The VLDB Journal manuscript No.
(will be inserted by the editor)

Continuous Reverse k Nearest Neighbors Queries in Euclidean Space and in Spatial Networks

Muhammad Aamir Cheema · Wenjie Zhang · Xuemin Lin · Ying Zhang · Xuefei Li

Received: date / Accepted: date

Abstract In this paper, we study the problem of continuous monitoring of reverse k nearest neighbors queries in Euclidean space as well as in spatial networks. Existing techniques are sensitive towards objects and queries movement. For example, the results of a query are to be re-computed whenever the query changes its location. We present a framework for continuous reverse k nearest neighbor (RkNN) queries by assigning each object and query with a *safe region* such that the expensive recomputation is not required as long as the query and objects remain in their respective safe regions. This significantly improves the computation cost. As a by-product, our framework also reduces the communication cost in client-server architectures because an object does not report its location to the server unless it leaves its safe region or the server sends a location update request. We also conduct a rigid cost analysis for our Euclidean space RkNN algorithm. We show that our techniques can also be applied to answer *bichromatic* RkNN queries in Euclidean space as well as in spatial networks. Furthermore, we show that our techniques can be extended for the spatial networks that are represented by directed graphs. The extensive experiments demonstrate that our techniques outperform the existing techniques by an order of magnitude in terms of computation cost and communication cost.

Muhammad Aamir Cheema, Wenjie Zhang, Ying Zhang
School of Computer Science and Engineering,
The University of New South Wales, Australia
E-mail: {macheema,zhangw,yingz,}@cse.unsw.edu.au

Xuemin Lin
School of Computer Science and Engineering,
The University of New South Wales, Australia
and
Software College, East China Normal University
E-mail: lxue@cse.unsw.edu.au

Xuefei Li
School of Computer Science,
Fudan University, Shanghai, China
E-mail: xuefei.li89@gmail.com

1 Introduction

Given a query point q, a reverse k nearest neighbor (RkNN) query retrieves all the data points that have q as one of their k nearest neighbors (k closest points). Throughout this paper, we use RNN queries to refer to RkNN queries for which $k = 1$. We give a formal definition of the RkNN problem in Section 2. Consider the example of Fig. 1 where q is a RNN query in Euclidean space. The nearest neighbor (the closest object in Euclidean space) of q is o_1. However, o_1 is not the RNN of q because the closest point of o_1 is not q. The RNNs of q are o_3 and o_4 because q is the nearest neighbor for both of these points.

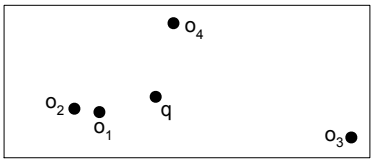


Fig. 1 o_3 and o_4 are RNNs of q in Euclidean space

RNN has received considerable attention [21,39,1,37, 23,42,44,53,52,48] from database research community based on the applications such as decision support, location based service, resource allocation, profile-based management, etc.

With the availability of inexpensive mobile devices, position locators and cheap wireless networks, location based services are gaining increasing popularity. Consider the example of a battlefield. A backup army unit may issue a RNN query to monitor other units for which this is the closest army unit. Such army units may seek help from the backup army unit in the case of an emergency event. Therefore the backup army unit may is-

sue a RNN query to retrieve such army units and may observe their status from time to time (e.g., current location, ammunition etc.).

Note that in the above example, the query objects and the data objects both belong to the same type of objects (i.e., army units). Such queries are called *monochromatic* queries. The queries where the query objects and the data objects belong to two different types of objects are called *bichromatic* queries (formally defined in Section 2.1). Consider the example of a user that needs a taxi and sends her location to a taxi company's dispatch center. The company owns several taxis and wants to send this job to a taxi for which she is the closest passenger. Hence, the company notifies the taxis that are among the bichromatic RNNs of the user. Cabspotting[1] and Zhiing[2] are two examples of such location based services.

Other examples of location based services include location based games, traffic monitoring, location based SMS advertising, enhanced 911 services and army strategic planning etc. These applications may require continuous monitoring of reverse nearest moving objects. For instance, in reality games (e.g., BotFighters, Swordfish), players with mobile devices search for other mobile devices in neighborhood. For example, in the award winning game BotFighters, a player gets points by shooting other nearby players via mobiles. In such an application, some players may want to continuously monitor her reverse nearest neighbors in order to avoid being shot by other players.

Driven by such applications, the continuous monitoring of reverse nearest neighbors has been investigated and several techniques have been proposed recently [1, 18,47,49,41] in the light of location-based services. The existing continuous monitoring techniques [1,18,47,49] adopt two frameworks based on different applications. In [1], the velocity of each object is assumed to be explicitly expressed while [18,47,49] deal with a general situation where the object velocities may be impossible to be explicitly expressed. In this paper, our research is based on the general situation; that is, object velocities are not explicitly expressible.

The techniques in [18,47,49] adopt a two-phase computation. In the *filtering* phase, objects are pruned by using the existing pruning paradigms from [39,42] and the remaining objects are considered as the candidate objects. In the *verification* phase, every candidate object for which the query is its closest point is reported as the RNN. To update the results, at each time-stamp, if the set of candidate objects is detected to be unchanged then only the verification phase is called to verify the results. Nevertheless, both the filtering and verification phases are required if one of the candidate objects changes its location or other objects move into the candidate region. Similarly, a set of candidate objects is needed to be re-computed (recall filtering) if the query changes its location.

As mentioned earlier, previous techniques [18,49,47] require expensive filtering if the query or any of the candidate objects changes its location. Our initial experiment results show that the cost of verification phase is much lower than the cost of filtering phase. In our technique, we assign each query and object a safe region. The safe region is a rectangular area for the queries in Euclidean space and is an edge (or a part of the edge) for the queries in spatial networks. The filtering phase for a query is not required as long as the query and its candidate objects remain in their corresponding safe regions. This significantly reduces the computation time of continuously monitoring RkNN queries.

As a by-product, our proposed framework also significantly reduces the communication cost in a client-server architecture. In the existing techniques, every object reports its location to the server at every time-stamp regardless whether query results will be affected or not. Consequently, such a computation model requires transmission of a large number of location updates; doing this has a direct impact on the wireless communication cost and power consumption - the most precious resources in mobile environment [15]. In our framework, each moving object reports its location update only when it leaves the region. This significantly saves the communication costs.

Depending on the users' needs, applications may require RNN queries to be monitored in Euclidean space or in spatial networks (e.g., a road network). While several algorithms have been proposed to monitor RNN queries in Euclidean space there does not exist any algorithm that efficiently updates RNNs in spatial networks after the objects and queries change their locations. In this paper, we present efficient algorithms to monitor RNN queries in Euclidean space as well as in spatial networks.

Below, we summarize our contributions:

Query processing in Euclidean space

1. We present a framework for continuously monitoring RNN together with a novel set of effective pruning and efficient increment computation techniques. It not only reduces the total computation cost of the system but also reduces the communication cost.
2. We extend our algorithm for the continuous monitoring of RkNN. Our algorithm can be used to monitor both *mono-chromatic* and *bichromatic* RkNN (to be formally defined in Section 2.1).
3. We provide a rigid analysis on the computation and communication costs of our algorithm that helps us to understand the effect of the size of the safe region on the costs of our algorithm.

[1] http://cabspotting.org/faq.html
[2] http://www.zhiing.com/how.php

4. Our extensive experiments demonstrate that the developed techniques outperform the previous algorithms by an order of magnitude in terms of computation cost and communication cost.

Query processing in spatial networks

5. We are first to present a continuous RNN monitoring algorithm for moving objects and queries in spatial networks. The proposed algorithm is computationally efficient and has low communication cost.
6. We show that our technique can be easily extended to monitor *mono-chromatic* and *bichromatic* RNN queries. The algorithm can also be extended to continuously monitor RkNN queries.
7. We conduct extensive experiments on a real road network and demonstrate that our algorithm gives an order of magnitude improvement over an algorithm that does not use the safe regions.

This paper is the extended version of our previous work on RkNN query processing in Euclidean space [7]. In this extended paper, we extend the techniques in [7] to process RkNN queries in spatial networks and present extensive experimental results to evaluate the efficiency. In addition, we also provide a brief survey of the previous work related to the query processing in spatial networks.

The rest of the paper is organized as follows. In Section 2, we give the problem statement, related work and motivation. Section 3 presents RNN monitoring techniques for Euclidean space including a detailed theoretical analysis. Section 4 presents our technique for continuously monitoring RNN queries and its variants in spatial networks. The experiment results are reported in Section 5. Section 6 concludes the paper.

2 Background Information

In this section, we first formally define the problem in Section 2.1 followed by a brief description of related work in Section 2.2. We present the motivation of our work in Section 2.3.

2.1 Problem Definition

There are two types of RkNN queries [21] namely, *monochromatic* and *bichromatic* RkNN queries. Below we define both.

Monochromatic RkNN query: Given a set of points P and a point $q \in P$, a monochromatic RkNN query retrieves every point $p \in P$ s.t. $dist(p,q) \leq dist(p,p_k)$ where $dist()$ is a distance function, and p_k is the kth nearest point to p according to the distance function $dist()$. In Euclidean space, $dist(x,y)$ returns the Euclidean distance between any two points x and y. In spatial networks, $dist(x,y)$ returns the minimum network distance between any two points lying on the spatial network.

Note that, in such queries, both the data objects and the query objects belong to the same class of objects. Consider an example of the reality game BotFighters, where a player issues a query to find other players for whom she is the closest person.

Bichromatic RkNN query: Given two sets O and P each containing different types of objects, a bichromatic RkNN query for a point $q \in O$ is to retrieve every object $p \in P$ such that $dist(p,q) \leq dist(p,o_k)$ where o_k is the kth nearest point of p in O according to the distance function $dist()$.

In contrast to monochromatic queries, the query and data objects belong to two different classes. Consider the example of a battlefield where a medical unit might issue a bichromatic RNN query to find the wounded soldiers for whom it is the closest medical unit.

Main focus of our paper is to present the techniques to continuously monitor monochromatic queries. However, we show that the proposed techniques can be easily extended to answer bichromatic queries. In rest of the paper, we use RNN query to refer to a monochromatic RNN query unless mentioned otherwise.

2.2 Related Work

2.2.1 Spatial Queries in Euclidean Space

First, we present pruning techniques for *snapshot* RNN queries. Snapshot RNN queries report the results only once and do not require continuous monitoring.

Snapshot RNN Queries: Korn *et al.* [21] were first to study RNN queries. They answer RNN query by precalculating a circle for each data object p such that the nearest neighbor of p lies on the perimeter of the circle. RNN of a query q are the points that contain q in its circle. Techniques to improve their work were proposed in [51, 23].

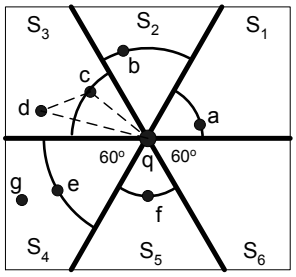


Fig. 2 Pruning based on six-regions

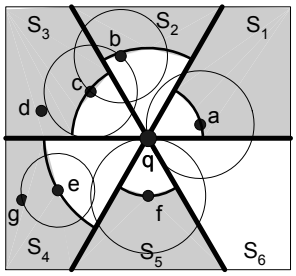


Fig. 3 Filtering and verification

First work that does not need any pre-computation was presented by Stanoi *et al.* [39]. They solve RNN queries by partitioning the whole space centred at the query q into six equal regions of 60° each (S_1 to S_6 in

Fig. 2). It can be proved that only the nearest point to q in each partition can possibly be the RNN. This also means that, in two-dimensional space, there are at most six possible RNNs of a query. Consider the region S_3 where c is the nearest object to q and d cannot be the RNN because its distance to c is smaller than its distance to q. This can be proved by the triangle Δqcd where $\angle dqc \leq 60°$ and $\angle dcq \geq 60°$, hence $dist(d,c) \leq dist(d,q)$. Fig. 3 shows the area (shown shaded) that cannot contain RNN of q.

In *filtering* phase, the candidate RNN objects (a, b, c, e and f in our example) are selected by issuing nearest neighbor queries in each region. In *verification* phase, any candidate object for which q is its nearest neighbor is reported as RNN (a and f). In this paper, we call this approach *six-regions pruning* approach.

Tao *et al.* [42] use the property of perpendicular bisectors to answer RkNN queries. Consider the example of Fig. 4, where a bisector between q and c is shown that divides the space into two half-spaces (the shaded half-space and the white half-space). Any point that lies in the shaded half-space $H_{c:q}$ is always closer to c than to q and cannot be the RNN for this reason. Their algorithm prunes the space by the half-spaces drawn between q and its neighbors in the unpruned region. Fig. 5 shows the example where half-spaces between q and a, c and f ($H_{a:q}$, $H_{c:q}$ and $H_{f:q}$, respectively) are shown and the shaded area is pruned. Then, the candidate objects (a, c and f) are verified as RNNs if q is their closest object. We call this approach *half-space pruning* approach. It is shown in [42] that the half-space pruning is more powerful than the six-regions pruning and it prunes larger area (compare the shaded areas of Fig. 3 and Fig. 5).

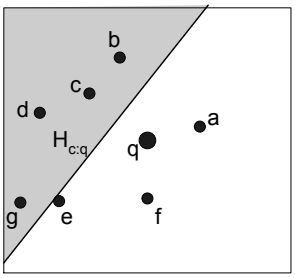


Fig. 4 Pruning based on half-spaces

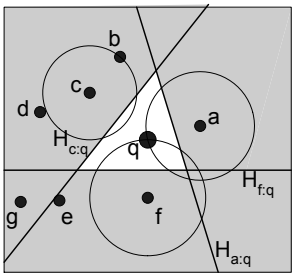


Fig. 5 Filtering and verification

Wu *et al.* [48] propose an algorithm for RkNN queries in $2d$-space. Instead of using bisectors to prune the objects, they use a convex polygon obtained from the intersection of the bisectors. Any object that lies outside the polygon can be pruned.

