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-kNN is repeatedly called to ensure that the $(k+x)$ maintained objects always include the k NNs to $\boldsymbol{q}$. Let $\boldsymbol{q_b}$ be the position of $\boldsymbol{q}$ where the latest BF-kNN call is made to retrieve the $(k+x)$ NNs to $\boldsymbol{q_b}$. Let $\boldsymbol{z}$ be the $(k+x)^{th}$ NN to $\boldsymbol{q_b}$. The search sphere corresponding to $\boldsymbol{z}$ centered at $\boldsymbol{q_b}$ is the latest one (since BF-kNN stops when $\boldsymbol{z}$ is obtained), and we call this search sphere the *known region*, denoted by $W(\boldsymbol{q_b}, \boldsymbol{z})$. We highlight $\boldsymbol{z}$ because it determines the boundary of the known region.

Figure 5 gives an example. The known region $W(\boldsymbol{q_b}, \boldsymbol{z})$ is actually a sphere centered at $\boldsymbol{q_b}$ with the radius $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{z})$. Point object $\boldsymbol{p}$ is one of the $(k+x-1)$ NNs of $\boldsymbol{q_b}$ and other objects in $W(\boldsymbol{q_b}, \boldsymbol{z})$ are not shown.

4.1 Safe region with regard to a data object

Next, we formulate a region within which $\boldsymbol{q}$ can move while $\boldsymbol{p}$ remains one of the $(k+x)$ NNs of $\boldsymbol{q}$. Let $\boldsymbol{q'}$ denote a later position of $\boldsymbol{q}$ after $\boldsymbol{q_b}$. Suppose $\boldsymbol{q'}$ is at the position as shown in Figure 5. We extend the line segment $\overline{\boldsymbol{q_b q'}}$ and it intersects $W(\boldsymbol{q_b}, \boldsymbol{z})$ at χ. Let $sph(\boldsymbol{v}, l)$ denote a sphere with center $\boldsymbol{v}$ and radius l. As long as $\boldsymbol{p}$ is in $sph(\boldsymbol{q'}, \text{DIST}(\boldsymbol{q'}, \boldsymbol{\chi}))$, it is one of the $(k+x)$ NNs of $\boldsymbol{q'}$. This is because any object outside $sph(\boldsymbol{q'}, \text{DIST}(\boldsymbol{q'}, \boldsymbol{\chi}))$ must be farther to $\boldsymbol{q'}$ than $\boldsymbol{p}$ to $\boldsymbol{q'}$ and there are at most $(k+x)$ objects inside $sph(\boldsymbol{q'}, \text{DIST}(\boldsymbol{q'}, \boldsymbol{\chi}))$. Since any object in $sph(\boldsymbol{q'}, \text{DIST}(\boldsymbol{q'}, \boldsymbol{\chi})$ remains one of the $(k+x)$ NNs of $\boldsymbol{q'}$, we call $sph(q', \text{DIST}(\boldsymbol{q'}, \boldsymbol{\chi}))$ the *reliable region* with regard to $\boldsymbol{q}$ [3] and any object in the reliable region a *reliable object*. If $\boldsymbol{p}$ is a reliable object, $\boldsymbol{p}$ is said to be *reliable*; otherwise, it is said to be *unreliable*. Mathematically, $\boldsymbol{p}$ being in the reliable region with regard to $\boldsymbol{q'}$ is expressed as

$$\text{DIST}(\boldsymbol{q'}, \boldsymbol{p}) \leq \text{DIST}(\boldsymbol{q_b}, \boldsymbol{z}) - \text{DIST}(\boldsymbol{q_b}, \boldsymbol{q'}), \qquad (1)$$

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

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

Definition 3 (Safe region with regard to a data object) *Given a known region $W(\boldsymbol{q_b}, \boldsymbol{z})$ and a data object $\boldsymbol{p}$ within $W(\boldsymbol{q_b}, \boldsymbol{z})$, the safe region with regard to $\boldsymbol{p}$ is*

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

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

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

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

Proof For any $\boldsymbol{q'}$ in $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{z})$,

$$\text{DIST}(\boldsymbol{q'}, \boldsymbol{z}) + \text{DIST}(\boldsymbol{q_b}, \boldsymbol{q'}) \leq \text{DIST}(\boldsymbol{q_b}, \boldsymbol{z}). \quad (2)$$

By the triangle inequality, we have $\text{DIST}(\boldsymbol{q'}, \boldsymbol{z}) + \text{DIST}(\boldsymbol{q_b}, \boldsymbol{q'}) \geq \text{DIST}(\boldsymbol{q_b}, \boldsymbol{z})$. The set of points that satisfies both inequalities is $\{\boldsymbol{q'} : \text{DIST}(\boldsymbol{q'}, \boldsymbol{z}) + \text{DIST}(\boldsymbol{q_b}, \boldsymbol{q'}) = \text{DIST}(\boldsymbol{q_b}, \boldsymbol{z})\}$. In Euclidean space, this is the line segment $\overline{\boldsymbol{q_b z}}$. □

Corollary 2 *Object $\boldsymbol{p}$ is reliable to $\boldsymbol{q'}$ iff $\boldsymbol{q'}$ is inside of $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$.*

Proof We prove this by showing that for any object $\boldsymbol{y}$ outside $W(\boldsymbol{q_b}, \boldsymbol{z})$, $\text{DIST}(\boldsymbol{q'}, \boldsymbol{p})$ is smaller than $\text{DIST}(\boldsymbol{q'}, \boldsymbol{y})$. The two conditions: (i) $\boldsymbol{q'}$ inside of $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p})$; (ii) $\boldsymbol{y}$ outside of $W(\boldsymbol{q_b}, \boldsymbol{z})$, imply that

$$\text{DIST}(\boldsymbol{q'}, \boldsymbol{p}) + \text{DIST}(\boldsymbol{q_b}, \boldsymbol{q'}) < \text{DIST}(\boldsymbol{q_b}, \boldsymbol{y}).$$

The triangle inequality dictates that

$$\text{DIST}(\boldsymbol{q_b}, \boldsymbol{y}) < \text{DIST}(\boldsymbol{q'}, \boldsymbol{y}) + \text{DIST}(\boldsymbol{q_b}, \boldsymbol{q'}).$$

We can conclude that $\text{DIST}(\boldsymbol{q'}, \boldsymbol{p})$ is smaller than $\text{DIST}(\boldsymbol{q'}, \boldsymbol{y})$, and hence $\boldsymbol{p}$ is reliable to $\boldsymbol{q'}$. □

4.2 Fixed-rank region

As discussed in Section 3.2, the fixed-rank region (FRR) and the incremental rank update (IRU) algorithm were introduced in [17] to denote a set of possible query-point locations that share a specific ranking of all objects. In this paper, we use the IRU algorithm to maintain the ranks of the $(k + x)$ objects retrieved.

Given a list L of objects, $\langle \boldsymbol{p_1}, ..., \boldsymbol{p_m} \rangle$, the FRR of L is the set of all points v such that the objects in L are sorted in ascending order according to their distances to $\boldsymbol{v}$. Let $H_{\boldsymbol{p_i p_j}}$ be defined as a set of points $\{\boldsymbol{v} \in \mathcal{D} : \text{DIST}(\boldsymbol{v}, \boldsymbol{p_i}) \leq \text{DIST}(\boldsymbol{v}, \boldsymbol{p_j})\}$, where $\mathcal{D}$ is the data space. An FRR is a function of a list of objects and is defined as follows.

Definition 4 (Fixed-rank region)

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

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

In Figure 3(a), the ranking of the objects according to their distances to $\boldsymbol{q_1}$ is $L = \langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f}, \boldsymbol{e}, \boldsymbol{d} \rangle$. $F(L)$ is defined as $H_{\boldsymbol{ac}} \cap H_{\boldsymbol{cb}} \cap H_{\boldsymbol{bf}} \cap H_{\boldsymbol{fe}} \cap H_{\boldsymbol{ed}}$. The boundary of $F(L)$ is defined by five bisectors, $B_{\boldsymbol{ac}}$, $B_{\boldsymbol{cb}}$, $B_{\boldsymbol{bf}}$, $B_{\boldsymbol{ef}}$ and $B_{\boldsymbol{de}}$.

We use the IRU algorithm described in Section 3.2 to incrementally compute the FRR $F(L)$ in which $\boldsymbol{q}$ currently resides for the $(k + x)$ maintained objects. A FRR is represented as (i) a list B of the $(k + x - 1)$ rank-adjacent bisectors, $B_{\boldsymbol{p_i p_{i+1}}}$, for $i = 1, 2, ..., k + x - 1$, and (ii) a reference point, which could be the current location of $\boldsymbol{q}$. The FRR is incrementally maintained by (i) checking whether $\boldsymbol{q}$ 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 $\boldsymbol{q}$. One alternative solution to IRU is to compute distances between the $(k + x)$ objects and $\boldsymbol{q}$ for every position of $\boldsymbol{q}$, 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) kNN does not change.

4.3 Integrated safe region

We are now ready to formulate the safe region for the MkNN 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 MkNN safe-region requirements.

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

$$F(L) \cap (\bigcap_{i=1}^{k} S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i})). \quad (3)$$

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

Lemma 1

$$F\langle \boldsymbol{p_i}, \boldsymbol{p_j} \rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_j}) \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i}) = F\langle \boldsymbol{p_i}, \boldsymbol{p_j} \rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_j}).$$

Proof For any point $\boldsymbol{v} \in F\langle \boldsymbol{p_i}, \boldsymbol{p_j} \rangle$, $\boldsymbol{v}$ satisfies

$$\text{DIST}(\boldsymbol{v}, \boldsymbol{p_i}) \leq \text{DIST}(\boldsymbol{v}, \boldsymbol{p_j}). \quad (4)$$

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

$$\text{DIST}(\boldsymbol{v}, \boldsymbol{p_j}) + \text{DIST}(\boldsymbol{q_b}, \boldsymbol{v}) \leq \text{DIST}(\boldsymbol{q_b}, \boldsymbol{z}). \quad (5)$$

For any $\boldsymbol{v} \in F\langle \boldsymbol{p_i}, \boldsymbol{p_j} \rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i})$, it satisfies inequalities (4) and (5). By adding the two inequalities,

$$\text{DIST}(\boldsymbol{v}, \boldsymbol{p_i}) + \text{DIST}(\boldsymbol{q_b}, \boldsymbol{v}) \leq \text{DIST}(\boldsymbol{q_b}, \boldsymbol{z}). \quad (6)$$

Inequality (6) shows that $\boldsymbol{v} \in S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i})$, given that $\boldsymbol{v} \in F\langle \boldsymbol{p_i}, \boldsymbol{p_j} \rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_j})$. It can be concluded that: $F\langle \boldsymbol{p_i}, \boldsymbol{p_j} \rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_j}) \subseteq S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i})$ and hence $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i})$ can be discarded in $F\langle \boldsymbol{p_i}, \boldsymbol{p_j} \rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_j}) \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_i})$. □

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

Theorem 1 *For $k \geq 2$,*

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

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,

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

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

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

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

$$F\langle \boldsymbol{p_1}, ..., \boldsymbol{p_l}\rangle \cap F\langle \boldsymbol{p_l}, \boldsymbol{p_{l+1}}\rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_{l+1}}) = F\langle \boldsymbol{p_1}, ..., \boldsymbol{p_{l+1}}\rangle \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_{l+1}}).$$

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

Since $F(L) = F\langle \boldsymbol{p_1}, ..., \boldsymbol{p_k}\rangle \cap F\langle \boldsymbol{p_k}, \boldsymbol{p_{k+1}} ..., \boldsymbol{p_{k+x}}\rangle$, based on Theorem 1, Expression (3) 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 5 (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*

$$I(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k}, L) = F(L) \cap S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k}). \quad (7)$$

