Continuous Spatial Query Processing: A Survey of Safe Region Based Techniques

JIANZHONG QI, University of Melbourne
RUI ZHANG, University of Melbourne
CHRISTIAN S. JENSEN, Aalborg University
KOTAGIRI RAMAMOHANARAO, University of Melbourne
JIAYUAN HE, University of Melbourne

In the past decade, positioning system-enabled devices such as smartphones have become most prevalent. This functionality brings the increasing popularity of *location-based services* in business as well as daily applications such as navigation, targeted advertising, and location-based social networking. *Continuous spatial queries* serve as a building block for location-based services. As an example, an Uber driver may want to be kept aware of the nearest customers or service stations. Continuous spatial queries require updates to the query result as the query or data objects are moving. This poses challenges to the query efficiency, which is crucial to the user experience of a service. A large number of approaches address this efficiency issue using the concept of *safe region*. A safe region is a region within which arbitrary movement of an object leaves the query result unchanged. Such a region helps reduce the frequency of query result update and hence improves query efficiency. As a result, safe region based approaches have been popular for processing various types of continuous spatial queries. Safe regions have interesting theoretical properties and are worth in-depth analysis. We provide a comparative study of safe region based approaches. We describe how safe regions are computed for different types of continuous spatial queries, showing how they improve query efficiency. We compare the different safe region based approaches and discuss possible further improvements.

CCS Concepts: **•General and reference → Surveys and overviews; •Information systems → Multidimensional range search; Location based services; Geographic information systems;**

Additional Key Words and Phrases: Continuous *k*NN query, continuous range query, moving query, moving object

ACM Reference Format:
Jianzhong Qi, Rui Zhang, Christian S. Jensen, Kotagiri Ramamohanarao, and Jiayuan He, 2018. Continuous Spatial Query Processing: A Survey of Safe Region Based Techniques *ACM Comput. Surv.* 1, 1, Article 1 (January 2018), 35 pages.
DOI: 10.1145/3193835

1. INTRODUCTION

Location-based services (LBS) have become increasingly popular due to the proliferation of mobile devices. Applications of LBS include navigation, augmented reality gaming, targeted advertising, and location-based social networking, to name but a few. Different LBS require the support of different types of queries, among which *continuous spatial queries* (CSQ) [Hu et al. 2005; Mouratidis et al. 2009; Chow et al. 2011] constitute an important type. Unlike traditional spatial queries, where the objects are static and queries are computed only once, CSQs must maintain up-to-date query results as the objects move. Processing CSQs is challenging, as query efficiency is critical to the user experience. Many efforts have been devoted to CSQ processing. In what follows, we give an overview of CSQs and the related query processing techniques.

This work is supported in part by Australian Research Council (ARC) Future Fellowships Project FT120100832 and Discovery Projects DP180102050 and DP180103332. C. S. Jensen was supported in part by a grant from the Obel Family Foundation. We thank Lu Fan and Dongge Liu for the useful discussions.
Authors' address: J. Qi, R. Zhang, K. Ramamohanarao, and J. He, School of Computing and Information Systems, University of Melbourne, Australia, Email: {jianzhong.qi, rui.zhang, kotagiri}@unimelb.edu.au, hjhe@student.unimelb.edu.au; C. S. Jensen, Department of Computer Science, Aalborg University, Denmark, Email: csj@cs.aau.dk.

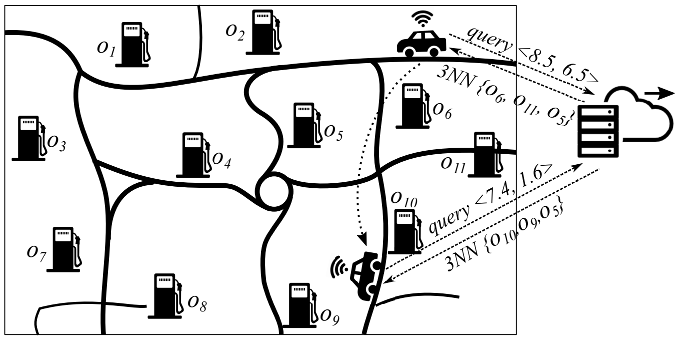
DOI: 10.1145/3193835

1.1. Continuous Spatial Queries

The idea of querying moving objects emerged in the early 1990s [Imielinski and Badrinath 1992], when the deployment of large mobile cellular networks started. Queries such as "*find Alice's nearest petrol stations while she is driving*" or "*find all the taxi cabs within 1 Km distance from Alice*" were envisioned. Ten years later, advanced mobile devices such as *personal digital assistants* had become popular, and CSQs started to attract more attention in the spatial database community. A large number of studies are conducted on various aspects of CSQs, including access methods [Pfoser et al. 2000; Saltenis et al. 2000; Gedik et al. 2004], query algorithms [Kalashnikov et al. 2002; Iwerks et al. 2006; Mokbel and Aref 2008], and new query types [Lee et al. 2009; Ali et al. 2010; Huang et al. 2012], just to name a few examples. A CSQ has been referred to as a *moving query* [Gedik and Liu 2006; Yiu et al. 2011; Huang et al. 2012], an *active query* [Jensen et al. 2003], a *(continuous) spatio-temporal query* [Pfoser et al. 2000; Tao and Papadias 2002; Agarwal et al. 2003], and a *continuous location-based (or location-dependent) query* [Ilarri et al. 2006, 2010; Wang and Zimmermann 2011; Afyouni et al. 2012, 2014]. These names reflect small differences mainly in the targeted settings. The terms moving query, spatio-temporal query, and active query were introduced in part to distinguish the new functionality from that of traditional spatial queries where the objects are static and time-independent. The terms continuous location-based or location-dependent query were introduced in part to emphasize the LBS application context. However, these queries share the following core characteristics:

(1) The queries maintain up-to-date answers from the query issue time until deactivation.
(2) The queries involve a (query) location parameter that may change continuously.
(3) The queries concern the locations of objects capable of continuous movement.
(4) The query processing is typically shared between a client that receives updates to the query location parameter and a server that maintains updates to the moving objects being queried.

In this survey, we focus on the query processing techniques rather than the subtle difference among the query names. We simply refer to queries that satisfy the characteristics above as continuous spatial queries (or CSQs) as long as the context is clear.



Station	Location
o_1	<2.6, 6.5>
o_2	<5.2, 6.7>
o_3	<0.5, 4.5>
o_4	<3.8, 4.1>
o_5	<6.2, 4.7>
o_6	<8.3, 5.0>
o_7	<1.2, 2.2>
o_8	<3.3, 1.2>
o_9	<5.9, 0.9>
o_{10}	<8.2, 2.5>
o_{11}	<9.8, 4.0>

Fig. 1: A typical continuous spatial query

Figure 1 shows a CSQ example. Alice, represented by a car, is driving. She wants to know continuously her three nearest petrol stations. She issues a query containing her location $\langle 8.5, 6.5\rangle$ through her mobile phone. A service provider, represented by a server in the cloud, maintains a database of petrol station locations $o_1, o_2, ..., o_{11}$ (Characteristic 3). The service provider searches the database to find Alice's three nearest petrol stations (Characteristic 4). The result $\{o_6, o_{11}, o_5\}$ is returned to Alice. As Alice drives to $\langle 7.4, 1.6\rangle$ (Characteristic 2), her new location is sent to the service provider to request a query answer update (now $\{o_{10}, o_9, o_5\}$). This process continues and Alice is kept updated with the query answer until she deactivates her query (Characteristic 1).

1.2. Efficiency Issues in Continuous Spatial Query Processing

Processing CSQs involves a wide range of challenges, among which query efficiency is an important aspect since it has significant impact on the user experience of LBS. Therefore, this survey focuses on the efficiency of CSQ processing, i.e., how to maintain up-to-date query results at low costs, including server, client, and communication costs, and with low response time.

Consider again the example in Fig. 1. Assume that Alice requests her nearest petrol stations over a time period that may be partitioned into T time points. A straightforward query system computes and sends the updated nearest petrol stations to Alice at every time point. Let C_p and C_m be the computation and communication costs at one time point, respectively. In the worst case, computing k nearest petrol stations requires a scan over all petrol stations to compute and compare their distances from Alice with the aid of a priority queue of size k. Assume that there are n petrol stations. Then, $C_p = O(n \cdot \log k)$, and the worst-case computation cost for T time points is $O(T \cdot n \cdot \log k)$. The cost of transmitting k nearest petrol stations (without compression) is linear to k, i.e., $C_m = O(k)$. The worst-case communication cost is $O(T \cdot k)$. Popular LBS may involve millions of data objects and query users, e.g., Foursquare has 100 million venues and 55 million active users [Smith 2016]. Straightforward algorithms are not attractive in such settings.

Many studies have been devoted to reduce the costs of CSQ processing. The *safe region* technique that is used commonly in many of these studies is the focus of this survey. A safe region is a region that allows arbitrary movements of an object without causing any changes to the query result. Safe regions allow the query processor to ignore some location updates of the query and data objects. A popular implementation of this technique called *geo-fencing* [Andriod 2017; Apple 2017] is used by mobile applications to reduce location tracking costs, including battery consumption, which is an important aspect in the user experience of mobile applications.

Note that most of the studies surveyed involve some index structure over the objects. For static objects, hierarchical indexes such as R-trees [Guttman 1984] and Quadtrees [Finkel and Bentley 1974], and grid-based indexes such as Grid files [Nievergelt et al. 1984] and space-filling curves [Orenstein and Merrett 1984] are frequently used. For moving objects, TPR-trees [Saltenis et al. 2000] and B-tree based indexes [Jensen et al. 2004] have been used. These index structures are not the focus of this survey. Interested readers are referred to surveys on these index structures [Gaede and Günther 1998; Böhm et al. 2001; Mokbel et al. 2003; Nguyen-Dinh et al. 2010].

Several existing surveys [Krumm 2009; Ilarri et al. 2010; Hendawi and Mokbel 2012; Silva et al. 2014] cover other aspects of CSQ processing. Hendawi and Mokbel [2012] discuss predictive CSQs and their challenges; Krumm [2009] discusses privacy issues in CSQs; Silva et al. [2014] discuss characteristics of CSQs in wireless sensor networks. For a comprehensive discussion of CSQ processing, interested readers are referred to the survey by Ilarri et al. [2010], which covers topics from hardware system architecture to different CSQ processing frameworks and algorithms. Their survey focuses on classifying the existing literature. Thus, the studies covered there are categorized based on criteria such as the problem setting, e.g., whether or not object trajectories are known. Their survey helps choose query algorithms in different categories. The focus is not to cover safe region techniques, which are the topic of the present survey.

Specifically, we cover the key idea underlying safe region techniques and how it is used in different CSQ algorithms to support a variety of query types. We also cover developments in CSQ processing (based on safe region techniques) that have appeared since the survey by Ilarri et al. [2010]. We aim for a tutorial-type survey that communicates key ideas and concepts and enables readers to apply safe region techniques to process new types of CSQs.

A related concept in CSQ processing is that of a *safe period*. While safe regions guarantee the query result validity in the spatial dimension, safe periods guarantee this in the temporal dimension. A safe period is a time period during which the query result is guaranteed to stay valid, thus not requiring reevaluation. Safe periods are often computed based on a given maximum moving speed of the objects. Examples of this technique can be found in the literature [Tao and Papadias 2002; Gedik and Liu 2004; Hu et al. 2005; Gedik and Liu 2006; Do et al. 2009; Zhang et al. 2012].

1.3. Problem Formulation

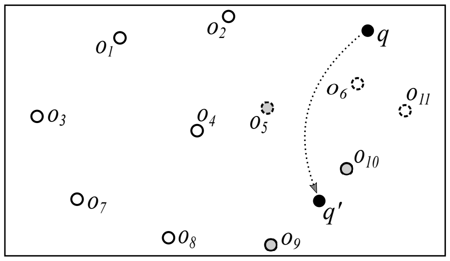
We first present terms and definitions used in this survey. We use q to denote a query object (query location parameter). We use $\mathsf{O} = \{o_1, o_2, ..., o_n\}$ to denote a database of n data objects which can be either points or objects with non-zero extent. The data objects are represented by their coordinates in a data space DS. By default, a two-dimensional Euclidean space is used, i.e., $DS \in \mathbb{R}^2$. A point object o_i is represented as $\langle o_i.c_1, o_i.c_2\rangle$ where c_j denotes the coordinate in dimension j $(j = 1,2)$. If o_i has non-zero extent, it is represented by its *minimum bounding rectangle* (MBR) $\langle o_i.c_{1-}, o_i.c_{1+}, o_i.c_{2-}, o_i.c_{2+}\rangle$. Here, c_{j-} and c_{j+} denote lower and upper bounds of the MBR in dimension j, respectively. Either q or the data objects (or both) may update their locations. We use $\texttt{dist}(\cdot)$ to denote the distance function. By default, it is the Euclidean distance:

$$\texttt{dist}(q, o_i) = \sqrt{\sum_{j=1}^{2} (q.c_j - o_i.c_j)^2}$$

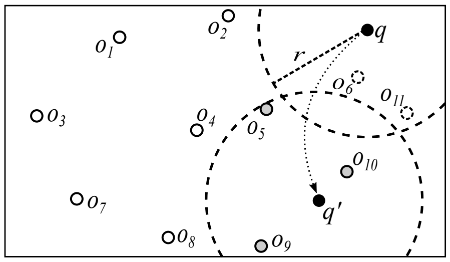
We focus on two types of queries due to their importance. These are also the building blocks for many other types of CSQs.

Definition 1 (Continuous k Nearest Neighbor Query, CkNN)**.** *Given a query object q, a set of data objects O, and a parameter k, the continuous k nearest neighbor query maintains (from being issued until deactivated) a size-k subset $\mathsf{S} \subseteq \mathsf{O}$ such that $\forall o_i \in \mathsf{S}, o_j \in \mathsf{O}\backslash\mathsf{S} : \texttt{dist}(q, o_i) \le \texttt{dist}(q, o_j)$.*

Figure 1 provides an example, where Alice is the query object and the petrol stations are the data objects. To highlight the essential components of the query, we redraw the figure to be Fig. 2(a), where q represents the query object and $\{o_1, o_2, ..., o_{11}\}$ represent the data objects. To simplify the discussion, we omit the underlying road network for now. The current 3NNs are $\{o_6, o_{11}, o_5\}$ (the dashed circles). As q moves to q', the 3NNs change to $\{o_{10}, o_9, o_5\}$ (the gray dots).



(a) A CkNN query



(b) A CR query

Fig. 2: Examples of continuous spatial queries

The CkNN query [Song and Roussopoulos 2001; Tao et al. 2002; Iwerks et al. 2003; Kolahdouzan and Shahabi 2005; Mouratidis et al. 2005a,b; Yu et al. 2005; Benetis et al. 2006; Nutanong et al. 2010; Zhang et al. 2010; Li et al. 2014, 2016] is one of the most popular types of CSQs. Many studies are devoted to the CkNN query and its variants, such as the continuous reverse kNN queries [Benetis et al. 2006; Cheema et al. 2012; Zeberga et al. 2017].

Definition 2 (Continuous Range Query, CR)**.** *Given a query object q and a region r, a set of data objects O, and a query predicate $\texttt{pred}(\cdot)$, the continuous range query maintains (from being issued until deactivated) the subset $\mathsf{S} \subseteq \mathsf{O}$ that contains every object $o_i \in \mathsf{O}$ where $\texttt{pred}(q, r, o_i) = \texttt{true}$.*

The CR query [Kalashnikov et al. 2002; Zhang et al. 2003; Cai et al. 2004; Wang and Zimmermann 2007; Hsueh et al. 2009; Cheema et al. 2010, 2011; Wang and Zimmermann 2011; AL-Khalidi et al. 2013; Huang and Huang 2013; Cho et al. 2014] is also an important type of CSQs. The query object q here is usually a point, and the query predicate $\texttt{pred}(q, r, o_i)$ tests whether o_i is within (or intersects) distance r (or region r) of q. A query example is to "*find the points-of-interest (POI) within 1 Km of Alice as she is sightseeing in a city.*" Here, the query region is the region within 1 Km of Alice. Figure 2(b) illustrates a CR query with a circular query region, i.e., the dashed circle centered at q with radius r. The data objects within the query region are $\{o_6, o_{11}\}$. When the query object moves to q', the data objects in the query region become $\{o_5, o_9, o_{10}\}$.

Circular and rectangular query regions have been mostly used. When a rectangular query region is used, the query is also referred to as a *continuous window query* [Huang and Huang 2013]. Different query predicates pred(·) have been used in CR queries, with the most common one being *containment* [Kalashnikov et al. 2002; Zhang et al. 2003, 2004; Hsueh et al. 2009; Cheema et al. 2010, 2011; Huang and Huang 2013; Cho et al. 2014]. This predicate finds data objects that are entirely contained in the query region. In the case of a circular query region (where q and r represent the query center and radius, respectively):

$$\texttt{pred}(q, r, o_i) = \begin{cases} \texttt{true}, \text{ if } \texttt{dist}(q, o_i) \le r \\ \texttt{false}, \text{ otherwise} \end{cases}$$

In the case of a rectangular query region, where the query region r is a rectangle:

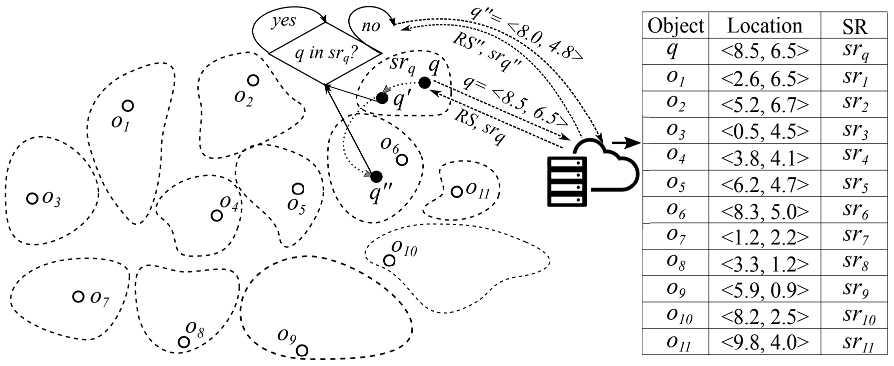
$$\texttt{pred}(q, r, o_i) = \begin{cases} \texttt{true}, \text{ if } o_i \text{ is contained in } r \\ \texttt{false}, \text{ otherwise} \end{cases}$$

Similar to the C*k*NN queries, a large number of studies consider variants of the CR query such as the continuous spatial join query [Tao and Papadias 2002; Zhang et al. 2008, 2012; Ward et al. 2014], which can be seen as a group of CR queries.

Content organization. The rest of the survey is organized as follows. We first present common principles of safe region based query processing in Section 2. Then, we describe how safe regions have been used to process two basic types of CSQs in Sections 3 and Section 4. We conclude the survey by discussing further developments of safe region based techniques in Section 5.

2. SAFE REGION BASED CONTINUOUS SPATIAL QUERY PROCESSING

Most CSQ systems assume a query object (user) and a set of data objects, both of which may be moving. The client-server model, as illustrated in Fig. 3, is commonly used for these systems (e.g., [Gedik and Liu 2006; Ilarri et al. 2006]). The query and/or the data objects are the clients, which report their locations to a server. The server computes the query answer and sends it back to the query object. This computation process repeats, and up-to-date answers are produced by each recomputation. We focus on query processing techniques based on this centralized model.



Object	Location	SR
q	<8.5, 6.5>	sr_q
o_1	<2.6, 6.5>	sr_1
o_2	<5.2, 6.7>	sr_2
o_3	<0.5, 4.5>	sr_3
o_4	<3.8, 4.1>	sr_4
o_5	<6.2, 4.7>	sr_5
o_6	<8.3, 5.0>	sr_6
o_7	<1.2, 2.2>	sr_7
o_8	<3.3, 1.2>	sr_8
o_9	<5.9, 0.9>	sr_9
o_{10}	<8.2, 2.5>	sr_{10}
o_{11}	<9.8, 4.0>	sr_{11}

Fig. 3: An overview of the safe region technique

The frequency of location changes and query recomputation affects the query costs. Here, the frequency of location changes is an inherent problem parameter. It is not controllable by the query server. The frequency of query recomputation, on the other hand, is a solution parameter and depends on the techniques used by the system.

Time point sampling is a simple way to control the frequency of query recomputation, i.e., the query is recomputed only at sample time points. However, this approach only produces accurate answers at the sample time points. Between two sample time points, the query is merely an approximation. When objects move with a high speed, this approximation may have low accuracy.

The safe region technique is another way to control the frequency of query recomputation and the workload at each recomputation. The dashed closed curves in Fig. 3 exemplify safe regions.