Cheema *et al.* [5] introduce the concept of *influence zone* to answer both snapshot and continuous RkNN queries. Influence zone of a query q is an area such that every point that is inside it is the RkNN of q and every point that does not lie inside the influence zone is not the RkNN. They present efficient techniques to compute the influence zone and RkNN queries and demonstrate improvement over existing algorithms.

Continuous RNN Queries: Computation-efficient monitoring of continuous range queries [11,22], nearest neighbor queries [25,54,50,16,43] and reverse nearest neighbor queries [1,49,18,47] has received significant attention. Although there exists work on communication-efficient monitoring of range queries [15] and nearest neighbor queries [15,26], there is no prior work that reduces the communication cost for continuous RNN queries. Below, we briefly describe the RNN monitoring algorithms that improve the computation cost.

Benetis *et al.* [1] presented the first continuous RNN monitoring algorithm. However, they assume that velocities of the objects are known. First work that does not assume any knowledge of objects' motion patterns was presented by Xia *et al.* [49]. Their proposed solution is based on the six-regions approach. Consider the examples of Fig. 3, the results of the RNN query may change in any of the following three scenarios:

1. the query or one of the candidate objects changes its location
2. the nearest neighbor of a candidate object is changed (an object enters or leaves the circles shown in Fig. 3)
3. an object moves into the unpruned region (the areas shown in white in Fig. 3)

Xia *et al.* [49] use this observation and propose a solution for continuous RNN queries based on the six-regions approach. They answer RNN queries by monitoring six pie-regions (the white areas in Fig. 3) and the circles around the candidate objects that cover their nearest neighbors.

Kang *et al.* [18] use the concept of half-space pruning and apply the same observation that the results may change in any of three scenarios mentioned above (please see the three scenarios shown above and consider Fig. 5 instead of Fig. 3). They continuously monitor the RNN queries by monitoring the unpruned region (white area in Fig. 5) and the circles around the candidate objects that cover their nearest neighbors. The proposed approach uses a grid structure to store the locations of the objects and queries. They mark the cells of the grid that lie or overlap with the area to be monitored. Any object movement in these cells triggers the update of the results.

Wu *et al.* [47] are the first to propose a solution for continuous monitoring of RkNN which is similar to the six-regions based RNN monitoring presented in [49]. Wu *et al.* [47] issue k nearest neighbor (kNN) queries in each region instead of single nearest neighbor queries. The kNNs in each region are the candidate objects and they are verified if q is one of their k closest objects. To monitor the results, for each candidate object, they

continuously monitor the circle around it that contains k nearest neighbors.

As mentioned earlier, Cheema *et al.* [5] use the concept of influence zone to answer RkNN queries. Their approach can also be used to answer continuous queries. However, they focus on continuous bichromatic RkNN queries where only the data objects move (the query objects do not change their locations).

Note that the problem of RkNN queries is different from all-nearest neighbor queries [8] where nearest neighbors of *every* object in a given dataset is to be found from another dataset. It is important to mention that several safe region based approaches have been proposed for continuous kNN queries [38,31,55,28,13, 15] and continuous range queries [55,15,3] in Euclidean space and continuous range queries in spatial networks [4]. However, these techniques are not applicable for RkNN queries.

2.2.2 Spatial Queries in Spatial Networks

Significant research attention has been given to developing techniques for spatial queries in spatial networks. The shortest path queries [33,34,9], k-NN queries [17, 10,20,19,30,27,35,36] and range queries [40,30,24,46] are among the most studied queries in spatial networks.

To the best of our knowledge, Safar *et al.* [32] are the first to study the snapshot RNN queries in spatial networks. They use Network Voronoi Diagram (NVD) [29] to efficiently process the RNN queries in spatial networks. A Network Voronoi Diagram (NVD) is similar to a Euclidean space Voronoi Diagram in the sense that every point in each Voronoi cell is closer to the generator point of the cell than any other point. However, a NVD considers minimum network distances instead of Euclidean distances between the points. More specifically, a Voronoi cell in a NVD is the set of nodes and edges that are closer to the generator point (in terms of minimum network distance) than any other point. Safar *et al.* [32] use the properties of NVD to efficiently process the RNN queries in network. In a following work [45], they extend their technique to answer RkNN queries and reverse k furthest neighbor queries in spatial network. Please note that their technique cannot be extended to answer continuous RNN queries because the NVD changes as the locations of underlying points change. It is computationally expensive to update NVD whenever the underlying dataset changes.

Sun *et al.* [41] study the continuous monitoring of RNN queries in spatial networks. The main idea is that for each query a multi-way tree is created that helps in defining the monitoring region. Only the updates in the monitoring region affect the results. Their approach is only applicable for the bichromatic RNN queries. Moreover, the proposed approach assumes that the query points do not move. The extension to the case when the query point is also moving is either non-trivial or inefficient because the multi-way trees may be changed as the query points move.

To the best of our knowledge, we are the first to present a technique for continuously monitoring RkNN queries (monochromatic and bichromatic) in spatial networks where both the objects and queries continuously change their locations.

2.3 Motivation

First, we briefly describe limitations of existing techniques that monitor RNNs in Euclidean space. Both the six-regions [49] and the half-space [18] based solutions have two major limitations.

1. As illustrated in the three scenarios presented in Section 2.2.1, the existing techniques are sensitive to object movement. If a query or any of its candidate objects changes its location, filtering phase is called again which is computationally expensive. For example, if a query is continuously moving, at each timestamp both of the approaches will have to compute the results from scratch. For example, in the half-space based approach, the half-spaces between q and its previous candidates are redrawn and the pruning area is adjusted. In our initial experiments, we find that the cost of redrawing the half-spaces (and marking and unmarking the relevant cells) is computationally almost as expensive as the initial computation of the results.

2. The previous techniques require every object to report its exact location to the server at every timestamp regardless whether it affects the query result or not. This has a direct impact on the two most precious resources in mobile environment, wireless communication cost and power consumption. Ideally, only the objects that affect the query results should report their locations. For example, in Fig. 5, as long as objects d, e and g do not enter into the white region or the three circles, they do not affect the results of the query.

Motivated by these, we present a framework that provides a computation and communication efficient solution. Note that, in some applications, the clients may have to periodically report their locations to the server for other types of queries. In this case, saving the communication cost is not possible. Nevertheless, our framework significantly reduces the computation costs for such applications[3].

[3] In rest of the paper, we present our technique assuming that the clients send their locations only for the RkNN query. For the case when the clients periodically send their locations for other types of queries, our techniques can be easily applied. The only change is that the safe regions are stored on the server which ignores the location updates from the objects that are still in

3 Query Processing in Euclidean Space

In this section, we present our technique to continuously monitor RNN queries in spatial networks. In Section 3.1, we present the framework of our proposed technique. A set of novel pruning techniques is presented in Section 3.2. Our continuous RNN monitoring algorithm is presented in Section 3.3. In Section 3.4, we present a detailed theoretical analysis to analyse the computation and communication cost of our proposed algorithms. We present the extensions of our approach to monitor other variants of RNN queries in Section 3.5.

3.1 Framework

Each moving object and query is assigned a safe region of a rectangular shape. Although other simple shapes (e.g., circles) could be used as safe regions, we choose the safe region of a rectangular shape mainly because defining effective pruning rules is easier for the rectangular safe regions. The clients may use their motion patterns to assign themselves better safe regions. However, we assume that such information is not utilized by the clients or the server because we do not assume any knowledge about the motion pattern of the objects.

In our framework, the server recommends the side lengths of the safe regions (a system parameter) to the clients. When a client leaves its safe region, the client assigns itself a new safe region such that it lies at the center of the safe region and reports this safe region to the server.

An object reports its location to the server only when it moves out of its safe region. Such updates issued by the clients (objects) are called *source-initiated* updates [15]. In order to update the results, the server might need to know the exact location of an object that is still in its safe region. The server sends a request to such object and updates the results after receiving its exact location. Such updates are called *server-initiated* updates [15].

If an object stops moving (e.g., a car is parked), it notifies the server and the server reduces its safe region to a point until it starts moving again. Client devices such as GPS can be programmed to notify the server when the device stops moving (e.g., the GPS notifies the server if the car engine is turned off or if the car did not move in last T time units).

In the previous approaches [49,18], the pruned area becomes invalid if the query point changes its location. On the other hand, in our framework, the query is also assigned with a safe region and the pruned area remains valid as long as the query and its candidate objects remain in their respective safe regions and no other object enters in the unpruned region. Although the query is also assigned with a safe region, it reports its location at every timestamp. This is because its location is important to compute the exact results and a server-initiated update would be required (in most of the cases) if it does not report its location itself. Moreover, the number of queries in the system is usually much smaller than the number of objects. Hence, the location updates by the queries do not have significant effect on the total communication cost.

Table 1 defines the notations used throughout this section.

Notation	Definition
$B_{x:q}$	a perpendicular bisector between point x and q
$H_{x:q}$	a half-space defined by $B_{x:q}$ containing point x
$H_{q:x}$	a half-space defined by $B_{x:q}$ containing point q
$H_{a:b} \cap H_{c:d}$	intersection of the two half-spaces
$A[i]$	value of a point A in the i^{th} dimension
$maxdist(x, y)$	maximum distance between x and y (each of x and y is either a point or a rectangle)
$mindist(x, y)$	minimum distance between x and y (each of x and y is either a point or a rectangle)
R_{fil}, R_{cnd}, R_q	rectangular region of the filtering object, candidate object and query, respectively
$R_H[i]$	highest coordinate value of a rectangle R in i^{th} dimension
$R_L[i]$	lowest coordinate value of a rectangle R in i^{th} dimension

Table 1 Notations

Like existing work on continuous spatial queries [25, 18,49], we assume that the errors due to the measuring equipments are insignificant and can be ignored. Our continuous monitoring algorithm consists of the following two phases.

Initial computation: When a new query is issued, the server first computes the set of candidate objects by applying pruning rules presented in Section 3.2. This phase is called *filtering* phase. Then, for each candidate object, the server verifies it as RkNN if the query is one of its k closest points. This phase is called *verification* phase.

Continuous monitoring: The server maintains the set of candidate objects throughout the life of a query. Upon receiving location updates, the server updates the candidate set if it is affected by some location updates. Otherwise, the server calls verification module to verify the candidate objects and reports the results.

3.2 Pruning Rules

In this section, we present novel pruning rules for RNN queries that can be applied when locations of the objects are unknown within their rectangular regions. Al-

their safe regions. Experiment results shown in Section 5 show the superiority of our approach for both of the cases.

though the proposed pruning rules work in any multidimensional space, to keep the discussion simple, we focus on two dimensional space in this section. The pruning rules for higher dimensionality are similar and we refer the interested readers to see [7] for details.

We also remark that the proposed pruning rules can be applied on the minimum bounding rectangles of the spatial objects that have irregular shapes (in contrast to the assumption that the spatial objects are points). In Section 3.5, we extend the pruning rules for RkNN queries.

Throughout this section, an object that is used for pruning other objects is called a *filtering* object and the object that is being considered for pruning is called a *candidate* object.

3.2.1 Half-space Pruning

First, we present the challenges in defining this pruning rule by giving an example of a simpler case where the exact location of a filtering object p is known but the exact location of q is not known on a line MN (shown in Fig. 6). Any object x cannot be the RNN of q if $mindist(x, MN) \geq dist(x, p)$ where $mindist(x, MN)$ is the minimum distance of x from the line MN. Hence, the boundary that defines the pruned area consists of every point x that satisfies $mindist(x, MN) = dist(x, p)$. Note that for any point x in the space on the right side of the line L_N, $mindist(x, MN) = dist(x, N)$. Hence, in the space on the right side of the line L_N, the bisector between p and the point N satisfies the equation of the boundary (because for any point x on this bisector $dist(x, N) = dist(x, p)$).

Similarly, on the left side of L_M, the bisector between p and M satisfies the condition. In the area between L_M and L_N, a parabola (shown in Fig. 6) satisfies the equation of the boundary. Hence the shaded area defined by the two half-spaces and the parabola can be pruned. Note that the intersection of half-spaces $H_{p:N}$ and $H_{p:M}$ does not define the area correctly. As shown in Fig. 6 , a point p' lying in this area may be closer to q than to the point p.

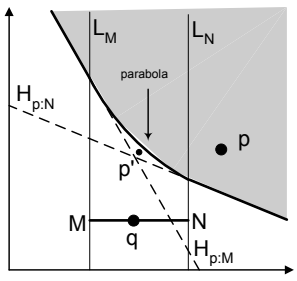


Fig. 6 Exact location of q on line MN is not known

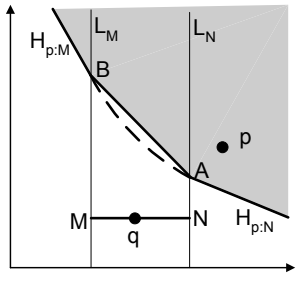


Fig. 7 Approximation of parabola by a line

Unfortunately, the pruning of the shaded area may be expensive due to presence of the parabola. One solution is to approximate the parabola by a line AB where A is the intersection of $H_{p:N}$ and L_N and B is the intersection of $H_{p:M}$ and L_M. Fig. 7 shows the line AB and the pruned area (the shaded area).

Another solution is to move the half-spaces $H_{p:M}$ and $H_{p:N}$ such that both pass through a point c that satisfies $mindist(c, MN) \geq dist(c, p)$ (e.g., any point lying in the shaded area of Fig. 6). This approximation of the pruning area is tighter if the point c lies on the boundary. Fig. 8 shows the half-spaces $H_{p:M}$ and $H_{p:N}$ moved to such point c. A half-space that is moved is called *normalized* half-space and a half-space $H_{p:M}$ that is moved is denoted as $H'_{p:M}$. Fig. 8 shows the normalized half-spaces $H'_{p:M}$ and $H'_{p:N}$ and their intersection can be pruned (the shaded area).