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

Theorem 2 *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 5, (i) 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; (ii) 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, all points $\boldsymbol{q'}$ in I share the same order-sensitive k NNs. □

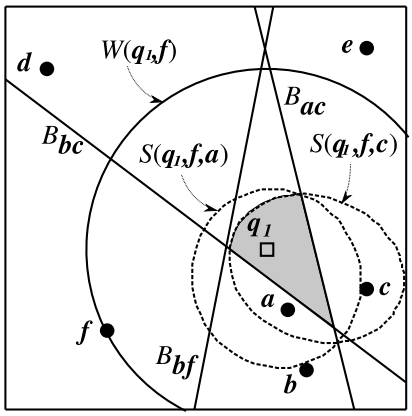


Fig. 6 Integrated safe region example ($k = 2$, $x = 2$)

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 a $(k + x)$NN search (BF-kNN) is performed. 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 gray 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*-kNN, an algorithm for MkNN 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*-kNN 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 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*-kNN continuously produces answers 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*-kNN($\boldsymbol{q_0}$, k, x)

```
1  q_b ← 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 ← 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 ≤ k then
12                 ReportResult(L.Head(k))
13             if Bisector.Index ∈ [k − 1, k] then
14                 p_k ← L.Item(k)
15                 S_k ← S(q_b, z, p_k)
16             ISR ← ConstructISR(S_k,B,q)
17         case ReliabilityUpdate
18             q_b ← 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-kNN 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 3. 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 3 when we did not consider deletions.

The continuous processing part of V*-kNN 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)

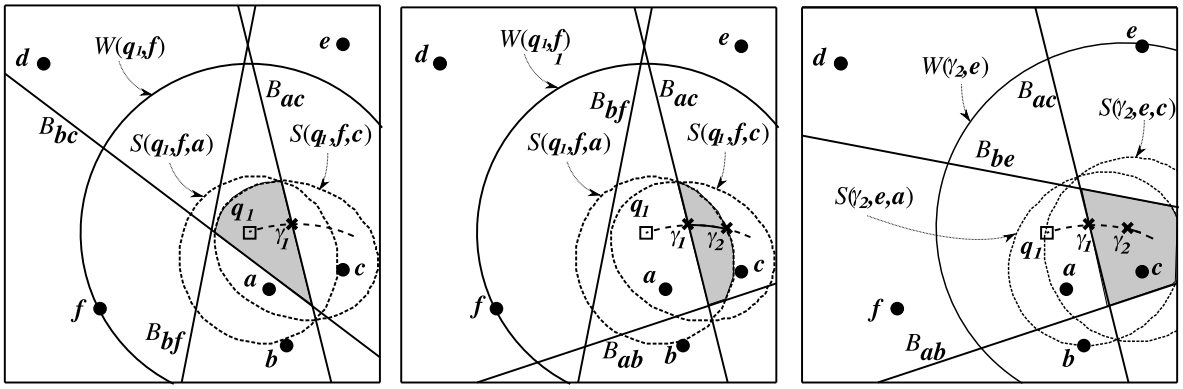
```
1  L ← BF-kNN(q_b, k + x)
2  z ← L.Item(k + x)
3  p_k ← L.Item(k)
4  S_k ← S(q_b, z, p_k)
5  B ← CreateBisectorList(L)
6  ISR ← ConstructISR(S_k,B,q_b)
7  return (L, z, S_k, B, ISR)
```

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 handlby reference to Algorithm 1 (in the algorithm, $\boldsymbol{q}$ is used to denote the position of the query point when an event takes place).

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 (Line 12). Moveover, if the rank update changes $\boldsymbol{p_k}$ (Line 13), S_k also needs to be updated (Lines 14 to 15).

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 num-



(a) Step 1: $F\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f} \rangle \cap S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{c})$ (b) Step 2: $F\langle \boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{f} \rangle \cap S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{a})$ (c) Step 3: $F\langle \boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{e} \rangle \cap S(\boldsymbol{\gamma_2}, \boldsymbol{e}, \boldsymbol{a})$

Fig. 7 Example for Algorithm 1, ($k = 2$, $x = 2$)

ber 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 kNN set nor their ordering changes.

Figure 7 gives an example run of the algorithm. According to Figure 7(a), 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 $F\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{b}, \boldsymbol{f} \rangle \cap S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{c})$.

Figure 7(b) 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 gray region). The current 2 NNs, $\boldsymbol{c}$ and $\boldsymbol{a}$, are reported to the user in that order.

Figure 7(c) shows how the ISR changes after $\boldsymbol{q}$ exits its $S(\boldsymbol{q_1}, \boldsymbol{f}, \boldsymbol{a})$. 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 gray 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 MkNN query. In this subsection, we discuss possible values of x, the number of auxiliary objects. Generally, we find that x should not assume the value of 0. This is explained as follows.

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 S_k with regard to $\boldsymbol{z}$ is the line segment $\overline{\boldsymbol{q_b z}}$. Unless $\boldsymbol{q}$ moves along $\overline{\boldsymbol{q_b z}}$, $\boldsymbol{q}$ exits S_k as it starts moving. Probabilistically, it is highly unlikely that $\boldsymbol{q}$ moves along $\overline{\boldsymbol{q_b z}}$ since $\overline{\boldsymbol{q_b z}}$ is just one direction among 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 repetitively. As a result, x should not be set to 0 for Euclidean space.

When x is a positive integer, the problem of premature triggering of the *ReliabilityUpdate* does not happen except under the coincidence described in the next paragraph. Therefore, any positive integer is a valid value for x. The effect of the value of x on performance is further investigated in Sections 6 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 $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{z})$. In this case, $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{p_k})$ is equal to $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{z})$. By definition, $S_k = \{\boldsymbol{q'} : \text{DIST}(\boldsymbol{q'}, \boldsymbol{p_k}) + \text{DIST}(\boldsymbol{q_b}, \boldsymbol{q'}) \leq \text{MINDIST}(\boldsymbol{q_b}, z)\}$. If we replace $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{z})$ by $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{p_k})$ in the inequality of the definition, we get $S_k = \{\boldsymbol{q'} : \text{DIST}(\boldsymbol{q'}, \boldsymbol{p_k}) + \text{DIST}(\boldsymbol{q_b}, \boldsymbol{q'}) \leq \text{DIST}(\boldsymbol{q_b}, \boldsymbol{p_k})\}$, which is also a line segment in Euclidean space. To completely avoid this problem, we can check whether $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{p_k})$ is equal to $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{z})$ after we retrieve $(k + x)$ objects by a BF-kNN call. If they are equal, then we increase the value of x until $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{p_k})$ is different from $\text{DIST}(\boldsymbol{q_b}, \boldsymbol{z})$.

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 frequent BF-kNN calls. On the other hand, a too large x value also incurs the overhead of retrieving more objects in every BF-kNN call and more computation for maintaining them.

5.2 Insertions and deletions of objects

In this subsection, we describe the algorithm to perform updates (that is, insertions and deletions) to the dataset for V*-kNN. 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 occurs 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 between $\boldsymbol{q}$ 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:

- **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 7 to 9). The new result is reported (Line 10).
- **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).
- **The update has no effects to the kNN set (but to one of the auxiliary objects) (Line 20).** Only the ISR is updated to reflect the change in B.

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

```
1  if p ∈ W(q_b, z) then
2      if Operation = Insertion then
3          L ← Insert(L,p,q)
4      else
5          L ← Delete(L,p)
6      B.Update(L)
7      if k > L.Length() then
8          q_b ← q
9          (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
10         ReportResult(L.Head(k))
11     else if DIST(q, p) ≤ DIST(q, p_k) then
12         p_k ← L.Item(k)
13         S_k ← S(q_b, z, p_k)
14         if q ∉ S_k then
15             q_b ← q
16             (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
17         else
18             ISR ← ConstructISR(S_k,B,q)
19         ReportResult(L.Head(k))
20     else
21         ISR ← ConstructISR(S_k,B,q)
```

5.3 MkNN with dynamically changing k values

The ability to gracefully handle changes to the value of k is crucial for the *distance browsing* functionality [10]. For static kNN 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 computations.

Algorithm *KUpdate* (Algorithm 4) shows how V*-kNN 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*-kNN 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      q_b ← q
3      (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
4  else
5      p_k ← L.Item(k)
6      S_k ← S(q, z, p_k)
7      if q ∉ S_k then
8          q_b ← q
9          (L, z, S_k, B, ISR) ← Compute-V*(q_b, k, x)
10     else
11         ISR ← ConstructISR(S_k,B,q)
12 ReportResult(L.Head(k))
```

6 Cost Models and Comparative Performance Analysis

In this section, we give a comparative performance analysis on V*-kNN and its best competitor, RIS-kNN [33]. For ease of exposition, we assume a 2D unit space. The analysis can be extended to higher dimensional spaces with arbitrary sizes by applying the same procedure. We also assume that the data objects are uniformly distributed points. In addition to the symbols shown in Table 3, frequently used symbols in this section and the experimental study (Section 8) are presented in Table 4. In our analysis, we focus on data access costs, since this is commonly the dominating factor in many application domains.

Table 4 Additional symbols

Symbol	Meaning
l	The total trajectory length.
s	The distance between two consecutive points of location updates.
$\boldsymbol{q_e}$	The point where the query point exits the safe region with regard to $\boldsymbol{p_k}$.
d_e	The distance between $\boldsymbol{q_e}$ and $\boldsymbol{q_b}$.
n_b	The number of BF-kNN calls.
n_v	The number of kVD cells crossed.

6.1 V*-kNN

V*-kNN requires data access every time a *ReliabilityUpdate* event takes place and triggers a BF-kNN call (Line 17 of Algorithm 1). Therefore, the number of data accesses for V*-kNN is measured as the number n_b of BF-kNN calls. A *ReliabilityUpdate* event is triggered every time the query point $\boldsymbol{q}$ exits $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k})$. At the exit point $\boldsymbol{q_e}$, BF-kNN is performed and $\boldsymbol{q_e}$ becomes the new $\boldsymbol{q_b}$. A *ReliabilityUpdate* event reoccurs when $\boldsymbol{q}$ exits $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k})$ again. Let d_e be the distance between $\boldsymbol{q_b}$ and $\boldsymbol{q_e}$. The number n_b of BF-kNN calls has a negative correlation with d_e. In the worst case, when $\boldsymbol{q}$ moves in a straight line, n_b is inversely proportional to d_e.

The value of d_e is estimated as follows. When $\boldsymbol{q}$ moves to $\boldsymbol{q_e}$, $\boldsymbol{p_k}$ is on the boundary of the reliable region (with regard to $\boldsymbol{q_e}$). In Figure 8, for example, $\boldsymbol{b}$ is $\boldsymbol{p_k}$ and $\boldsymbol{q_e}$ is on the boundary of $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{b})$. We can see that d_e, which is $\textsc{Dist}(\boldsymbol{q_b}, \boldsymbol{q_e})$, is given by

$$d_e = \textsc{Dist}(\boldsymbol{q_b}, \boldsymbol{\chi}) - \textsc{Dist}(\boldsymbol{q_e}, \boldsymbol{\chi}).$$

Note that $\textsc{Dist}(\boldsymbol{q_b}, \boldsymbol{\chi})$ is the radius of $W(\boldsymbol{q}, \boldsymbol{z})$, which is a sphere that contains $(k + x)$ points; $\textsc{Dist}(\boldsymbol{q_e}, \boldsymbol{\chi})$ is the radius of the reliable region with regard to $\boldsymbol{q_e}$, which is a sphere that contains k points.