— A safe region for a query object q, denoted by sr_q, is a region inside which q can move arbitrarily without causing any change in the query answer, and hence no query recomputation is required. Only when q leaves sr_q (e.g., at q''), recomputation is required. This way, the frequency of query recomputation is reduced to the frequency of q leaving its safe regions.
— A safe region for a query answer and a data object o_i, denoted by sr_i, is a region inside which o_i can move arbitrarily without causing any change in the query answer. Any query recomputation can safely discard o_i as long as o_i remains inside sr_i. This way, the workload of each query recomputation can be reduced.

Figure 3 shows how the safe region (e.g., sr_q) is sent together with the query answer RS to the object of interest (e.g., q). Then the checking of whether the object remains inside its current safe region is delegated to the client. This will further reduce the costs at the server side.

When the safe region technique is used, *the key is to identify the spatial boundary where the spatial relationships between the objects change.* Consider the CkNN query as an example. A safe region for the query object should identify the boundary where the nearness ranks of the data objects change. When the query object moves within this boundary, the nearness ranks of the data objects do not change. Thus, the query answer is unaffected.

When applying the safe region technique, three questions must be answered:

(1) How to define a *tight* (i.e., maximized) safe region to minimize recomputation frequency?
(2) How to compute and update the safe region efficiently?
(3) How to verify efficiently whether the object of interest is still in its safe region?

These questions usually need to be considered together for maximal efficiency. Ideally, we may aim for a safe region that is tight and is efficient to compute, update, and verify. However, computing a tight safe region may require examining a large number of objects to determine where their spatial relationships change, which may have a high cost. Further, a tight safe region may have an irregular shape, which can render verifying whether an object is still in it expensive. On the other hand, a safe region that is efficient to compute and verify is less likely to be tight, which increases the frequency of query recomputation. A common approach is to use a safe region that is subset of a tight safe region (Any subset of a safe region is also a safe region), has a regular shape, and can be computed efficiently, e.g., a circle or a rectangle. Approximation and aggregation techniques are used in computing such safe regions.

How to design a safe region that is well-balanced in tightness and computational efficiency is query-dependent. In the next two sections, we detail how safe regions are designed for continuous k nearest neighbor queries and continuous range queries. The underlying ideas have been applied in the context of other types of CSQs, such as continuous reverse kNN queries [Attique et al. 2016; Zeberga et al. 2017], continuous visible kNN queries [Wang et al. 2014a; Li et al. 2015], and continuous k diversified NN queries [Gu et al. 2016a], to name but a few.

3. SAFE REGIONS IN CONTINUOUS *K* NEAREST NEIGHBOR QUERIES

For CkNN queries, the safe region of an object retains the nearness ranks among the data objects when this object is moving. Safe regions may be built for either query objects or data objects.

When built for a query object q, a safe region ensures that q is nearer to the kNN objects than to any non-kNN object. A simple safe region is defined by the *lazy search* technique [Song and Roussopoulos 2001] based on the minimum distance that q needs to move such that the $(k+1)^{st}$ NN would become nearer than the k^{th} NN, assuming that q lies on the line segment connecting the k^{th} and the $(k+1)^{st}$ NNs. Stricter safe regions bound the movement of q according to all non-kNN objects rather than just the $(k+1)^{st}$ NN. Such safe regions are computed based on the *perpendicular bisectors* between kNN objects and non-kNN objects. The perpendicular bisector between two objects o_i and o_j (in Euclidean space) partitions the space into two half-planes. When q is in the half-plane on o_i's side, it is nearer to o_i than to o_j, and vice versa. The perpendicular bisectors for all pairs of data objects create a partition of the space, i.e., an (order-k) *Voronoi diagram* [Okabe et al. 1992]. Every resulting region is an (order-k) Voronoi cell and is a safe region corresponding to a kNN answer. The query object can move arbitrarily in such a region without invalidating the corresponding kNN answer.

Safe regions based on Voronoi diagrams are tight. However, they are expensive to compute due to the $O(n^2)$ pairs of data objects to be considered. More advanced safe regions are essentially all based on the idea of Voronoi diagrams, but aim to reduce the computational costs. Three algorithms, *FindkNN* [Benetis et al. 2001, 2006], *TPkNN* [Tao and Papadias 2002], and *kCNN* [Tao et al. 2002], compute safe regions along a fixed moving direction of the query object. Then, only data objects that create perpendicular bisectors intersecting the query moving direction need to be considered. The *RIS-kNN* algorithm [Zhang et al. 2003] shows that a finite subset of all possible moving directions is sufficient to compute a safe region equivalent to an order-k Voronoi cell. Thus, only data objects related to this finite subset of moving directions need to be considered. The *IRU* algorithm [Kulik and Tanin 2006] sorts all data objects in ascending order of their distances to q and only computes the perpendicular bisector between every pair of adjacent data objects in the sorted order, i.e., computes $O(n)$ perpendicular bisectors. The kNN answer is valid as long as q does not cross any perpendicular bisector. The *V*-digram* [Nutanong et al. 2008] combines the ideas of lazy search and IRU such that when the $(k+1)^{st}$ NN remains far away from q, only the perpendicular bisectors between the first $k+1$ NNs need to be considered. The *INS-kNN* algorithm [Li et al. 2014] replaces order-k Voronoi-diagrams with order-1 Voronoi-diagrams, which are much more efficient to compute. It is shown that such a replacement does not jeopardize the tightness of the safe regions.

When built for a data object, e.g., the i^{th} NN object o^i, a safe region keeps o^i as the i^{th} NN by bounding its movement based on the locations of the $(i-1)^{st}$ and the $(i+1)^{st}$ NN objects, o^{i-1} and o^{i+1}. The *SRB* algorithm [Hu et al. 2005] computes such a safe region assuming query objects at fixed locations. It uses rectangular safe regions. The safe regions of o^{i-1} and o^{i+1} bound a ring centered at a query object in which o^i can move arbitrarily without invalidating its nearness rank. The safe region of o^i is a rectangle bounded by this ring.

For spatial networks, the safe regions are still based on the concept of perpendicular bisectors, although a perpendicular bisector now becomes a set of "bisector points" on network edges, where each point partitions the network into two halves. Safe regions for spatial networks are extensions of those for Euclidean space as discussed above, where the network shortest path distance is used as the distance metric instead of Euclidean distance. The *VN*^{3} algorithm [Kolahdouzan and Shahabi 2004b] extends Voronoi diagrams to *network Voronoi diagrams*; the *IE* algorithm [Kolahdouzan and Shahabi 2004a] extends kCNN, and the *UB* algorithm [Kolahdouzan and Shahabi 2004a] further integrates the idea of lazy search; the *UNICONS* algorithm [Cho and Chung 2005] extends IE to directed networks; and the *network V*-diagram* [Nutanong et al. 2010] extends the V*-digram. In these extensions, graph traversals are used to find bisector points.

Figure 4 summarizes the safe region techniques mentioned above. We cover safe regions for moving query objects and for moving data objects in Euclidean space in Sections 3.1 and 3.2, respectively. We cover safe regions for moving query objects in spatial networks in Section 3.3.

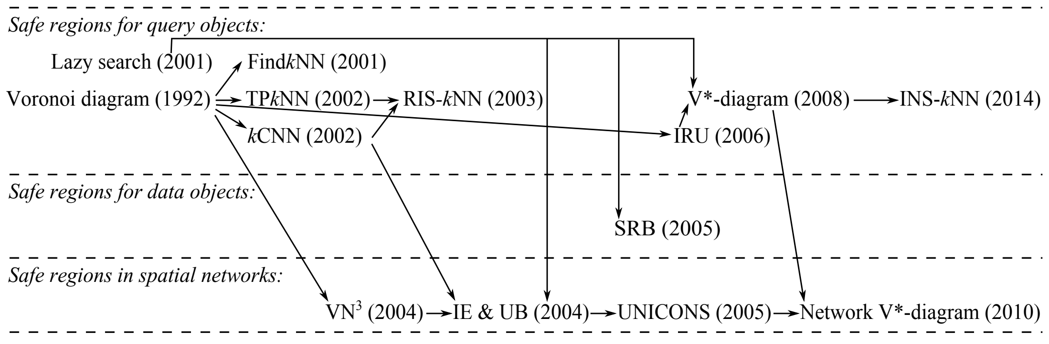


Fig. 4: Evolution of safe region techniques for CkNN queries

3.1. Safe Regions for Query Objects

We first consider safe regions for query objects that are moving. A safe region bounds the query object so that its movement in the region does not change the nearness ranks of the current kNN objects and non-kNN objects. All studies discussed except for those by Benetis et al. [2001, 2002, 2006] assume that the data objects are stationary.

We start with a simple safe region [Song and Roussopoulos 2001] of CkNN queries (Section 3.1.1). We then revisit a classic technique for nearest neighbor finding, the Voronoi diagram [Okabe et al. 1992], and discuss how to use it to construct safe regions (Section 3.1.2). Modern CkNN algorithms build more advanced safe regions based on the Voronoi diagram. We cover those algorithms in Sections 3.1.3, 3.1.4, and 3.1.5. In particular, Section 3.1.3 covers algorithms that compute safe regions based on a given moving direction of the query object, including the Find$k$NN [Benetis et al. 2006], TPkNN [Tao and Papadias 2002], and kCNN [Tao et al. 2002] algorithms. Safe regions are computed for a linear path along which the query object moves, resulting in linear safe regions. This subsection also covers the RIS-kNN algorithm [Zhang et al. 2003] that considers several moving directions to compute a non-linear safe region that is sufficient to cover arbitrary moving directions of the query object. Section 3.1.4 covers another algorithm, named V*-diagram [Nutanong et al. 2008], that computes a safe region that allows arbitrary moving directions of the query object. This algorithm computes an approximate safe region, the goal being to achieve higher computational efficiency. Section 3.1.5 covers the INS-kNN algorithm [Li et al. 2014] that computes safe regions implicitly via computing a set of safeguarding objects. The safeguarding objects define a tight safe region and are efficient to compute. Thus, INS-kNN achieves high computational efficiency without sacrificing the tightness of the safe regions.

3.1.1. The Lazy Search Technique. Song and Roussopoulos [2001] process a CkNN query by sampling points from the trajectory of the query object. For every sampled point, a kNN algorithm for stationary objects (e.g., [Roussopoulos et al. 1995]) is used to compute the query answer at that point. Three incremental computation techniques are proposed to use the kNN answer of the preceding sampled points to construct the next kNN answer. A fourth technique proposed, *lazy search*, employs the safe regions implicitly. This technique uses the distance that the query object needs to move before the current $(k+1)^{st}$ NN can become a new kNN to possibly skip the kNN computation for some sampled points. Assume that at a sampled point q, the k^{th} and $(k+1)^{st}$ NNs are o^k and o^{k+1}, respectively. Let q be on the line segment between o^k and o^{k+1} (cf. Fig. 5). The distances between q and the other two points are $\texttt{dist}(q, o^k)$ and $\texttt{dist}(q, o^{k+1})$. Let q move towards o^{k+1} by a distance of δ. For q to become nearer to o^{k+1}, δ needs to satisfy:

$$\texttt{dist}(q, o^k) + \delta > \texttt{dist}(q, o^{k+1}) - \delta \quad \Rightarrow \quad \delta > (\texttt{dist}(q, o^{k+1}) - \texttt{dist}(q, o^k))/2$$

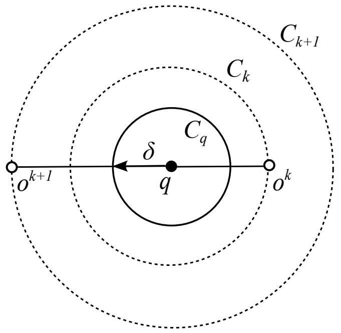


Fig. 5: Safe region of the lazy search technique

Therefore, the minimum distance $\delta_\perp$ that q needs to move to become nearer to o^{k+1} is $(\texttt{dist}(q, o^{k+1}) - \texttt{dist}(q, o^k))/2$. Further, $\texttt{dist}(q, o^k)$ is an upper bound on the distance between q and any kNN object—it defines a circle C_k enclosing the kNN objects. Distance $\texttt{dist}(q, o^{k+1})$ is a lower bound on the distance between q and any non-kNN object. It also defines a circle C_{k+1}.

All non-kNN objects are on or outside this circle. When moving no farther away than distance $\delta_\perp$, q will not become nearer to any non-kNN object than to any current kNN object. This yields a circular safe region centered at q with $\delta_\perp$ as the radius (cf. the solid circle C_q in Fig. 5). As long as the query object remains inside this circle, no kNN computation is required.

Subsequent studies use safe regions more explicitly and try to provide accurate kNN answers continuously instead of only at sampled points. However, the core idea in safe region computation is similar, i.e., to identify the boundary where a data object may become nearer to q than any existing kNN object. Traditionally, the *Voronoi diagram* has been used for this purpose.

3.1.2. The Voronoi Diagram. Voronoi diagrams are widely used in computational geometry. We focus only on how this technique is employed in CkNN algorithms. For a comprehensive discussion, interested readers are referred to excellent textbooks on this topic [Preparata and Shamos 1985; Okabe et al. 1992; Berg et al. 2000].

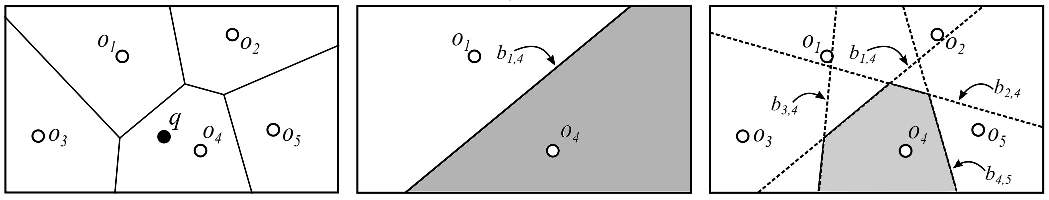
A Voronoi diagram is a partition of a space where the resultant regions are called *Voronoi cells.*

Definition 3 (Voronoi Cell). *Given a space DS and a set of objects* $\mathsf{O} = \{o_1, o_2, ..., o_n\}$, *the Voronoi cell of* o_i, *denoted by* $vc(o_i)$, *is the set of all points in DS for which* o_i *is the nearest object among all the objects in* O. *Formally, let p be a point in DS. Then:*

$$vc(o_i) = \{p \in DS | \forall o_j \in \mathsf{O} \setminus \{o_i\} : \mathtt{dist}(p, o_i) \le \mathtt{dist}(p, o_j)\}$$

Figure 6(a) shows an example of the Voronoi cells for objects $\{o_1, o_2, ..., o_5\}$, where each cell encloses its corresponding object, e.g., the cell enclosing o_4 is the Voronoi cell of o_4. The Voronoi cells of all the objects form the Voronoi diagram of O, denoted by $\mathsf{V_O}$:

$$\mathsf{V_O} = \bigcup_{o_i \in \mathsf{O}} vc(o_i)$$



(a) A Voronoi diagram (b) Dominance region of o_i over o_j (c) Dominance region intersection

Fig. 6: Voronoi diagram computation by intersection

Voronoi cells are natural safe regions for continuous NN queries. In Fig. 6(a), the query object q is in $vc(o_4)$, and o_4 is the NN. As long as q stays in the same cell, the NN remains the same.

In two-dimensional Euclidean space, a Voronoi diagram is formed by the perpendicular bisectors[1] between the objects. As Fig. 6(b) shows, a bisector $b_{1,4}$ between two objects o_1 and o_4 divides the space into two half planes. Each of o_1 and o_4 is the nearest object in its own half plane. The two half planes are called the *dominance regions* of o_1 over o_4 and o_4 over o_1 (the gray region), respectively. In Fig. 6(c), the gray region is the intersection of the dominance regions of o_4 over all the other objects. This region is $vc(o_4)$. Computing a Voronoi diagram by computing the bisectors between every pair of objects has an $O(n^2)$ time complexity. Many efforts have been devoted to reduce this time complexity. The *plan sweep* algorithm [Fortune 1987; Seidel 1988], for example, reduces the time complexity to $O(n \log n)$. For a detailed discussion on Voronoi diagram computation, interested readers are referred to [Berg et al. 2000].

Order-k Voronoi diagrams. The Voronoi diagrams shown above are *order*-1 *Voronoi diagrams*, i.e., every cell corresponds to one object only. In general, for an *order-k Voronoi diagram* ($k \in \mathbb{N}^+$), every cell corresponds to k objects, and the cell is called an *order-k Voronoi cell*.

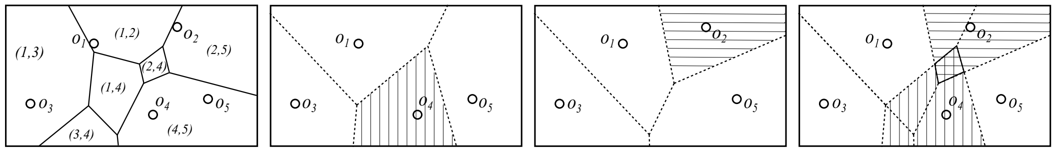
Definition 4 (Order-k Voronoi Cell). *Given a space DS and a set of objects* $\mathsf{O} = \{o_1, o_2, ..., o_n\}$, *let* $\mathsf{S} = \{o^1, o^2, ..., o^k\}$ *be a size-k subset of* O. *The order-k Voronoi cell of* S, *denoted by* $vc_k(\mathsf{S})$, *is the set*

[1]We use bisector in the following discussion when the context is clear.

of all points in DS that have S *as their set of k nearest objects among all the objects in* O*. Formally, let p be a point in DS. Then:*

$$vc_k(\mathsf{S}) = \{p \in DS | \max_{o^i \in \mathsf{S}} \mathtt{dist}(p, o^i) \leq \min_{o_j \in \mathsf{O} \setminus \mathsf{S}} \mathtt{dist}(p, o_j)\}$$

Figure 7(a) shows an order-2 Voronoi diagram, where each cell is labeled with a pair (i, j) that indicates the corresponding pair of objects o_i and o_j. For example, the cell in the middle labeled with $(2, 4)$ is the order-2 Voronoi cell $vc_2(\{o_2, o_4\})$. When q is in this cell, o_2 and o_4 are the 2NN. As the figure shows, higher order Voronoi cells do not always enclose their corresponding objects.



(a) Order-2 Voronoi diagram (b) Diagram without o_2 (c) Diagram without o_4 (d) Order-2 Voronoi cell

Fig. 7: Order-2 Voronoi diagram

A straightforward way to compute an order-k Voronoi cell is to compute the intersection of k order-$(k-1)$ Voronoi cells. Each order-$(k-1)$ Voronoi cell guarantees a size-$(k-1)$ subset to be the $(k-1)$NN set. The intersection then guarantees the k objects to be the kNN set. Figures 7(b) to 7(d) illustrate the computation of an order-2 Voronoi cell. Two order-1 Voronoi diagrams are computed, one excluding o_2 (Fig. 7(b)) and the other excluding o_4 (Fig. 7(c)), respectively. The intersection of $vc(o_4)$ and $vc(o_2)$ is $vc_2(\{o_2, o_4\})$, i.e., the cross-lined region in Fig. 7(d). Chazelle and Edelsbrunner [1987] propose an algorithm to compute an order-k Voronoi diagram without the lower order Voronoi diagrams. It has a time complexity of $O(n^2 \log n + k(n-k) \log^2 n)$ or $O(n^2 + k(n-k) \log^2 n)$, depending on the data structures used. This algorithm is not the focus of this survey and is not discussed further. Nevertheless, such an algorithm is still too expensive to be used in applications that require frequent updates and recomputations of Voronoi diagrams.

Continuous query processing. The order-k Voronoi cells can be precomputed and stored in a spatial index, e.g., the VoR-tree [Sharifzadeh and Shahabi 2010], which is an R-tree augmented to store data objects together with their Voronoi cells. When a CkNN query is issued, we locate the order-k Voronoi cell that the query object q is in and return the corresponding data objects as the kNN answer. As long as q remains in the same cell, no further processing is required. Otherwise, we identify the new cell that q has entered and update the kNN answer accordingly.

Updates such as a change in the value of k or data objects being inserted or removed require a recomputation of the Voronoi diagram. The high cost of computing the order-k Voronoi diagram renders it impractical in a dynamic environment with frequent updates. This has led to studies that compute local safe regions on-the-fly rather than precomputing the full safe regions.

3.1.3. Query Object Moving Direction Dependent Safe Regions.

A series of studies [Benetis et al. 2001, 2002, 2006; Tao and Papadias 2002; Tao et al. 2002; Zhang et al. 2003] have been conducted on computing safe regions locally based on the concept of *influence points*.