Among the two possible solutions discussed above, we choose normalized half-spaces in developing our pruning rules for the following reason. In our relatively simple example, the number of half-spaces required to prune the area by using the normalized half-spaces is two (in contrast to three lines for the other solution). The difference between this number becomes significant when both the query and the filtering object are represented by rectangles especially in multidimensional space. This makes the pruning by normalized half-spaces a less expensive choice.

Now, we present our pruning rule that defines the pruned area by using at most four half spaces in two dimensional space. This pruning rule uses the normalized half-spaces between 4 selected pairs of corners of the two rectangles to prune the space. We first give a formal description of our pruning rule and then we briefly describe the reason of its correctness. First, we define the following concepts:

Antipodal Corners Let C be a corner of rectangle $R1$ and C' be a corner in $R2$. The two corners are called *antipodal corners* if both of the followings hold: i) if C is a corner on the lower side of $R1$ then C' is a corner on the upper side of $R2$ and vice versa; ii) if C is a corner on the right side of $R1$ then C' is a corner on the left side of $R2$ and vice versa.

For example, a lower-left corner of $R1$ is the antipodal corner of the upper-right corner of $R2$. Similarly, an upper-left corner of $R1$ is the antipodal corner of the lower-right corner of $R2$. Fig. 9 shows two rectangles $R1$ and $R2$. The corners B and M are two antipodal corners. Similarly, other pairs of antipodal corners are (D, O), (C, N) and (A, P).

Antipodal Half-Space A half-space that is defined by the bisector between two antipodal corners is called *antipodal half-space*. Fig. 9 shows two antipodal half-spaces $H_{M:B}$ and $H_{O:D}$.

Higher and lower midpoints. Let $R1$ and $R2$ be two rectangles. Let $R1_L[i]$ denote the lowest coordinate value and $R1_H[i]$ denote the highest coordinate value

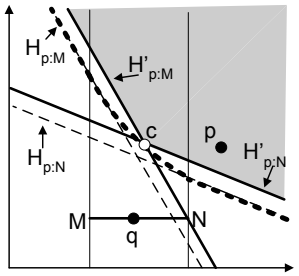


Fig. 8 Defining pruned region by moving half-spaces

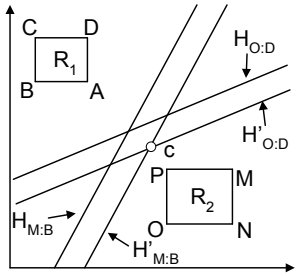


Fig. 9 Antipodal corners and normalized half-spaces

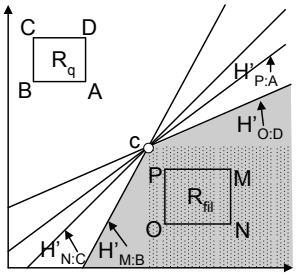


Fig. 10 Half-space pruning and dominance pruning

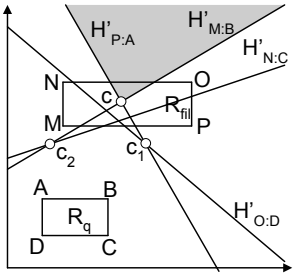


Fig. 11 Any point in shaded area cannot be RNN of q

of $R1$ in i^{th} dimension. The higher midpoint $M_H[i]$ of two rectangles $R1$ and $R2$ in i^{th} dimension is $(R1_H[i] + R2_H[i])/2$. Similarly, the lower midpoint $M_L[i]$ of two rectangles $R1$ and $R2$ in i^{th} dimension is $(R1_L[i] + R2_L[i])/2$.

Assume that for a point P, we denote its x and y coordinate values as $P.x$ and $P.y$, respectively. In the example of Fig. 9, the higher midpoint of $R1$ and $R2$ along x-axis is $(N.x + A.x)/2$ (see $c.x$). Similarly, the lower midpoint along y-axis is $(O.y + A.y)/2$ (see $c.y$).

Normalized Half-Space Let B and M be two points in the rectangles $R1$ and $R2$, respectively. The normalized half-space $H'_{M:B}$ is a half-space defined by the bisector between M and B that passes through a point c such that $c[i] = M_L[i]$ (lower midpoint) for every dimension i for which $B[i] > M[i]$ and $c[j] = M_H[j]$ (higher midpoint) for every dimension j for which $B[j] \leq M[j]$. A normalized antipodal half-space can be represented by a mathematical inequality and we refer the interested readers to [7] for details.

Fig. 9 shows the normalized (antipodal) half-spaces $H'_{M:B}$ which is obtained by moving the half-space $H_{M:B}$ to the point c where $c.x$ is the higher midpoint of the two rectangles along x-axis (because $B.x < M.x$) and $c.y$ is the lower midpoint along y-axis because $B.y > M.y$. Fig. 9 also shows another normalized half-space $H'_{O:D}$ that also passes through the same point c.

PRUNING RULE 1 : Let R_q and R_{fil} be the rectangular regions of the query q and a filtering object p, respectively. For any point p' that lies in $\bigcap_{i=1}^{4} H'_{C_i:C'_i}$, $mindist(p', R_q) > maxdist(p', R_{fil})$ where $H'_{C_i:C'_i}$ is normalized half-space between C_i (the i^{th} corner of the rectangle R_{fil}) and its antipodal corner C'_i in R_q. Hence p' can be pruned.

Fig. 10 shows an example of the half-space pruning where the four normalized antipodal half-spaces define the pruned region (the area shown shaded). The proof of correctness is non-trivial and is given in our technical report (Lemma 5) [6]. Below, we present the intuitive justification of the proof.

Intuitively (as in the example of Fig. 8), if we draw all possible half-spaces between all points of R_q and R_{fil} and move them to a point c for which $mindist(c, R_q) \geq maxdist(c, R_{fil})$, then the intersection of these half-spaces correctly approximates the pruned region. Also note that in two dimensional space, at most two normalized spaces define such area. Consider the example of Fig. 10, where only $H'_{O:D}$ and $H'_{M:B}$ define the pruned region (the reason is that these two have largest and smallest slopes among all other possible half-spaces). In fact, the antipodal corners are defined such that the half-spaces having largest and smallest slopes are among the four antipodal half-spaces. Moreover, the point c shown in Fig. 10 satisfies $mindist(c, R_q) = maxdist(c, R_{fil})$ because normalized half-spaces are defined such that c lies at the middle of the line that joins the corners A and N. Hence the four normalized antipodal half-spaces correctly approximate the pruned region.

For ease of explanation, in Fig. 10, we have shown an example where the two rectangles R_q and R_{fil} do not overlap each other in any dimension. If the two rectangles overlap each other in any dimension (as in Fig. 11), the four half-spaces do not meet at the same point. In Fig. 11, $H'_{O:D}$ and $H'_{P:A}$ are moved to c_1 and $H'_{N:C}$ and $H'_{M:B}$ are moved to point c_2. However, it can be verified by calculating the intersection that the half-spaces that define the pruned region ($H'_{M:B}$ and $H'_{P:A}$) meet at a point c that satisfies $mindist(c, R_q) \geq maxdist(c, R_{fil})$.

3.2.2 Dominance Pruning

We first give the intuition behind this pruning rule. Consider the example of Fig. 10 again. The normalized half-spaces are defined such that if R_{fil} and R_q do not overlap each other in any dimension then all the normalized antipodal half-spaces meet at the same point c. This is because the point c is constructed using either the upper or the lower midpoint in each dimension depending on the x and y coordinate values of the two corners (see the definition of normalized half-spaces and the four normalized half-spaces in Fig. 10).

We also observe that the angle between the half-spaces that define the pruned area (shown in grey) is always greater than 90°. Based on these observations, it can be verified that the space dominated by c (the dotted-shaded area) can be pruned. Formal proof is given in our technical report (Lemma 6) [6].

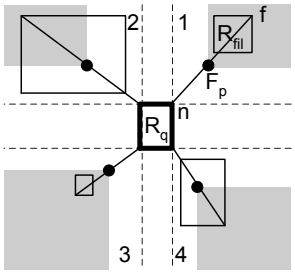


Fig. 12 Shaded areas can be pruned

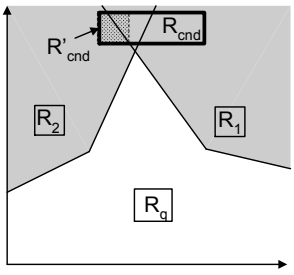


Fig. 13 R_{cnd} can be pruned by R_1 and R_2

Let R_q be the rectangular region of q. We can obtain four regions as shown in Fig. 12. Let R_{fil} be the rectangular region of a filtering object that lies completely in one of the 4 regions. Let f be the furthest corner of R_{fil} from R_q and n be the nearest corner of R_q from f (as shown in region 1 of Fig. 12). A point F_p that lies at the centre of the line joining f and n is called a *frontier point*.

Pruning Rule 2 : Any candidate object p' that is dominated by the frontier point F_p of a filtering object cannot be RNN of q.

Fig. 12 shows four examples of dominance pruning (one in each region). In each partition, the shaded area is dominated by the frontier point of that partition and can be pruned. Note that if R_{fil} overlaps R_q in any dimension, we cannot use this pruning rule because the normalized antipodal half-spaces in this case do not meet at the same point. For example, the four normalized antipodal half-spaces intersect at two points in Fig. 11. In general, the pruning power of this rule is less than that of the half-space pruning. Fig. 10 shows the area pruned by the half-space pruning (the shaded area) and dominance pruning (the dotted area). The main advantage of this pruning rule is that the pruning procedure is computationally more efficient than the half-space pruning, as checking the dominance relationship is easier.

3.2.3 Metric Based Pruning

Pruning Rule 3 : A candidate object can be pruned if $maxdist(R_{cnd}, R_{fil}) < mindist(R_{cnd}, R_q)$ where R_{cnd} is the rectangular region of the candidate object.

This pruning approach is the least expensive because it requires a simple distance comparison. Recall that the half-space (or the dominance) pruning defines a region such that any point p' that lies in it is always closer to the filtering object than to q. Metric based pruning checks this by a simple distance comparison. However, this does not mean that the metric based pruning has at least as much pruning power as half-space or dominance pruning. This is because the half-space and dominance pruning can trim the rectangular region of a candidate object that lies in the pruned region. It may lead to pruning of a candidate object when more than one filtering objects are considered.

Consider the example of Fig. 13, where two rectangles R_1 and R_2 of two filtering objects are shown. The rectangle R_{cnd} cannot be pruned when half-space pruning is applied on R_1 or R_2 alone. However, the rectangle R_{cnd} can be pruned when both R_1 and R_2 are considered. As in [42], we use loose trimming of the rectangle by using trimming algorithm [12]. The trimming algorithm trims a part of the rectangle that cannot be pruned. First, R_{cnd} is pruned by the half-spaces of R_1 and the trimming algorithm trims the rectangle that lies in the pruned region. The unpruned rectangle R'_{cnd} (shown with dotted shaded area) is returned. This remaining rectangle completely lies in the area pruned by R_2 so the candidate object is pruned. Note that metric based pruning cannot prune R_{cnd}.

Also note that if the exact location of a candidate object is known (R_{cnd} is a point) and metric based pruning fails to prune the object then half-space pruning and dominance pruning also fail to prune the object. Hence, half-space pruning and dominance pruning are applied only when the exact location of a candidate object is not known.

3.2.4 Pruning if exact location of query is known

If the exact location of the query or a filtering object is known, previous pruning rules can be applied by reducing the rectangles to points. However, a tighter pruning is possible if the exact location of the query is known. Below, we present a tighter pruning rule for such case.

Pruning Rule 4 : Let R_{fil} be a rectangle and q be a query point. For any point p that lies in $\bigcap_{i=1}^{4} H_{C_i:q}$ (C_i is the i^{th} corner of R_{fil}), $dist(p,q) > maxdist(p, R_{fil})$ and thus p cannot be the RNN of q.

Proof Maximum distance between a rectangle R_{fil} and any point p is the maximum of distances between p and the four corners, i.e., $maxdist(p, R_{fil}) = max(dist(p, C_i))$ where C_i is the i^{th} corner of R_{fil}. Any point p that lies in a half-space $H_{C_i:q}$ satisfies $dist(p,q) > dist(p, C_i)$ for the corner C_i of R_{fil}. Hence a point p lying in $\bigcap_{i=1}^{2^d} H_{C_i:q}$, satisfies $dist(p,q) > maxdist(p, R_{fil})$. □

Consider the example of Fig. 14 that shows the half-spaces between q and the corners of R_{fil}. Any point that lies in the shaded area is closer to every point in rectangle R_{fil} than to q.

It is easy to prove that the pruned area is tight. In other words, any point p' that lies outside the shaded area may possibly be the RNN of q. Fig. 14 shows such point p'. Since it does not lie in $H_{P:q}$ it is closer to q than to the corner P. Hence it may be the RNN of q if the exact location of the filtering object is at corner P.

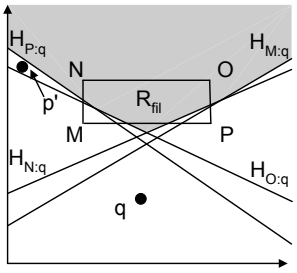


Fig. 14 Half-space pruning when exact location of query is known

3.2.5 Integrating the pruning rules

Algorithm 1 is the implementation of all the pruning rules. Specifically, we apply pruning rules in increasing order of their computational costs (i.e., metric based pruning, dominance pruning and then half-space pruning). While simple pruning rules are not as restricting as more expensive ones, they can quickly discard many non-promising candidate objects and save the overall computational time.