We use the cost model proposed by Tao et al. [30] to estimate the distance between $\boldsymbol{q_b}$ and $\boldsymbol{q_e}$. According to the model, the distance between the query point $\boldsymbol{q_b}$ and the k^{th} NN (in a 2D unit space with n objects uniformly distributed) is estimated as $\frac{2}{\sqrt{\pi}}\left(1 - \sqrt{1 - \sqrt{\frac{k}{n}}}\right)$. Therefore d_e, i.e., $\textsc{Dist}(\boldsymbol{q_b}, \boldsymbol{\chi}) - \textsc{Dist}(\boldsymbol{q_e}, \boldsymbol{\chi})$, is given by

$$d_e = \frac{2}{\sqrt{\pi}}\left(\sqrt{1 - \sqrt{\frac{k}{n}}} - \sqrt{1 - \sqrt{\frac{k+x}{n}}}\right). \quad (8)$$

Another factor that may affect n_b is the directionality of the trajectory. We show our analysis on two typical trajectory types: *straight line* and *random*. For straight-line trajectories, n_b is inversely proportional to d_e, and therefore

$$n_b = l/d_e, \quad (9)$$

where l is the trajectory length.

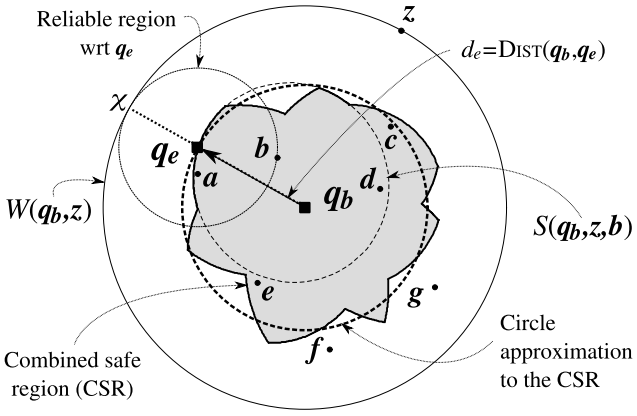


Fig. 8 Combined safe region ($k = 2$, $x = 6$) and its circle approximation

For random trajectories, we use the *random-walk* model to help estimate n_b. A random walk is a sequence of fixed-size steps where the orientation of each step is selected at random. We first estimate the region within which the query point can move without requiring any data access (termed as the *combined safe region* (CSR)). Figure 8 shows that the CSR (the gray region) consists of sections of safe regions with regard to different objects when they are the k^{th} NN. For example, part of the CSR in the figure is a set of points $\boldsymbol{v}$ such that $\boldsymbol{v}$ is in $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{b})$ and $\boldsymbol{b}$ is the k^{th} NN of $\boldsymbol{v}$. Since the distance between $\boldsymbol{q_b}$ and $\boldsymbol{q_e}$ is d_e (see Figure 8), we can approximate the CSR as a circle centered at $\boldsymbol{q_b}$ with a radius of d_e. Second we estimate the number n_s of steps the query point $\boldsymbol{q}$ has to take from $\boldsymbol{q_b}$ in order to cross the circle approximation to the CSR for the first time. This turns out to be a classic random-walk problem [26] and n_s is given by

$$n_s = (d_e/s)^2, \quad (10)$$

where s is the step size. Therefore, for a random trajectory with length l, n_b is given by

$$n_b = ls/d_e^2. \quad (11)$$

6.2 RIS-kNN

RIS-kNN processes the MkNN query as follows. Every time the query point $\boldsymbol{q}$ exits the current kVD cell, RIS-kNN is executed to obtain the new kVD cell and the corresponding k NNs. Therefore, the number of data accesses for RIS-kNN is equal to the number n_v of kVD cells crossed by $\boldsymbol{q}$. In the worst case, $\boldsymbol{q}$ moves along a straight line, and n_v is proportional to the average linear density[4] of the kVD cells. Assuming that n is much greater than k, the number of the kVD cells in 2D space is approximated as $2kn$ [20]. The area of the space is 1 square unit, so the density of kVD

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

cells is $2kn$, which corresponds to a linear density of $\sqrt{2kn}$. For a given trajectory length l, the number n_v of kVD cells crossed by the trajectory is given by

$$n_v = l\sqrt{2kn}. \tag{12}$$

Figure 9 shows that the number of kVD cells that $\boldsymbol{q}$ crosses as it moves from $\boldsymbol{q_1}$ to $\boldsymbol{q_2}$ is roughly the square root of the number of kVD cells overlapped with (or contained by) the gray square with a side length of $\text{DIST}(\boldsymbol{q_1}, \boldsymbol{q_2})$.

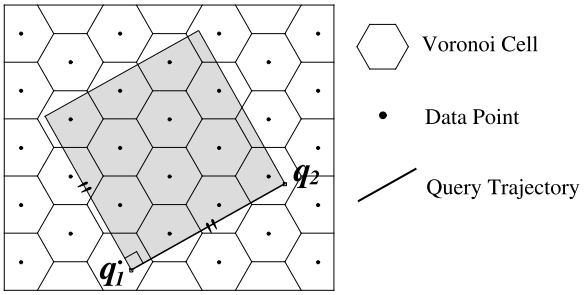


Fig. 9 The quadratic relation between the number of crossed kVD cells and the density of kVD cells

The RIS-kNN cost model (Equation (12)) is used to estimate the number of data accesses for both trajectory types in the experiments (Section 8.5). This is because, for the considered values of parameters, kVD cells are too small for the two different trajectory types to produce a noticeable difference in terms of the number n_v of encountered kVD cells. We will see from the experimental results that the cost model can accurately estimate the number of data accesses for both trajectory types.

6.3 Comparison of V*-kNN and RIS-kNN

Since the trajectory type that maximizes n_b is a straight line, we use the cost model for straight-line trajectories to represent the processing cost of V*-kNN in this comparison. The value of n_b in terms of k, n, x and l can be expressed as

$$n_b = \frac{l\sqrt{\pi}}{2\left(\sqrt{1-\sqrt{\frac{k}{n}}} - \sqrt{1-\sqrt{\frac{k+x}{n}}}\right)}.$$

The expression of n_b can be relaxed as follows:

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

Therefore, we obtain $2\sqrt{\pi}\left(l\sqrt{\frac{(k+x)n}{x^2}}\right)$ as an upper bound of n_b. Typically, x is comparable to k and $(k+x)$ is much smaller than xk. As a result,

$$n_b \leq 2\sqrt{\pi}\left(l\sqrt{\frac{kn}{x}}\right).$$

Let C_{nn} be the cost of a BF-kNN call. An upper bound of the total cost of V*-kNN is

$$C_{nn}\left(2\sqrt{\pi}\cdot l\sqrt{\frac{kn}{x}}\right).$$

The total cost is determined by the number n_v of RIS-kNN calls and the cost of each RIS-kNN call. Each RIS-kNN run requires 12 TPkNN queries on average [33]. Let C_{tpnn} be the cost of a TPkNN call. Then the total cost of RIS-kNN for the MkNN query is

$$12C_{tpnn}\left(\sqrt{2}\cdot l\sqrt{kn}\right).$$

We now compare the costs of V*-kNN and RIS-kNN. For the number of data accesses, V*-kNN has a lower count than RIS-kNN especially when x is large. For the cost per data access, V*-kNN executes a single BF-kNN query to retrieve $(k+x)$ NNs and RIS-kNN executes 12 TPkNN queries on average. To compare the costs of a BF-kNN call and a TP-kNN call, we refer to the experimental results reported by Tao et al. [28], which suggest that the I/O cost of each TPkNN call is comparable to that of a BF-kNN call. Although V*-kNN retrieves x auxiliary objects in addition to the k NNs for each BF-kNN call, this small addition to k has no significant effect on the cost of BF-kNN [10]. For the practical range of x values suggested by our experiments, a single BF-kNN call (issued by V*-kNN) has a lower cost than 12 TPkNN calls (issued by RIS-kNN). Therefore, V*-kNN has both smaller number of data accesses and lower cost per data access.

In terms of the CPU cost, V*-kNN maintains the rank of $(k+x)$ objects, which has more computation than RIS-kNN. However, given today's mobile devices (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 realistic settings, the benefits in the data access cost outweigh this overhead.

7 The V*-Diagram in Spatial Networks

When the movement of the query point 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. A spatial network is also a metric space [23]. The V*-Diagram and V*-kNN are applicable to the domain of spatial networks because we assume a metric space in their formulations.

The essence of the V*-Diagram is the ISR, which consists of two key components: the safe region S_k with regard to k^{th} NN and the fixed-rank region (FRR). We focus our discussions on how to determine these two components. We reuse the algorithms to process MkNN queries presented in Section 5.

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 two points $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ in the network, $\textsc{Dist}(\boldsymbol{p_1}, \boldsymbol{p_2})$ is the length of $path(\boldsymbol{p_1}, \boldsymbol{p_2})$, where $path(\boldsymbol{p_1}, \boldsymbol{p_2})$ denotes the shortest path between $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$. Figure 10(a) shows a spatial network of eight nodes, $\boldsymbol{a}$ to $\boldsymbol{h}$. The distance $\textsc{Dist}(\boldsymbol{q_1}, \boldsymbol{s})$ is 2 using the path via $\boldsymbol{b}$.

We use $pnt(\boldsymbol{p_1}, \boldsymbol{p_2}, l)$ to denote a point on $edge(\boldsymbol{p_1}, \boldsymbol{p_2})$ with a distance l along the edge to $\boldsymbol{p_1}$, and $seg(\boldsymbol{p_1}, \boldsymbol{p_2}, l)$ to denote a section on $edge(\boldsymbol{p_1}, \boldsymbol{p_2})$ with two endpoints of $\boldsymbol{p_1}$ and $pnt(\boldsymbol{p_1}, \boldsymbol{p_2}, l)$. For example, $\boldsymbol{u}$ is at $pnt(\boldsymbol{a}, \boldsymbol{e}, 3)$, and the segment between Vertex $\boldsymbol{a}$ and $\boldsymbol{q_1}$ can be represented as $seg(\boldsymbol{a}, \boldsymbol{b}, 3)$.

Figure 10(a) shows a MkNN query with k and x values of 1 and 2, respectively. The $(k+x)$ NNs of $\boldsymbol{q_1}$, i.e., $\boldsymbol{s}$, $\boldsymbol{t}$ and $\boldsymbol{u}$, are retrieved via a spatial-network kNN query [12, 13, 21, 24]. We can apply the known region definition in Section 4 and express the known region with regard to $\boldsymbol{q_1}$ and $\boldsymbol{u}$ as

$$W(\boldsymbol{q_1}, \boldsymbol{u}) = \{\boldsymbol{v} \in \mathcal{D} : \textsc{Dist}(\boldsymbol{q_1}, \boldsymbol{v}) \leq \textsc{Dist}(\boldsymbol{q_1}, \boldsymbol{u})\}.$$

This region is mapped to a collection of the following edges and segments in the network: $edge(\boldsymbol{a}, \boldsymbol{b})$, $edge(\boldsymbol{a}, \boldsymbol{c})$, $edge(\boldsymbol{b}, \boldsymbol{d})$, $seg(\boldsymbol{a}, \boldsymbol{e}, 3)$, $seg(\boldsymbol{b}, \boldsymbol{f}, 5)$, $seg(\boldsymbol{d}, \boldsymbol{c}, 2)$ and $seg(\boldsymbol{d}, \boldsymbol{g}, 2)$. The region $W(\boldsymbol{q_1}, \boldsymbol{u})$ is shown as thick line segments in Figure 10.