Tao and Papadias [2002] assume a query object with a fixed velocity and static data objects. They compute a kNN answer and a *safe time span* during which the answer remains valid. Given a fixed velocity, the safe time span defines a line segment as a safe region for the query object. Benetis et al. [2001, 2002, 2006] assume fixed velocities for both the query and data objects and compute the kNN result for a given query time interval. The result contains a set of pairs. Each pair contains a set of objects and a time interval such that (i) the time intervals partition the query interval and (ii) the objects in a set is the kNN answer during the corresponding interval. This functionality generalizes that of Tao and Papadias [2002] in that the time interval of the first pair is the safe time span, if the safe time span is contained in the query interval. Tao et al. [2002] assume a query object moving along a line segment and break it into sub-segments each of which is a safe region for a different kNN answer. Zhang et al. [2003] explore all possible moving directions of the query object and obtain a safe region equivalent to an order-k Voronoi cell.

The Find$k$NN algorithm. Benetis et al. [2001, 2002, 2006] model the movement of a moving object by its position $\langle x, y\rangle$ at time t_0 and a velocity vector $\overrightarrow{v} = \langle v_x, v_y\rangle$, where v_x and v_y denote the speed in the x- and y-dimension, respectively. Given a set O of such objects, Benetis et al. propose the *FindkNN* algorithm to compute the kNN objects for a moving query object q for a query time interval $[t_-, t_+]$, where $t_- \geq t_{issue}$ and t_{issue} denotes the query issue time.

The Find$k$NN algorithm partitions $[t_-, t_+]$ into multiple intervals, where the kNN result during the i^{th} interval T_i is denoted by $\mathsf{S}_i = \langle o_{i1}, o_{i2}, \ldots, o_{ik}\rangle$. The objects in S_i are ordered by their Euclidean distances to q, i.e., object o_{ik} is the k^{th} nearest object to q throughout T_i:

$$\forall i : \texttt{dist}(q, o_{i1}) \leq \texttt{dist}(q, o_{i2}) \leq \ldots \leq \texttt{dist}(q, o_{ik}) \wedge \left(\forall o \in \mathsf{O} \setminus \mathsf{S}_i : \texttt{dist}(q, o_{ik}) \leq \texttt{dist}(q, o)\right)$$

Each interval T_i corresponds to a line segment that is a safe region for the query object, i.e., the segment that the linearly moving query object will traverse from the start to the end of T_i.

We proceed to describe how Find$k$NN partitions the query interval using $k = 1$. The algorithm assumes that the set O is indexed by a TPR-tree [Saltenis et al. 2000], which is an R-tree extended to index moving objects; in the TPR-tree, the sides of bounding rectangles expand linearly with time to bound the rectangles and objects they contain. To compute the result, the algorithm traverses the TPR-tree on O using a temporal generalization of the branch-and-bound procedures described by Roussopoulos et al. [1995] and Hjaltason and Samet [1999].

The algorithm uses the squared Euclidean distance, which is simpler than Euclidean distance and yields the same result because it preserves relative distances. Function $d(q, o, t)$ denotes the distance between the query object q and a data object o at time t. As q and o move linearly, $d(q, o, t) = at^2 + bt + c$, where $t \in [t_-, t_+]$, and a, b, and c are constant parameters. To facilitate the index traversal, an additional function $d(q, R, t)$ is used that denotes the distance between q and its nearest point on a rectangle R at time t. A time interval $[t_-, t_+]$ can be partitioned into at most five intervals so that each interval T_i corresponds to a distance function $d_i(q, R, t) = a_i t^2 + b_i t + c_i$, where $t \in T_i$ and a_i, b_i, and c_i are constant parameters. When q enters R, $d_i(q, R, t)$ becomes zero.

The algorithm uses a min-heap Q to control the order of tree traversal. Two functions *push* and *pop* are used to insert an entry (a TPR-tree node) into and remove an entry from Q, respectively. When *pop* is called, the entry with the smallest key value from Q is returned. To compute the key value of an entry e, two factors are considered: (i) the level of e in the tree, and (ii) a *representative distance* between e and q during $[t_-, t_+]$, denoted by $rd(q, e)$. If only the representative distance is used to define the key, the traversal is best-first, where the next entry to be visited has the smallest representative distance in Q. If both the representative distance and the level of the entry are used, the traversal is depth-first, assuming that the root has the largest level number.

The representative distance $rd(q, e)$ is a temporal version of `mindist` [Roussopoulos et al. 1995; Hjaltason and Samet 1999]. Given a query time interval $[t_-, t_+]$ and the MBR R of e, distance $rd(q, e)$ is defined as follows, where the integral computes the average of the squared Euclidean distance between the MBR of e and q, multiplied by the query time interval length.

$$rd(q, e) = \int_{t_-}^{t_+} d(q, R, t)\, dt$$

When the algorithm starts, the result interval has one partition which is the query interval. As the TPR-tree is traversed, distances are repeatedly compared to find nearest objects. As the distance functions are quadratic, the comparison of distances corresponds to solving quadratic inequalities. Such inequalities can have at most two roots, and these roots are then used for further partitioning of the result intervals. When the algorithm terminates, the result contains a partition for each different nearest neighbor during the query interval.

Continuous query processing. A Find$k$NN result is temporal—the query time interval $[t_-, t_+]$ is partitioned into disjoint intervals each of which corresponds to a different kNN result set. The kNN result sets may become invalid if the data set is updated. Algorithms are provided that enable incremental result updates when objects are inserted into or removed from the data set.

The TPkNN algorithm. Tao and Papadias [2002] also model the movement of the query object q by a velocity vector. Based on the current location and velocity of q, they compute a triple

$\langle \mathsf{S}, t_\perp, \mathsf{S}_\Delta \rangle$, where S denotes the kNN answer, $t_\perp$ denotes the time when this answer will change, and S_Δ denotes the set of data objects that will trigger this change, i.e., data objects that will enter or leave the kNN answer. The time $t_\perp$ is called the *minimum influence time*. It is the safe time span that the current kNN answer stays valid. At time $t_\perp$, or if the velocity of q changes before that, the current kNN answer expires and the triple $\langle \mathsf{S}, t_\perp, \mathsf{S}_\Delta \rangle$ is recomputed. This procedure repeats and up-to-date kNN answers are generated continuously. This query algorithm is named the *time-parameterized kNN* (TPkNN) algorithm since the kNN answers are time dependent.

The key of TPkNN is the computation of the minimum influence time $t_\perp$. For each object o_i in the kNN set S, the algorithm computes the *influence time* $t_{inf}(o_i)$, which is the earliest time when o_i becomes farther away from q than some non-kNN object $o_j \in \mathsf{O} \setminus \mathsf{S}$. The minimum of the influence time of all the objects in S is $t_\perp$:

$$t_\perp = \min_{o_i \in \mathsf{S}} t_{inf}(o_i)$$

The time when o_i becomes farther away from q than o_j is computed by the following inequality:

$$\mathtt{dist}(o_i, q.c + q.\overrightarrow{v} \cdot t) > \mathtt{dist}(o_j, q.c + q.\overrightarrow{v} \cdot t)$$

Here, $q.c$ and $q.\overrightarrow{v}$ denote the current location and velocity of q. This inequality is easily solvable since every parameter is a known constant. Solving the inequality yields a point $b^*_{i,j}$ on the projected trajectory of q (denoted by l) where $\mathtt{dist}(o_i, b^*_{i,j}) \geq \mathtt{dist}(o_j, b^*_{i,j})$. This is the point where the bisector $b_{i,j}$ intersects l. In Fig. 8, object o_3 is the current nearest neighbor of q. Point $b^*_{3,1}$ is the point where o_1 becomes closer to q.

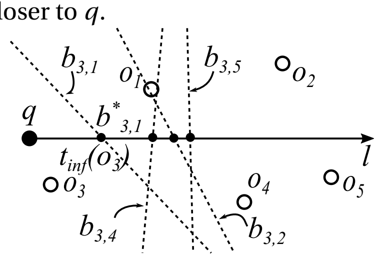


Fig. 8: A TPkNN query

By computing the above inequality between o_i and every non-kNN object, $t_{inf}(o_i)$ is obtained as the minimum value of t from all the inequalities:

$$t_{inf}(o_i) = \min\{t | \mathtt{dist}(o_i, q.c + q.\overrightarrow{v} \cdot t) > \mathtt{dist}(o_j, q.c + q.\overrightarrow{v} \cdot t), o_j \in \mathsf{O} \setminus \mathsf{S}\}$$

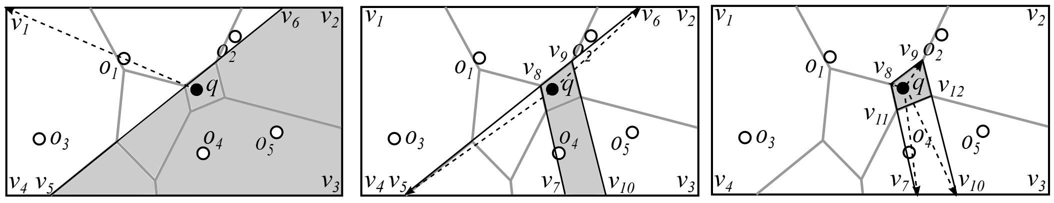
The influence time $t_{inf}(o_i)$ corresponds to the first point where l intersects some bisector of o_i, i.e., $\min\{b^*_{i,j} | o_j \in \mathsf{O} \setminus \mathsf{S}\}$. This point is named the *influence point* of o_i. Object o_i is "safe" until q reaches this point. Figure 8 shows the bisectors between o_3 and every other data object, and their intersection points with l. Point $b^*_{3,1}$ is the first intersection point and is the influence point of o_i. When considering kNNs, the current query answer is "safe" until q reaches the influence point of any kNN object, i.e., $\min\{b^*_{i,j} | o_i \in \mathsf{S}, o_j \in \mathsf{O} \setminus \mathsf{S}\}$, which corresponds to the time $t_\perp$.

TPkNN traverses an R-tree on O in a branch-and-bound fashion. During the traversal, $t_\perp$ is progressively updated as more data objects are accessed. It helps prune the branches that cannot produce smaller influence time estimated based on the `mindist` metric [Roussopoulos et al. 1995]. When all data objects have been either accessed or pruned, the value of $t_\perp$ is returned.

Continuous query processing. Initially a static kNN query is computed to find the kNN set S. Then TPkNN computes the minimum influence time $t_\perp$ and the corresponding pair of objects $o_i \in \mathsf{S}$ and $o_j \in \mathsf{O} \setminus \mathsf{S}$ which forms the set S_Δ. At $t_\perp$, S is updated by replacing o_i with o_j. TPkNN is run again to update $t_\perp$ and S_Δ. This procedure repeats, and updated kNN answers are produced continuously. In the event of an update in the value of k, the velocity of q, or the object data set, TPkNN re-runs to update the triple $\langle \mathsf{S}, t_\perp, \mathsf{S}_\Delta \rangle$.

The *k*CNN algorithm. Tao et al. [2002] consider the case where the query object q moves on a line segment $\overline{se}$. This can be handled by running TPkNN repeatedly to find the influence points on $\overline{se}$ and break $\overline{se}$ into sub-segments, where each sub-segment is a safe region. However, since the trajectory of the query object is known in this case, an algorithm named *kCNN* is proposed to compute all the influence points on $\overline{se}$ in one traversal of the R-tree of O rather than by using multiple runs of TPkNN. An influence point s_i computed by the kCNN algorithm is called a *split point* and is associated with a kNN set corresponding to the sub-segment $\overline{s_i s_{i+1}}$. At the start, there are two split points, namely the two end points s and e of $\overline{se}$, and their kNN sets are empty. To compute the split points, kCNN traverses the R-tree in a branch-and-bound manner. During the traversal, when a data object o is found to be closer to a split point s_i than some existing kNN object of s_i, o is added to the kNN set of s_i. It is also possible that o is only a new kNN object for a portion of the sub-segment $\overline{s_i s_{i+1}}$. In this case, a new split point is added between s_i and s_{i+1}. This new split point breaks the sub-segment into two, where one views o as a new kNN object while the other does not. The `mindist` metric and the maximum distance between a split point and its k^{th} NN are used to prune tree branches from traversal.

The Retrieve-Influence-Set *K*NN algorithm. Zhang et al. [2003] propose another CkNN algorithm, named *Retrieve-Influence-Set kNN* (RIS-kNN), based on TPkNN, which lifts the constraint of a linear query trajectory. The algorithm runs TPkNN in different moving directions from q to find the first data objects that will enter the kNN answer in those directions. The resulting set of objects bounds a region that is equivalent to an order-k Voronoi cell and is used as the safe region. This set is named the *influence set*, and the objects are named the *influence objects*.



(a) Processing v_1 (b) Processing v_5 and v_6 (c) Processing subsequent vertices

Fig. 9: Computing an order-2 Voronoi cell

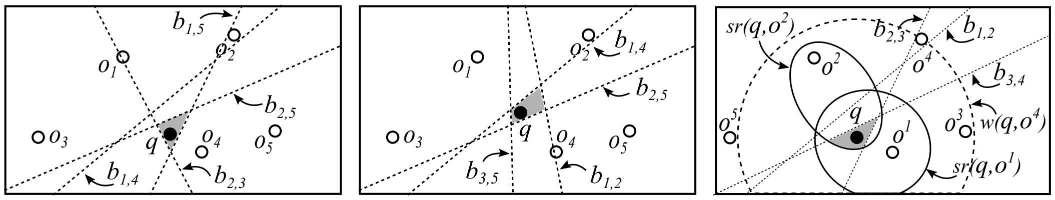
We use Fig. 9 to illustrate RIS-kNN. Let k be 2. The current 2NN set of q is $\{o_2, o_4\}$. The entire data space is a safe region at first. Let v_1, v_2, v_3, and v_4 be its vertices. The algorithm then uses TPkNN to find the influence object in the direction of a randomly selected vertex, e.g., v_1 in Fig. 9(a). Object o_1 is returned by TPkNN, since it is the first object to become nearer to q than an existing 2NN object (i.e., o_4) towards v_1. The bisector between o_1 and o_4 is drawn, which intersects the boundary of the current safe region (i.e., the entire data space). This creates the new vertices v_5 and v_6. The safe region becomes the polygon $v_2 v_3 v_5 v_6$. Next, another vertex is randomly selected from this polygon, e.g., v_5 in Fig. 9(b). TPkNN is run towards v_5, where o_1 is again found as the influence object. The bisector between o_1 and o_2 further shrinks the safe region, generating the new vertices v_7 and v_8. This process continues. Note that a vertex does not lead to a new influence object if it is too close to q, as exemplified by v_8 and v_9 in Fig. 9(c). These vertices are called *confirmed vertices*. When every vertex is confirmed, the algorithm terminates. The confirmed vertices form a polygon equivalent to $vc_2(\{o_2, o_4\})$ (cf. Fig. 9(c)).

Continuous query processing. When a CkNN query is issued, RIS-kNN computes the current kNN set S and the Voronoi cell vc_k(S). Then only when q leaves vc_k(S), recomputation of S and vc_k(S) is required. When an update changes the kNN set S, such as a change of the value of k or a data object being added to (or removed from) the data set, S and vc_k(S) are also recomputed.

3.1.4. Query Object Moving Direction Independent Safe Regions. RIS-kNN avoids the computation of a full Voronoi diagram, but it remains relatively expensive to compute an exact Voronoi cell locally, as multiple runs of TPkNN are needed. We proceed to consider the *V*-diagram* algorithm [Nutanong et al. 2008] that achieves higher efficiency by computing less strict safe regions.

The V*-diagram algorithm computes an *integrated safe region* (ISR). The ISR is a combination of two safe regions, one to guarantee the validity of the kNN objects, and the other to guarantee the nearness ranks within the kNN objects. The latter safe region has an idea similar to that of the *incremental rank update* (IRU) technique [Kulik and Tanin 2006]. We describe IRU briefly first.

The Incremental Rank Update algorithm. IRU maintains the complete nearness rank of all data objects, $o^1, o^2, ..., o^n$, to the query object q. This is done based on the bisectors between the data objects. However, IRU does not compute all the bisectors to form a full Voronoi diagram. Rather, it is sufficient to compute a bisector between every pair of adjacent data objects o^i and o^{i+1} in the nearness rank, meaning that $n-1$ bisectors are required for maintaining a complete ranking of n data objects. The intuition is straightforward: a bisector $b_{i,i+1}$ guarantees that q is nearer to o^i than to o^{i+1}; the $n-1$ bisectors $b_{1,2}$, $b_{2,3}$, ..., $b_{n-1,n}$ guarantee that q is nearer to o^1 than to o^2, and nearer to o^2 than to o^3, ..., and nearer to o^{n-1} than to o^n.



(a) Fixed-rank region (b) Rank update (c) Integrated safe region

Fig. 10: The V*-diagram

Figure 10(a) shows an example, where the data objects are ranked as o_4, o_1, o_5, o_2, o_3. The gray region enclosed by the bisectors $b_{1,4}, b_{1,5}, b_{2,5}$, and $b_{2,3}$ is a safe region. As long as q stays in this region, the nearness rank of the data objects stays unchanged. When q crosses a bisector (e.g., $b_{2,5}$), as exemplified in Fig. 10(b), IRU updates the nearness rank by swapping the corresponding data objects o_2 and o_5. This creates two (or one if swapping the first/last pair) new adjacent pairs of data objects, (o_1, o_2) and (o_5, o_3). Their bisectors are then computed (i.e., $b_{1,2}$ and $b_{3,5}$).

The V*-diagram algorithm. The V*-diagram algorithm uses IRU for rank maintenance within the kNN answer and uses an additional safe region for validity checking of the current kNN answer. Figure 10(c) illustrates how the safe region is computed. The algorithm performs a best-first search on an R-tree of the data objects to retrieve the nearest neighbors. It retrieves $(k+x)$ NNs instead of k, where x is a system parameter. Let these NNs be $o^1, o^2, ..., o^k, o^{k+1}, ..., o^{k+x}$. The x extra NNs are called *auxiliary neighbors*. They will serve as a cache for reducing the kNN recomputation frequency. In the figure, we use $k = 2$ and $x = 2$. The $(k+x)^{th}$ NN, o^4, defines a circle centered at q with $\texttt{dist}(q, o^4)$ as radius. The region enclosed by the circle, denoted by $w(q, o^4)$, is called the *known region*, since the $(k+x)$ NNs are "known" to be in this region. No other data object is in this region. Next, a *safe region w.r.t. a data object*, denoted by $sr(q, o^i)$, is computed for every kNN object o^i $(i \le k)$. When the query object is in this region, o^i is closer to the query object than any object o outside the known region. A point q' in this region satisfies:

$$\forall o \in \mathcal{O} \setminus \{o^1, o^2, ..., o^{k+x}\} : \texttt{dist}(q', o^i) \le \texttt{dist}(q', o) \tag{1}$$

Based on triangle inequality, $\texttt{dist}(q, o) - \texttt{dist}(q, q') \le \texttt{dist}(q', o)$. Equation 1 is tightened to be:

$$\forall o \in \mathcal{O} \setminus \{o^1, o^2, ..., o^{k+x}\} : \texttt{dist}(q', o^i) \le \texttt{dist}(q, o) - \texttt{dist}(q, q') \tag{2}$$

Since o is outside the known region, we know $\texttt{dist}(q, o^{k+x}) < \texttt{dist}(q, o)$. Equation 1 is further tightened to be:

$$\texttt{dist}(q', o^i) \le \texttt{dist}(q, o^{k+x}) - \texttt{dist}(q, q') \tag{3}$$

Therefore, the safe region w.r.t. a data object o^i is defined as follows:

$$sr(q, o^i) = \{q' \in DS | \texttt{dist}(q', o^i) + \texttt{dist}(q, q') \le \texttt{dist}(q, o^{k+x})\}$$

This region is an ellipse where q and o^i are the two focal points and $\texttt{dist}(q, o^{k+x})$ is the major axis length. In Fig. 10(c), the two ellipses are the safe regions w.r.t. o^1 and o^2. Their intersection guarantees that the query object is closer to them than to any object outside the known region.

Rank maintenance is done using a concept called the *fixed-rank region*. This region is formed by the bisector between each pair of adjacent objects in the $k+x$ NNs. It is computed by the IRU algorithm. If the query object is in this region, the nearness ranks of the $k+x$ NNs do not change. In Fig. 10(c), three bisectors are computed for the 4 NNs to bound the fixed-rank region.