Algorithm 1 : Prune(R_q, S_{fil}, R_{cnd})

Input: R_q: rectangular region of q ; S_{fil}: a set of filtering objects ; R_{cnd}: the rectangular region of candidate object
Output: returns true if R_{cnd} is pruned; otherwise, returns false
Description:

```
1: for each R_fil in S_fil do
2:    if maxdist(R_cnd, R_fil) < mindist(R_q, R_cnd) then    // Pruning rule 3
3:       return true
4:    if mindist(R_cnd, R_fil) > maxdist(R_q, R_cnd) then
5:       S_fil = S_fil − R_fil  // R_fil cannot prune R_cnd
6: if exact location of cnd is known then
7:    return false  // the object cannot be pruned
8: for each R_fil in S_fil do
9:    if R_fil is fully dominated by R_q in a partition P then  // Pruning rule 2
10:      trim the part of R_cnd that is dominated by F_p
11:   return true if R_cnd is pruned
12: return
13: for each R_fil in S_fil do
14:   Trim using half-space pruning  // Pruning rule 1
15:   return true if R_cnd is pruned
16: return false
```

Three subtle optimizations in the algorithm are:

1. As stated in Section 3.2.3, if the exact location of the candidate object is known then only metric based pruning is required. So, we do not consider dominance and half-space pruning for such candidates (line 7).

2. If $mindist(R_{cnd}, R_{fil}) > maxdist(R_q, R_{cnd})$ for a given MBR R_{fil}, then R_{fil} cannot prune any part of R_{cnd}. Hence such R_{fil} is not considered for dominance and half-space pruning (lines 4-5).

3. If the frontier point F_{p_1} of a filtering object R_{fil_1} is dominated by the frontier point F_{p_2} of another filtering object R_{fil_2}, then F_{p_1} can be removed from S_{fil} because the area pruned by F_{p_1} can also be pruned by F_{p_2}. However, note that a frontier point cannot be used to prune its own rectangle. Therefore, before deleting F_{p_1}, we use it to prune the rectangle belonging to F_{p_2}. This optimization reduces the cost of dominance pruning. To maintain the simplicity, we do not show this optimization in Algorithm 1.

3.3 Continuous RNN Monitoring

3.3.1 Data Structure

Our system has an object table and a query table. Object table (query table) stores the id and the rectangular region for each object (query). In addition, the query table stores a set of candidate objects S_{cnd} for each query.

Main-memory computation is the main paradigm in on-line/real-time query processing [25,18,49]. Grid structure is preferred when updates are intensive [25] because complex data structures (e.g., R-tree, Quad-tree) are expensive to update. For this reason, we choose grid-based data structure to store the locations and rectangular regions of moving objects and queries. Each cell contains two lists: 1) *object list*; 2) *influence list*. Object list of a cell c contains object id of every object whose rectangular region overlaps the cell c. This list is used to identify the objects that may be located in this cell. Influence list of a cell c contains query ids of all queries for which this cell lies in (or overlaps with) the unpruned region. The intuition is that if an object moves into this cell, we know that the queries in the influence list of this cell are affected.

Range queries and constrained NN queries (nearest neighbors in constrained region) are issued to compute RNNs of a query (e.g., six constrained nearest neighbor queries are issued in the six-regions based approach). In our algorithm, we also need an algorithm to search the nearby objects in a constrained area (the unpruned region). Several continuous nearest neighbors algorithms [54, 25,50] based on grid-based index have been proposed. However, the extension of these grid-access methods for queries on constrained area becomes inefficient. i.e., the cells around queries are retrieved even if they lie in the pruned region. To efficiently search nearest neighbors in a constrained area, we use *conceptual grid tree* which we introduced in [7] and then further studied in [14].

Fig. 15 shows an example of the conceptual grid-tree of a 4×4 grid. For a grid-based structure containing $2^n \times 2^n$ cells where $n \geq 0$, the root of our conceptual grid-tree is a rectangle that contains all $2^n \times 2^n$ cells. Each entry at l-th level of this grid-tree contains $2^{(n-l)} \times 2^{(n-l)}$ cells (root being at level 0). An entry at l-th level is divided into four equal non-overlapping rectangles such that each such rectangle contains $2^{(n-l-1)} \times 2^{(n-l-1)}$ cells. Any n-th level entry of the tree corresponds to one cell of the grid structure. Fig. 15 shows root entry, intermediate entries and the cells of grid.

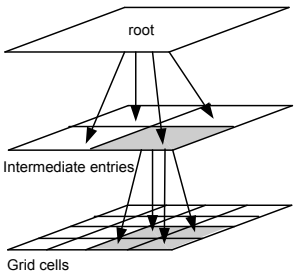


Fig. 15 Conceptual grid-tree of a 4×4 grid

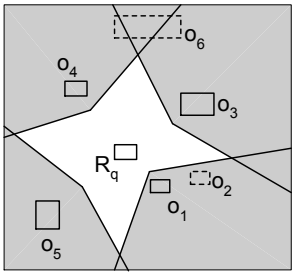


Fig. 16 Illustration of filtering phase

Note that the grid-tree does not exist physically, it is just a conceptual visualisation of the grid.

The spatial queries algorithms that can be applied on R-tree can easily be applied on the conceptual grid tree. The advantage of using this grid-tree over previously used grid-based access methods is that if an intermediate entry of the tree lies in the pruned region, none of the cells inside it are accessed.

3.3.2 Initial Computation

The initial computation consists of two phases namely *filtering* and *verification*. Below we discuss them in detail.

Filtering

In this phase (Algorithm 2), the grid-tree is traversed to select the candidate objects and these objects are stored in S_{cnd}. These candidate objects are also used to prune other objects. Initially, root entry of the grid-tree is inserted in a min-heap H. We try to prune every de-heaped entry e (line 6) by using the pruning rules presented in the previous section. If e is a cell and cannot be pruned, we insert the objects into heap that are in its object list. Otherwise, if e is an intermediate entry of the grid-tree, we insert its four children into the heap H with key $mindist(c, R_q)$. If e is an object and is not pruned, we insert it into S_{cnd}. The algorithm stops when the heap becomes empty.

Algorithm 2 : Filtering

```
1:  for each query q do
2:    S_cnd = φ
3:    Initialize a min-heap H with root entry of Grid-Tree
4:    while H is not empty do
5:      de-heap an entry e
6:      if (not Pruned(R_q, S_cnd, e)) then   // Algorithm 1
7:        if e is a cell in Grid then
8:          for each object o in object list of e do
9:            insert o into H if not already inserted
10:       else if e is an intermediate entry of grid-tree then
11:         for each of its four children c do
12:           insert c into H with key mindist(c, R_q)
13:       else if e is an object then
14:         S_cnd = S_cnd ∪ {e}
```

Fig. 16 shows an example of the filtering phase. For better illustration, the grid is not shown. Objects are numbered in order of their proximity to q. Algorithm iteratively finds the nearest objects and prunes the space accordingly. In the example of Fig. 16, the algorithm first finds o_1 and prunes the space. Since the next closest object o_2 lies in the pruned space, it is not considered and o_3 is selected instead. The algorithm continues and retrieves o_4 and o_5 and the shaded area is pruned. The algorithm stops because there is no other object in the unpruned area (the white area). The rectangles of the pruned objects are shown in broken lines.

One important note is that in this phase, the call to pruning algorithm at line 6 does not consider the exact locations of any object or query for pruning even if the exact location is known. This is because we want to find a set of candidate objects S_{cnd} such that as long as all of them remain in their rectangular regions and no other object enters in the unpruned area, the set of candidate objects is not affected. For example, the set of candidate objects $\{o_1, o_3, o_4, o_5\}$ will not change unless q or any candidate object moves out of its rectangular region or any of the remaining objects (o_2 and o_6) moves in the unpruned area (the white area).

Marking the cells in unpruned area: To quickly identify that an object has moved into the unpruned area of a query q, each cell that lies in the unpruned area is marked. More specifically, q is added in the influence list of such cell. We mark these cells in a hierarchical way by using the grid-tree. For example, if an entry completely lies in the unpruned region, all the cells contained by it are marked. The cells are unmarked similarly.

Verification

At this stage, we have a set of candidate objects S_{cnd} for each query. Now, we proceed to verify the objects. Since every query q reports its location to the server at every timestamp, we can use its location to further refine its S_{cnd}. More specifically, any object $o \in S_{cnd}$ cannot be the RNN of q for which $mindist(o, q) \geq maxdist(o, o')$ for any other $o' \in S_{cnd}$. If the object cannot be pruned by this distance based pruning, we try to prune it by using pruning rule 4. For every query q, its candidate objects that cannot be pruned are stored in a list S_{global}.

The server sends messages to every object in S_{global} for which the exact location is not known. The objects send their exact locations in response. For each query q, the list of candidate objects is further refined by using these exact locations. As noted in [39], at this stage, the number of candidate objects for a query cannot be greater than six in two dimensional space. We verify these candidate objects as follows.

For a candidate object o, we issue a *boolean range query* [37] centered at o with range $dist(o, q)$. In contrast to the conventional range queries, a boolean range query does not return all the objects in the range. It returns true if an object is found within the range, otherwise it returns false. Fig. 17 shows an example, where

Algorithm 3 : Verification

1: Refine S_{cnd} using the exact location of q
2: Request objects in S_{cnd} to send their exact locations
3: Select candidate objects based on exact location of the objects
4: Verify candidate objects (at most six) by issuing boolean range queries

candidate objects are o_1 to o_4. Any object for which its exact location in its rectangular region is not known is shown as a shaded rectangle (see objects o_6, o_7 and o_8). The rectangular regions of the objects for which we know the exact locations are shown in dotted rectangles (see objects o_1 to o_5 and the query q).

The object o_3 cannot be the RNN because o_5 (for which we know the exact location) is found within the range. Similarly, o_4 cannot be the RNN because the rectangular region of o_6 completely lies within the range. The object o_2 is confirmed as RNN because no object is found within the range. The only candidate object for which the result is undecided is o_1 because we do not know the exact location of object o_8 which may or may not lie within the range. The server needs its exact location in order to verify o_1. For each query q, the server collects all such objects. Then, it sends messages to all these objects and verifies all undecided candidate objects upon receiving the exact locations.

3.3.3 Continuous Monitoring

The set of candidate objects S_{cnd} of a query changes only when the query or one of the candidate objects leaves its rectangular region or when any other object enters into the unpruned region. If S_{cnd} is not affected, we simply call the verification phase to update the results. Otherwise, we have to update S_{cnd}.

Consider the running example of Fig. 17 that shows a query q and its four candidates (o_1 to o_4). Assume that after several timestamps, one of the candidate objects (see o_1 in Fig. 18) moves out of its rectangular region. We need to call the filtering phase again because the pruned region is not valid anymore and S_{cnd} may have changed.

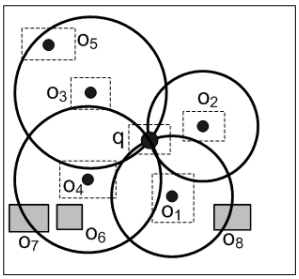


Fig. 17 Illustration of verification phase

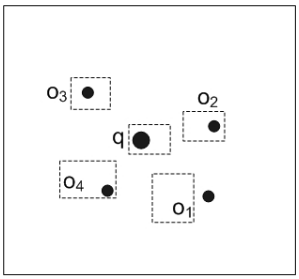


Fig. 18 Continuous monitoring

One possible approach to update S_{cnd} is to call the filtering phase (Algorithm 2) from scratch. Second possible approach to update S_{cnd} is to call Algorithm 2 with S_{cnd} set to $\{o_2, o_3, o_4\}$ instead of initializing an empty S_{cnd}. Note that the object that moves out of its rectangular region (e.g., o_1) has not been considered in S_{cnd}. If it is still the candidate object it will be retrieved during the execution of Algorithm 2. In our initial experiments, we found that the second approach to update S_{cnd} is almost as expensive as the first approach. Below, we show that if we choose to compute S_{cnd} from scratch, we may save computation cost in upcoming timestamps.

Consider the example of Fig. 18 where the candidate object o_1 leaves its rectangular region. Since the query and other candidate objects are also moving, they are likely to leave their regions in next few timestamps which will trigger the expensive filtering phase again. For example, it is possible that the object o_4 leaves its rectangular region in the next timestamp and we have to call the expensive filtering phase again. To overcome this problem, we request all the candidate objects to send their exact locations as well as their new rectangular regions (note that this does not increase the communication cost because in any case we need to contact these candidate objects in the verification phase at line 2 of Algorithm 3). After receiving these new rectangular regions, we update S_{cnd} by calling the filtering phase from scratch. Now the candidate objects have new rectangular regions and they are expected to remain in their respective rectangular regions for longer.

Suppose that an object o is a candidate for two queries q_1 and q_2 and S_{cnd} of q_1 is affected by a location update of any other object o'. We cannot ask o to update its rectangular region because it will affect S_{cnd} of query q_2 as well. Hence, the server only asks an object to update its rectangular region if it does not affect other queries.

3.4 Cost Analysis

In this section, we analyse the computation and communication cost for our proposed solution. First, we present a pruning rule based on six-regions approach and compute the communication cost. Then, we show that the pruning rules used in our technique are superior. Hence the communication cost gives an upper bound. Then, we analyse the computation cost.

Assumptions: We assume that the system contains N objects in a unit space (extent of the space on both dimensions is from 0 to 1). Each rectangular region is a square and width of each side is w. The centers of all rectangular regions are uniformly distributed.

Communication cost: Consider the example of Fig. 19 where a 60° region bounded by the angle $\angle EqC$ is shown in thick lines. Suppose that we find a filtering object whose rectangular region R_{fil} is fully contained in the region. Any object o' can be pruned if $dist(o', q) \geq$

$maxdist(R_{fil}, q)$. In other words, the possible candidates may lie only in the space defined by qEC where EC is an arc and $qC = qE = maxdist(R_{fil}, q)$.