Next, we determine the safe region S_k of the k^{th} NN of $\boldsymbol{q}$ (which is $\boldsymbol{s}$ in Figure 10(a)). The safe region with regard to $\boldsymbol{s}$ is given by

$$S(\boldsymbol{q_1}, \boldsymbol{u}, \boldsymbol{s}) = \{\boldsymbol{q'} : \textsc{Dist}(\boldsymbol{q'}, \boldsymbol{s}) + \textsc{Dist}(\boldsymbol{q_1}, \boldsymbol{q'}) \leq \textsc{Dist}(\boldsymbol{q_1}, \boldsymbol{u})\}.$$

We formulate an algorithm to identify the boundary of S_k based on the observation that the safe region with regard to $\boldsymbol{p}$ in Euclidean space is an elliptic region with the focal points of $\boldsymbol{q_b}$ and $\boldsymbol{p}$. Specifically, we apply the *incremental network expansion* (INE) algorithm [21] around the two focal points, $\boldsymbol{q_b}$ and $\boldsymbol{p}$, to identify edges that cross the boundary of S_k. As seen in Algorithm 5, a priority queue is used to order vertices to be examined according to the sum aggregate distance (SUMDIST) to $\boldsymbol{q_b}$ and $\boldsymbol{p}$. We explain the algorithm via the following example.

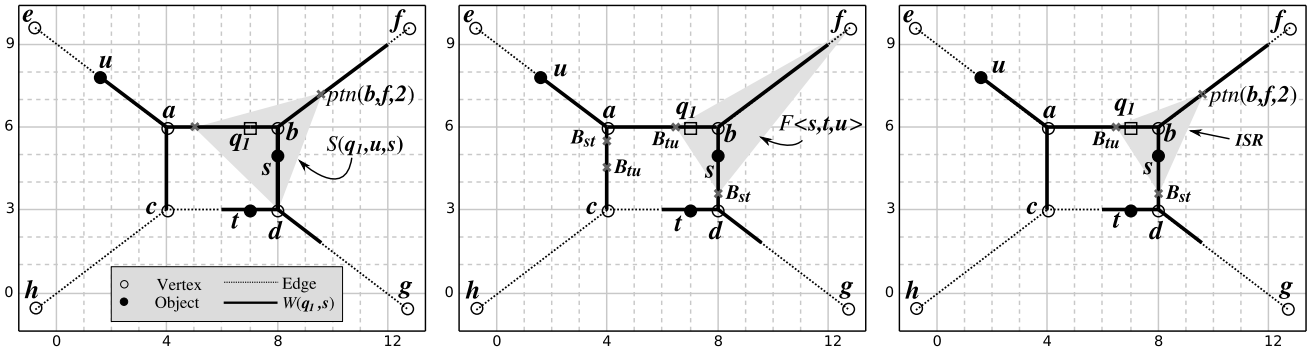
Algorithm 5: Compute-S_k-Boundary($\boldsymbol{q_b}$, $\boldsymbol{z}$, $\boldsymbol{p_k}$)

```
1  Create an empty list L_b of boundaries
2  Create an empty priority queue pq of vertices
3  Insert vertices adjacent to path(q_b, p_k) into pq
4  while (pq is not empty) do
5      r ← pq.PopHead()
6      if (Dist(q_b, r) + Dist(r, p_k) ≤ Dist(q_b, z)) then
7          Insert all immediate neighbors of r into pq
8      else
9          Identify the boundary points on path(q_b, r) and
           path(p_k, r), then insert them into L_b
10 return L_b
```

In this example, Algorithm 5 is applied to the example in Figure 10(a) to calculate the boundaries of $S(\boldsymbol{q_1}, \boldsymbol{u}, \boldsymbol{s})$. A summary of execution steps is given in Table 5 and detailed explanations are given as follows.

- **Step 0.** A list L_b and a priority queue pq are initialized (Lines 1 and 2). Vertices $\boldsymbol{d}$, $\boldsymbol{a}$ and $\boldsymbol{f}$, which are adjacent to $path(\boldsymbol{q_1}, \boldsymbol{s})$, are inserted into pq (Line 3).
- **Step 1.** Vertex $\boldsymbol{d}$ is retrieved from pq (Line 5). Since the SUMDIST of $\boldsymbol{d}$ is smaller than or equal to $\textsc{Dist}(\boldsymbol{q_1}, \boldsymbol{u})$ (Line 6), we insert the neighbors of $\boldsymbol{d}$, i.e., $\boldsymbol{c}$ and $\boldsymbol{g}$, into pq (Line 7).
- **Step 2.** We examine $\boldsymbol{a}$. Since the SUMDIST of $\boldsymbol{a}$ is greater than $\textsc{Dist}(\boldsymbol{q_1}, \boldsymbol{u})$, we identify the boundaries of S_k on $path(\boldsymbol{q_1}, \boldsymbol{a})$ and $path(\boldsymbol{s}, \boldsymbol{a})$. The two paths provide the same boundary at $pnt(\boldsymbol{b}, \boldsymbol{a}, 3)$. The boundary is inserted into L_b (Line 9).
- **Step 3.** We examine $\boldsymbol{c}$ and find two boundaries at $pnt(\boldsymbol{b}, \boldsymbol{a}, 3)$ and Vertex $\boldsymbol{d}$. Since $pnt(\boldsymbol{b}, \boldsymbol{a}, 3)$ is a duplicate, only $\boldsymbol{d}$ is inserted into L_b.
- **Step 4.** We examine $\boldsymbol{f}$. The SUMDIST of f is greater than $\textsc{Dist}(\boldsymbol{q_1}, \boldsymbol{u})$. A boundary of S_k is identified as $pnt(\boldsymbol{b}, \boldsymbol{f}, 2)$ and is inserted into L_b (Line 9).
- **Step 5.** We examine $\boldsymbol{g}$ and find a boundary at Vertex $\boldsymbol{d}$. Since Vertex $\boldsymbol{d}$ is a duplicate, it is not inserted into L_b.

We finally obtain the boundaries of $S(\boldsymbol{q_1}, \boldsymbol{u}, \boldsymbol{s})$ as $\boldsymbol{d}$, $pnt(\boldsymbol{b}, \boldsymbol{a}, 3)$ and $pnt(\boldsymbol{b}, \boldsymbol{f}, 2)$. These boundaries corre-



(a) Safe region $S(\boldsymbol{q_1}, \boldsymbol{u}, \boldsymbol{s})$ (b) Fixed-rank region $F\langle \boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u} \rangle$ (c) ISR $I(\boldsymbol{q_1}, \boldsymbol{u}, \boldsymbol{s}, \langle \boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u} \rangle)$ is the intersection of $S(\boldsymbol{q_1}, \boldsymbol{u}, \boldsymbol{s})$ and $F\langle \boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u} \rangle$

Fig. 10 V*-Diagram in a spatial network ($k = 1$ and $x = 2$)

spond to the region containing $edge(\boldsymbol{b}, \boldsymbol{d})$, $seg(\boldsymbol{b}, \boldsymbol{a}, 3)$ and $seg(\boldsymbol{b}, \boldsymbol{f}, 2)$. The safe region $S(\boldsymbol{q_1}, \boldsymbol{u}, \boldsymbol{s})$ is shown as the segments in the gray triangle in Figure 10(a).

Table 5 Example run of Algorithm 5

Step	Vertex	pq	L_b
0	-	$\langle \boldsymbol{d}, \boldsymbol{a}, \boldsymbol{f} \rangle$	$\langle\rangle$
1	$\boldsymbol{d}$	$\langle \boldsymbol{a}, \boldsymbol{c}, \boldsymbol{f}, \boldsymbol{g} \rangle$	$\langle\rangle$
2	$\boldsymbol{a}$	$\langle \boldsymbol{c}, \boldsymbol{f}, \boldsymbol{g} \rangle$	$\langle pnt(\boldsymbol{b}, \boldsymbol{a}, 3) \rangle$
3	$\boldsymbol{c}$	$\langle \boldsymbol{f}, \boldsymbol{g} \rangle$	$\langle pnt(\boldsymbol{b}, \boldsymbol{a}, 3), \boldsymbol{d} \rangle$
4	$\boldsymbol{f}$	$\langle \boldsymbol{g} \rangle$	$\langle pnt(\boldsymbol{b}, \boldsymbol{a}, 3), \boldsymbol{d}, pnt(\boldsymbol{b}, \boldsymbol{f}, 2) \rangle$
5	$\boldsymbol{g}$	$\langle\rangle$	$\langle pnt(\boldsymbol{b}, \boldsymbol{a}, 3), \boldsymbol{d}, pnt(\boldsymbol{b}, \boldsymbol{f}, 2) \rangle$

The FRR of $\langle \boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u} \rangle$ is given by $H_{\boldsymbol{st}} \cap H_{\boldsymbol{tu}}$. As discussed in Section 4.2, the FRR is determined by the rank-adjacent bisectors of the $(k + x)$ objects. In a spatial network, a bisector reduces to points on edges. Algorithm 6 is an algorithm to find the boundary of an FRR $F(L)$ as a list L_b of bisecting points. Specifically, the algorithm finds all bisecting points of all rank-adjacent bisectors of L on edges in L_e (Lines 2 and 3). A bisecting point on an edge is calculated by checking whether the two endpoints are on either side of the bisector[5] (Line 4). If yes, the bisecting point is calculated and inserted into L_b (Lines 5 and 6).

Unlike the boundary of S_k, bisecting points of rank-adjacent bisectors are not localized around the query point. We have to limit the search space by specifying the list L_e of edges we want to check to avoid accessing all edges in the data space. If the FRR is to be used in conjunction with the S_k to construct the ISR, L_e can be the list of edges overlapped with (or contained by) S_k. A "lazier" approach is having L_e as a singleton list of the edge that the query point currently occupies and executing the algorithm every time the query point enters a new edge. In addition, this lazy approach can be applied to evaluations of S_k.

[5] We omit the handling a marginal case where one or both of the vertices are on the bisector.

Algorithm 6: Compute-FRR-Boundary(L, L_e)

```
1 Create an empty list L_b of boundaries
2 for all (edge(a, b) in L_e) do
3     for all (rank adjacent pairs (s, t) in L) do
4         if a and b are on either side of B_st then
5             Find Point v on edge(a, b) such that
              DIST(v, s) is equal to DIST(v, t)
6             Insert v into L_b
7 return L_b
```

By applying Algorithm 6 to all edges in the network in Figure 10(b), two following bisectors are detected: (i) $B_{\boldsymbol{st}}$ as $pnt(\boldsymbol{b}, \boldsymbol{d}, 2.5)$ and $pnt(\boldsymbol{a}, \boldsymbol{c}, 0.5)$, and (ii) $B_{\boldsymbol{tu}}$ as $pnt(\boldsymbol{b}, \boldsymbol{a}, 1.5)$ and $pnt(\boldsymbol{a}, \boldsymbol{c}, 1.5)$. The two bisecting points $pnt(\boldsymbol{a}, \boldsymbol{c}, 0.5)$ and $pnt(\boldsymbol{a}, \boldsymbol{c}, 1.5)$ are outside S_k and hence are irrelevant to the ISR. Now let L_e be the edges overlapped with (or contained by) S_k. The two detected bisecting points are $pnt(\boldsymbol{b}, \boldsymbol{d}, 2.5)$ and $pnt(\boldsymbol{b}, \boldsymbol{a}, 1.5)$ from $B_{\boldsymbol{st}}$ and $B_{\boldsymbol{tu}}$, respectively. Figure 10(c) shows that two bisecting points are sufficient to function as the FRR boundary to form part of the ISR boundary.