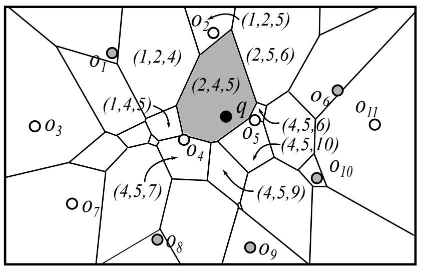
The final safe region, the *integrated safe region* (ISR), is the intersection of the safe regions w.r.t. the kNN objects and the fixed-rank region, as exemplified by the gray region in Fig. 10(c). To compute this region, only the safe region w.r.t. o^k needs to be considered. If the query object is in $sr(q, o^k)$, o^k is closer to it than any object outside the known region. The fixed rank region further guarantees $o^1, o^2, ..., o^{k-1}$ to be closer to the query object than o^k, and $o^{k+1}, o^{k+2}, ..., o^{k+x}$ to be farther away from the query object than o^k. Thus, the kNN set is correct.

Continuous query processing. Once the $k+x$ NNs and the ISR are computed, the V*-diagram algorithm monitors two types of events continuously as the query object moves. These events may change the kNN set or the ISR: (i) When the query object leaves $sr(q, o^k)$, the kNN set becomes invalid. A *reliability update* is performed to recompute both the kNN set and the ISR. (ii) When the query object crosses a bisector, e.g., between o^i and o^{i+1}, a *rank update* is performed to swap the two objects in the ranking. If o^k is affected, the safe region $sr(q, o^k)$ is recomputed as well.

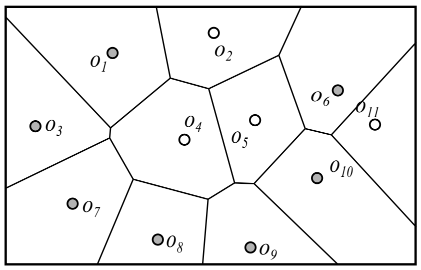
The V*-diagram also handles query and data updates. If the k value changes, there are two cases: (i) The new k value exceeds $k + x$. Then the kNN set and the ISR are recomputed. (ii) Otherwise, the k^{th} NN, o^k, and its safe region, $sr(q, o^k)$, are computed. If the query object is outside the new $sr(q, o^k)$, the kNN set and the ISR are recomputed. For data object updates (insertion or deletion), a query recomputation is required if the updates relate to the known region.

3.1.5. Implicit Safe Regions. The V*-diagram computes safe regions efficiently which are not tight. In comparison, RIS-kNN computes tight safe regions (i.e., order-k Voronoi cells) but is less efficient. To overcome the limitations of both algorithms, Li et al. [2014] compute safe regions based on the *influential neighbor set* (INS), which is a set of data objects surrounding the current kNNs that define an order-k Voronoi cell implicitly. They name their algorithm *INS-kNN*.

INS-kNN computes *safeguarding objects* and checks whether the kNNs are closer to the query object q than the safeguarding objects. If yes, the kNNs are valid. A natural choice of safeguarding objects is the objects contributing edges to the order-k Voronoi cell of the kNNs. In Fig. 11(a), q is in $v_3(\{o_2, o_4, o_5\})$, which is the gray cell labeled by the subscripts (2,4,5) of the corresponding data objects $\{o_2, o_4, o_5\}$. The kNNs are $\{o_2, o_4, o_5\}$. There are 8 neighboring order-3 Voronoi cells, which are also labeled by the subscripts of their corresponding data objects. These corresponding data objects form a set $\{o_1, o_2, o_4, o_5, o_6, o_8, o_9, o_{10}\}$. Subtracting the kNN set $\{o_2, o_4, o_5\}$ from this set leaves $\{o_1, o_6, o_8, o_9, o_{10}\}$. These objects (the gray points) are the safeguarding objects. The edges of $v_3(\{o_2, o_4, o_5\})$ are formed by the bisectors between the kNN objects and these objects, e.g., the bottom edge is formed by the bisector between o_2 and o_8. As long as q is closer to the kNN objects than to these objects, q is in $v_3(\{o_2, o_4, o_5\})$. This set of safeguarding objects, identified from the neighboring order-k Voronoi cells, is named the *minimal influential set* (MIS).



(a) MIS



(b) INS

Fig. 11: MIS and INS of an order-3 Voronoi cell

The MIS defines a tight safe region. However, to compute the MIS is too expensive as it requires computing an order-k Voronoi cell. Li et al. [2014] propose to use a superset of the MIS called the influential neighbor set (INS), which defines a tight safe region with little computation overhead.

The influential neighbor set. The INS is formed by (order-1) *Voronoi neighbors*. In an order-1 Voronoi diagram, two data objects are Voronoi neighbors if their Voronoi cells share an edge. For example, in Fig. 11(b), o_1 and o_2 are Voronoi neighbors. The set of Voronoi neighbors of an object o_i in the Voronoi diagram of O is denoted by $N_{\mathsf{O}}(o_i)$. Let S be a kNN set. Then the INS of S, denoted by $I_{\mathsf{O}}(\mathsf{S})$, is the union of the Voronoi neighbor sets of the objects in S minus S:

$$I_{\mathsf{O}}(\mathsf{S}) = (\bigcup_{o_i \in \mathsf{S}} N_{\mathsf{O}}(o_i)) \setminus \mathsf{S}$$

As shown in Fig. 11(b), the INS of $\mathsf{S} = \{o_2, o_4, o_5\}$ is $N_{\mathsf{O}}(o_2) \cup N_{\mathsf{O}}(o_4) \cup N_{\mathsf{O}}(o_5) \setminus \mathsf{S}$, which is:

$$\{o_1, o_4, o_5, o_6\} \cup \{o_1, o_2, o_3, o_5, o_7, o_8, o_9\} \cup \{o_2, o_4, o_6, o_9, o_{10}\} \setminus \{o_2, o_4, o_5\} = \{o_1, o_3, o_6, o_7, o_8, o_9, o_{10}\}$$

The INS is always a super set of the MIS [Li et al. 2014]. Essentially, INS-kNN uses the neighbors of the kNNs in an order-1 Voronoi diagram to approximate those in an order-k Voronoi diagram. INS-kNN precomputes the order-1 Voronoi diagram and indexes the data objects together with the Voronoi cells in a VoR-tree [Sharifzadeh and Shahabi 2010], which enables fast retrieval of the Voronoi neighbors. Meanwhile, the average number of Voronoi neighbors per data objects is a small constant, as constrained by Euler's formula. Thus, the INS can be computed online with high efficiency. As a result, INS-kNN outperforms V*-diagram consistently in practical settings.

Continuous query processing. INS-kNN computes a few extra NNs to reduce the recomputation frequency. It computes a ρkNN set S_ρ together with the INS $I_{\mathsf{O}}(\mathsf{S}_\rho)$. Here, ρ is a system parameter ($\rho > 1$). Its value is obtained empirically. The top kNNs in S_ρ are returned as the query answer, while the rest of the objects are used as safeguarding objects. When the kNN objects are closer to the query object than the safeguarding objects, the query answer is valid. If a kNN object becomes invalid, there are two cases: (i) If the kNN object is invalidated by a new NN object in S_ρ, we simply replace the invalidated kNN object with the new NN object, and the invalidated object becomes a safeguarding object. This ensures that the kNN set and the safeguarding objects are up-to-date because the Voronoi neighbors of the new NN object is already computed and in $I_{\mathsf{O}}(\mathsf{S}_\rho)$ at the very beginning. (ii) If the kNN object is invalidated by an object outside S_ρ, a full recomputation of the sets S_ρ and $I_{\mathsf{O}}(\mathsf{S}_\rho)$ is required.

INS-kNN also allows setting the value of k at query time, since there is no precomputation based on the value of k. When handling data updates, INS-kNN first updates the VoR-tree using the built-in update functions. Then the kNN set and INS are recomputed accordingly.

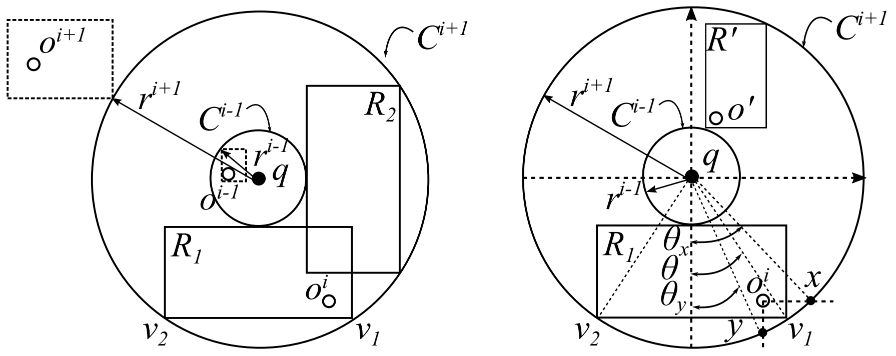
3.2. Safe Regions for Data Objects

Section 3.1 covers studies on safe regions for the query object. Those studies, with the exception of that by Benetis et al. [2001, 2002, 2006], assume stationary data objects. Other studies assume moving (or streaming) data objects [Prabhakar et al. 2002; Jensen et al. 2004; Mokbel et al. 2004; Koudas et al. 2004; Hu et al. 2005; Xiong et al. 2005]. Moving data objects make building (tight) safe regions more challenging as their nearness rank is more difficult to maintain. Periodic query reevaluation is needed, where incremental computation is applied to reduce the query costs. Only the objects updated since last query evaluation are considered in the next round of query reevaluation. SINA [Mokbel et al. 2004] and SEA-CNN [Xiong et al. 2005] are typical algorithms in this category. Incremental computation algorithms, however, are not the focus of the survey. One study [Hu et al. 2005] builds safe regions on moving data objects. We discuss it next.

The SRB Algorithm. Hu et al. [2005] assume a set of CkNN queries registered at the server, where the query points are fixed. A *combined safe region* is computed for each data object so that an object's movement inside this region does not affect the answer validity of any registered query. The combined safe region is the intersection of a set of *individual safe regions*, each of which guarantees that the movement of the object does not affect one query. We only discuss how an individual safe region is computed and refer to it simply as a safe region. The query algorithm using this safe region is the *safe region based* (SRB) algorithm.

SRB uses rectangular safe regions, which are computed incrementally for multiple queries. Assume that the data objects are already enclosed by their respective safe regions for some queries.

When a new query located at q is issued, the safe regions need to be updated. To update the safe region of the i^{th} NN, o^i, we need to examine the safe regions of the preceding and the following NNs, o^{i-1} and o^{i+1}. The updated safe region ensures that o^i is bounded between o^{i-1} and o^{i+1}.



(a) SRB safe regions (b) SRB safe region computation

Fig. 12: Safe region for a data object

Figure 12(a) gives an example, where the safe regions of o^{i-1} and o^{i+1} are denoted by the two dashed rectangles. The *minimum* distance between q and the safe region of o^{i+1}, $\texttt{mindist}(q, o^{i+1})$, defines a circle C^{i+1} centered at q and with a radius $r^{i+1} = \texttt{mindist}(q, o^{i+1})$; the *maximum* distance between q and the safe region of o^{i-1}, $\texttt{maxdist}(q, o^{i-1})$, defines a circle C^{i-1} centered at q and with a radius $r^{i-1} = \texttt{maxdist}(q, o^{i-1})$. These two circles form a ring. Any rectangle inside this ring enclosing o^i forms a safe region. SRB uses the *inscribed rectangle of the ring with the longest perimeter* as the safe region for safe region maximization. This safe region has an edge tangent to C^{i-1} and two vertices on C^{i+1}, denoted by v_1 and v_2. The edge of the safe region may be tangent to C^{i-1} either horizontally or vertically, as illustrated by the two solid rectangles R_1 and R_2, respectively. We discuss only the horizontal case R_1. The vertical case has the same computation procedure. As illustrated in Fig. 12(b), assume a rectangular coordinate system where q is the origin and o^i is in the fourth (southeast) quadrant. Let v_1 be the vertex of R_1 in the fourth quadrant and let θ be the angle from $\overline{qv_1}$ to the y-axis. Then, the perimeter of R_1 can be computed as a function of θ:

$$\texttt{perimeter}(R_1, \theta) = 4r^{i+1}\sin\theta + 2(r^{i+1}\cos\theta - r^{i-1})$$

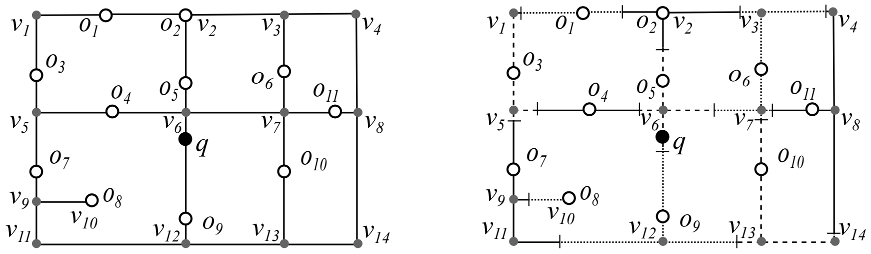
The first order derivative shows that this function is monotonic and has a maximum value at $\theta = \arctan 2$. If the rectangle defined by $\theta = \arctan 2$ encloses o^i, it is used as the safe region. However, o^i may be outside this rectangle. In this case, the rectangle with the smallest θ that encloses o^i is used. This is identified by comparing $\arctan 2$ with the two angles θ_x and θ_y that represent the angles between $\overline{qx}$ and $\overline{qy}$ and the y-axis, respectively. Here, x and y are the two points where a vertical line and a horizontal line through o^i intersect C^{i+1}. If $\theta_y < \arctan 2$, o^i is close to C^{i+1} and the y-axis; thus, θ_y should be used as θ to define the safe region. Similarly, if $\theta_x > \arctan 2$, o^i is close to C^{i+1} and the x-axis, meaning that θ_x should be used as θ.

Two questions remain: (i) What if o^i is too close to C^{i-1} (e.g., at o') and cannot be covered by R_1 or R_2? (ii) What if it is the first query where there is no safe region and hence no `mindist` or `maxdist` to define the circles? Hu et al. [2005] did not discuss these questions, but they can be addressed easily. For Question (i), we can use a rectangle tangent to C^{i-1} at a vertex instead of an edge (e.g., R'). The point where $\overline{qo'}$ intersects C^{i-1} can be used as this vertex. For Question (ii), we can use $\frac{\texttt{dist}(q,o^i)+\texttt{dist}(q,o^{i+1})}{2}$ as $\texttt{mindist}(q, o^{i+1})$ and $\texttt{maxdist}(q, o^i)$ to define the circles.

Continuous query processing. When a CkNN query is issued, a static kNN query is performed to retrieve the kNN answer. Safe regions of the data objects are computed following the above procedure. A *quarantine region* (QR) is computed, which is a circle centered at the query point that encloses *all and only* the safe regions of the kNN objects. When an object moves out its safe region, if it enters or leaves the QR, a recomputation of the kNN answer and the safe regions is required. Otherwise, only the safe region of the object itself is recomputed. Similarly, if an object has been inserted into or deleted from the QR, a query recomputation is needed.

3.3. Safe Regions in Spatial Networks

Most types of spatial queries have a variant where the objects are constrained in a spatial network. This is due to the proliferation of spatial networks, most notably road networks. The CkNN query is no exception. In this section, we review safe region techniques to process CkNN queries in spatial networks: the *network Voronoi diagram* technique [Kolahdouzan and Shahabi 2004b], the *split point* technique [Kolahdouzan and Shahabi 2004a; Cho and Chung 2005], and the *network V*-diagram* technique [Nutanong et al. 2010]. These techniques have similarities to the Voronoi diagram technique (Section 3.1.2), the influence point based technique (Section 3.1.3), and the V*-diagram technique (Section 3.1.4), respectively. The main difference is that network distance is used instead of Euclidean distance. All three techniques assume static data objects and a moving query object. The network Voronoi diagram technique precomputes safe regions over the entire spatial network, while the split point technique computes safe regions locally, edge-by-edge over a given trajectory of the query object. The network V*-diagram technique also computes safe regions locally, but it does not require the trajectory of the query object to be given. These three techniques are discussed in Sections 3.3.1, 3.3.2, and 3.3.3, respectively.



(a) A spatial network (b) A network Voronoi diagram

Fig. 13: Construction of a network Voronoi diagram

We start with the data representation of CkNN queries in spatial networks. As Fig. 13(a) shows, a spatial network is modeled as a graph $G = \langle V, E, \mathsf{W}\rangle$. The set $V = \{v_1, v_2, ..., v_h\}$ is a set of h vertices (e.g., road intersections), shown as gray dots. The set $E = \{e_1, e_2, ..., e_m\}$ is a set of m edges (e.g., road segments), shown as line segments between vertices. An edge $e_{i,j}$ connects two vertices v_i and v_j. The weight function $\mathsf{W} : E \rightarrow \mathbb{R}^+$ maps an edge to a real number which is the weight (e.g., length) of the edge. Both directed and undirected graphs have been considered in the literature. We consider undirected graphs. The data objects $\{o_1, o_2, ..., o_n\}$ are points in the network, shown as hollow circles. They may be at the vertices (e.g., o_2) or on the edges (e.g., o_1). A query object q is a point moving in the network, shown as the black dot. A CkNN query maintains the k objects with the smallest *shortest path distances* to q. Here, the shortest path distance between two objects is defined as the "length" (sum of the edge weights) of the shortest path between the two objects. We use $\mathtt{dist_n}(\cdot)$ to represent the shortest path distance. In Fig. 13(a), a C3NN query returns $\{o_5, o_9, o_4\}$ as they are the nearest at this particular moment.

3.3.1. The Network Voronoi Diagram Approach. A Voronoi diagram can also be obtained on a spatial network, where nearness is defined by the shortest path distance. The resulting diagram is a *network Voronoi diagram* [Okabe et al. 1992], where a *network Voronoi cell* consists of a set of edge segments. Figure 13(b) represents the edge segments of the same network Voronoi cell with same-style line segments, e.g., the dashed line segments connected to o_5 form the cell of o_5. The query object q is on one of these line segments. Thus, its NN is o_5. We denote the network Voronoi cell of o_i by $vc_n(o_i)$:

$$vc_n(o_i) = \{p \in G | \mathtt{dist_n}(p, o_i) \leq \mathtt{dist_n}(p, o_j), \forall\ o_j \neq o_i, o_j \in \mathsf{O}\}$$

Here, p represents a point in the spatial network. The boundary of a cell is formed by a set of boundary points, as denoted by the bars in Fig. 13(b).

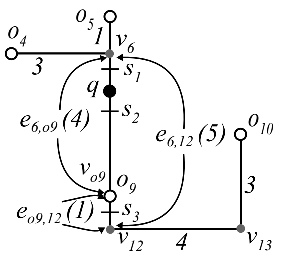
The network Voronoi diagram can be computed by the parallel Dijkstra algorithm [Erwig and Hagen 2000]. This algorithm expands simultaneously a shortest path tree from every object until the trees meet. The meeting points are the boundary points of the cells.

Continuous query processing. Network Voronoi cells can be used as safe regions to process C1NN queries. Processing a CkNN query where $k > 1$ is more difficult because higher order network Voronoi diagrams are expensive to compute.

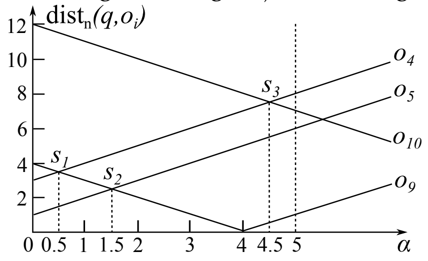
Kolahdouzan and Shahabi [2004b] propose the *Voronoi-based network nearest neighbor* (VN^3) algorithm to compute *static* kNNs based on the network Voronoi diagram. However, applying VN^3 to process a CkNN query has very high cost. The algorithm starts by locating the first NN by searching from the precomputed network Voronoi cells. It has been shown that the $(k+1)^{st}$ NN must be a neighbor of a kNN object. Thus, the algorithm proceeds to iteratively put the neighboring objects of the NNs found into an *NN candidate set* and then finds the next NN from this set. To find the next NN, the shortest path distances between the query object and the NN candidates are computed and compared. The first kNNs found are returned as the query answer. When the query object moves continuously, the graph needs to be traversed constantly to update the shortest path distances and the kNNs, which is expensive.

3.3.2. The Split Point Approach. The influence point based approach for Euclidean space (Section 3.1.3) assumes that the query object has a linear trajectory and finds the influence points to partition the trajectory into segments, each of which has a different kNN set. This approach extends naturally to spatial networks where the query trajectory is constrained by linear network edges. Kolahdouzan and Shahabi [2004a] propose the *Intersection Examination* (IE) algorithm to split an edge into segments that each has a different kNN set. They further propose the *Upper Bound* (UB) algorithm using the lazy search technique (Section 3.1.1) to reduce the number of edges to split. Both IE and UB, however, cannot split edges with data objects on them. They have to first break the edges by the data objects. Cho and Chung [2005] overcome this limitation with the *Unique Continuous Search* (UNICONS) algorithm. We discuss these three algorithms next.