Let r be the distance between q and the center of R_{fil}. Then, $maxdist(R_{fil}, q) \leq r + w/\sqrt{2}$ where $w/\sqrt{2}$ is the half of the diagonal length of R_{fil}. Since, all objects are represented by rectangular regions, any object is possible RNN candidate that has its centre at a distance not greater than $w/\sqrt{2}$ from the region qEC. So, the range becomes $(r + \sqrt{2}w)$. Total number of candidates that overlap or lie within the region qEC is

$$\frac{\pi(r + \sqrt{2}w)^2 N}{6}$$

Let R be the maximum of r of all six regions, the total number of candidate objects is bounded by

$$| S_{cnd} | = \pi(R + \sqrt{2}w)^2 N \quad (1)$$

The server sends request to all these candidate objects and receives their exact locations. So the total number of messages M_1 at this stage is bounded by

$$M_1 = 2\pi(R + \sqrt{2}w)^2 N \quad (2)$$

After receiving the updates, the server eliminates the candidate objects that cannot be the RNN (based on their exact locations). As proved in [39], the number of candidate objects cannot be greater than six. Hence, the server needs to verify those six candidate objects. In order to verify a candidate object o, the server issues a range query of distance $dist(o, q)$ centered at o. In worst case, all the objects that lie within this range must report their exact locations. Total number of objects that overlap or lie within the range is

$$\pi(dist(o, q) + w/\sqrt{2})^2 N$$

Since these candidate objects belong to the nearest neighbors in each region, $dist(o, q)$ corresponds to the distance of closest object in the region. For all six regions, the maximum of $dist(o, q)$ is the distance of sixth nearest neighbor from q (assuming uniform distribution). So the maximum range is the radius of a circle around q that contains six objects. As we assume a unit space, the radius of such circle that contains six objects is $\sqrt{\frac{6}{N\pi}}$. So the maximum number of messages M_2 required to verify all six candidate objects is

$$M_2 = 6 \times 2\pi(\sqrt{\frac{6}{N\pi}} + w/\sqrt{2})^2 N$$

$M_1 + M_2$ are the messages required to retrieve the *server-initiated* updates. Let M_3 be the number of *source-initiated* updates (the objects that leave their rectangular regions). Let v be the average speed of objects. An object starting at center of the square of width w and moving with speed v will take at least $w/2v$ time to leave the region. So, total number of updates M_3 at each timestamp is

$$M_3 = N \times min(\frac{2v}{w}, 1)$$

Note that the equation bounds the number of source-initiated updates by N. The total communication cost per timestamp is $(M_1+M_2+M_3+1)$ where 1 denotes the location update of the query. Note that if w is small, the number of source-initiated updates M_3 increases and if w is large, the number of server-initiated updates $(M_1 + M_2)$ increases.

Now, we find R. Note that to use the pruning of Fig. 19, we had assumed that R_{fil} completely lies in the 60 degree region EqC. Hence r in Equation (1) corresponds to the distance of the closest object in each region that completely lies in it. Similarly, R is the maximum of r of each region.

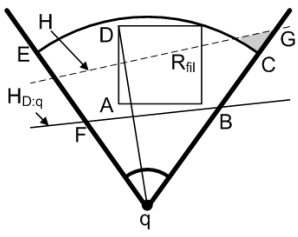


Fig. 19 Half-space pruning vs six-regions based pruning

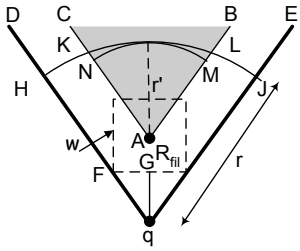


Fig. 20 An object completely lying in the 60° region

Fig. 20 shows a region DqE and a rectangular region R_{fil} of a filtering object (shown in broken line). Note that any rectangular region of side length w with center lying in ABC (the shaded area) will completely lie in the region DqE. In other words, r corresponds to the closest object of q in the region that has center lying in ABC.

Let $r = qH = qJ$ as shown in Fig. 20. Let the radius belonging to area AMN be r'. The radius r' can be computed as $r' = r - qA$ where $qA = qG + GA = qG + w/2$. The length of $qG = 0.866w$ which can be found by the triangle FGq where $FG = w/2$ and $\angle GFq = 60°$. Hence $r' = r - 1.366w$.

It can be verified that when $r = \sqrt{\frac{6}{N\pi}} + 1.366w$, then $\pi(r')^2 N = 6$. In other words when radius is r, one object in each region will be found such that it completely lies in the region. So M_1 can be rewritten as

$$M_1 = 2\pi(\sqrt{\frac{6}{N\pi}} + 2.78w)^2 N$$

The cost $(M_1 + M_2 + M_3 + 1)$ is the cost for one RNN query. The cost of multiple RNN queries is $| Q |$

$\cdot(M_1 + M_2 + 1) + M_3$ where $\mid Q \mid$ is the number of queries.

Now, we show that the area pruned by our proposed approach (pruning rule 4) contains the area pruned by previously described six regions based approach. Consider the example of Fig. 19 where R_{fil} completely lies in the region. The area pruned by six-regions approach is the area of region outside qCE where CE is an arc and $qC = maxdist(R_{fil}, q)$. Our pruning approach prunes the area defined by the intersection of the four half-spaces between q and the corners of R_{fil}. Fig. 19 shows a half-space H (shown in broken line) that crosses the region at a point G such that $qG > qC$. This half-space fails to prune some area pruned by the six region based approach (the six region based approach prunes the shaded area which this half-space H fails to prune).

In order to prove that our pruning approach always contains the area pruned by the six-region based approach, we need to show that all four half-spaces between q and the corners of R_{fil} cross the region at a point B such that $qB \leq qC$. Fig. 19 shows a half-space $H_{D:q}$ between corner D and q. Consider the right triangle qAB where $\angle BqA \leq 60°$. The length of qB is $\frac{qA}{cos(\angle BqA)}$. The maximum possible value of qB is $2 \times qA$ when $\angle BqA$ is 60°. Since $2 \times qA = qD$ and $qD \leq qC = maxdist(R_{fil}, q)$, so $qB \leq qC$. Similarly, it can be proved that $qF \leq qE$. Hence all the four half-spaces contain the area pruned by the region based approach.

Computation cost: Let C_{fil} and C_{ver} be the costs of the filtering phase and the verification phase, respectively. The computation cost at each timestamp is $\rho \times C_{fil} + C_{ver}$ where ρ is the probability that at a given timestamp at least one of the following two events happens: i) the query or any of the candidate objects leaves its safe region; ii) any other object enters in the unpruned region of the query.

The verification cost includes using the exact locations of M_1 objects to further refine the set of candidate objects and using boolean range queries to verify the remaining candidate objects (at most six). Let the cost of refining an object be C_{ref} and the cost of a boolean range query be C_{br}, the verification cost is $C_{ver} = M_1 \times C_{ref} + \mid S_{cnd} \mid \times C_{br}$ where $\mid S_{cnd} \mid \leq 6$.

3.5 Extensions

Since our proposed pruning rules can be applied in multidimensional space, the extension of our algorithm to arbitrary dimensionality is straightforward. Below, we present extension of our algorithm to RkNN monitoring.

RkNN Pruning: An object cannot be RkNN of a query if it is pruned by at least k filtering objects. We initialize a counter to zero and trim R_{cnd} by each filtering object. When the whole rectangle is trimmed, the counter is incremented and the original rectangle is restored. We continue this process by trimming with remaining filtering objects. If the counter becomes equal to k, the object is pruned.

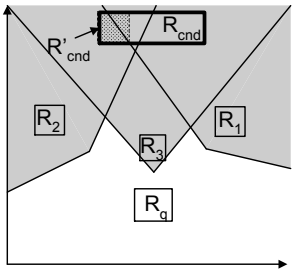


Fig. 21 RkNN Pruning

Suppose k is 2 and consider the example of Fig. 21 where R_{cnd} and three filtering objects R_1, R_2 and R_3 are shown. Filtering objects are considered in order R_1, R_2 and R_3. R_{cnd} is trimmed to R'_{cnd} when R_1 is used for pruning. R'_{cnd} is completely pruned by R_2. The counter is incremented to one and the original rectangle R_{cnd} is restored. Now, R_{cnd} is trimmed by R_3 and the counter is incremented to two because whole rectangle is trimmed. The algorithm prunes R_{cnd} because it has been pruned two times.

Note that if the filtering objects are processed in order R_1, R_3 and R_2, the candidate object cannot be pruned. Finding the optimal order is difficult and trying all possible orders is computationally expensive. This will make filtering of this candidate object more expensive than its verification. Hence, if a candidate object is not pruned by the above mentioned pruning, we consider it for verification.

RkNN Verification: An object o cannot be RkNN if the range query centered at o with range $dist(o, q)$ contains greater than or equal to k objects. Otherwise, the object is reported as RkNN. Suppose k is 2 and consider the example of Fig. 17 again. The candidate objects o_2 and o_3 are confirmed as R2NNs because there are less than 2 objects within their ranges. The object o_1 is also confirmed because at most one object (o_5) lies within the range. The result for o_4 is undecided, so the location of o_7 is requested. Note that we do not need to request the exact location of o_6.

Bichromatic Queries: Now, we briefly present the extension of our proposed solution to bichromatic queries. Let there be two sets of objects O and P and query q belongs to O. The area is pruned by iteratively finding nearby filtering objects that belong to O and lie in the unpruned region. The pruning of area is stopped when there is no filtering object in the unpruned region. The objects of type P that lie in the unpruned region are the candidate objects. The server asks these candidate objects to report their exact locations. Upon receiving

the exact locations, any candidate object p is reported as RNN if there does not lie an object of type O within a circle with radius $dist(p, q)$ centered at p. If the result is undecided, type O objects that have rectangles overlapping with the circles are requested to send their locations. Based on these received locations, the result is computed and reported to the client.

4 Query Processing in Spatial Networks

In this section, we present our technique to continuously monitor RNN queries in Euclidean space. First we introduce basic concepts and notations in Section 4.1. In Section 4.2, we study the problem characteristics. Section 4.3 presents the framework of our technique. Filtering and verification techniques are presented in Section 4.4 and Section 4.5, respectively. We present the extensions of our RNN monitoring algorithm to other variants of RNN queries in Section 4.7.

4.1 Terminology

First we define few terms and notations.

Spatial network G is a weighted graph consisting of *vertices* and *edges*. An edge between two vertices v_1 and v_2 is denoted as $e(v_1, v_2)$. Each edge has a positive weight that denotes the cost of travelling on that edge (e.g., length of the edge, time taken to travel along the edge etc.). The weight of an edge $e(v_1, v_2)$ is denoted as $|e(v_1, v_2)|$.

Segment $s_{[x,y]}$ is the part of an edge between x and y where both x and y are points on the edge. By definition, an edge is also a segment defined by the end points (vertices) of the edge. The weight of a segment $s_{[x,y]}$ is denoted as $|s_{[x,y]}|$.

Fig. 22 shows an example of a road network with eight vertices (a to h). Six objects (o_1 to o_5 and q) are also shown. The query object q is shown as a black star. Several segments are also shown. For instance, the edge $e(b, g)$ consists of segments $s_{[b,o_5]}$, $s_{[o_5,o_4]}$, $s_{[o_4,m]}$, $s_{[m,o_3]}$ and $s_{[o_3,g]}$. The weights of edges and segments are also shown. For example, the weight of the edge $e(c, g)$ is 5 and the weight of the edge $e(b, g)$ is 2+4+2+2+2 = 12.

Shortest network distance $SNDist(x, y)$ between any two points x and y is the minimum network distance between x and y (i.e., total weight of the edges on the shortest path from x to y). In Fig. 22, the shortest path from q to o_4 is $q \rightarrow c \rightarrow g \rightarrow o_4$ and $SNDist(q, o_4)$ is 14.

In Section 2.1, we had formally defined the RNN queries based on the distance function $dist()$. In spatial networks, the RNN query uses the distance function such that it returns the shortest network distance between the points (i.e, $dist(x, y) = SNDist(x, y)$).

4.2 Problem Characteristics

In this section, we study the problem characteristics. The lemma below identifies the objects that cannot be the RNN of a query q.

Lemma 1 *An object o cannot be the RNN of q if the shortest path between q and o contains any other object o'.*

Proof If an object o' lies on the shortest path between q and o, this implies that $SNDist(o, o') < SNDist(o, q)$. Hence o is not the RNN of q. □

In Fig. 22, the object o_4 is not the RNN of q because the shortest path from q to o_4 is $q \rightarrow c \rightarrow g \rightarrow o_4$ which contains another object o_3.

Before we present next lemma, we define *dead* vertices. A vertex v is called a dead vertex if there exists an object o such that $SNDist(v, o) < SNDist(v, q)$. The object o is called the killer object of v because this is the object that makes the vertex v a dead vertex. In Fig. 22, the vertex g is a dead vertex and o_3 is its killer object. The vertex a is not a dead vertex. Note that a dead vertex may have more than one killer objects. For example, o_3, o_4 and o_1 are the killer objects of the vertex g.

Lemma 2 *An object o cannot be the RNN of q if the shortest path between q and o contains a dead vertex v with a killer object o' where $o' \neq o$.*

Proof Assume that a dead vertex v exists on the shortest path between q and o. The shortest network distance between o and q is $SNDist(o, q) = SNDist(o, v) + SNDist(v, q)$. Let o' be the killer object of vertex v. The shortest network distance between o and o' is $SNDist(o, o') \leq SNDist(o, v) + SNDist(v, o')$. By definition of a dead vertex v, $SNDist(v, o') < SNDist(v, q)$. Hence, $SNDist(o, o') < (SNDist(o, v) + SNDist(v, q) = SNDist(o, q))$. Hence, o cannot be the RNN of q. □

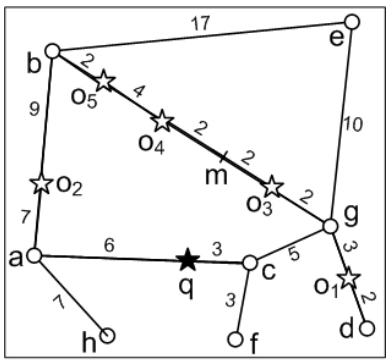


Fig. 22 RNN query in a spatial network

In Fig. 22, the shortest path from q to o_3 contains a dead vertex g with a killer object o_1. The object o_3 is

not the RNN of the query q because $SNDist(o_1, o_3) < SNDist(q, o_3)$. Similarly, the object o_1 is also not the RNN because the shortest path between q and o_1 contains the dead vertex g with a killer object o_3.