The ISR definition given in Section 4.3 can be applied to the spatial-network domain. The spatial-network ISR retains the kNN safe-region property, since Theorems 1 and 2 are applicable to any metric space. The boundary of the ISR $I(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k}, L)$, i.e., the V*-Diagram, can be obtained by using Algorithm 5 and Algorithm 6 to construct $S(\boldsymbol{q_b}, \boldsymbol{z}, \boldsymbol{p_k})$ and $F(L)$, respectively. As shown in Figure 10(c), the ISR is the intersection of $S(\boldsymbol{q_1}, \boldsymbol{u}, \boldsymbol{s})$ and $F\langle \boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u} \rangle$, and 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 gray region. According to Algorithm 1, exiting the ISR via $pnt(\boldsymbol{b}, \boldsymbol{a}, 1.5)$ ($B_{\boldsymbol{tu}}$) or $pnt(\boldsymbol{b}, \boldsymbol{d}, 2.5)$ ($B_{\boldsymbol{st}}$) triggers a *RankUpdate* event, and exiting the ISR via $pnt(\boldsymbol{b}, \boldsymbol{f}, 2)$ triggers a *ReliabilityUpdate* event.

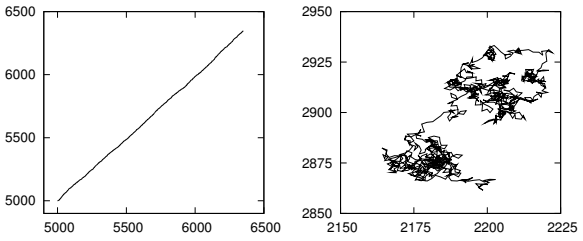
8 Experimental Study

This section presents the results of our experimental study, which includes the followings: (i) the effect of the number x of auxiliary objects on V*-kNN; (ii) a performance comparison between V*-kNN and its best competitor, RIS-kNN [33], in terms of the total response time, the I/O cost and the communication cost; (iii) validation of the three cost models in Section 6. (iv) experimental study on a spatial-network implementation of the V*-kNN algorithm.

8.1 Experimental setup

The experiments were conducted on a 2.4GHz Intel Core 2 Quad machine with a 4GB main memory. An R*-tree [1] was used to index the data objects. The R*-tree, V*-kNN and RIS-kNN were implemented in C++. 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)

Fig. 11 Trajectory types

We generated two different types of query trajectories, *directional* (D) and *random* (R), as shown in Figure 11. The trajectory length is denoted as l and the distance between two consecutive location updates (step size) is denoted as s. The number u of location updates is given by $u = l/s$. For example, a trajectory with l of $2,000$ units and s of 2 units has $1,000$ query point updates. Between two query point updates, the trajectory is assumed to be a straight line segment. For each type of trajectory, we generated 20 different trajectories. We ran these 20 trajectories as a query set for each experiment and presented the average result. We measured both the cumulative total response time (the wall-clock time) and the cumulative number of page accesses for a whole trajectory as the performance metric.

Note that in Section 6, we considered two types of trajectories, straight-line and random. A straight-line trajectory can be considered as a special case of a directional trajectory where all sampling points are co-linear. In our experiments, directional trajectories were used instead of straight-line trajectories, since the former are more realistic than the latter. For random trajectories, each trajectory is generated as a random walk [26], which conforms with the analysis given in Section 6.

Table 6 gives a summary on the ranges and the default values of the parameters used from Section 8.2 to Section 8.5: (i) the number of auxiliary objects x, (ii) the cache size c, (iii) the trajectory length l, (iv) the cardinality n of the dataset, (v) the value of k, (vi) the step size s, and (vii) the probability δ that the k value is altered by 1. The settings for the spatial-network experimental study are given in Section 8.6.

Table 6 Summary of parameters

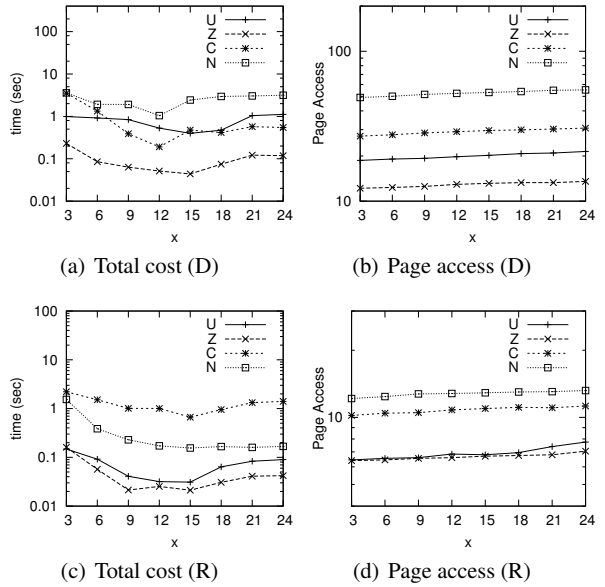
Parameter	Default Value	Range	Increment
x	15	[3:24]	3
c	32	[0:32]	8
l	2,000	[400:2,000]	400
n	25,000	[25,000:100,000]	25,000
k	20	[10:40]	10
s	2	[2:3]	0.25
δ	0	[0:0.1]	0.025

8.2 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*-kNN. The value of x was varied from 3 to 24. 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*-kNN effective. Figure 12 shows the response time and the number of page accesses as functions of x for both query trajectory types.

The general trend of the results is observed as follows. The response time first decreases as x increases and the response time starts to increase when x becomes larger than approximately 15. This is because an increase in x reduces the number of BF-kNN calls, i.e., the number of data accesses, but increase the CPU cost due to the higher number of objects that V*-kNN has to maintain. When x becomes too large, the computational overhead of maintaining more objects becomes more significant and may outweigh the savings in the number of data accesses.

According to Figures 12(b) and 12(d), the page-access cost increases as x increases. This is because the footprint of V*-kNN becomes larger. Although the number of data accesses is higher for smaller values of x, consecutive accesses are localized. This spatial locality results in repetitive accesses on pages already stored in the cache.



(a) Total cost (D) (b) Page access (D)

(c) Total cost (R) (d) Page access (R)

Fig. 12 Effect of x

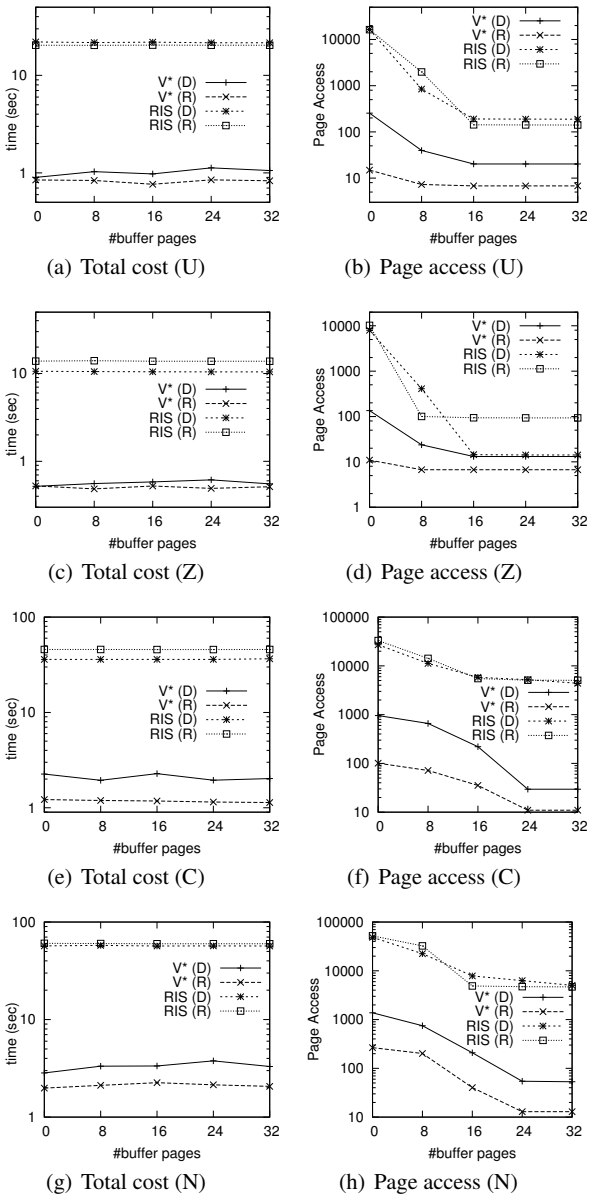
These results confirms our discussion in Section 5.1 and the analysis in 6. In all these experiment, the x value of 15 provides a good performance, so 15 is used as the default value of x for the rest of the experiments.

8.3 Comparative study: centralized

In this subsection, we compare V*-kNN with RIS-kNN in terms of their total response times.

The effect of c. In this set of experiments, we use the buffer sizes between 0 and 32 pages. Figure 13 shows the results for two synthetic datasets with the default dataset size and the two real datasets. V*-kNN outperforms the RIS-kNN in all settings in terms of both total response time and number of page accesses. In most cases, the improvement factor is one order of magnitude. For both methods, the page access cost decreases as the buffer size increases as expected. The total response time remains relatively stable as the buffer size increases. The main reason is that the R-tree buffer has the same functionality as the page cache [8], which is a transparent disk buffer (on the main memory) maintained by the operating system. Therefore, changes in the buffer size have no effect on the total response time.

The effect of l. In this set of experiments, we vary the trajectory length l from 400 to 2,000 units. Figure 14 shows the response time on the four datasets. In all experiments, V*-kNN outperforms RIS-kNN and the improvement factor is one order of magnitude in most cases. The results in terms 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. For both techniques, the total response time increases as l increases.



(a) Total cost (U) (b) Page access (U)

(c) Total cost (Z) (d) Page access (Z)

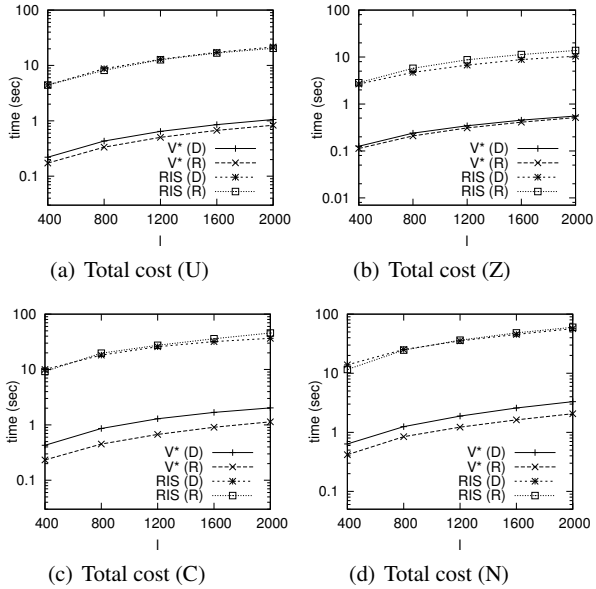
(e) Total cost (C) (f) Page access (C)

(g) Total cost (N) (h) Page access (N)

Fig. 13 Centralized processing: effect of c

The effect of n. In this set of experiments, the number of objects in the dataset is varied from 25,000 to 100,000 for the synthetic datasets. Figure 15 shows the response time results. Again, V*-kNN outperforms RIS-kNN in all settings and the improvement factor is one order of magnitude in most cases. For both techniques, the total response time increases as the number of objects increases.