The Intersection Examination algorithm. Kolahdouzan and Shahabi [2004a] show that the kNN answer at any point on an edge $e_{i,j}$ between vertices v_i and v_j must be a subset of the data objects on $e_{i,j}$ and the kNNs of v_i and v_j. As shown in Fig. 14(a), which is a version of Fig. 13(a) zoomed-in on edge $e_{6,12}$, the 2NN set of q on $e_{6,12}$, $\{o_5, o_9\}$, is a subset of $U = \{o_9\} \cup \{o_4, o_5\} \cup \{o_9, o_{10}\}$, which is the union set of the data objects on $e_{6,12}$, the 2NNs of v_6, and the 2NNs of v_{12}. As q moves from v_6 to v_{12}, some objects in U will become closer to q (e.g., o_{10}); some will become farther away (e.g., o_4 and o_5); others will first become closer and then farther away (e.g., o_9). The objects with constantly increasing or constantly decreasing distance to q, i.e., $\{o_4, o_5, o_{10}\}$, must *not* be on $e_{6,12}$. The constant distance changing patterns enables finding points on $e_{6,12}$ where the nearness ranks of the data objects change, which are the *split points* used to split the edge into segments (safe regions). Object o_9 does not have a constant distance changing pattern. Its nearness rank is more difficult to compute. This is addressed by breaking an edge into multiple edges using the data objects as vertices, e.g., $e_{6,12}$ is broken into two edges $e_{6,o9}$ and $e_{o9,12}$. The weights of these edges are labeled in parentheses in Fig. 14(a), e.g., $e_{6,12}$ has a weight of 5.



(a) IE and UNICONS



(b) Split point computation in UNICONS

Fig. 14: The split point based algorithms

Let $e_{6,o9}$ be the edge for which the split points are to be computed. The IE algorithm first generates a list L storing the kNNs of v_6 and v_{o9} sorted in ascending order of their distances to v_6. Each element in L is represented by a 3-tuple $\langle o_i, \mathtt{dist_n}(o_i, v_6), \uparrow \text{ or } \downarrow\rangle$ where "$\uparrow$" or "$\downarrow$" denote whether the object gets closer or farther away as q moves from v_6 to v_{o9}. In the figure, the integers by the edge segments represent the weights of the segments, e.g., $|e_{6,o5}| = 1$. Based on these weights, the list L is computed as $L = \langle\langle o_5, 1, \uparrow\rangle, \langle o_4, 3, \uparrow\rangle, \langle o_9, 4, \downarrow\rangle, \langle o_{10}, 12, \downarrow\rangle\rangle$. In this list, an "$\uparrow$" element $\langle o_i, \mathtt{dist_n}(o_i, v_6), \uparrow\rangle$ followed by a "$\downarrow$" element $\langle o_j, \mathtt{dist_n}(o_j, v_6), \downarrow\rangle$ will yield a split point, e.g., $\langle o_4, 3, \uparrow\rangle$ and $\langle o_9, 4, \downarrow\rangle$. This is because o_i is currently closer but is getting farther away while o_j is currently farther away but is getting closer. At some point the two objects will swap ranks in L. This point is a split point. The split point is a point on $e_{6,o9}$ with the following distance to v_6:

$$\frac{1}{2}(\mathtt{dist_n}(o_j, v_6) - \mathtt{dist_n}(o_i, v_6)) \tag{4}$$

For the pair $\langle o_4, 3, \uparrow\rangle$ and $\langle o_9, 4, \downarrow\rangle$, the split point has a distance of $\frac{1}{2}(4-3) = 0.5$ from v_6, as denoted by s_1 (the horizontal bar) on $e_{6,o9}$. Adjacent pairs with other patterns (e.g., both with "$\uparrow$") do not have this property. They do not yield any split points and can be omitted.

The IE algorithm finds all adjacent pairs with the "$\uparrow$"-"$\downarrow$" pattern and compute the split points. The split point closest to v_6 is recorded as the first split point s_1, yielding a new edge $e_{s1,o9}$. Now the algorithm finds the split points for $e_{s1,o9}$. The distance values in L are updated from $\mathtt{dist_n}(o_i, v_6)$ to $\mathtt{dist_n}(o_i, s_1)$. The ranking of the elements is updated accordingly. This procedure repeats until no more split points can be found. The kNNs of the segments created by the split points are the top-k elements in L when the split points are generated. In Fig. 14(a), after split point s_1 is created at a distance of 0.5 from v_6, list L is updated to $\langle\langle o_5, 1.5, \uparrow\rangle, \langle o_9, 3.5, \downarrow\rangle, \langle o_4, 3.5, \uparrow\rangle, \langle o_{10}, 11.5, \downarrow\rangle\rangle$. Two "$\uparrow$"-"$\downarrow$" patterns are found: $(\langle o_5, 1.5, \uparrow\rangle, \langle o_9, 3.5, \downarrow\rangle)$ and $(\langle o_4, 3.5, \uparrow\rangle, \langle o_{10}, 11.5, \downarrow\rangle)$, which produce two split points at distances of 1 and 4 from s_1, respectively. The one closer to s_1 is recorded as s_2. The list L is then updated to $\langle\langle o_9, 2.5, \downarrow\rangle, \langle o_5, 2.5, \uparrow\rangle, \langle o_4, 4.5, \uparrow\rangle, \langle o_{10}, 10.5, \downarrow\rangle\rangle$. Subsequently, split point s_3, which has a distance of 3 from s_2, is found. This split point is outside edge $e_{6,o9}$, and is omitted. Then the algorithm terminates.

The Upper Bound algorithm. The IE algorithm requires a kNN search for every vertex v_i (and object) on the query trajectory to find candidate kNN objects. The kNN search is expensive since it requires a run of Dijkstra's algorithm. Thus, reducing the number of vertices to be considered is essential to reduce the query cost. The UB algorithm proposes an improvement based on the lazy search technique (Section 3.1.1). Let the current k^{th} and $(k+1)^{st}$ NNs be o^k and o^{k+1}, respectively. The lazy search technique states that, before q moves for a distance of $\delta = \frac{1}{2}(\mathtt{dist}(q, o^{k+1}) - \mathtt{dist}(q, o^k))$, the kNN answer stays valid. In a spatial network, a similar observation holds. Let the edge $e_{i,j}$ between v_i and v_j be the edge to be processed. We compute the $(k+1)$NNs of v_i and put them into list L. Before doing the same for v_j, we compute the first split point using L and the distance of this split point from v_i. This distance is computed by Equation 4, which has the same form as that of δ. If this distance exceeds the weight of $e_{i,j}$, the kNN set at v_i is valid throughout $e_{i,j}$. In this case, no kNN computation is needed for v_j.

The Unique Continuous Search algorithm. Cho and Chung [2005] follow the same overall algorithmic approach as IE, but use an alternative way to compute the split points. Assume that the query object q moves from v_i to v_j on an edge $e_{i,j}$ and that its distance to v_i is α. To compute the split points on $e_{i,j}$, first, the distance between q and a kNN candidate object o in the list L, $\mathtt{dist_n}(q, o)$, is formulated as a function of α:

$$\mathtt{dist_n}(q, o) = \mathtt{dist_n}(o, v^*) + |\alpha - \mathtt{dist_n}(v^*, v_i)| \quad \alpha \in [0, |e_{i,j}|] \tag{5}$$

This distance has two terms: (i) $\mathtt{dist_n}(o, v^*)$, which represents the distance between o and the vertex $v^*(v^* = v_i \text{ or } v_j)$ of which o is a kNN object; (ii) $|\alpha - \mathtt{dist_n}(v^*, v_i)|$, where $\mathtt{dist_n}(v^*, v_i)$ represents the distance between v^* and v_i. There are three different cases for the values of $\mathtt{dist_n}(o, v^*)$ and $\mathtt{dist_n}(v^*, v_i)$. We again use a C2NN query in Fig. 14(a) to illustrate these cases. Now edge $e_{6,12}$ is considered as a whole instead of two edges, i.e., $v_i = v_6$ and $v_j = v_{12}$.

(1) For a kNN object of v_i: $\texttt{dist}_\texttt{n}(o, v^*) = \texttt{dist}_\texttt{n}(o, v_i)$ and $\texttt{dist}_\texttt{n}(v^*, v_i) = \texttt{dist}_\texttt{n}(v_i, v_i) = 0$.
Object o_5 is an example, where $\texttt{dist}_\texttt{n}(o_5, v^*) = \texttt{dist}_\texttt{n}(o_5, v_6) = 1$ and $\texttt{dist}_\texttt{n}(v^*, v_i) = \texttt{dist}_\texttt{n}(v_6, v_6) = 0$. Thus, $\texttt{dist}_\texttt{n}(q, o_5) = 1 + |\alpha| = 1 + \alpha$.
(2) For a kNN object of v_j: $\texttt{dist}_\texttt{n}(o, v^*) = \texttt{dist}_\texttt{n}(o, v_j)$ and $\texttt{dist}_\texttt{n}(v^*, v_i) = \texttt{dist}_\texttt{n}(v_j, v_i) = |e_{i,j}|$.
Object o_{10} is an example, where $\texttt{dist}_\texttt{n}(o_{10}, v^*) = \texttt{dist}_\texttt{n}(o_{10}, v_{12}) = 7$ and $\texttt{dist}_\texttt{n}(v^*, v_i) = \texttt{dist}_\texttt{n}(v_{12}, v_6) = 5$. Thus, $\texttt{dist}_\texttt{n}(q, o_{10}) = 7 + |\alpha - 5|$.
(3) For an object on $e_{i,j}$: $\texttt{dist}_\texttt{n}(o, v^*) = 0$ and $\texttt{dist}_\texttt{n}(v^*, v_i) = \texttt{dist}_\texttt{n}(o, v_i)$.
Object o_9 is an example, where $\texttt{dist}_\texttt{n}(o_9, v^*) = 0$ and $\texttt{dist}_\texttt{n}(v^*, v_i) = \texttt{dist}_\texttt{n}(o_9, v_6) = 4$. Thus, $\texttt{dist}_\texttt{n}(q, o_9) = |\alpha - 4|$.

Note that Equation 5 handles data objects on $e_{i,j}$ as well (Case 3). Thus, UNICONS does not need to treat these data objects as vertices and hence requires fewer kNN searches.

In all three cases, $\texttt{dist}_\texttt{n}(o, v^*)$ and $\texttt{dist}_\texttt{n}(v^*, v_i)$ are constants once the edge $e_{i,j}$ and the object o are given; the distance function in Equation 5 is a (piecewise) linear function of α. This distance function can be plotted as a polyline in a two-dimensional coordinate system. If we draw the polylines for the candidate kNN objects in list L, the points where these polylines intersect are the points where the nearness ranks of the objects change, which are the split points.

Figure 14(b) shows an example, where UNICONS computes the split points for a C2NN query from v_6 to v_{12} in Fig. 14(a). The kNN candidates are $\{o_5, o_4, o_9, o_{10}\}$. Their corresponding distance functions are: $\texttt{dist}_\texttt{n}(q, o_5) = 1 + \alpha$, $\texttt{dist}_\texttt{n}(q, o_4) = 3 + \alpha$, $\texttt{dist}_\texttt{n}(q, o_9) = |\alpha - 4|$, and $\texttt{dist}_\texttt{n}(q, o_{10}) = 7 + |\alpha - 5|$, respectively. These distance functions are drawn in the figure. Three intersection points, $s_1 = 0.5, s_2 = 1.5$, and $s_3 = 4.5$, are found where $\alpha \le |e_{6,12}| = 5$. They are the split points. The 2NN answers corresponding to each segment produced by the split points can be identified easily by checking which lines are the lowest in the segment. For example, in the segment $[0, 0.5)$, the 2NN set is $\{o_5, o_4\}$ since these objects have the lowest lines in the range.

Continuous query processing. The three algorithms above split the query trajectory into segments and compute the kNN answer for each segment. As long as the query object stays on the same segment, no answer update is required. Only when the query object crosses a split point, the answer is updated and sent to the query user. If the value of k changes or a data object update occurs, a recomputation of the split points is needed.

3.3.3. The Network V-diagram Approach.* The *network V*-diagram* approach [Nutanong et al. 2010] is a relatively straightforward extension of the V*-diagram (Section 3.1.4). This approach follows the same overall algorithmic procedure as the V*-diagram. It also computes the known region, the safe region w.r.t. a data object, the fixed ranked regions, and the integrated safe region, but now network distance is used in the computation, not Euclidean distance.

To compute the known region, $(k + x)$NNs are computed. Let o^k and o^{k+x} be the k^{th} and $(k + x)^{th}$ NNs, respectively. The edge segments within the range of $\texttt{dist}_\texttt{n}(q, o^{k+x})$ from q are identified with a Dijkstra's algorithm like traversal starting from q. These edge segments form the known region. Similarly, the edge segments where any point q' on the segments satisfies $\texttt{dist}(q', o^k) + \texttt{dist}(q, q') \le \texttt{dist}(q, o^{k+x})$ are identified. These edge segments form the safe region w.r.t. o^k. Computing the fixed rank regions corresponds to finding the boundary points where the nearness ranks of the $(k + x)$NNs change, which are essentially the split points used in IE, UB, and UNICONS. They can be computed by any of these algorithms.

4. SAFE REGIONS IN CONTINUOUS RANGE QUERIES

We proceed to consider safe regions for continuous range (CR) queries. In a CR query with a static query object, the query range is a straightforward safe region for the data objects: as long as a data object does not enter or leave the query range, it does not cause the query result to change. Existing studies focus on computing safe regions for a moving query object where the data objects are static. Two types of query range in Euclidean space have been considered, namely rectangular ranges and circular ranges centered at a query location. In spatial networks, the query range is formed by edge segments surrounding a query object.

Regardless of the type of query range, a safe region for a CR query should ensure that the query range (i) encloses the current result objects and (ii) does not enclose any non-result objects. Intuitively, Condition (i) can be achieved by ensuring that the query range encloses the *minimum bounding region* of the current result objects. Condition (ii) can be achieved by ensuring that the query range is enclosed by the *maximum bounding region* of the current result objects. The minimum and maximum bounding regions are the smallest and largest regions (with a regular shape such as a rectangle) centered at the query location that enclose, and *only* enclose, the result objects. Movements of the query range between these two bounding regions does not invalidate the query answer. The *estimated window vector* (EWV) [Huang and Huang 2013] and the *range safe region* [AL-Khalidi et al. 2013] employ this idea and compute safe regions for rectangular and circular range queries, respectively.

The minimum and maximum bounding regions are rough estimates of the locations of the data objects. A safe region based on bounding regions will not be tight. One way to define a tight safe region is to use the *Minkowski region* (MR). The MR is a region that is identical to the query range, but centers at a data object rather than the query object. If a data object is enclosed by the query range, its MR must enclose the query object, and vice versa. Thus, as long as the MRs of the result objects *all* enclose the query object, and no MR of any non-result object encloses the query object, the result stays valid. This means that the region obtained as the intersection of the MRs of the result objects minus the MRs of the non-result objects defines a tight safe region. The *validity region* [Zhang et al. 2003], the *safe zone* [Cheema et al. 2010], the *safe exits* [Yung et al. 2012], and the *safe exits (directed)* [Cho et al. 2014] employ this idea and compute safe regions for rectangular, circular, undirected network, and directed network range queries, respectively.

Figure 15 summarizes the query range types and the corresponding safe region techniques. Next, we cover safe region techniques for rectangular and network range queries in Sections 4.1 and 4.2, respectively. Safe region techniques for circular range queries are similar to those for rectangular range queries and are discussed in Appendix A for completeness and conciseness.

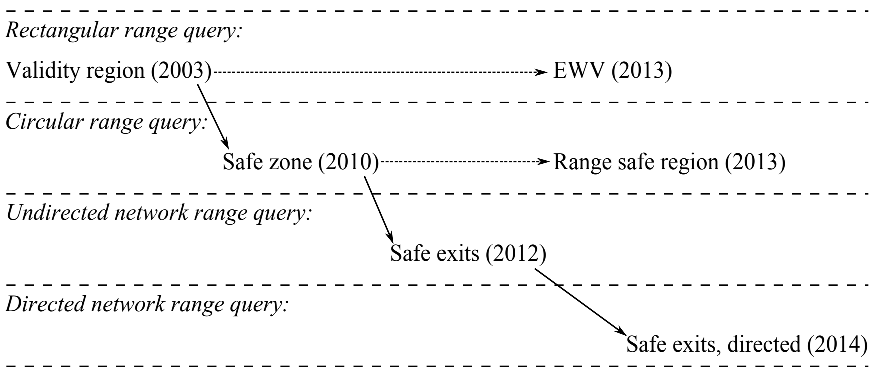


Fig. 15: Evolution of safe region techniques for CR queries

4.1. Safe Region for Rectangular Range Queries

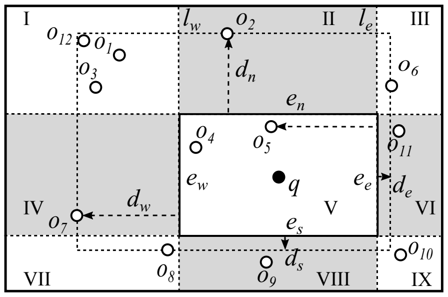
CR queries with a rectangular query range are commonly referred to as *continuous window queries*. We start with the bounding region based technique, estimated window vector, and then discuss the Minkowski region based technique, validity region.

4.1.1. The Estimated Window Vector. Huang and Huang [2013] propose a proxy-based approach to process the continuous window query, assuming a moving query object and static data points. This approach partitions the data space by a grid and assigns a proxy to handle the queries in a few grid cells. Query results are cached at the proxies and are used to answer subsequent queries in the same cells. Only when no result can be found in a local proxy's cache, a new query is forwarded to a central server for processing. This way, the query workload is distributed, and higher query efficiency is achieved. How the queries are delegated is beyond the scope of the survey. Huang and Huang [2013] propose to use the *estimated window vector* (EWV) as the safe region to avoid query recomputation when the query object moves at a proxy.

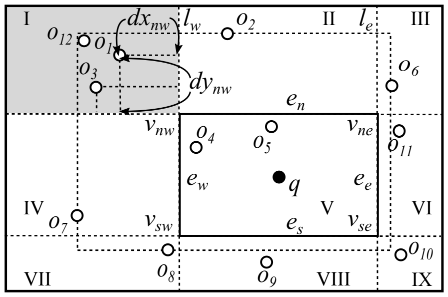
The EWV, denoted by ewv, is a vector representing the safe distances that the query object q (and the query window) can move in four different directions, namely, northeast (ne), northwest (nw), southeast (se), and southwest (sw), such that the query window stays between the *minimum* and *maximum* bounding rectangles of the result objects:

$$ewv = \langle\langle dx_{ne}, dy_{ne}\rangle, \langle dx_{nw}, dy_{nw}\rangle, \langle dx_{se}, dy_{se}\rangle, \langle dx_{sw}, dy_{sw}\rangle\rangle$$

Here, the distance that q can move in a direction, e.g., nw, is represented by a tuple, e.g., $\langle dx_{nw}, dy_{nw}\rangle$. The two values dx_{nw} and dy_{nw} denote the distances that q can move along the x-axis and y-axis in the nw direction, respectively.



(a) Computation of d_e, d_w, d_n, d_s



(b) Computation of dx_{nw}, dy_{nw}

Fig. 16: The estimated window vector

Figure 16 illustrates the computation of the EWV, where the solid rectangle in each sub-figure is the query window. The EWV takes two steps to compute. In Fig. 16(a), the four edges e_w, e_e, e_n, and e_s of the query window are extended to four lines l_w, l_e, l_n, and l_s that partition the data space into 9 regions labeled I to IX. The first step considers the west, east, north, and south regions (i.e., the shaded Regions IV, VI, II, and VIII). A vector $\langle d_w, d_e, d_n, d_s\rangle$ is computed, where d_w, d_e, d_n, d_s represent the safe distances that the query window can move in the four regions without reaching a data object. Take d_w as an example. Edge e_w can move west for a distance of $\texttt{dist}(e_w, o_7)$. Here, object o_7 is the closest object in the west region (Region IV). Since the query window has a fixed size, when e_w moves to the west, e_e moves to the west as well. The distance d_w needs to also guarantee that e_e does not reach o_5 which is a result object closest to e_e. Formally:

$$\begin{aligned}
&d_w = \min\{\texttt{dist}(e_w, o_7), \texttt{dist}(e_e, o_5)\}, \text{ where}\\
&o_7 = \underset{o_i}{\arg\max}\, o_i.x, \forall o_i \in \mathsf{O} \cap o_i.x < e_w.x \cap e_s.y \le o_i.y \le e_n.y, \text{ and}\\
&o_5 = \underset{o_i}{\arg\max}\, o_i.x, \forall o_i \in \mathsf{O} \cap e_w.x \le o_i.x \le e_e.x \cap e_s.y \le o_i.y \le e_n.y
\end{aligned} \quad (6)$$

Here, "$.x$" and "$.y$" denote the x- and y-coordinates of an object (a data object or an edge), respectively. The two objects o_7 and o_5 defining d_w can be identified with a simple scan on the object data set O. The other three distances d_e, d_n, and d_s are computed similarly.