Lemma 3 *A vertex v' is a dead vertex if the shortest path between q and v' contains a dead vertex v.*

Proof Let o be the killer object of the vertex v. Then, $SNDist(v', o) \leq SNDist(v', v)+SNDist(v, o)$. By definition of a dead vertex v, $SNDist(v, o) < SNDist(v, q)$. Hence, $SNDist(v', o) \leq (SNDist(v', v)+SNDist(v, q)$ $= SNDist(v', q))$. Hence v' is a dead vertex. □

In Fig. 22, the shortest path from q to e is $q \rightarrow c \rightarrow g \rightarrow e$ which contains a dead vertex g. Hence, e is also a dead vertex.

Lemma 4 *Consider an edge $e(v_1, v_2)$ that contains at least two objects on it and assume that the query q does not lie on it. The edge cannot contain any RNN if $SNDist(v_1, q) \geq |e(v_1, v_2)|$ and $SNDist(v_2, q) \geq |e(v_1, v_2)|$ where $|e(v_1, v_2)|$ is the weight of the edge.*

Proof $|e(v_1, v_2)| \geq SNDist(o, o')$ for any two objects o and o' that lie on the edge $e(v_1, v_2)$. For any object o on the edge $e(v_1, v_2)$, the shortest path between o and q either passes through v_1 or v_2. Hence, $SNDist(o, q) \geq min(SNDist(v_1, q), SNDist(v_2, q)) \geq |e(v_1, v_2)| \geq SNDist(o, o')$. Hence, o cannot be the RNN of q. □

Before we present next lemma, we define extreme objects of an edge. An object o is called an extreme object of an edge $e(v_1, v_2)$ if either the segment $s_{[o,v_1]}$ or the segment $s_{[o,v_2]}$ does not contain any other object o'. In Fig. 22, the objects o_3 is an extreme object of the edge $e(b, g)$ because the segment $s_{[o_3,g]}$ does not contain any other object. Similarly, the object o_5 is also an extreme object because the segment $s_{[o_5,b]}$ does not contain any other object. However, the object o_4 is not an extreme object because both the segments $s_{[o_4,b]}$ and $s_{[o_4,g]}$ contain an object other than o_4. By definition of extreme objects, each edge contains at most two extreme objects. This holds true even if more than one objects lie at the same location. For instance, in Fig. 22, if there was an object o' at the same location as of o_3 then both o_3 and o' would not be the extreme objects.

Lemma 5 *Only the extreme objects of an edge can be the RNN of a query q given that q does not lie on the edge.*

Proof Let o be an object on the edge $e(v_1, v_2)$ and o be not an extreme object. Since q does not lie on the edge $e(v_1, v_2)$, the shortest path between o and q either passes through v_1 or v_2. Since o is not an extreme object, each of the segment $s_{[o,v_1]}$ and $s_{[o,v_2]}$ contains at least one object other than o. Hence, the shortest path from o to q contains at least one other object and o cannot be the RNN of q as implied by Lemma 1. □

In Fig. 22, the object o_4 cannot be the RNN of q because it is not an extreme object.

Lemma 6 *Regardless of the number of queries in the system, an edge that does not contain any query has at most two objects that can be the RNNs of any of the queries.*

Proof From Lemma 5, only the extreme objects can be the RNN of a query q. Since each edge contains at most two extreme objects, only at most two objects can be the RNNs of any of the queries. □

In Fig. 22, assume that the object o_2 is also a query point. Only the extreme objects (o_3 and o_5) of the edge $e(b, g)$ can be the RNNs of the query points q and o_2. Lemma 5 and Lemma 6 imply that the extreme objects of an edge are the only possible candidate objects for the queries that do not lie on the edge. Moreover, several queries may share same candidate objects.

Based on the problem characteristics we studied in this section, we develop an algorithm to continuously monitor RNN queries. The next section presents the framework of our proposed technique.

4.3 Framework

To simplify the presentation, we assume that the safe regions of the objects and queries are segments. Later in Section 4.6, we show that our technique can support the safe regions that consist of more than one edges and segments.

Each object and query is assigned a segment that is its safe region. The safe region of an object o is denoted as $o.s_{[x,y]}$. Since the safe region of an object is a segment, we use $o.x$ and $o.y$ to denote the end points of this segment. Each object and query reports its location to the server whenever it leaves its safe region. Such updates are called *source-initiated* updates. In order to update the results, the server might need to know the exact locations of some objects. The server receives the exact location of each such object by requesting its current location. Such updates are called *server-initiated* updates.

The safe region of a query q is chosen such that $q.s_{[x,y]}$ does not overlap with the safe region of any other object. The segment $q.s_{[x,y]}$ is considered as an edge and the end points $q.x$ and $q.y$ are considered as vertices. This is to simplify the presentation because Lemma 5 and Lemma 6 are applicable to every edge if the segment that contains the query is considered as a different edge.

The continuous monitoring algorithm consists of two phases.

1. Filtering. In filtering phase, the set of candidate objects are retrieved by pruning the objects that cannot

be the RNN of a query q. The edges and segments of the network that cannot contain any RNN are also pruned. The part of the network that is not pruned is called unpruned network. The set of candidate objects remain valid unless at least one of the following happens: i) the query or a candidate object leaves its safe region; ii) an object enters in the unpruned network. Hence, the filtering phase is called only when at least one of the above two happens.

2. Verification. In verification phase, for each candidate object, the server checks if the candidate object is the RNN of q or not. More specifically, if q is the closest object of o (in terms of $SNDist$), the object o is reported as the RNN. The verification phase is called at each timestamp.

In the following, we present the details of both the filtering and verification phases.

4.4 Filtering

The main idea is to incrementally expand the network around the query in a way similar to Dijkstra's algorithm. More specifically, the vertices are accessed in increasing order of their $SNDist$ from q (a min-heap is used). Whenever a vertex v is de-heaped, its adjacent vertices are inserted in the heap if v is not a dead vertex. Lemma 1 and Lemma 2 are used to identify the candidate objects lying on the adjacent edges of v. The algorithm stops when the heap becomes empty.

Algorithm 4 : Filtering

```
1:  S_cnd = φ
2:  initialize a min-heap H
3:  insert q.x and q.y with keys set to zero
4:  while H is not empty do
5:     deheap a vertex v from H and mark it as visited
6:     for each unvisited adjacent vertex v' of v do
7:        if there exists at least one object on e(v, v') then
8:           get the object o closest to v
9:           Assign o a safe region o.s_[x,y] and insert o in S_cnd
10:          d = max(|s_[v,o.x]|, |s_[v,o.y]|)
11:          if d < v.key then
12:             mark v as dead; break;
13:    if v is not marked dead then
14:       for each unvisited adjacent vertex v' of v do
15:          if e(v, v') does not contain any object then
16:             if v' is not present in the heap H then
17:                insert v' in H with key v'.key = v.key + |e(v, v')|
18:             else
19:                v'.key = min(v'.key, v.key + |e(v, v')|)
```

Algorithm 4 presents the details. The set of candidate objects is S_{cnd} and is initialized to an empty set. Let $q.s_{[x,y]}$ be the safe region of the query. As mentioned earlier, the end points of the safe regions of the queries are treated as the vertices. A min-heap is initialized and $q.x$ and $q.y$ are inserted with keys set to zero. The entries from the heap are retrieved iteratively (line 5). When a vertex v is de-heaped, we consider its adjacent vertices iteratively (line 6). Let v' be an adjacent vertex of v. We obtain an object o that lies on the edge $e(v, v')$ and is closest to v and assign it a safe region $o.s_{[x,y]}$ (lines 7 to 9).

Based on the safe region of the object o, we determine if the vertex v is a dead vertex or not (lines 10 to 12). Recall that a vertex v is a dead vertex if its $SNDist$ from an object o is smaller than its $SNDist$ from the query q. Since, we have assigned safe regions to both the query object q and the data object o, we need to make sure that a vertex v is marked dead only if it satisfies the condition regardless of the location of q and o in their respective safe regions. In other words, a vertex v is marked dead if its maximum $SNDist$ from the safe region of o is less than its minimum $SNDist$ from the safe region of q. The maximum $SNDist$ of the safe region of o from v is the maximum of the weights of the segments $s_{[v,o.x]}$ and $s_{[v,o.y]}$ where $o.x$ and $o.y$ are the end points of the safe region of o (line 10). To be more precise, this gives an upper bound on the maximum $SNDist$ between v and the safe region of o. The upper bound on the minimum distance of v from the safe region of q is the key value $v.key$ of v (the value with which it was inserted in heap).

We use d to denote the maximum $SNDist$ between v and the safe region of o (line 10). The value of d is compared with the key $v.key$ of the vertex v. If $v.key$ is greater than d, the vertex is marked as dead (lines 11 and 12). Please note that the vertex v will remain dead as long as both the query object and the object o remain in their respective safe regions.

If the vertex v is marked dead, we do not need to consider other adjacent vertices and the objects on the adjacent edges (Lemmas 1, 2 and 3). If the vertex v is not a dead vertex, then each of its adjacent vertex v' that has not been visited earlier is considered (line 14). If the edge $e(v, v')$ contains at least one object, the vertex v' is ignored (line 15). This is because if the shortest path of v' from q passes through v, the vertex v' is a dead vertex because an object o exists on the shortest path. If the edge $e(v, v')$ does not contain any object and v' is not present in the heap then v' is inserted in the heap with key set to $v.key + |e(v, v')|$ (line 17). On the other hand, if v' is already present in the heap then its key is updated to $v.key + |e(v, v')|$ if $v.key + |e(v, v')|$ is less than its existing key (line 19). The algorithm stops when the heap becomes empty.

The edges and segments that are explored during the execution of the algorithm form the unpruned network. Fig. 23 shows an example of filtering phase called for the query q. The unpruned network is shown in thick lines. The objects o_1, o_2 and o_3 are the candidate objects.

Proof of correctness can be obtained by proving that the algorithm shortlists every object that may possibly be the RNN of q (by applying Lemma 1 and Lemma 2).

We omit the details of the proof. However, it is important to mention that the key $v.key$ of a vertex v may not necessarily be the shortest network distance of v from the safe region of q because the dead vertices are not inserted in the heap. However, this does not affect the correctness of the algorithm because if the shortest path between an object and the query passes through a dead vertex, the object cannot be the RNN. Hence, if $v.key$ is not the shortest network distance between v and the safe region of q then this implies that v is a dead vertex (Lemma 3) and we do not miss any possible RNN of q.

4.5 Verification

An object o is the RNN of q if and only if there does not exist any other object o' such that $SNDist(o, o') < SNDist(o, q)$. If there exists such an object o', the object o is not the RNN and we say that the object o' invalidates the candidate object o.

A straight forward approach to check if a candidate object o is the RNN is to issue a *boolean range query* on the spatial network with range set to $SNDist(o, q)$. A conventional range query returns every object o' for which $SNDist(o, o')$ is less than a given range r. In contrast to conventional range queries, a boolean range query returns true if there exists at least one object o' for which $SNDist(o, o')$ is less than r otherwise it returns false. To check if an object o is the RNN of q, a boolean range query can be issued with range set to $SNDist(o, q)$. If the boolean range query returns true, the object is not the RNN. Otherwise o is reported as the RNN.

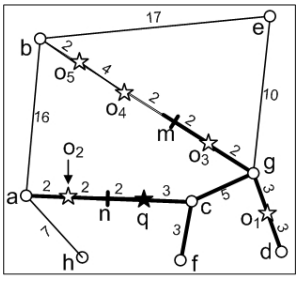


Fig. 23 Illustration of filtering phase

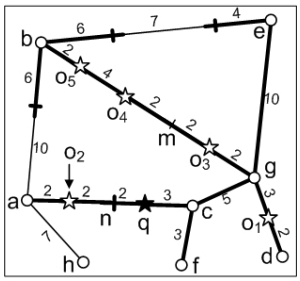


Fig. 24 Computing monitored network

Next, we show that some candidate objects may be verified without issuing a boolean range query.

As implied by Lemma 4, a candidate object o that lies on an edge $e(v_1, v_2)$ that contains at least one other object o' and does not contain q cannot be the RNN of q if $|e(v_1, v_2)| \leq SNdist(v_1, q)$ and $|e(v_1, v_2)| \leq SNDist(v_2, q)$. Hence, we compare $|e(v_1, v_2)|$ with the shortest network distances of v_1 and v_2 from the safe region of q and if the edge satisfies the above conditions then the object o does not require verification. For each such candidate object o, we keep a counter that records the number of objects on the edge $e(v_1, v_2)$ and we verify the object o only if the counter is equal to one (i.e., o is the only object on this edge).

There may be several candidates that cannot be verified by using the strategy presented above. One possible way to verify such a candidate object is to issue a boolean range query. Note that the cost of the verification phase may dominate the cost of the filtering phase if a boolean range query is issued for each candidate object. Since verification is to be called at each timestamp regardless of the underlying data movement, the safe region based approach may not improve the performance significantly if the verification phase is expensive.

Next, we present a technique based on the concept of *monitored network*. Once the monitored network for an object o is computed, the verification becomes computationally cheap. We show that the monitored network of a candidate object does not require to be recomputed at every timestamp. In fact, the monitored network of a candidate object remains valid as long as the unpruned network (obtained during the filtering phase) remains valid. In other words, the monitored network is required to be computed only when the filtering phase is called.

Let o be an unverified candidate object. Let $o.s_{[x,y]}$ and $q.s_{[x,y]}$ denote the safe regions of the object o and the query q, respectively. We use $MaxSNDist(o, q)$ to denote the maximum $SNDist$ between the safe regions of o and q (i.e., maximum $SNDist$ between any two points a and b where a is a point in $o.s_{[x,y]}$ and b is a point in $q.s_{[x,y]}$). In the example of Fig. 24, the safe regions of o_3 and q are $s_{[m,g]}$ and $s_{[n,c]}$, respectively. The $MaxSNDist(o_3, q)$ is 14 (i.e., the shortest network distance between m and n).