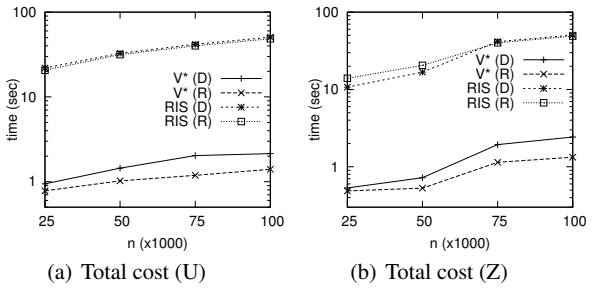
The effect of k. In this set of experiments, we vary the value of k from 10 to 40 for the four datasets. Figure 16 shows the response time results. We observe similar results as in previous experiments. V*-kNN outperforms RIS-kNN in all settings and the improvement factor is one order of magnitude in most cases. For both techniques, the total response time increases as the value of k increases, but the



(a) Total cost (U) (b) Total cost (Z)

(c) Total cost (C) (d) Total cost (N)

Fig. 14 Centralized processing: effect of l



(a) Total cost (U) (b) Total cost (Z)

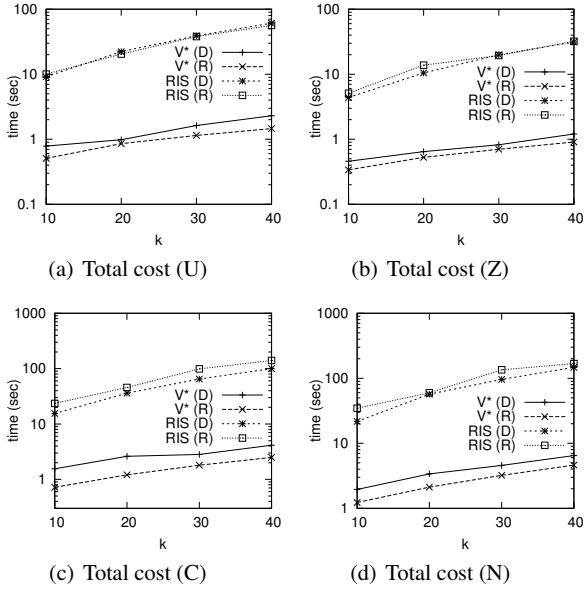
Fig. 15 Centralized processing: effect of n

total response time of V*-kNN increases slower than that of RIS-kNN.

The effect of s. Figure 17 shows the response time results as the step size s is varied from 2 to 3 units while the trajectory length is fixed to $2,000$ units. The total cost for RIS-kNN remains stable as s increases, since s only affect the validity-check frequency whose cost is negligible. We also observe that there is no absolute increase for V*-kNN as s is varied from 2 to 3 units.

The effect of δ. The value δ is the probability that the k value is altered (decreased or increased) by 1 for each location update. Specifically, for a trajectory of u location updates, the value of k is expected to change $u\delta$ times. Table 7 shows the expected number of times the k value changes and the expected range of k for each experimented value of δ, where u is equal to 1,000 updates.

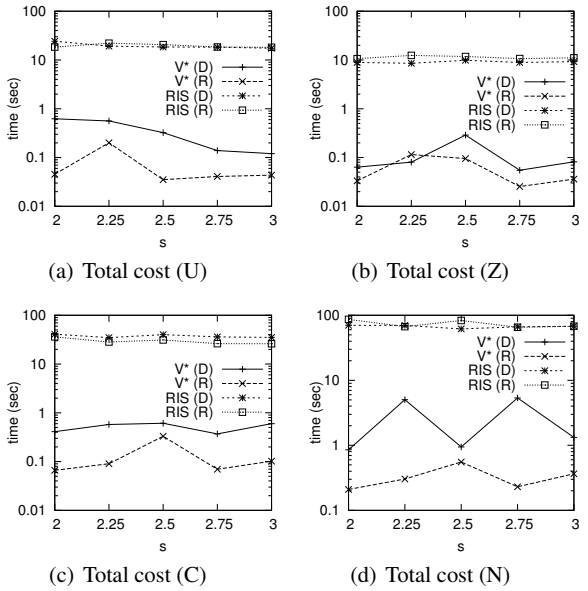
Figure 18 shows the response times as the probability δ is varied from 0 to 0.1 with an increment of 0.025. For RIS-kNN, the total response time has a positive correlation with δ. For V*-kNN, although the correlation is less apparent than that of RIS-kNN, we observe that a higher value of δ has a tendency to produce a greater response time. This is



(a) Total cost (U) (b) Total cost (Z)

(c) Total cost (C) (d) Total cost (N)

Fig. 16 Centralized processing: effect of k



(a) Total cost (U) (b) Total cost (Z)

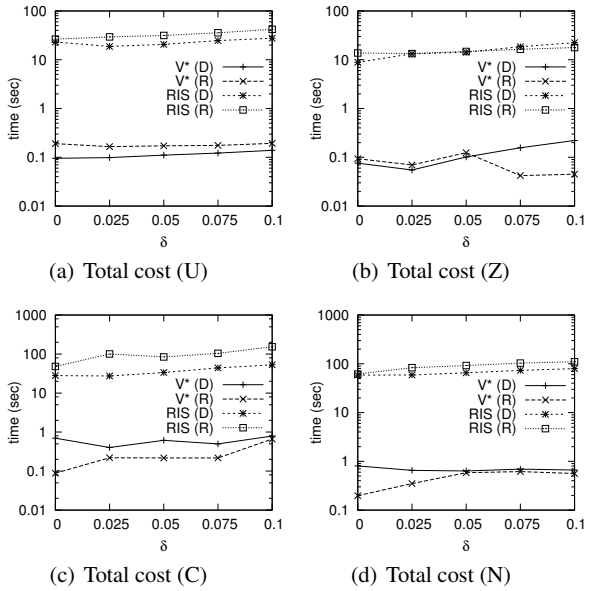
(c) Total cost (C) (d) Total cost (N)

Fig. 17 Centralized processing: effect of s

Table 7 Effect of δ

δ	Expected number of changes	Expected range of k
0	0	[20:20]
0.025	25	[17:23]
0.050	50	[15:25]
0.075	75	[14:26]
0.100	100	[13:27]

because a greater fluctuation in k incurs a higher access cost and more CPU time for both algorithms. V*-kNN consistently outperforms RIS-kNN in all settings.



(a) Total cost (U) (b) Total cost (Z)

(c) Total cost (C) (d) Total cost (N)

Fig. 18 Centralized processing: effect of δ

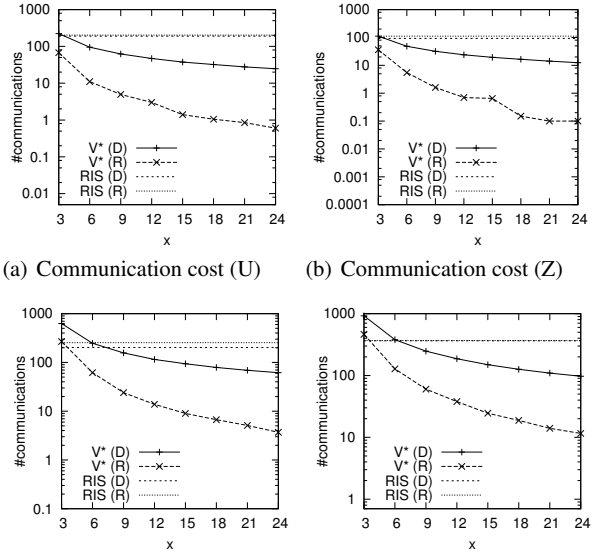
8.4 Comparative study: client-server

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*-kNN and RIS-kNN. The number of times a client has to communicate with the server is used as the performance measure.

For V*-kNN, the communication cost is measured by the number of times the BF-kNN query is executed because other operations are local. For RIS-kNN, the communication cost is the number of times the query itself is executed, i.e., the number of kVD cells crossed. We ignore the data size because query answers in the experiments can be accommodated by a single packet with a typical size of 500 bytes (approximately 62 data points). V*-kNN requires a storage for only x data points, since the first k can be reused. RIS-kNN requires k objects for the query answer and 6 point locations on average for the corresponding kVD cell's corners.

The effect of x. Figure 19 shows the communication costs with the increasing x values in the four datasets and two trajectory types. Although the parameter x does not apply to RIS-kNN, we use its results from the default values of c, l, n and k for comparison purposes. For all datasets, we can see that the communication cost of V*-kNN decreases as the value of x increases. This conforms with the cost analysis in Section 6, which suggests that the retrieval cost of the V*-kNN should decrease as x increases. It can be seen that when x is greater than 6, V*-kNN outperforms RIS-kNN in all settings.

The difference between the communication cost and total response time (Figure 12) is notable. Unlike the total response time, the communication cost measure disregards the CPU cost. Only the number of data accesses is considered, and thus there is no penalty for large values of x. In practice, however, the value of x will be limited by the computational capability of each individual mobile device, since the cost to maintain the $(k + x)$ objects is positively correlated to x.



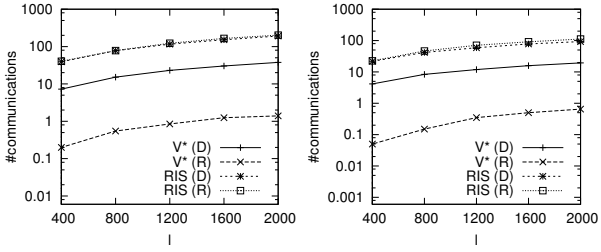
(a) Communication cost (U) (b) Communication cost (Z)

(c) Communication cost (C) (d) Communication cost (N)

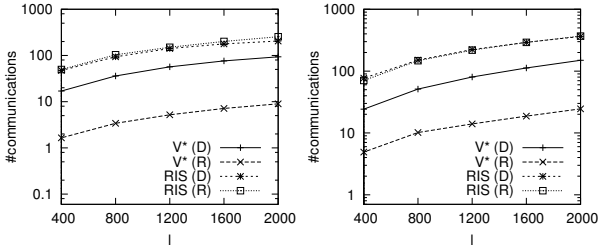
Fig. 19 Communication cost: effect of x

The effect of l. Figure 20 shows the communication costs as we vary the trajectory length l from 400 to $2,000$ units. The communication costs increases as l increases. This is because l is proportional to the number of BF-kNN calls for V*-kNN, and the number of crossed kVD cells for RIS-kNN.

The effect of n. Figure 21 shows the communication costs as we vary the value of n from $25,000$ to $100,000$. Since a larger value of n produces denser kVD cells in the data space, the cost of RIS-kNN increases as n increases due to more frequent cell crossings. The communication cost of V*-kNN also has a positive correlation with n as suggested by the analysis in Section 6. V*-kNN still outperforms RIS-kNN in all settings, as shown in Figure 21. The effect of n on the communication and total costs are very similar. This is because n has positive correlations with: communication, tree-traversal and computation costs.

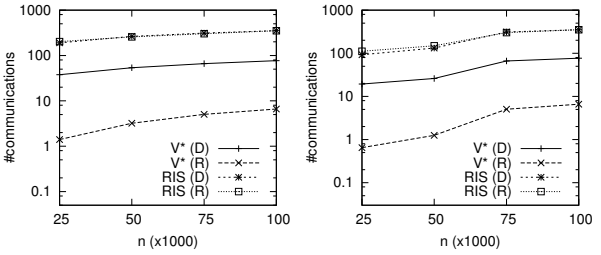


(a) Communication cost (U) (b) Communication cost (Z)



(c) Communication cost (C) (d) Communication cost (N)

Fig. 20 Communication cost: effect of l



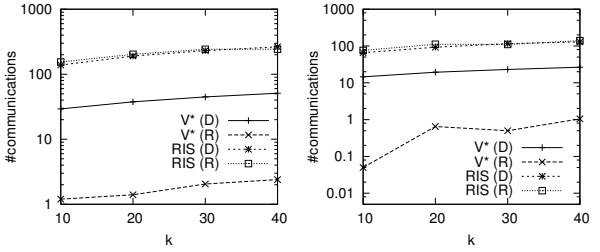
(a) Communication cost (U) (b) Communication cost (Z)

Fig. 21 Communication cost: effect of n

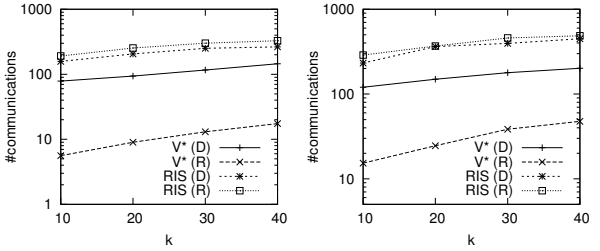
The effect of k. Figure 22 shows the communication costs as we vary the value of k from 10 to 40. Similar to the parameter n, the communication cost increases as k increases, since the k has a positive correlation to both the density of kVD cells and the number of times that V*-kNN executes BF-kNN. V*-kNN continues to outperform RIS-kNN in all settings.