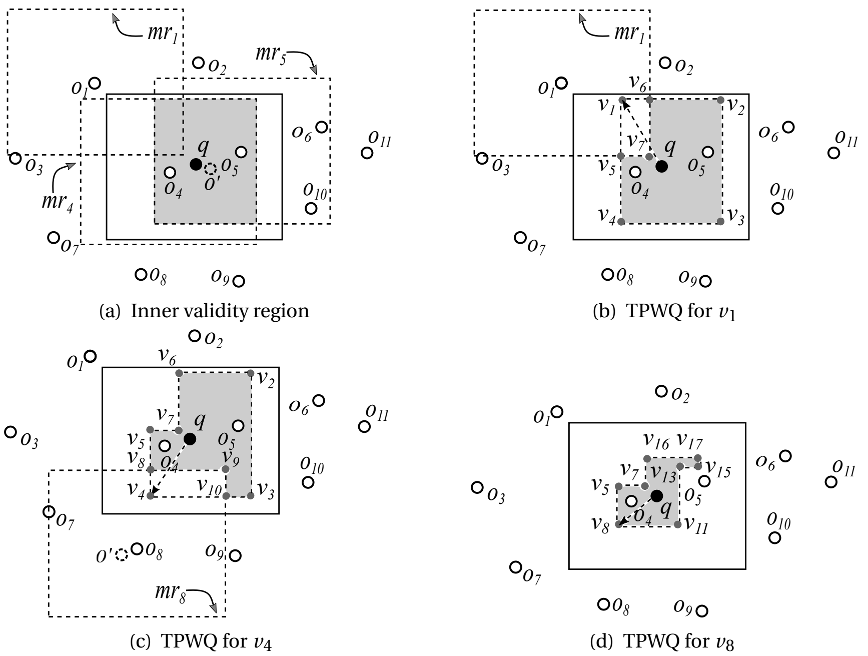
The vector $\langle d_w, d_e, d_n, d_s\rangle$ defines a rectangular region, i.e., the large dashed rectangle in Fig. 16(a). This rectangular region is safe in the region overlapping with Regions II, IV, V, VI, and VIII. However, this region may also overlap with the northwest, northeast, southeast, and southwest regions (Regions I, III, IX, and VII). The overlapping regions are *not* safe, since data objects in the overlapping regions were not considered when computing $\langle d_w, d_e, d_n, d_s\rangle$. The second step considers those data objects. Take the northwest region (Region I, the gray region in Fig. 16(b)) as an example. We find the *skyline objects* in this region. A skyline object forms a rectangle with the northwest vertex of the query window v_{nw} that does *not* enclose any data objects. Objects o_1 and o_3 are skyline objects as there is no data object inside the two small dashed rectangles formed by them and v_{nw}. They allow the query window to move towards them without reaching any non-result object. Object o_1 is chosen to define dx_{nw} and dy_{nw} as it forms a larger rectangle.

$$dx_{nw} = e_w.x - o_1.x, dy_{nw} = o_1.y - e_n.y$$

The other components of EWV are computed with the same procedure. The skyline objects themselves may be computed by any skyline algorithms (e.g., [Börzsönyi et al. 2001]), or simply by iterating through the objects and testing whether they satisfy the skyline object condition.

Continuous query processing. Once the EWV is computed, as long as the query window is bounded by the EWV, the current query result stays valid, and no query recomputation is required. When the query window leaves the region bounded by the EWV or changes its size, or if a data object update occurs, the query and the EWV are recomputed.

4.1.2. The Validity Region. The EWV is simple to compute. However, its safe region is not tight. This is because the EWV derives safe distances from only a limited number of moving directions to bound query window movement in all directions. As shown in Fig. 16(b), the direction towards o_1 is used to define dx_{nw} and dy_{nw}, while the query window can move towards o_3 without invalidating the query result. To compute a tight safe region, Zhang et al. [2003] use *Minkowski regions* (MR). Their safe region is named the *validity region.* They also assume a moving query object q, a rectangular query window centered at q, and a set of static data objects O. As mentioned earlier, the MR of a data object o_i, denoted by mr_i, is a rectangle identical to the query window centered at o_i. In Fig. 17(a), the solid rectangle represents the query window and the dashed rectangles represent MRs. The data objects o_4 and o_5 in the query window have MRs mr_4 and mr_5 that enclose q. A data object not in the query window, e.g., o_1, has an MR that does not enclose q either. In general, as long as the MRs of the result objects *all* enclose q and *no* MR of any non-result object encloses q, the result objects stay valid. The validity region is the intersection of the MRs of the result objects minus the MRs of the non-result objects. This safe region is tight by definition.



(a) Inner validity region (b) TPWQ for v_1

(c) TPWQ for v_4 (d) TPWQ for v_8

Fig. 17: Computation of the validity region

To compute the validity region, Zhang et al. [2003] first compute the *inner validity region*, which is the intersection of the MRs of the result objects. Since each MR is a rectangle, the intersection of multiple MRs remains a rectangle, e.g., the gray region in Fig. 17(a). To compute the intersection, only the MRs of the left-, right-, top-, and bottom-most result objects need to be considered. These few objects that contribute to the inner validity region are named the *inner influence objects*. In Fig. 17(a), o_4 and o_5 are influence objects. If there were another object o' between o_4 and o_5 (the dashed circle), its MR would not contribute to the inner validity region.

The MR of a non-result object only needs to be considered if it intersects the inner validity region. Such a non-result object is called an *outer influence object*. Even for an outer influence object, its MR may not have any impact on the final validity region, since the MR may be shadowed by the MRs of other outer influence objects. To identify the MRs that are not shadowed by others and compute the final validity region, the following procedure is used. At first, both the inner influence objects and the outer influence objects are added to a *candidate influence object set*, denoted as C_{inf}. In Fig. 17(a), $C_{inf} = \{o_1, o_2, o_4, o_5, o_6, o_8, o_9, o_{10}\}$. A *final influence object set*, denoted as S_{inf}, is initialized to be empty. Next, as illustrated in Fig. 17(b), a vertex is randomly chosen from the four vertices of the inner validity region v_1, v_2, v_3, and v_4. Let the vertex chosen be v_1. A *time-parameterized window query* (TPWQ) [Tao and Papadias 2002] is computed from q towards v_1. Similar to TPkNN, a TPWQ returns the first data object that may enter or leave the query window if q moves in a given direction. In Fig. 17(b), o_1 is returned since it is the first object to enter the query window when q moves towards v_1. The validity region is subtracted by mr_1. Object o_1 is then removed from C_{inf} and added to S_{inf}. Now the inner validity region has 6 vertices. Another vertex is randomly chosen, e.g., v_4, as shown in Fig. 17(c). A TPWQ towards v_4 discovers o_8, which is removed from C_{inf} and added to S_{inf}. Then mr_8 is subtracted from the validity region. Note that if there were another outer influence object $o' \in C_{inf}$ near o_8 (the dashed circle), the intersection of the MR of o' and the query window would be shadowed by mr_8. Thus, o' should be removed from C_{inf} as well. This procedure continues. The vertices of the remaining inner validity region are used to construct more TPWQs. Not every TPWQ returns an outer influence object. It may return an inner influence object as well. In Fig. 17(c), if the query object moves towards v_5, o_5 will be the first object to leave the query window. The inner validity region needs not to be updated, and v_5 is *confirmed*. Object o_5 is simply removed from C_{inf} and added to S_{inf}. Further, when few objects are left in C_{inf}, a TPWQ may not return any object at all. In this case, the corresponding vertex is also confirmed. When all the vertices are confirmed, the remaining inner validity region is returned as the final validity region, as illustrated in Fig. 17(d).

Continuous query processing. When a continuous window query is issued, the current query result is first computed (this can be done by any static window query algorithm). The validity region and the influence object set S_{inf} are then computed. Set S_{inf} is returned together with the query result. As long as q stays in the validity region, no computation occurs. This is checked by testing whether q remains in the MRs of the inner influence objects in S_{inf} and outside the MRs of the outer influence objects in S_{inf}. This testing is simpler than testing whether q is in the validity region, which may have an irregular shape. When the size of the query window changes or if a data object update occurs, a recomputation is needed to update the query result and S_{inf}.

4.2. Safe Regions in Spatial Networks

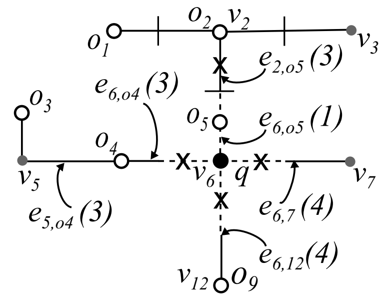
Given a spatial network $G = \langle V, E, \mathsf{W} \rangle$ as formulated in Section 3.3, a CR query on G with a moving query object q and a query range r reports all the data objects within distance r of q continuously. For a CR query on G, the idea of the validity region still applies. However, the Minkowski region must be replaced by a set of edge segments within distance r of a data object o_i. We call such edge segments the *Minkowski edge segments* (MES). The intersection of the MES of the result objects minus those of the non-result objects defines a tight safe region. Yung et al. [2012] call the boundary points of such a safe region *safe exits* and propose an algorithm to compute them. Cho et al. [2014] extend the algorithm to directed graphs. We discuss these two studies in Sections 4.2.1 and 4.2.2, respectively. Afyouni et al. [2012, 2014] partition an indoor space with a grid and build a grid graph over the space. A CR query on this graph is processed by continuously updating the *boundary nodes*, which are the graph vertices on the boundary of the query range. Only data objects inside the region bounded by the boundary nodes need to be considered for the query answer. Afyouni et al. organize the graph vertices in a multi-granule graph such that boundary nodes can be computed efficiently with a *hierarchical network expansion* technique (a procedure similar to a breadth-first traversal of the multi-granule graph). The boundary nodes define a natural safe region, i.e., the query range. We do not discuss them further.

4.2.1. Safe Exits. Safe exits are formulated based on the Minkowski edge segments. Let $\texttt{mes}(o_i)$ be the set of MES of a data object o_i:

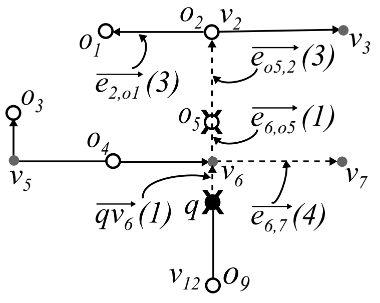
$$mes(o_i) = \{es | es \text{ is a segment of an edge } e_{i,j} \in E, \forall p \text{ on } es : \texttt{dist}_\texttt{n}(o_i, p) \le r\}$$

In an undirected graph G, if a data object o_i is in the query range then the query object q must be on the MES of o_i, and vice versa. The intersection of the MES of the result objects minus those of the non-result objects defines a tight safe region, denoted by $sr(q,r)$. Let O and S denote the object data set and the result set, respectively. Then:

$$sr(q,r) = \bigcap_{o_i \in \mathsf{S}} mes(o_i) \setminus \bigcup_{o_j \in \mathsf{O} \setminus \mathsf{S}} mes(o_j)$$



(a) Safe exits on an undirected network



o_1
o_2
v_2
v_3
$\overrightarrow{e_{2,o1}}$ (3)
$\overrightarrow{e_{o5,2}}$ (3)
o_3
o_5
$\overrightarrow{e_{6,o5}}$ (1)
o_4
v_5
v_6
v_7
$\overrightarrow{qv_6}$ (1)
q
$\overrightarrow{e_{6,7}}$ (4)
v_{12}
o_9

(b) Safe exits on a directed network

Fig. 18: Safe exits in spatial networks

The safe exits, denoted by $se(q,r)$, are the boundary points of $sr(q,r)$. Figure 18(a) gives an example. The query object q is represented by the black dot located at v_6 and the query range r is 2. The weights of the edge segments are: $e_{6,o5} = 1, e_{2,o5} = 3, e_{6,o4} = 3, e_{5,o4} = 3, e_{6,7} = 4$, and $e_{6,12} = 4$. The dashed line segments around q denote the query range. One object, o_5, is in the query range. Thus, the current result set is $\mathsf{S} = \{o_5\}$, and the set of non-result objects, $\mathsf{O} \setminus \mathsf{S}$, is the rest of the objects. The MES of o_5 consist of the edge segments around o_5 bounded by the X's:

$$mes(o_5) = \{\langle v_6, v_5, 0, 1\rangle, \langle v_6, v_7, 0, 1\rangle, \langle v_6, v_{12}, 0, 1\rangle, \langle v_6, v_2, 0, 3\rangle\}$$

Here, each edge segment is represented by a 4-tuple. The first two elements are the vertices of the edge that the segment lies on, e.g., the first segment lies on $e_{6,5}$. The last two elements specify the location of the segment on the edge, e.g., the first segment is at distance 0 to distance 1 from v_6. The MES of non-result object o_2, as denoted by the edge segments bounded by bars in the figure, overlap with $mes(o_5)$. A subtraction is then needed to get the safe region. Thus, the safe region is $sr(q,r) = \{\langle v_6, v_5, 0, 1\rangle, \langle v_6, v_7, 0, 1\rangle, \langle v_6, v_{12}, 0, 1\rangle, \langle v_6, v_2, 0, 2\rangle\}$, which consists of segments of $e_{6,o4}$, $e_{6,7}$, $e_{6,12}$ between v_6 and the X's, and the segment of $e_{6,2}$ between v_6 and the horizontal bar. The safe exits are $se(q,r) = \{\langle v_6, v_5, 1\rangle, \langle v_6, v_7, 1\rangle, \langle v_6, v_{12}, 1\rangle, \langle v_6, v_2, 2\rangle\}$ (three X's and the horizontal bar) . Here, every safe exit is represented by a 3-tuple: two vertices of the edge that the safe exit lies on, e.g., $\langle v_6, v_5\rangle$, and the distance from the first vertex to the safe exit, e.g., 1.

Yung et al. [2012] propose three rules to prune some data objects from consideration when computing the safe regions. These rules are detailed in Appendix B.

Continuous query processing. As earlier, once the safe exits have been computed, a query re-computation is only required if the query object moves across a safe exit, the query range r changes, or a data object update occurs within distance $3r$ of the previous query location.

4.2.2. Safe Exits in Directed Spatial Networks. The algorithm above uses a search starting from a data object o_i to find the MES, which considers only the distance from o_i to the edge segments. When edges are directed, the distance *from o_i to q* may not be the same as that *from q to o_i*. To extend safe exits to directed spatial networks, Cho et al. [2014] propose the *CRUISE* algorithm.

A spatial network with directed edges can be formulated as a directed graph $\overrightarrow{G} = \langle V, \overrightarrow{E}, \mathsf{W}\rangle$, where an edge $\overrightarrow{e_{i,j}} \in \overrightarrow{E}$ only allows travel from vertex v_i to vertex v_j.

The CRUISE algorithm starts from the query object and traverses $\overrightarrow{G}$ in a breadth-first manner until the safe exits are found. For each directed edge $\overrightarrow{e_{i,j}}$ visited, if a safe exit is found on $\overrightarrow{e_{i,j}}$ then the edges starting at v_j do not need to be visited. Otherwise, the edges starting at v_j are added to a queue Q of edges to be visited. The algorithm terminates when Q is empty, at which point all the safe exits have been found. When $\overrightarrow{e_{i,j}}$ is being processed, CRUISE computes the segment of the edge where any point can reach every current result object within distance r. This segment, denoted by $\overrightarrow{es_1}$, will be part of the intersection of the MES of the result objects. The algorithm computes another segment of the edge where any point can reach a non-result object within distance r. This segment, denoted by $\overrightarrow{es_2}$, is an MES of a non-result object. If $\overrightarrow{es_1}$ intersects $\overrightarrow{es_2}$ then the intersection point is a safe exit. The edges starting at v_j do not need to be visited further. It is also possible that $\overrightarrow{es_1}$ fully covers $\overrightarrow{e_{i,j}}$ and that $\overrightarrow{es_2}$ is empty. Then edge $\overrightarrow{e_{i,j}}$ is safe. It is added to the safe region. There is no safe exit on this edge, and the edges starting at v_j must be visited.

Figure 18(b) gives an example, where the black dot on edge $\overrightarrow{e_{12,6}}$ denotes the query object q and $r = 5$ (dashed edge segments are in the query range). The result objects are o_5 and o_2. Query object q breaks edge $\overrightarrow{e_{12,6}}$ into $\overrightarrow{v_{12}q}$ and $\overrightarrow{qv_6}$, both of which are treated as edges and are inserted into queue Q. No points on $\overrightarrow{v_{12}q}$ can be reached from q directly. A safe exit s_1 (denoted by X) is placed at q. No more edges starting at q need to be inserted into Q. The next edge to be processed is $\overrightarrow{qv_6}$. The maximum distance from any point on $\overrightarrow{qv_6}$ to the result objects o_5 and o_2 is $\texttt{dist}_\texttt{n}(q, o_2) = 5$, which does not exceed $r = 5$. The minimum distance from any point on $\overrightarrow{qv_6}$ to any non-result object is $\texttt{dist}_\texttt{n}(v_6, o_1) = 7$, which exceeds r. The entire edge $\overrightarrow{qv_6}$ is in the safe region, and no safe exit is found on $\overrightarrow{qv_6}$. Edges starting at v_6, $\overrightarrow{e_{6,7}}$ and $\overrightarrow{e_{6,2}}$, are then added to Q. No points on $\overrightarrow{e_{6,7}}$ can reach o_5 or o_2 within a distance of 5. This edge and any edge starting at v_7 are pruned. For $\overrightarrow{e_{6,2}}$, there is a safe exit at o_5 (denoted by X) because once passing o_5, it cannot be reached again within a distance of 5. The algorithm to find the safe exits then terminates.

To compute the segments $\overrightarrow{es_1}$ and $\overrightarrow{es_2}$, the key is to compute for a point p on $\overrightarrow{e_{i,j}}$ its distance to any given object o. Recall that in Section 3.3.2, the UNICONS algorithm uses a piecewise linear function to model the distance between q and a data object o when q moves along an edge. A similar approach is used by CRUISE to model the distance between p and o, assuming that p moves along $\overrightarrow{e_{i,j}}$. We omit the detailed distance function computation for conciseness.

Continuous query processing. The query processing procedure is similar to that used by Yung et al. [2012]. The difference lies in the way that the safe exits are computed as discussed above.

5. CONCLUSIONS AND FUTURE DEVELOPMENTS

We conclude the article with a summary of the papers reviewed and a discussion on future research directions of the safe region technique.

5.1. Conclusions

We have reviewed how safe regions are applied to process two fundamental continuous spatial queries (CSQ): continuous k nearest neighbor (CkNN) queries and continuous range (CR) queries. We gave an overview of the CSQs and the query processing challenges. We then presented the general notion of safe region and the key issues to consider when applying this technique. We detailed how different safe region based approaches enable CkNN and CR query processing. We conclude with a comparative summary of the techniques presented and then discuss future developments of safe region based query processing.

Table I summarizes the safe region based techniques for the CkNN queries. The key to safe region design is to maintain the nearness ranks of objects. Most techniques assume moving query objects and static data objects, the exception being that Find$k$NN [Benetis et al. 2001, 2002, 2006] assumes moving query and data objects, and builds safe regions that bound the query object such that the nearness ranks of the data objects stay unchanged. In contrast, SRB [Hu et al. 2005] assumes moving data objects and builds safe regions to bound the data objects.