Monitored network of an object o is the part of the network such that for every point p that does not lie on it, minimum $SNDist$ between p and the safe region of o is greater than $MaxSNDist(o, q)$.

Fig. 24 shows the monitored network of object o_3 (shown in thick lines) and it consists of every point of the network that has minimum $SNDist$ from the safe region of o_3 at most 14. Intuitively, the monitored network is defined as the network such that, for any object o' that does not lie on it, $SNDist(o, o') > SNDist(o, q)$ regardless of the locations of o and q in their respective safe regions. Hence, only the objects that lie on the monitored network are required to be considered in order to verify the candidate object o.

Note that once the monitored network is computed it remains valid as long as q and o remain in their respective safe regions. Hence, the recomputation of the monitored network is not required unless at least one of q and o does not leave its safe region. Recall that if q or o leaves its safe region, the filtering phase is required to be called again. Hence, the monitored network remains

valid as long as the filtering phase is not required to be called again.

To compute the monitored network, we use an algorithm similar to Dijkstra's algorithm and Algorithm 4 and expand the network starting from the safe region of o until we visit every vertex v such that minimum $SNDist$ of v from the safe region of o is at most equal to $MaxSNDist(o, q)$.

To enable efficient computations of $SNDist$ between o and other objects in the monitored network, we maintain the minimum $SNDist$ of each explored vertex from the safe region of o. In addition, we also maintain the list of objects that lie on the monitored network of o. An object o' notifies the server when it enters or leaves the monitored network of the object o and the list of the objects that lie on the monitored network of o is updated accordingly.

Optimizations. We present two optimizations that can help in terminating the computation of the monitored network earlier.

1. During the computation of the monitored network, we do not need to expand the network beyond a vertex v if the shortest path between v and the safe region of o contains the query q. This is because any object o' that lies beyond the vertex v satisfies $SNDist(o, o') > SNDist(o, q)$. For example, in Fig. 24, an object that lies on segment $s_{[q,a]}$ (e.g., o_2) cannot help in verifying the object o_3 because the shortest path from the safe region of o_3 to the segment $s_{[q,a]}$ contains the query q. Hence, we do not need to include this segment in the monitored network.

2. Similar to the definition of $MaxSNDist(o, q)$, we define $MinSNDist(o, q)$ as the minimum $SNDist(o, q)$ between the safe regions of o and q. In the example of Fig. 24, $MinSNDist(o_3, q)$ is 5 and $MaxSNDist(o, q)$ is 14 where the safe regions of o_3 and q are $s_{[m,g]}$ and $s_{[n,c]}$, respectively.

Note that if there exists an object o' such that it satisfies $MaxSNDist(o, o') < MinSNDist(o, q)$ then the object o cannot be the RNN of q as long as the objects o and o' and the query q remain in their respective safe regions. We utilize this observation and expand the monitored network such that it covers every point p such that $SNDist(o, p) < MaxSNDist(o, o')$. Here o' corresponds to the object that has smallest $MaxSNDist(o, o')$ among all the objects discovered so far. For instance, in Fig. 24, $MaxSNDist(o_3, o_4)$ is 12 (the safe region of o_4 is $s_{[b,m]}$) and $MaxSNDist(o_3, o_1)$ is 9. Hence, during the computation of the monitored network, we may stop expanding the network when the expanded network contains every point p that has minimum $SNDist$ from the safe region of o_3 at most 9. To verify the object o, we compute $SNDist$ between o and every other object (including the query) that lies on the expanded network.

Note that if we use the optimization presented above, the computed monitored network does not need to be updated as long as o, o' and q remain in their respective safe regions. If o' is also a candidate object (e.g., $o' = o_3$ in Fig. 24), the monitored network remains valid as long as the filtering phase is not called again. Otherwise, when the object o' moves out of its safe region the monitored network is recomputed to guarantee the correctness.

4.6 Safe regions consisting of more than one edges

In this section, we show that our proposed algorithm can support the safe regions consisting of more than one edges and segments. Assume that the safe region of an object o consists of more than one edges and segments. We denote its safe region $o.s$ by its end points (i.e., boundary points). Consider the example of Fig. 23 and assume that the safe region of the object o_3 is shown in thick lines. The end points of the safe region of o_3 are m, g, d, f and a.

The safe region of a query q is always chosen such that it does not overlap with the safe region of any other object. The algorithm 4 is modified as follows. At line 3, all the end points of the safe region of q are inserted in the heap with keys set to zero. Moreover, the vertices that lie inside the safe region of q are marked as visited so that they are not considered during the network expansion. Note that as the key of every end point of $q.s$ is set to zero, the key of a de-heaped vertex v denotes minimum $SNDist$ from the safe region of q to the vertex v. At line 10, d is set as the maximum distance between v and the safe region of o. Note that these changes guarantee that a vertex v remains dead as long as q and o remain inside their respective safe regions.

The rest of the filtering algorithm does not require any changes. The techniques and optimizations we presented in the verification phase can be immediately applied and do not require any change.

4.7 Extensions

4.7.1 Queries on directed graphs

In the previous section, our main focus was on the RNN queries in the spatial networks that are represented by undirected graphs. Our proposed techniques can be easily extended for the directed graphs. Below, we highlight the changes we need to make to extend our techniques for the RNN queries on directed graphs.

1. $SNDist(x, y)$ is defined as the total weights of the edges and the segments on the shortest path *from* the point x *to* the point y.

2. Lemma 1 and Lemma 2 are the same except that we use the shortest path *from o to q* instead of the shortest path between q and o. The definition of the dead vertex remains unchanged.
3. Lemma 3 is the same except that we use shortest path *from* v' *to* q instead of shortest path between q and v'.
4. Lemma 4 is not applicable whereas Lemma 5 and Lemma 6 remain unchanged.
5. Filtering phase (Algorithm 4) is similar except that we expand the network from any vertex v to v' (see line 6) only if there is a directed edge from the vertex v' to v. Please note that we expand the network in the direction opposite to the direction of edges. This is because the lemmas are applicable on the path from an object o to q and not on the path from q to o.
6. The verification phase remains same and we compute the monitored network by expanding the network in the direction of the edges.

4.7.2 RkNN queries

We briefly discuss the necessary changes that are required to extend our proposed techniques for RkNN queries.
1. Lemma 1 is restated as follows. An object o cannot be the RkNN of q if the shortest path from o to q contains *at least* k other objects.
2. A dead vertex v is redefined as a vertex v for which there exist at least k objects such that for every such object o, $SNDist(v, o) < SNDist(v, q)$. After redefining the dead vertices, Lemma 2 and Lemma 3 do not require any modification.
3. Lemma 4 is the same except that it is applicable only on the edges that contain *at least* $(k + 1)$ objects on it. Recall that Lemma 4 is only applicable for RkNN queries on undirected graphs. For the RkNN queries on the directed graphs, we do not consider Lemma 4.
4. Extreme objects are redefined. An object o is called an extreme object of an edge $e(v_1, v_2)$ if either the segment $s_{[o,v_1]}$ or the segment $s_{[o,v_2]}$ contains *at most* $k-1$ other objects. Lemma 5 holds after the extreme objects are redefined as above.
5. Lemma 6 is restated as follows. Regardless of the number of queries in the system. An edge that does not contain any query has at most $2k$ objects that can be the RkNNs of any of the queries.
6. Filtering phase is similar except that we mark the vertices as dead according to the redefined definition of the dead vertices.
7. Verification phase is similar except that an object o is reported as RkNN iff there are at most $k-1$ other objects closer to o than q. Computation of the monitored network remains unchanged.

4.7.3 Bichromatic queries

Let P and O be the two sets of objects and assume that the query q belongs to O. In the filtering phase, only the objects of type O are considered to prune the network. The objects of type O that are discovered during the filtering phase are called *filtering* objects. The objects of type P that lie on the unpruned network are called the candidate objects. The set of candidate objects remain valid unless at least one of the following three happens: i) the query leaves its safe region; ii) one of the filtering objects leaves its safe region; iii) one of type P objects enters or leaves the unpruned network. In the case when one of the first two events happens, the filtering and verification phases are called. If only the third event happens, we do not need to call the filtering phase again because the unpruned network is not affected by the movement of type P objects. Instead, we update the set of candidate objects by adding the objects of type P that enter the unpruned network and removing the type P objects that leave the unpruned network.

5 Experiment Results

All the experiments were conducted on Intel Xeon 2.4 GHz dual CPU with 4 GBytes memory. All the algorithms (including the competitors) were implemented in C++. Our algorithm is called SAC (**S**wift **A**nd **C**heap) due to its computational efficiency and communication cost saving.

As discussed in Section 2.3, there may be some applications where the objects have to report their locations to the server for other types of queries like range queries, nearest neighbor queries etc. In such case, the server is responsible for checking whether an object lies in the safe region or not. In order to show the superiority of our technique in all kinds of applications, the computation costs shown in the experiments include the cost of checking whether each object lies in its safe region or not. Obviously, the computation cost would be less for the case when the clients report their locations only when they leave their safe regions.

In Section 5.1, we evaluate the performance of our Euclidean space algorithm. The performance of our spatial network algorithm is evaluated in Section 5.2.

5.1 Query Processing in Euclidean Space

For RNN queries ($k = 1$), we compare our algorithm with state-of-the-art algorithm (IGERN) [18] which has been shown superior in [18] to other RNN monitoring algorithms [49,47]. For RkNN queries ($k > 1$), we compare our algorithm with CRkNN [47] which is the

only available RkNN monitoring algorithm. In accordance with work in [18] and [47], we choose 64×64 grid structure for IGERN and 100×100 grid structure for CRkNN. For our algorithm, the grid cardinality is 64×64.

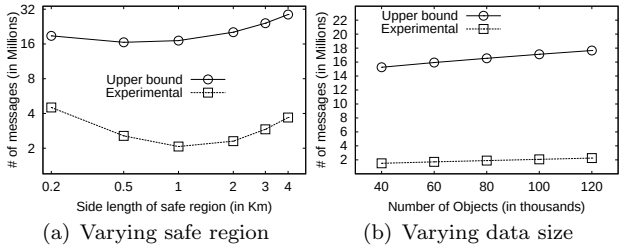
Similar to previous work, we simulated moving cars by using the spatio-temporal data generator [2]. Input to the generator is road map of Texas[4] and output is a set of cars (objects and queries) moving on the roads. The size of data universe is 1000 $Km \times 1000$ Km. The parameters of datasets are shown in Table 2 and default values are shown in bold.

Parameter	Range
Number of objects (×1000)	40, 60, 80, **100**, 120
Number of queries	100, 300, **500**, 700, 1000
Average speed (in Km/hr)	40, 60, **80**, 100, 120
Side length of safe region (in Km)	0.2, 0.5, **1**, 2, 3, 4
Mobility (%)	5, 20, 40, 60, **80**, 100

Table 2 System parameters for experiments in Euclidean space

The server reports the results continuously after every one second (i.e., the timestamp length is 1 sec). Both the objects and queries are cars moving on roads and they have similar properties (e.g., average speed, mobility). Mobility refers to the percentage of objects and queries that are moving at any timestamp (percentage of objects and queries that change their locations between two consecutive timestamps). All queries are continuously monitored for five minutes (300 timestamps) and the results shown correspond to total CPU time and communication cost. Communication cost is the total number of messages sent between clients and server.

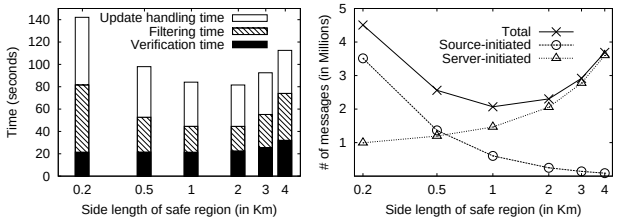
In Fig. 25, we conduct experiments to verify the cost analysis presented in Section 3.4. The experiments show that the actual cost is around 12% to 25% of the upper bound. We also observe that the actual results follow a trend similar to the trend anticipated by our theoretical analysis (e.g., the cost increases if the safe region is too small or too large).



(a) Varying safe region (b) Varying data size

Fig. 25 Verifying theoretical upper bound

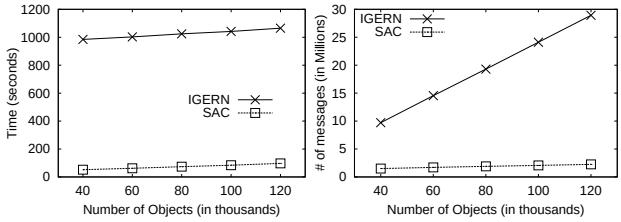
Fig. 26(a) shows the effect of the safe region size on computation time our algorithm and IGERN [18]. The computation cost consists of update handling cost, filtering cost and verification cost. The update handling cost includes the cost of checking whether an object/query is in its safe region or not and updating the underlying grid structure if the object/query leaves the safe region. If the safe region is too small, the set of candidate objects is affected frequently and the filtering is required more often. Hence, the cost of the filtering phase increases. On the other hand, if the safe region is too large, the number of candidate objects increases and the verification of these candidates consumes more computation time. Also, the cost of filtering phase increases because less space can be pruned if the safe region is large. The update handling cost is larger for smaller safe regions because the objects and queries leave the safe regions more frequently.



(a) Computation time (b) Communication cost

Fig. 26 Effect of safe region size

Fig. 26(b) studies the effect of safe region size on communication cost. As studied in Section 3.4, the number of source-initiated updates increases if the side length of the safe region is small. On the other hand, if the safe region is large, the number of server-initiated updates increases. Fig. 26(b) verifies this. In current experiment settings, our algorithm performs best when the side length of the safe region is 1Km so we choose this value for the remaining experiments.