The effect of s. Figure 23 shows the communication costs as we vary the step size s from 2 to 3 units. For V*-kNN on directional trajectories, the parameter s has no effect on the communication cost, the results hence remain stable as s changes. For V*-kNN on random trajectories, the communication cost slightly increases as s increases. For RIS-kNN, we observe no significant change in the communication cost as s increases.

The effect of δ. Figure 24 shows the communication costs as we vary the probability δ from 0 to 0.1. For both algorithms, the value of δ has a positive correlation with the communication cost. This is because RIS-kNN has to recompute the kVD cell every time k is altered. For V*-kNN, an increase in δ has a greater effect on the communication cost for random trajectories than directional trajectories. This is because of the correlation between the expected radius d_e of the combined safe region (CSR) and the expected number n_s of steps between two consecutive data accesses. According to Equation (10), n_s is quadratic with respect to d_e for random trajectories, while n_s is proportional to d_e for directional trajectories. As d_e has a negative correlation with k, a greater fluctuation in the value of k has a greater effect on random trajectories than directional trajectories for V*-kNN.

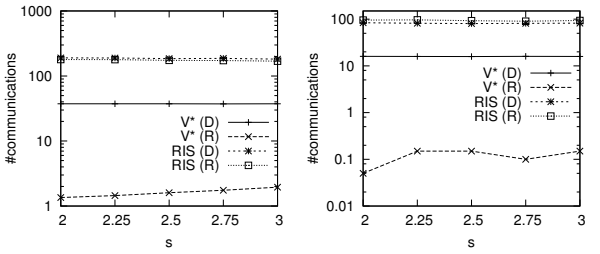


(a) Communication cost (U) (b) Communication cost (Z)

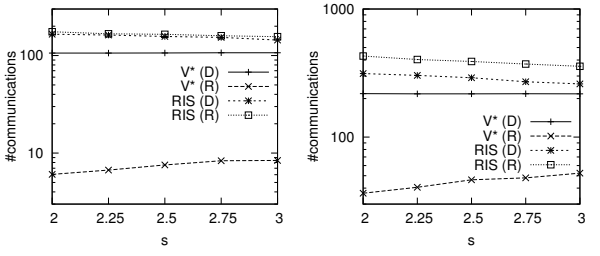


(c) Communication cost (C) (d) Communication cost (N)

Fig. 22 Communication cost: effect of k

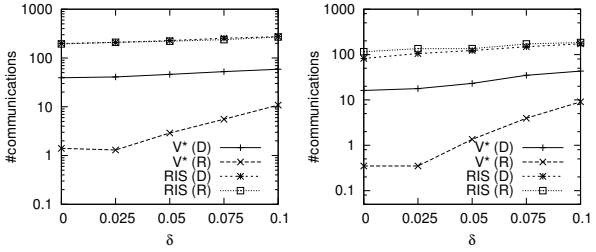


(a) Communication cost (U) (b) Communication cost (Z)

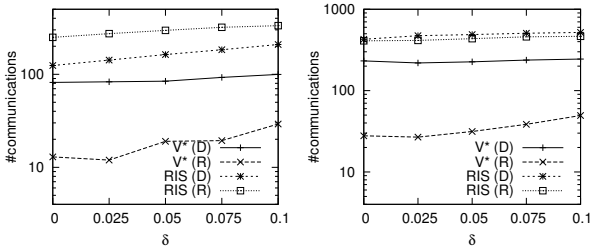


(c) Communication cost (C) (d) Communication cost (N)

Fig. 23 Communication cost: effect of s



(a) Communication cost (U) (b) Communication cost (Z)



(c) Communication cost (C) (d) Communication cost (N)

Fig. 24 Communication cost: effect of δ

8.5 Cost model validation

In this subsection, we validate our cost models described in Section 6. For V*-kNN, the number n_b of BF-kNN calls is estimated as:

$$n_b = l/d_e, \text{(for straight-line trajectories);} \quad (13)$$

$$n_b = ls/d_e^2, \text{(for random-walk trajectories).} \quad (14)$$

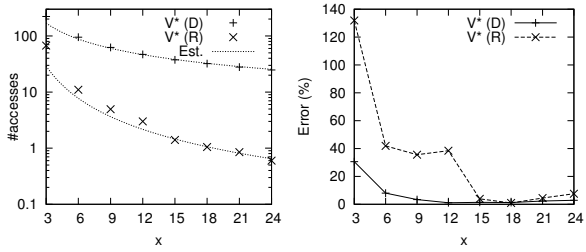
The value of d_e is given by Equation (8).

For RIS-kNN, according to Section 6, the number n_v of kVD cells crossed by a query point moving in a straight line trajectory with a length of l is estimated by

$$n_v = l\sqrt{2kn}. \quad (15)$$

We use Equation (13) for V*-kNN on directional trajectories and Equation (14) for V*-kNN on random trajectories. Equation (15) is used for RIS-kNN on both types of trajectories. We show the estimated data access counts produced by the three cost models in comparison to the actual results, as well as the relative errors. The relative error of a measurement is given by $|(v_0 - v)/v|$, where v is the measured value and v_0 is the theoretical estimate.

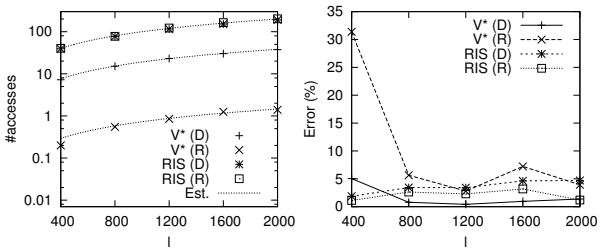
The effect of x. Figure 25 shows a comparison between experimental results and the two cost models (for V*-kNN) as the value of x is varied from 3 to 24. Figure 25(a) shows that the estimates produced by the two V*-kNN cost models conform with the experimental results. That is, the number of data accesses reduces as x increases. Figure 25(b) displays the relative errors between the theoretical estimates and the measured values. The accuracy of the cost models for both trajectory types have a tendency to improve as x increases. When x assumes a relatively large value (15), the relative errors are less than 10%.



(a) Effect of x (b) Relative errors

Fig. 25 Cost model validation for x on synthetic uniform data

The effect of l. Figure 26 provides a comparison between the experimental results and the three cost models as the value of l is varied from 400 to 2,000 units. Figure 26(a) shows that the number of data accesses increases as l increases for both algorithms and the experimental results conform with the estimates produced by the cost models. The relative errors are presented in Figure 26(b). For V*-kNN on random trajectories, we observe a drastic accuracy improvement as l is increased from 400 to 800 units and the relative error remains below 10% for the rest of l values. For V*-kNN on directional trajectories and RIS-kNN on both trajectory types, their relative errors remain below 5% for the whole range of l values.

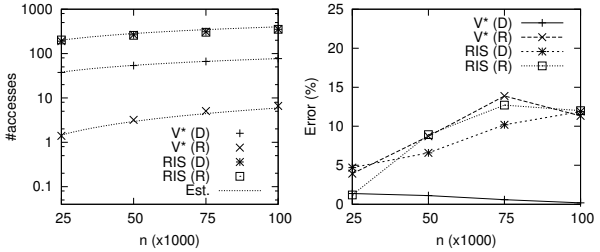


(a) Effect of l (b) Relative errors

Fig. 26 Cost model validation for l on synthetic uniform data

The effect of n. Figure 27 provides a comparison between the experimental results and the three cost models as the value of n is varied from 25,000 to 100,000 objects. Figure 27(a) shows that the communication cost increases as n increases. The cost models accurately capture the number of data accesses for both algorithms. Figure 27(b) shows that for the considered range of n, the relative errors are below 15%.

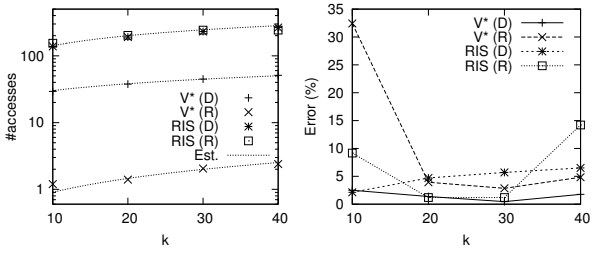
The effect of k. Figure 28 provides a comparison between the experimental results and the three cost models as the value of k is varied from 10 to 40. Figure 28(a) shows that the access cost increases as k increases and the cost



(a) Effect of n (b) Relative errors

Fig. 27 Cost model validation for n on synthetic uniform data

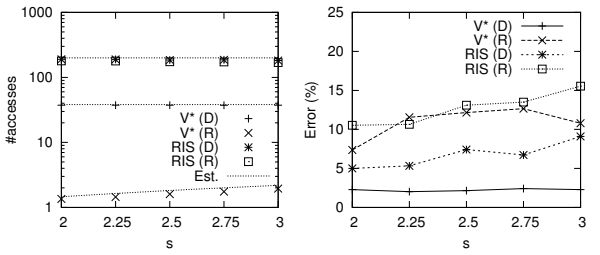
models accurately capture the number of data accesses for both algorithms. Figure 28(b) shows that when k assumes a relatively large value (20), the relative errors are less than 15% and in most cases, the relative errors are below 10%.



(a) Effect of k (b) Relative errors

Fig. 28 Cost model validation for k on synthetic uniform data

The effect of s. Figure 29 shows a comparison between the three cost models and the experimental results as the step size s is varied from 2 to 3. The cost models correctly capture the stable number of accesses for the case of RIS-kNN on both trajectory types and V*-kNN on directional trajectories. The slight increase in the number of accesses for V*-kNN on random trajectories is also captured by the corresponding cost model. Figure 29(b) shows that the relative errors are smaller than 15% in most cases.



(a) Effect of s (b) Relative errors

Fig. 29 Cost model validation for s on synthetic uniform data

8.6 Spatial-network experimental study

The problem of spatial-network MkNN is concerned with maintaining the k objects with the smallest network distances, while the movement of the query point is constrained by the network connectivity. In this experiment, we simulate a scenario of a vehicle travelling in a road network while continuously keeping track of the k nearest gas stations.

Figure 30 illustrates the road network used in this set of experiments. The road network consists of 175,813 nodes and 179,179 edges. The average degree of the network is 2.04. The data objects are uniformly distributed among the edges in the data space of 10,000 $\times$ 10,000 square units.