Most approaches, except for SRB [Hu et al. 2005], lazy search [Song and Roussopoulos 2001], and the V*-diagram [Nutanong et al. 2008], use tight safe regions. To obtain tight safe regions,

Table I: Safe region based techniques for CkNN queries

Technique	Object moving	Tight	Precomputation free	Unknown query trajectory	Computation at query time	Incremental update	Data space	Time efficiency
SRB [Hu et al. 2005]	data	×	✓	✓	✓	✓	E	–
Voronoi diagram [Okabe et al. 1992]	query	✓	×	✓	×	×	E/M	↑↑↑↑
Lazy search [Song and Roussopoulos 2001]	query	×	✓	✓	✓	×	E/M	–
Find$k$NN [Benetis et al. 2001, 2002, 2006]	both	✓	✓	×	✓	✓	E/M	↑↑
TPkNN [Tao and Papadias 2002]	query	✓	✓	×	✓	✓	E/M	↑
kCNN [Tao et al. 2002]	query	✓	✓	×	✓	×	E/M	↑↑
RIS-kNN [Zhang et al. 2003]	query	✓	✓	✓	✓	×	E	↑↑
IRU [Kulik and Tanin 2006]	query	✓	✓	✓	✓	✓	E/M	–
V*-diagram [Nutanong et al. 2008]	query	×	✓	✓	✓	✓	E/M	↑↑↑
INS-kNN [Li et al. 2014]	query	✓	×	✓	✓	✓	E/M	↑↑↑↑
VN^3 [Kolahdouzan and Shahabi 2004b]	query	✓	×	✓	×	×	N/M	–
IE [Kolahdouzan and Shahabi 2004a]	query	✓	✓	×	✓	×	N	↑
UB [Kolahdouzan and Shahabi 2004a]	query	✓	✓	×	✓	×	N	↑↑
UNICONS [Cho and Chung 2005]	query	✓	✓	×	✓	×	N	↑↑↑
Network V*-diagram [Nutanong et al. 2010]	query	×	✓	✓	✓	✓	N/M	↑↑↑

those approaches require either precomputation of a (network) Voronoi diagram or a pre-known query path. Find$k$NN [Benetis et al. 2001, 2002, 2006], TPkNN [Tao and Papadias 2002], RIS-kNN [Zhang et al. 2003], and IRU [Kulik and Tanin 2006] are exceptions. However, without precomputation or a pre-known query path, these approaches incur higher safe region computation costs at query time. Find$k$NN and TPkNN compute safe regions assuming given movement vectors for the query (and data) objects. When these vectors are updated, the query results and safe regions need to be updated. RIS-kNN does not have this restriction for the query object, but requires multiple runs of TPkNN to cover all possible movement directions of the query object. Nutanong et al. [2008] show that the V*-diagram outperforms RIS-kNN in query efficiency due to cheaper online safe region computation, even though the safe regions computed are not tight. IRU also suffers in safe region computation efficiency because it needs to examine all the data objects. INS-kNN [Li et al. 2014] is the only approach that requires precomputation and computation of the safe region at query time. Due to the relatively lightweight precomputation of an order-1 Voronoi diagram, INS-kNN achieves a tight safe region that can be computed efficiently and updated online. This approach is the state-of-the-art for CkNN queries in Euclidean space. It is a good example of how to balance the tightness and computation costs of safe regions. The approaches for spatial networks all have a Euclidean counterpart. VN^3 [Kolahdouzan and Shahabi 2004b] and the network V*-diagram [Nutanong et al. 2010] are extensions of the Voronoi diagram and the V*-diagram where the graph shortest path distance replaces the Euclidean distance. IE, UB [Kolahdouzan and Shahabi 2004a], and UNICONS [Cho and Chung 2005] are network versions of TPkNN and lazy search, where influence points are replaced by split points.

Most of the approaches also apply to metric spaces, as long as the safe region computation is based on distance comparisons between the objects directly and the distance function is a metric. These approaches are noted as “E/M” or “N/M” in Table I. Some of these approaches, such as Find$k$NN, TPkNN, and kCNN, use distances between objects and R-tree nodes to accelerate the computation, which may not satisfy the requirements of a metric. This can be addressed by replacing the R-tree with a metric index, e.g., the M-tree [Ciaccia et al. 1997]. SRB, RIS-kNN, IE, UB, and UNICONS do not apply to metric space. SRB and RIS-kNN are for Euclidean space (noted as “E”). SRB is based on Euclidean distance and RIS-*k*NN is based on that the vertices of a safe region may be enumerated. IE, UB, and UNICONS are for spatial networks (noted as “N”). They are based on that the NNs of any point on an edge can be found from the NNs of the two vertices of the edge. These conditions do not generally hold in metric space.

Usage guidelines. In spatial networks, all C*k*NN query approaches surveyed assume a moving query object and a set of static data objects. UNICONS is the most efficient approach when the query trajectory is known, while the network V*-diagram is more efficient when the query trajectory is unknown. We indicate the comparative efficiency of the different approaches by ↑'s in Table I. Under the same settings, approaches with more ↑'s indicate a higher query time efficiency. For example, UNICONS has three ↑'s while UB and IE have only two and one ↑, respectively. VN^3 is an earlier approach and has the lowest efficiency among the approaches surveyed for C*k*NN queries in spatial networks. This is noted by “−” in the table. In Euclidean space, (i) when both the query object and the data objects are moving, Find*k*NN is the only approach that can compute safe regions for C*k*NN queries; (ii) when only the data objects are moving, SRB is a simple and easy-to-use approach; and (iii) when only the query object is moving, INS-*k*NN is the state-of-the-art. This is noted by four ↑'s for INS-*k*NN in the table as opposed to three ↑'s for the V*-digram and two ↑'s for RIS-*k*NN. For Case (iii), if the query trajectory is known, *k*CNN can be applied to break it into linear safe regions, which is more efficient than re-running TP*k*NN. We note this by two ↑'s for *k*CNN and only one ↑ for TP*k*NN. Find*k*NN generalizes the functionalities of *k*CNN and TP*k*NN to moving query and data objects. We annotate this approach with two ↑'s as well. Note that Find*k*NN, *k*CNN, and RIS-*k*NN have different query efficiency even though they all have two ↑'s, as they run for different settings. IRU does not have an experimental study while SRB and lazy search do not compare with any surveyed approaches. We annotate these approaches with “−”. The Voronoi digram has four ↑'s for its high query time efficiency (i.e., only requiring simple lookups), although its practicability is low due to its high precomputation costs.

Table II summarizes the safe region based techniques for CR queries. In CR queries, the query range is a natural safe region for the data objects. Existing studies thus focus on safe regions for query objects. The key is to define a region that *bounds and only bounds* the current result objects. In Euclidean space, two types of query range have been studied, rectangular and circular. In spatial networks, the query range is given as a distance of *r* from the query object. Regardless of the query range types, the safe regions proposed can be categorized into two classes. In Class 1, safe regions are defined in terms of the minimum and maximum bounding rectangles of the result objects [Huang and Huang 2013; AL-Khalidi et al. 2013]. These safe regions are not tight, but they are efficient to compute. In Class 2, safe regions are defined in terms of the Minkowski regions of the result objects [Zhang et al. 2003; Cheema et al. 2010; Yung et al. 2012; Cho et al. 2014]. These safe regions are tight, but they are less efficient to compute. Efforts have been devoted to improve the computation efficiency. Cheema et al. [2010] propose means to incrementally update query results and safe regions. The network version of Class 2 safe region approaches [Yung et al. 2012; Cho et al. 2014] share very similar ideas to those of the Euclidean space approaches, but further require a known query path. The range safe region, safe zone, and safe exits approaches also apply to metric space, again because they are based on direct distance comparison between the objects. The EWV and validity region approaches require bounding rectangles, which may not be formulated in metric space. The safe exits approach by Cho et al. [2014] assumes directed edges, which does not hold in metric space either.

Usage guidelines. All CR query approaches surveyed focus on computing safe regions to bound the movement of the query object, while the data objects are assumed to be static. In spatial networks, the two safe exit based approaches share similar ideas. The one by Yung et al. [2012] is proposed for undirected networks, while the one by Cho et al. [2014] can handle both undirected and directed networks. In Euclidean space, the validity region and safe zone approaches, for rectangular and circular queries, respectively, are to be used if minimizing the query answer recomputation and communication frequency has priority. The EWV and safe zone approaches are to be used if the safe region computation cost has priority. These papers do not contain per-

Table II: Safe region based techniques for CR queries

Technique	Object moving	Range type	Tight	Unknown query trajectory	Incremental update	Data space	Time efficiency
EWV [Huang and Huang 2013]	query	rectangular	×	✓	×	E	–
Validity region [Zhang et al. 2003]	query	rectangular	✓	✓	×	E	–
Range safe region [AL-Khalidi et al. 2013]	query	circular	×	✓	×	E/M	–
Safe zone [Cheema et al. 2010]	query	circular	✓	✓	✓	E/M	–
Safe exits [Yung et al. 2012]	query	network	✓	×	×	N/M	–
Safe exits (directed) [Cho et al. 2014]	query	network	✓	×	×	N	–

formance comparisons with each other. As discussed above, the algorithms have strengths in relation to different aspects, and it is not straightforward to predict their overall comparative performance. Thus, we indicate their query time efficiency by "–" in Table II.

5.2. Future Developments

The survey reveals three trends for future developments of the safe region technique.

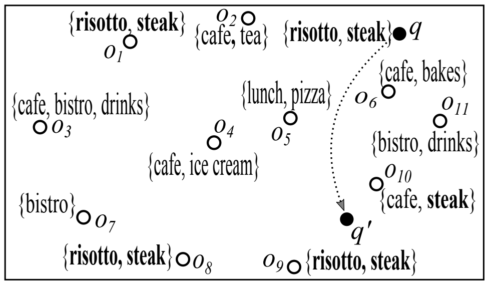
— To pursue safe regions that are both tight and efficient to compute. The safe region based approaches for CkNN queries show a typical trend. They have gone through a history from "tight but low computational efficiency (Voronoi diagram [Okabe et al. 1992])" to "far from tight but high computational efficiency (lazy search [Song and Roussopoulos 2001])" then to "tight and relatively low computational efficiency (RIS-kNN [Zhang et al. 2003])" and to "closer to tight and high computational efficiency (V*-diagram [Nutanong et al. 2008])" and finally "tight and high computational efficiency (INS-kNN [Li et al. 2014])". The safe region based approaches for CR queries have gone from "tight but relatively low computational efficiency (validity region [Zhang et al. 2003] and safe zone [Cheema et al. 2010])" to "high computational efficiency but far from tight (EWV [Huang and Huang 2013] and range safe region [AL-Khalidi et al. 2013])." Safe regions that are tight and efficient to compute await exploration. Further, when safe regions have irregular or complex shapes, it can be expensive to verify whether an object is in its safe region. Safe region verification for CR queries in the context of irregularly shaped safe regions is studied by Al-Khalidi et al. [2014]. This study assumes a known velocity of the query object. Lifting this assumption to achieve a more general safe region verification technique may be an interesting direction to explore.

— To handle more types of problem settings by the safe region technique. Examples include studies of CkNN query processing in constrained Euclidean space where obstacles may block the movements of objects [Wang et al. 2014a; Li et al. 2015], linearly shaped data objects [Gu et al. 2016b], query preferences over kNN objects far away from each other for diversification considerations [Gu et al. 2016a], or multiple CkNN and CR queries being monitored simultaneously [Choudhury et al. 2017]. Safe regions have been applied successfully in these settings. Other common problem settings include objects with uncertain locations [Sistla et al. 2015], queries with privacy considerations [Hashem et al. 2013], and queries in indoor settings [Yang et al. 2010; Afyouni et al. 2012, 2014]. CR queries with both a moving query object and moving data objects have not been considered in the context of safe region based query processing. How safe regions can be exploited in these settings needs to be studied.

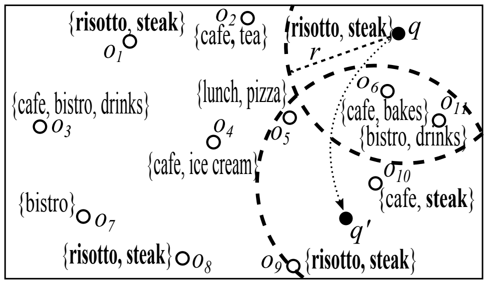
— To explore more types of queries that may be processed using safe regions. The CkNN and CR queries are fundamental types of CSQs. A variety of other types of CSQs have been considered, where safe region based query processing techniques have been proposed. For example, Ohsawa and Htoo [2016] propose a generalized safe region computation framework for CkNN queries, CR queries, and CRkNN queries in road networks. In the past few years, spatial keyword queries [Cong et al. 2009; Cao et al. 2011; Li et al. 2011; Cao et al. 2012; Chen et al. 2013; Cong and Jensen 2016] have attracted substantial interest. These queries extend traditional spatial queries by adding a set of textual keywords to every object, including the query object, which adds semantics to the objects. This kind of extension has been introduced to CSQs, resulting in *continuous top-k spatial keyword queries* [Rocha-Junior et al. 2011; Wu et al. 2011; Huang et al. 2012; Wu et al. 2013; Amagata et al. 2015; Chen et al. 2015; Guo et al. 2015; Zheng et al. 2016] and *continuous range keyword queries* [Chen et al. 2013; Wang et al. 2014b, 2015; Yu et al. 2015]. Typical definitions of these two types of queries are as follows.

Definition 5 (Continuous Top-k Spatial Keyword (CkSK) Query)**.** *Given a query object q, a set of data objects O, a weighting parameter $\alpha \in [0,1]$, and a query parameter k, the continuous top-k spatial keyword query maintains (from being issued until deactivated) a size-k subset $\mathsf{S} \subseteq \mathsf{O}$, such that $\forall o_i \in \mathsf{S}, o_j \in \mathsf{O} \backslash \mathsf{S} : \alpha\mathtt{dist}(q,o_i) + (1-\alpha)\mathtt{dist}_{\mathtt{w}}(q,o_i) \leq \alpha\mathtt{dist}(q,o_j) + (1-\alpha)\mathtt{dist}_{\mathtt{w}}(q,o_j)$.*

The CkSK query extends the concept of *nearness* to include both spatial proximity and textual relevance. An example is to "*find the k restaurants that are the nearest to Alice and serve risotto or steak as she is walking in a city.*" Figure 19(a) illustrates the query, where every object now is associated with a set of keywords. The top-k data objects may not be simply the spatial kNN objects any more. The result depends on $\mathtt{dist}(\cdot)$, $\mathtt{dist}_{\mathtt{w}}(\cdot)$, and α, which represent the spatial proximity, the textual relevance, and the preference between spatial proximity and textual relevance, respectively. The textual relevance here is usually measured by a document similarity metric such as the TFIDF metric [Wu et al. 2011; Huang et al. 2012; Wu et al. 2013].



(a) A CkSK query



(b) A CRK query

Fig. 19: Examples of continuous spatial keyword queries

Definition 6 (Continuous Range Keyword (CRK) Query)**.** *Given a query object q, a query range r, a set of data objects O, and a query predicate* $\mathtt{pred}(\cdot)$*, the continuous range keyword query maintains (from being issued until deactivated) the subset $\mathsf{S} \subseteq \mathsf{O}$ that contains every object $o_i \in \mathsf{O}$ where* $\mathtt{pred}(q,r,o_i) = \mathtt{true}$.

In the CRK query, the query predicate $\mathtt{pred}(\cdot)$ considers both spatial range and keyword set coverage. An example is to "*find all the restaurants serving risotto and steak that are within 1 Km distance from Alice as she is walking in a city.*" Figure 19(b) illustrates a CRK query. Now only the data objects that are within the query range and contain all the query keywords are in the query answer. At q there is no such data object, while at q', object o_9 enters the query answer. As location-based services are becoming increasingly popular, we envisage further research and industry efforts being devoted to these types of queries.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

References

I. Afyouni, C. Ray, S. Ilarri, and C. Claramunt. 2012. Algorithms for continuous location-dependent and context-aware queries in indoor environments. In *SIGSPATIAL*. 329–338.

I. Afyouni, C. Ray, S. Ilarri, and C. Claramunt. 2014. A PostgreSQL extension for continuous path and range queries in indoor mobile environments. *Pervasive and Mobile Computing* 15, C (2014), 128–150.

P. K. Agarwal, L. Arge, and J. Erickson. 2003. Indexing moving points. *Journal of Computer and System Science* 66, 1 (2003), 207–243.

H. AL-Khalidi, D. Taniar, J. Betts, and S. Alamri. 2013. On finding safe regions for moving range queries. *Mathematical and Computer Modelling* 58, 5–6 (2013), 1449–1458.

H. Al-Khalidi, D. Taniar, J. Betts, and S. Alamri. 2014. Monitoring moving queries inside a safe region. *The Scientific World Journal* 2014 (2014).

M. E. Ali, E. Tanin, R. Zhang, and L. Kulik. 2010. A motion-aware approach for efficient evaluation of continuous queries on 3D object databases. *The VLDB Journal* 19, 5 (2010), 603–632.

D. Amagata, T. Hara, and S. Nishio. 2015. Distributed top-k query processing on multi-dimensional data with keywords. In *SSDBM*. 10:1–10:12.

Andriod. 2017. *Creating and monitoring geofences.* https://developer.android.com/training/location/geofencing.html

Apple. 2017. *Region monitoring.* https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/LocationAwarenessPG/RegionMonitoring/RegionMonitoring.html

M. Attique, H.-J. Cho, and T.-S. Chung. 2016. *CORE: Continuous monitoring of reverse k nearest neighbors on moving objects in road networks.* Springer International Publishing, 109–124.

R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. 2001. *Nearest neighbor and reverse nearest neighbor queries for moving objects.* Technical Report TR-66. TimeCenter.

R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. 2002. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *International Database Engineering & Applications Symposium*. 44–53.

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

M. D. Berg, M. V. Kreveld, M. Overmars, and O. C. Schwarzkopf. 2000. *Computational geometry: Algorithms and Applications.* Springer.

C. Böhm, S. Berchtold, and D. A. Keim. 2001. Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. *Comput. Surveys* 33, 3 (2001), 322–373.

S. Börzsönyi, D. Kossmann, and K. Stocker. 2001. The skyline operator. In *ICDE*. 421–430.

Y. Cai, K. A. Hua, and G. Cao. 2004. Processing range-monitoring queries on heterogeneous mobile objects. In *MDM*. 27–38.

X. Cao, L. Chen, G. Cong, C. S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M. L. Yiu. 2012. Spatial keyword querying. In *ER*. 16–29.

X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. 2011. Collective spatial keyword querying. In *SIGMOD*. 373–384.

B. Chazelle and H. Edelsbrunner. 1987. An improved algorithm for constructing kth-order Voronoi diagrams. *IEEE Trans. Comput.* 100, 11 (1987), 1349–1354.

M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. 2010. Multi-guarded safe zone: an effective technique to monitor moving circular range queries. In *ICDE*. 189–200.

M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. 2011. Continuous monitoring of distance-based range queries. *IEEE Transactions on Knowledge and Data Engineering* 23, 8 (2011), 1182–1199.

M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li. 2012. Continuous reverse k nearest neighbors queries in Euclidean space and in spatial networks. *The VLDB Journal* 21, 1 (2012), 69–95.

L. Chen, G. Cong, and X. Cao. 2013. An efficient query indexing mechanism for filtering geo-textual data. In *SIGMOD*. 749–760.

L. Chen, G. Cong, X. Cao, and K.-L. Tan. 2015. Temporal spatial-keyword top-k publish/subscribe. In *ICDE*. 255–266.

L. Chen, G. Cong, C. S. Jensen, and D. Wu. 2013. Spatial keyword query processing: an experimental evaluation. In *PVLDB*. 217–228.

H.-J. Cho and C.-W. Chung. 2005. An efficient and scalable approach to CNN queries in a road network. In *VLDB*. 865–876.

H.-J. Cho, K. Ryu, and T.-S. Chung. 2014. An efficient algorithm for computing safe exit points of moving range queries in directed road networks. *Information Systems* 41 (2014), 1–19.

F. M. Choudhury, Z. Bao, J. S. Culpepper, and T. Sellis. 2017. Monitoring the top-m rank aggregation of spatial objects in streaming queries. In *ICDE*. 585–596.

C.-Y. Chow, M. F. Mokbel, and H. V. Leong. 2011. On efficient and scalable support of continuous queries in mobile peer-to-peer environments. *IEEE Transactions on Mobile Computing* 10, 10 (2011), 1473–1487.

P. Ciaccia, M. Patella, and P. Zezula. 1997. M-tree: an efficient access method for similarity search in metric spaces. In *VLDB*. 426–435.

G. Cong and C. S. Jensen. 2016. Querying geo-textual data: spatial keyword queries and beyond. In *SIGMOD*. 2207–2212.

G. Cong, C. S. Jensen, and D. Wu. 2009. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB* 2, 1 (2009), 337–348.

T. T. Do, K. A. Hua, and C. S. Lin. 2009. ExtRange: Continuous moving range queries in mobile peer-to-peer networks. In *MDM*. 317–322.