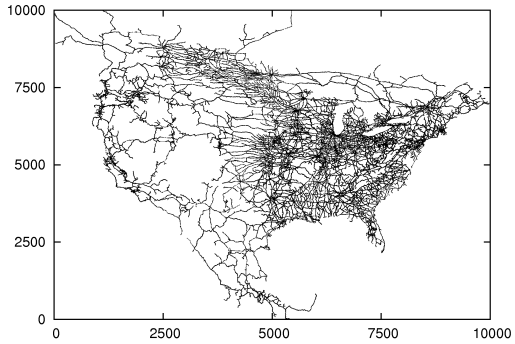


Fig. 30 Road network in North America

In our implementation of Algorithm 1, we use the *incremental network expansion (INE) kNN* algorithm [21] as the access method. To calculate the boundaries of the safe region S_k and the fixed-rank region (FRR), we employ the lazy-evaluation approach discussed in Section 7. That is, we evaluate only S_k and FRR boundaries along the edge that the query point currently occupies.

The results are presented in two cost measures, the total response time (the total cost) and the number of accesses. The total response time is the wall clock time taken to process one trajectory. The number of accesses is the number of INE-kNN calls, which is also indicative of the communication cost in the client-server paradigm[6].

Table 8 gives a summary on the range and default values of the parameters used in the experiments. To provide a practical range of k values in this gas-station scenario, we use the k values between 2 to 10. We use the x values from 2 to 10 to comply with the range of k values. In order to emulate typical interstate driving distances, the range of travelling distances l of [250:1,000] units is used. According to the 2002 economic census [31], the number of nearest gas stations in the U.S. was 121,446 in that year. We therefore set the range of n values to [25,000:125,000] objects. Due

[6] This is under the assumption that the road network is static and is stored on both client and server sides, so communication is required only during each INE-kNN call.

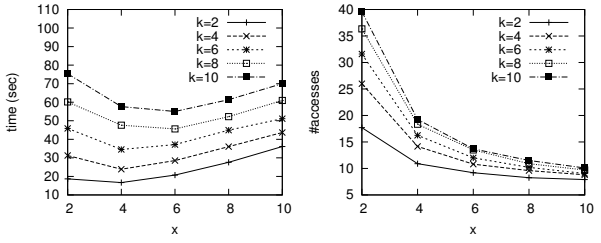
to space limitations, other parameters c, s and δ are set to constant values of 0, 0.5 and 0 respectively.

Table 8 Spatial network: summary of parameters

Parameter	Default Value	Range	Increment
k	-	[2:10]	2
x	6	[2:10]	2
l	250	[250:1000]	250
n	25,000	[25,000:125,000]	25,000

The effect of x. Figure 31(a) shows the total response times as the value of x is varied from 2 to 10 for the k values between 2 to 10. The figure shows that the optimal value of x gradually shifts from 4 to 6 as k increases. This is because the cost for each access via INE-kNN increases as k increases. It therefore becomes more beneficial to increase the number x of auxiliary objects in order to reduce the number of accesses. Since the x value of 6 generally provides a reasonable performance for all experimented k values, we set the default value of x to 6. Figure 31(b) shows the number of accesses for the same ranges of x and k in Figure 31(a). Similar to the Euclidean counterpart in Figure 25(a), the number of accesses reduces as x increases.

Although not displayed here, we have also conducted an experiment on a sampling based approach that does not utilize auxiliary objects and invokes INE-kNN at every location update. We have found that this method has greater response times than V*-kNN (with x equal to 6) by 5.9 times for k of 2 and 6.5 times for k of 10. The number of accesses of the sampling based approach is higher than that of V*-kNN (with x equal to 6) by 54.3 times for k of 2 and 36.5 times for k of 10.

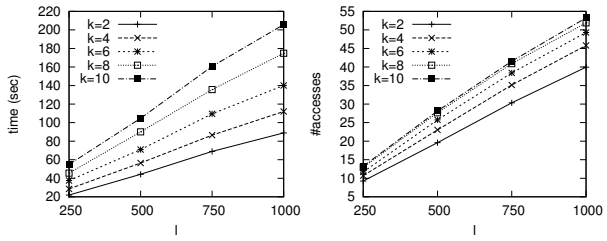


(a) Total cost (b) Number of accesses

Fig. 31 Spatial network: effect of x

The effect of l. Figure 32 shows that both of the total response time and the number of accesses increase as the trajectory length l increases for all values of k. This is because as l increases, the trajectory covers a larger portion of the network and the query point encounters more updates.

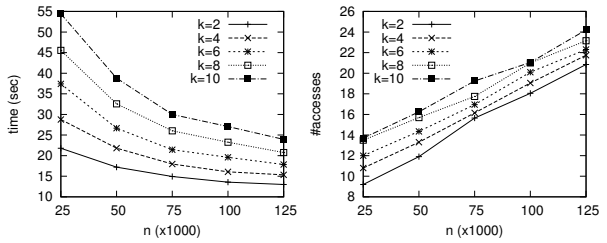
The effect of n. A greater value of n provides a higher object density, which produces two different effects. First,



(a) Total cost (b) Number of accesses

Fig. 32 Spatial network: effect of l

as shown in Figure 33(b), we observe a higher number of accesses, which conforms with the Euclidean counterpart in Figure 27(a). Second, we observe a reduction in the total cost as shown in Figure 33(a). This is because as the density of data objects increases, $(k + x)$ NNs can be retrieved with a smaller network-distance calculation cost during each data access via INE-kNN. This reduction in the cost per access outweighs the increase in the number of accesses and results in a reduction of the total cost.



(a) Total cost (b) Number of accesses

Fig. 33 Spatial network: effect of n

8.7 Summary

V*-kNN consistently outperforms RIS-kNN. The two factors that make V*-kNN superior to RIS-kNN are a lower cost per access and a smaller number of data accesses. V*-kNN, however, incurs an additional CPU cost in order to maintain the order of $(k + x)$ objects. This CPU overhead could be disadvantageous when using a slow mobile device especially with a large value of x.

For V*-kNN, a random trajectory tends to incur a smaller communication cost than a directional trajectory of the same length, since the query point on a random trajectory tends to stay in the same combined safe region for a longer period of time. This effect is also captured by our cost models in Equations (13) and (14). The cost differences between the two trajectory types are less apparent for the total time, which takes CPU costs into account. This is because a similar amount of work is required to keep the $(k + x)$ objects

in order. Finally, the experimental results have shown that for practical ranges of s, l, x, k and n values, the cost models can accurately capture the number of data accesses for V*-kNN and RIS-kNN.

For the spatial-network experiments, the results regarding the number of accesses conform with the Euclidean counterpart. That is, the number of accesses has a positive correlation with the parameters l, n and k, and has a negative correlation with x. For the total response time, choosing the value of x is still a trade off between the number of accesses and the CPU cost. Due to the behavior of the underlying access method, INE-kNN, the value of n has a negative correlation with the total response time, which is opposite to the case for the Euclidean space.

9 Conclusions

In this paper, we have discussed the V*-Diagram and the associated algorithm V*-kNN, which can be used to efficiently process moving k nearest neighbor queries (MkNN). Unlike other safe-region techniques, which only utilizes the knowledge of the data objects, the V*-Diagram exploits also the query location and the scope of the current search space. As a result, the V*-Diagram is more economical in terms of both I/O and CPU costs. We also showed that the V*-Diagram can be applied to spatial networks. We have derived cost models for V*-kNN and a competitive technique, RIS-kNN, and showed that they can accurately predict the number of data accesses for both techniques. We also performed an extensive experimental study and the results show that our algorithm outperforms RIS-kNN by one order of magnitude.

References

1. Beckmann, N., Kriegel, H., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: SIGMOD, pp. 322–331 (1990)
2. Benetis, R., Jensen, C.S., Karciauskas, G., Saltenis, S.: Nearest and reverse nearest neighbor queries for moving objects. VLDB J. **15**(3), 229–249 (2006)
3. Böhm, C.: A cost model for query processing in high dimensional data spaces. ACM Trans. Database Syst. **25**(2), 129–178 (2000)
4. Chávez, E., Navarro, G., Baeza-Yates, R.A., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. **33**(3), 273–321 (2001)
5. Cho, H.J., Chung, C.W.: An efficient and scalable approach to CNN queries in a road network. In: VLDB, pp. 865–876 (2005)
6. Dijkstra, E.W.: A note on two problems in connection with graphs. Numeriche Mathematik **1**, 269–271 (1959)
7. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: SIGMOD, pp. 47–57 (1984)
8. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann (2006)
9. Hjaltason, G.R., Samet, H.: Ranking in spatial databases. In: Symposium on Large Spatial Databases, pp. 83–95 (1995)
10. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Trans. Database Syst. **24**(2), 265–318 (1999)
11. Hu, H., Xu, J., Lee, D.L.: A generic framework for monitoring continuous spatial queries over moving objects. In: SIGMOD, pp. 479–490 (2005)
12. Huang, X., Jensen, C.S., Lu, H., Saltenis, S.: S-GRID: A versatile approach to efficient query processing in spatial networks. In: SSTD, pp. 93–111 (2007)
13. Huang, X., Jensen, C.S., Saltenis, S.: The islands approach to nearest neighbor querying in spatial networks. In: SSTD, pp. 73–90 (2005)
14. Iwerks, G.S., Samet, H., Smith, K.P.: Maintenance of *k*-nn and spatial join queries on continuously moving points. ACM Trans. Database Syst. **31**(2), 485–536 (2006)
15. Kolahdouzan, M.R., Shahabi, C.: Voronoi-based k nearest neighbor search for spatial network databases. In: VLDB, pp. 840–851 (2004)
16. Kolahdouzan, M.R., Shahabi, C.: Alternative solutions for continuous k nearest neighbor queries in spatial network databases. GeoInformatica **9**(4), 321–341 (2005)
17. Kulik, L., Tanin, E.: Incremental rank updates for moving query points. In: GIScience, pp. 251–268 (2006)
18. Mouratidis, K., Papadias, D., Bakiras, S., Tao, Y.: A threshold-based algorithm for continuous monitoring of k nearest neighbors. IEEE Trans. Knowl. Data Eng. **17**(11), 1451–1464 (2005)
19. Nutanong, S., Zhang, R., Tanin, E., Kulik, L.: The V*-Diagram: A query dependent approach to moving kNN queries. In: VLDB, pp. 1095–1106 (2008)
20. Okabe, A., Boots, B., Sugihara, K.: Spatial tessellations: concepts and applications of Voronoi diagrams. John Wiley & Sons, Inc. (1992)
21. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: VLDB, pp. 802–813 (2003)
22. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD, pp. 71–79 (1995)
23. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann (2006)
24. Samet, H., Sankaranarayanan, J., Alborzi, H.: Scalable network distance browsing in spatial databases. In: SIGMOD, pp. 43–55 (2008)
25. Song, Z., Roussopoulos, N.: K-nearest neighbor search for moving query point. In: SSTD, pp. 79–96 (2001)
26. Splitzer, F.: Principles of Random Walk. Springer (2001)
27. Tao, Y., Papadias, D.: Time-parameterized queries in spatio-temporal databases. In: SIGMOD, pp. 334–345 (2002)
28. Tao, Y., Papadias, D.: Spatial queries in dynamic environments. ACM Trans. Database Syst. **28**(2), 101–139 (2003)
29. Tao, Y., Papadias, D., Shen, Q.: Continuous nearest neighbor search. In: VLDB, pp. 287–298 (2002)
30. Tao, Y., Zhang, J., Papadias, D., Mamoulis, N.: An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. IEEE Trans. Knowl. Data Eng. **16**(10), 1169–1184 (2004)
31. U.S. Census Bureau: Economic census (retail trade United States) (2002)
32. Yu, X., Pu, K.Q., Koudas, N.: Monitoring k-nearest neighbor queries over moving objects. In: ICDE, pp. 631–642 (2005)
33. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based spatial queries. In: SIGMOD, pp. 443–454 (2003)