M. Erwig and F. Hagen. 2000. The graph Voronoi diagram with applications. *Networks* 36 (2000), 156–163.

R. A. Finkel and J. L. Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta Informatica* 4, 1 (1974), 1–9.

S. Fortune. 1987. A sweepline algorithm for Voronoi diagrams. *Algorithmica* 2, 1-4 (1987), 153–174.

V. Gaede and O. Günther. 1998. Multidimensional access methods. *Comput. Surveys* 30, 2 (1998), 170–231.

B. Gedik and L. Liu. 2004. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*. 67–87.

B. Gedik and L. Liu. 2006. MobiEyes: a distributed location monitoring service using moving location queries. *IEEE Transactions on Mobile Computing* 5, 10 (2006), 1384–1402.

B. Gedik, K.-L. Wu, P. Yu, and L. Liu. 2004. Motion adaptive indexing for moving continual queries over moving objects. In *CIKM*. 427–436.

Y. Gu, G. Liu, J. Qi, H. Xu, G. Yu, and R. Zhang. 2016a. The moving k diversified nearest neighbor query. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2778–2792.

Y. Gu, H. Zhang, Z. Wang, and G. Yu. 2016b. Efficient moving k nearest neighbor queries over line segment objects. *World Wide Web* 19, 4 (2016), 653–677.

L. Guo, J. Shao, H. Aung, and K.-L. Tan. 2015. Efficient continuous top-k spatial keyword queries on road networks. *GeoInformatica* 19, 1 (2015), 29–60.

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

T. Hashem, L. Kulik, and R. Zhang. 2013. Countering overlapping rectangle privacy attack for moving kNN queries. *Information Systems* 38, 3 (2013), 430–453.

A. M. Hendawi and M. F. Mokbel. 2012. Predictive spatio-temporal queries: a comprehensive survey and future directions. In *International Workshop on Mobile Geographic Information Systems*. 97–104.

G. R. Hjaltason and H. Samet. 1999. Distance browsing in spatial databases. *ACM Transactions on Database Systems* 24, 2 (1999), 265–318.

Y.-L. Hsueh, R. Zimmermann, and W.-S. Ku. 2009. Adaptive safe regions for continuous spatial queries over moving objects. In *DASFAA*. 71–76.

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

J.-L. Huang and C.-C. Huang. 2013. A proxy-based approach to continuous location-based spatial queries in mobile environments. *IEEE Transactions on Knowledge and Data Engineering* 25, 2 (2013), 260–273.

W. Huang, G. Li, K.-L. Tan, and J. Feng. 2012. Efficient safe-region construction for moving top-k spatial keyword queries. In *CIKM*. 932–941.

S. Ilarri, E. Mena, and A. Illarramendi. 2006. Location-dependent queries in mobile contexts: distributed processing using mobile agents. *IEEE Transactions on Mobile Computing* 5, 8 (2006), 1029–1043.

S. Ilarri, E. Mena, and A. Illarramendi. 2010. Location-dependent query processing: Where we are and where we are heading. *Comput. Surveys* 42, 3 (2010), 12:1–12:73.

T. Imielinski and B. Badrinath. 1992. Querying in highly mobile distributed environments. In *VLDB*. 41–52.

G. S. Iwerks, H. Samet, and K. Smith. 2003. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*. 512–523.

G. S. Iwerks, H. Samet, and K. P. Smith. 2006. Maintenance of k-nn and spatial join queries on continuously moving points. *ACM Transactions on Database Systems* 31, 2 (2006), 485–536.

C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. 2003. Nearest neighbor queries in road networks. In *GIS*. 1–8.

C. S. Jensen, D. Lin, and B. C. Ooi. 2004. Query and update efficient B^+-tree based indexing of moving objects. In *VLDB*. 768–779.

D. V. Kalashnikov, S. Prabhakar, S. E. Hambrusch, and W. G. Aref. 2002. Efficient evaluation of continuous range queries on moving objects. In *DEXA*. 731–740.

M. R. Kolahdouzan and C. Shahabi. 2004a. Continuous k-nearest neighbor queries in spatial network databases. In *STDBM*. 33–40.

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

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

N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. 2004. Approximate NN queries on streams with guaranteed error/performance bounds. In *VLDB*. 804–815.

J. Krumm. 2009. A survey of computational location privacy. *Personal Ubiquitous Comput.* 13, 6 (2009), 391–399.

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

J. Lee, S. Kang, Y. Lee, S. J. Lee, and J. Song. 2009. BMQ-Processor: a high-performance border-crossing event detection framework for large-scale monitoring applications. *IEEE Transactions on Knowledge and Data Engineering* 21, 2 (2009), 234–252.

C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and Q. Deng. 2016. INSQ: An influential neighbor set based moving kNN query processing system. In *ICDE*. 1338–1341.

C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and W. Yi. 2014. Processing moving kNN queries using influential neighbor sets. *PVLDB* 8, 2 (2014), 113–124.

C. Li, Y. Gu, J. Qi, R. Zhang, and G. Yu. 2015. A safe region based approach to moving KNN queries in obstructed space. *Knowledge and Information Systems* 45, 2 (2015), 417–451.

Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. Lee, and X. Wang. 2011. IR-Tree: an efficient index for geographic document search. *IEEE Transactions on Knowledge and Data Engineering* 23, 4 (2011), 585–599.

M. F. Mokbel and W. G. Aref. 2008. SOLE: scalable on-line execution of continuous queries on spatio-temporal data streams. *The VLDB Journal* 17, 5 (2008), 971–995.

M. F. Mokbel, T. M. Ghanem, and W. G. Aref. 2003. Spatio-temporal access methods. *IEEE Data Eng. Bull.* 26, 2 (2003), 40–49.

M. F. Mokbel, X. Xiong, and W. G. Aref. 2004. SINA: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*. 623–634.

K. Mouratidis, S. Bakiras, and D. Papadias. 2009. Continuous monitoring of spatial queries in wireless broadcast environments. *IEEE Transactions on Mobile Computing* 8, 10 (2009), 1297–1311.

K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. 2005b. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *IEEE Transactions on Knowledge and Data Engineering* 17, 11 (2005), 1451–1464.

K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. 2005a. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*. 634–645.

L.-V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. 2010. Spatio-temporal access methods: part 2 (2003 - 2010). *IEEE Data Eng. Bull.* 33, 2 (2010), 46–55.

J. Nievergelt, H. Hinterberger, and K. C. Sevcik. 1984. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems* 9, 1 (1984), 38–71.

S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. 2008. The V*-Diagram: A query dependent approach to moving kNN queries. *PVLDB* 1, 1 (2008), 1095–1106.

S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. 2010. Analysis and evaluation of V*-*k*NN: an efficient algorithm for moving *k*NN queries. *The VLDB Journal* 19, 3 (2010), 307–332.

Y. Ohsawa and H. Htoo. 2016. Versatile safe-region generation method for continuous monitoring of moving objects in the road network distance. In *DASFAA*. 377–392.

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

J. Orenstein and T. Merrett. 1984. A class of data structures for associative searching. In *PODS*. 181–190.

D. Pfoser, C. S. Jensen, and Y. Theodoridis. 2000. Novel approaches in query processing for moving object trajectories. In *VLDB*. 395–406.

S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. 2002. Query indexing and velocity constrained indexing: scalable techniques for continuous queries on moving objects. *IEEE Trans. Comput.* 51, 10 (2002), 1124–1140.

F. P. Preparata and M. Shamos. 1985. *Computational geometry: an introduction.* Springer.

J. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. 2011. Efficient processing of top-k spatial keyword queries. In *SSTD*. 205–222.

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

S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. 2000. Indexing the positions of continuously moving objects. In *SIGMOD*. 331–342.

R. Seidel. 1988. *Constrained Delaunay triangulations and Voronoi diagrams with obstacles.* Technical Report Technical Report 260. IIG-TU Graz, Austria.

M. Sharifzadeh and C. Shahabi. 2010. VoR-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. *PVLDB* 3, 1-2 (2010), 1231–1242.

R. I. D. Silva, D. F. Macedo, and J. M. S. Nogueira. 2014. Spatial query processing in wireless sensor networks - a survey. *Information Fusion* 15 (2014), 32–43.

A. P. Sistla, O. Wolfson, and B. Xu. 2015. Continuous nearest-neighbor queries with location uncertainty. *The VLDB Journal* 24, 1 (2015), 25–50.

C. Smith. 2016. *By the numbers: 20 important foursquare stats.* http://expandedramblings.com/index.php/by-the-numbers-interesting-foursquare-user-stats/

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

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

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

H. Wang and R. Zimmermann. 2007. Location-based query processing on moving objects in road networks. In *VLDB*. 321–332.

H. Wang and R. Zimmermann. 2011. Processing of continuous location-based range queries on moving objects in road networks. *IEEE Transactions on Knowledge and Data Engineering* 23, 7 (2011), 1065–1078.

X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. 2014b. Selectivity estimation on streaming spatio-textual data using local correlations. *PVLDB* 8, 2 (2014), 101–112.

X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. 2015. AP-tree: efficiently support continuous spatial-keyword queries over stream. In *ICDE*. 1107–1118.

Y. Wang, R. Zhang, C. Xu, J. Qi, Y. Gu, and G. Yu. 2014a. Continuous visible k nearest neighbor query on moving objects. *Information Systems* 44 (2014), 1 – 21.

P. G. D. Ward, Z. He, R. Zhang, and J. Qi. 2014. Real-time continuous intersection joins over large sets of moving objects using graphic processing units. *The VLDB Journal* 23, 6 (2014), 965–985.

D. Wu, M. L. Yiu, and C. S. Jensen. 2013. Moving spatial keyword queries: formulation, methods, and analysis. *ACM Transactions on Database Systems* 38, 1 (2013), 7:1–7:47.

D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. 2011. Efficient continuously moving top-k spatial keyword query processing. In *ICDE*. 541–552.

X. Xiong, M. F. Mokbel, and W. G. Aref. 2005. SEA-CNN: scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*. 643–654.

B. Yang, H. Lu, and C. S. Jensen. 2010. Probabilistic threshold k nearest neighbor queries over moving objects in symbolic indoor space. In *EDBT*. 335–346.

M. L. Yiu, E. Lo, and D. Yung. 2011. Authentication of moving kNN queries. In *ICDE*. 565–576.

M. Yu, G. Li, and J. Feng. 2015. A cost-based method for location-aware publish/subscribe services. In *CIKM*. 693–702.

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

D. Yung, M. L. Yiu, and E. Lo. 2012. A safe-exit approach for efficient network-based moving range queries. *Data & Knowledge Engineering* 72 (2012), 126–147.

K. Zeberga, R. Jin, H.-J. Cho, and T.-S. Chung. 2017. A safe-region approach to a moving *k*-rnn queries in a directed road network. *Journal of Circuits, Systems, and Computers* 26, 5 (2017), 1–18.

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

R. Zhang, H. V. Jagadish, B. T. Dai, and K. Ramamohanarao. 2010. Optimized algorithms for predictive range and kNN queries on moving objects. *Information Systems* 35, 8 (2010), 911–932.

R. Zhang, D. Lin, R. Kotagiri, and E. Bertino. 2008. Continuous intersection joins over moving objects. In *ICDE*. 863–872.

R. Zhang, B. C. Ooi, and K.-L. Tan. 2004. Making the pyramid technique robust to query types and workloads. In *ICDE*. 313–324.

R. Zhang, J. Qi, D. Lin, W. Wang, and R. C.-W. Wong. 2012. A highly optimized algorithm for continuous intersection join queries over moving objects. *The VLDB Journal* 21, 4 (2012), 561–586.

B. Zheng, K. Zheng, X. Xiao, H. Su, H. Yin, X. Zhou, and G. Li. 2016. Keyword-aware continuous kNN query on road networks. In *ICDE*. 871–882.

Received November 2016; revised August 2017; accepted March 2018

Online Appendix to: Continuous Spatial Query Processing: A Survey of Safe Region Based Techniques

JIANZHONG QI, University of Melbourne
RUI ZHANG, University of Melbourne
CHRISTIAN S. JENSEN, Aalborg University
KOTAGIRI RAMAMOHANARAO, University of Melbourne
JIAYUAN HE, University of Melbourne

A. SAFE REGION FOR CIRCULAR RANGE QUERIES

Safe regions and query processing procedures for CR queries with a circular query range are similar to those for window queries. The notions of *range safe region* [AL-Khalidi et al. 2013] and *safe zone* [Cheema et al. 2010] are counterparts of the estimated window vector and the validity region used for window queries. We discuss these two techniques in Sections A.1 and A.2 but omit the corresponding query processing procedures for conciseness.

A.1. The Range Safe Region

Similar to the estimated window vector, the range safe region is defined using the minimum and maximum bounding regions of the current result objects. Since the query range is now circular, bounding circles are used to form the safe regions. Figure 20(a) exemplifies the range safe region. The query object is at q, and the query range is enclosed by circle C_q with radius r (the solid circle). Two dashed circles, C_{min} and C_{max}, centered at q are drawn. The radius of C_{min}, r_{min}, is the *maximum* distance between q and any result object; the radius of C_{max}, r_{max}, is the *minimum* distance between q and any non-result object:

$$r_{min} = \max\{\mathtt{dist}(q, o_i) | o_i \in \mathcal{O}, \mathtt{dist}(q, o_i) \leq r\}; \quad r_{max} = \min\{\mathtt{dist}(q, o_i) | o_i \in \mathcal{O}, \mathtt{dist}(q, o_i) > r\}$$

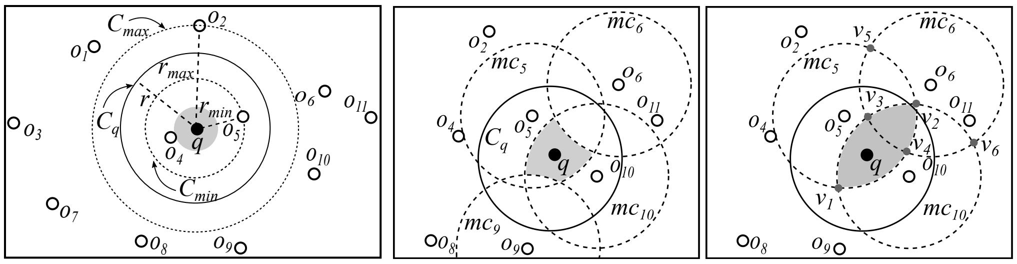
The two circles are the minimum and maximum bounding circles of the query result. As long as C_q is in the ring formed by these two circles, the query result does not change. This means that the query object should not move away from q farther than distance $\min\{r - r_{min}, r_{max} - r\}$. This distance defines a circular region centered at q (the gray region), which is the range safe region.

A.2. The Safe Zone

Like the estimated window vector, the range safe region is not tight. Cheema et al. [2010] propose the *safe zone* that is a tight safe region for circular CR queries. The safe zone is a variant of the validity region where the rectangular Minkowski regions are replaced by circular regions. Such circular regions center at the data objects, and their radii are all the same as the query range r. We call them *Minkowski circles* (MC) for simplicity. We denote the MC of object o_i by mc_i. Then o_i being inside the query range is equivalent to the query object being inside mc_i. The safe zone is the intersection of the MCs of the result objects minus the MCs of all non-result objects. Figure 20(b) shows an example, where the black dot is the query point q, the solid circle C_q with radius r is the query range, and the dashed circles are MCs. Objects o_5 and o_{10} are in C_q. The query point q is inside the intersection of their MCs, mc_5 and mc_{10}. As long as q stays in this intersection, o_5 and o_{10} stay as the query answer. This intersection overlaps with the MCs of o_6 and o_9. The query point q needs to stay outside the overlapping regions to keep o_6 and o_9 outside the query answer. Thus, the final safe zone is $mc_5 \cap mc_{10} \setminus (mc_6 \cup mc_9)$, which is the gray region.

The safe zone is computed progressively by adding (subtracting) one MC to (from) the current safe zone at a time. The MCs are processed in ascending order of the distances between their

© 2018 ACM. 0360-0300/2018/01-ART1 $15.00
DOI: 10.1145/3193835



(a) The range safe region (b) The safe zone (c) Computation of the safe zone

Fig. 20: Safe regions for circular range queries

corresponding data objects and q. This guarantees that no MC that can affect the safe zone is missed. As Fig. 20(c) shows, assume that MCs mc_5 and mc_{10} have already been processed, yielding the intermediate safe zone given by the gray region. The intersection points of the two circles, v_1 and v_2, are stored in a list V of safe zone vertices. The MC of object o_6, mc_6, is processed next. The intersection points between mc_6 and the two existing MCs are computed, which are v_3, v_4, v_5, and v_6. The points v_3 and v_4 that are on the boundary of the current safe zone are added to list V. Points v_5 and v_6 are discarded. Then the existing vertices v_1 and v_2 in V are checked against mc_6. Only vertices outside mc_6 are kept (i.e., v_1), since o_6 is a non-result object and anything inside mc_6 should be excluded from the safe zone. Had o_6 been a result object then only vertices inside mc_6 should be kept. Now the safe zone is updated to be the region $v_1 v_3 v_4$. This procedure continues until an MC with no intersection with the current safe zone is reached. The safe zone at that moment is the final safe zone.

Cheema et al. [2010] use an R-tree to index the data objects (not their MCs) to speed up computation of the safe zone. A best-first traversal is used to access the tree nodes and data objects. Pruning rules are used to reduce the search space for finding the data objects with MCs that may impact the safe zone. They use the relative position (e.g., the `mindist` and `maxdist` metrics) between a tree node and the safe zone computed so far to prune nodes from consideration. We omit the details of these pruning rules for conciseness.

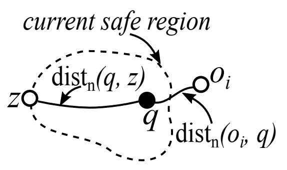
B. PRUNING RULES FOR FAST SAFE EXIT COMPUTATION

Computing the Minkowski edge segments (MES) of a data object requires a run of Dijkstra's algorithm on the spatial network, which is expensive. To reduce the cost, Yung et al. [2012] propose to only consider the data objects within distance $3r$ of the query object q. All other data objects can be discarded. The reason is as follows. The result objects are within distance r of q. Their MES must be within distance $2r$ of q. For any other object, if its MES interest with the MES of the result objects, it must be within distance $3r$ of q.

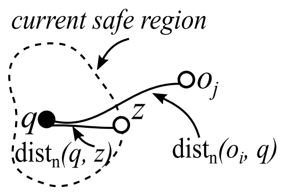
Dijkstra's algorithm is first run to identify the objects within distances r and $3r$ of q. The objects within distance r are returned as the result. Then the MES of every object within distance $3r$ of q are computed using Dijkstra's algorithm. The safe region and safe exits are computed incrementally as more MES are obtained. During this process, three pruning rules are applied to reduce cost. Let z be the farthest point from q of the current safe region.

— Pruning rule 1 (Fig. 21(a)): If the distance between a result object o_i and q satisfies $\mathtt{dist}_{\mathtt{n}}(o_i, q) + \mathtt{dist}_{\mathtt{n}}(q, z) \le r$ then $\mathtt{dist}_{\mathtt{n}}(o_i, z) \le r$, and $mes(o_i)$ is guaranteed to fully cover the intermediate safe region. Thus, the MES of o_i have no impact on the safe region. Such result objects can be pruned safely.

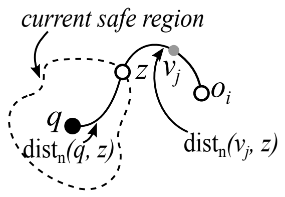
— Pruning rule 2 (Fig. 21(b)): If the distance between a non-result object o_j and q satisfies $\mathtt{dist}_{\mathtt{n}}(o_j, q) - \mathtt{dist}_{\mathtt{n}}(q, z) \ge r$ then $\mathtt{dist}_{\mathtt{n}}(o_j, z) \ge r$, and $mes(o_j)$ cannot intersect any edge segments in the safe region. Thus, the MES of o_j have no impact on the safe region. Such non-result objects can be pruned safely.



(a) Pruning a result object



(b) Pruning a non-result object



(c) Pruning irrelevant edges

Fig. 21: Pruning rules for safe exit computation

— Pruning rule 3 (Fig. 21(c)): Let o_i be the object that the MES are being computed for and v_j be the next vertex to be visited by Dijkstra's algorithm. If $\texttt{dist}_\texttt{n}(o_i, v_j) + \texttt{dist}_\texttt{n}(v_j, z) > r$ then any path from v_j can be pruned safely as it cannot reach the safe region by distance r. Here, computing $\texttt{dist}_\texttt{n}(v_j, z)$ needs another graph traversal. However, we can replace it by an estimate of $\texttt{dist}_\texttt{n}(v_j, q) - \texttt{dist}_\texttt{n}(q, z)$, where $\texttt{dist}_\texttt{n}(q, z)$ was computed when computing z, and $\texttt{dist}_\texttt{n}(v_j, q)$ was computed when searching for the objects within distance $3r$ of q.