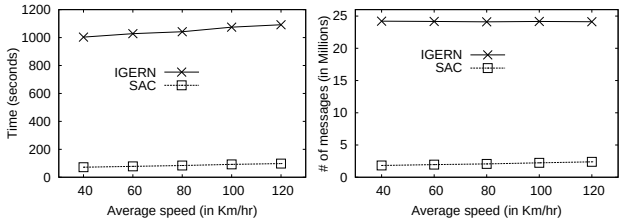
(a) Computation time (b) Communication cost

Fig. 27 Effect of datasize

Fig. 27 shows the effect of the number of objects. Our algorithm not only outperforms IGERN but also scales better. The composition of CPU time is not shown due to the huge difference in the performance of both algorithms. However, the composition of CPU time is similar to Fig. 26(a) for our algorithm. For IGERN, the filtering phase takes 95% to 99% of the total cost in all experiments. This is because the expensive filtering phase is called frequently.

Fig. 28 studies the effect of the average speed of queries and objects. Fig. 28(a) shows that the computation time increases for both of the approaches as the speed increases. For our approach, the time increases

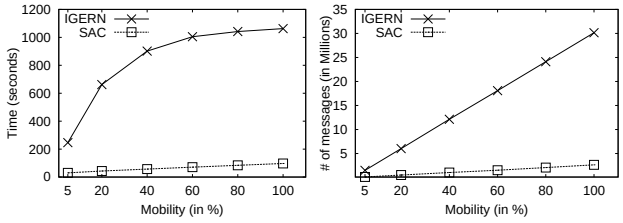
[4] http://www.census.gov/geo/www/tiger/



(a) Computation time (b) Communication cost

Fig. 28 Effect of Speed

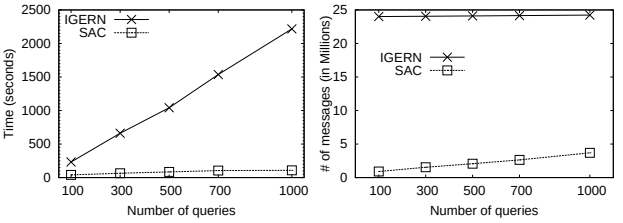
because the objects and queries leave their respective safe regions more frequently and the filtering phase is called more often. Fig. 28(b) shows that IGERN requires an order of magnitude more messages than our approach. The communication cost for our approach increases due to the larger number of source-initiated updates as the speed increases.



(a) Computation time (b) Communication cost

Fig. 29 Effect of data mobility

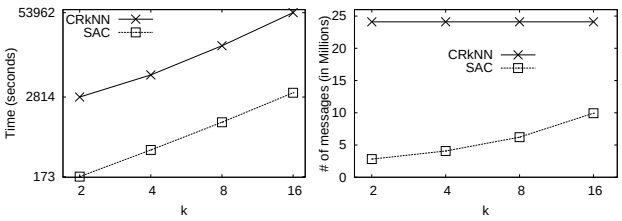
Fig. 29(a) compares the computation time for increasing data mobility. As expected, IGERN performs good when the object mobility is low (e.g., 5%). However, its computation cost increases significantly as the object mobility increases. Our algorithm performs better for all cases and scales decently. Fig. 29(b) studies the effect of objects and queries mobility on the communication cost. Since only the moving objects report their locations, the number of messages increase with the increase in mobility. However, our algorithm consistently gives improvement of more than an order of magnitude compared to IGERN.



(a) Computation time (b) Communication cost

Fig. 30 Effect of number of queries

Fig. 30 studies the effect of number of queries. In Fig. 30(a), we note that our algorithm gives more than an order of magnitude improvement over IGERN in terms of CPU time and scales better. In accordance with the analysis in Section 3.4, Fig. 30(b) show that the communication cost of our approach increases with the number of queries.



(a) Computation time (b) Communication cost

Fig. 31 Effect of k

Fig. 31 studies the effect of k on communication and computation time. Fig. 31(a) compares our approach with [47] referred as CRkNN. Computation cost of both approaches increases with increase in k. However, our algorithm scales better (note the log scale). CRkNN continuously monitors $6k$ range queries to verify the candidate objects. To monitor these queries, it keeps a counter for the number of objects leaving and entering within the range. However, this information becomes useless when the candidate object or query changes its location. As shown in Fig. 31(b), communication cost for our approach increases for larger values of k. This is mainly because the number of candidate objects that require verification increases with k. Communication cost of our algorithm reaches to 24 million when $k = 64$ (CPU time 23,000 sec). We were unable to run CRkNN for $k > 16$ due to its large main-memory requirement.

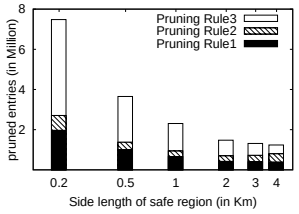


Fig. 32 Effectiveness of pruning rules

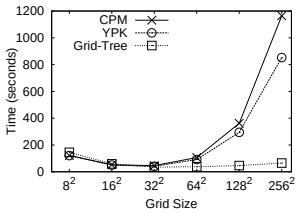


Fig. 33 Effectiveness of grid-tree

Fig. 32 shows the effectiveness of pruning rules for different safe region sizes. Pruning rules are applied in the same order as in Algorithm 1. If a pruning rule fails to prune an entry (an object or a node of the grid-tree), the next pruning rule is used to prune it. Fig. 32 shows that a greater number of entries are pruned if the safe region size is small. Majority of the entries are pruned by the metric based pruning (pruning rule 3) when the safe regions are small. The average time to prune an entry by metric based pruning, dominance pruning and half-space pruning is 1.1, 2.3 and 10.5 micro seconds, respectively.

Now, we show the effectiveness of grid-tree over previous proposed grid access methods CPM [25] and YPK [54]. Fig. 33 shows the total CPU time for our RNN monitoring algorithm when the underlying constrained nearest neighbor algorithm (and marking and unmarking of cells) use CPM, YPK and grid-tree. We change the grid size from 8×8 to 256×256. Grid-tree based

RNN monitoring algorithm scales much better with increase in number of cells.

5.2 Query Processing in Spatial Networks

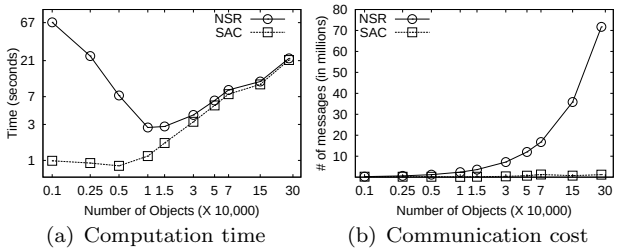
To the best of our knowledge, we are the first to propose an algorithm to continuously monitor RNN queries in spatial networks for the case where both the queries and data objects continuously change their locations. We compare our algorithm (SAC) with a naïve algorithm. The safe regions used by SAC consist of at most one edge. Naïve algorithm recomputes the results at every timestamp by applying our algorithm and setting the safe region size to zero (i.e., safe region is not used). We choose a better competitor of our algorithm and call it NSR (No Safe Region). NSR is the same as naïve algorithm except that it calls the filtering phase only when the query object or one of the candidate objects changes its location. As obvious, the naïve algorithm performs worse than NSR. Hence, we compare our algorithm with NSR.

We use the road network of California[5] that consists of around 22,380 road segments (edges). Each object in the data set randomly picks a vertex and starts moving towards it with a certain speed (a system parameter). When the object reaches at its destination vertex, it randomly chooses one of its adjacent vertices and continues travelling towards it. The queries are generated similarly. Table 3 shows the default parameters used in our experiments.

Parameter	Range
Number of objects (×1000)	1, 2.5, **5**, 10, 15, 30, 50, 70, 150, 300
Number of queries	25, **100**, 150, 250, 500, 700, 1000
Average speed (in Km/hr)	40, 60, **80**, 100, 120
Mobility (%)	5, 20, 40, 60, **80**, 100

Table 3 System Parameters for experiments in road network

Each query is continuously monitored for 300 timestamps (five minutes) and the results shown correspond to the total CPU time and total communication cost. The communication cost corresponds to the number of messages sent between the server and clients.



(a) Computation time (b) Communication cost

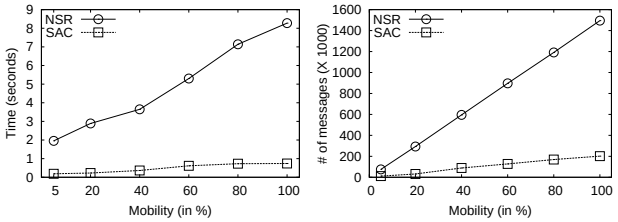
Fig. 34 Effect of datasize

[5] http://www.cs.fsu.edu/ lifeifei/SpatialDataset.htm

In Fig. 34, we study the effect of number of objects on the road network. Fig. 34(a) shows the effect on the computation time of both the algorithms (note that log-scale is used). Interestingly, the performance of both the algorithms is poor when the number of objects is too small or too large. When the number of objects is large, the performance becomes poor mainly because updates of more objects are needed to be handled. Since the dominant cost is handling these location updates, both of the algorithms perform similar when the number of objects is large.

When the number of objects is small, greater number of edges are to be explored for filtering and verification phases which results in greater computation time. Note that if each edge contains several objects, the RNN queries can be answered by visiting at most one or two edges. Hence, it would be more interesting to compare the performance of the algorithms where the density of objects (number of objects per edge) is low. For this reason, we choose 5000 objects for the rest of the experiments.

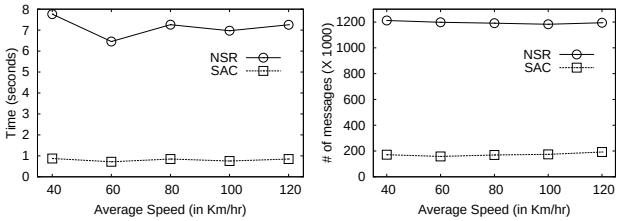
Fig. 34(b) shows the trend that the communication costs of both algorithms increase with the increase in number of objects (log scale is used for x-axis). However, the safe region based algorithm SAC scales much better than NSR.



(a) Computation time (b) Communication cost

Fig. 35 Effect of data mobility

Fig. 35 studies the effect of data mobility on both of the algorithms. As expected, both algorithms perform worse as the data mobility increases. However, SAC scales much better than NSR both in terms of computation time and communication cost.

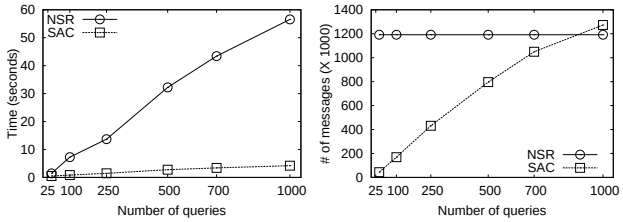


(a) Computation time (b) Communication cost

Fig. 36 Effect of speed

Fig. 36 studies the effect of speed of the objects and queries on the algorithms. The experiments demonstrate that the performance of the proposed technique is not significantly affected by the speed. Although the objects

and queries leave their safe regions more frequently as the speed increases, the communication cost is not significantly affected. This is because the total communication cost is dominated by the server initiated updates (e.g., when the server requests the objects to send their exact locations in order to verify if a candidate is the RNN or not). The number of server initiated updates does not depend on the speed hence the total communication cost is not significantly affected by the speed.



(a) Computation time (b) Communication cost

Fig. 37 Effect of number of queries

In Fig. 37, we change the number of queries and show the effect on the performances of both the algorithms. The computation times for both of the algorithms increase as the number of queries increases but SAC scales much better. The communication cost of NSR does not depend on the number of queries because each object reports its location whenever it changes its location. On the other hand, the communication cost of SAC increases mainly because more objects are required to be verified if the number of queries is large. To verify more objects, greater number of server initiated updates are required and this results in increased communication cost. Fig. 37(b) shows that the communication cost of SAC is more than the cost of NSR when a large proportion of the data objects are also the query objects (e.g., 1000 queries among 5000 objects).

We remark that in the worst case the communication cost of SAC can be at most two times the cost of NSR. This is because, for each object, at most two messages are to be sent (one to request the location and one to receive the server). Nevertheless, in the applications where the proportion of queries is large, the clients (data objects) may be configured to send their locations at every timestamp. The communication cost in this case would be the same as the cost of NSR.

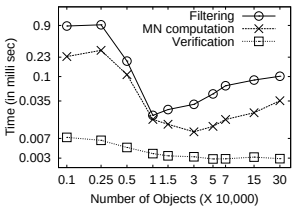


Fig. 38 Costs of different phases

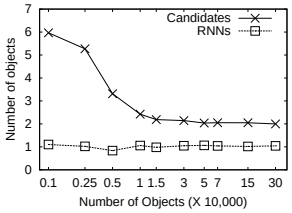


Fig. 39 Average number of candidates

Fig. 38 shows the average time taken by a call to filtering phase, a call to compute the monitored network (shown as MN computation) and a call to verify the objects (after the monitored network has been computed). As expected, the cost of filtering and computing the monitored network is high if the number of objects is too large or too small. The cost of verification increases when the data size is too small. This is mainly because the number of candidate objects increases when the data size is small. The next experiment confirms this trend.

In Fig. 39, we study the effect of data size on the number of candidate objects and the number of RNNs. We observe that the number of candidate objects is large when the data size is small. This is because the algorithm needs to explore more edges during the filtering phase and the pruning power decreases.

6 Conclusion

In this paper, we studied the problem of continuous reverse k nearest neighbor monitoring in Euclidean space and in spatial networks. Our proposed approach not only significantly improves the computation time but also reduces the communication cost for client-server architectures. We also present a thorough theoretical analysis for our Euclidean space RNN algorithm. Furthermore, we show that our algorithms can be extended to handle other variants of RNN queries in Euclidean space and in spatial networks. Experiment results demonstrate an order of magnitude improvement in terms of both the computation time and the communication cost. The theoretical analysis for our spatial network RNN algorithm and the automatic computation of the optimal size of the safe regions remain two open problems.

Acknowledgements We would like to thank the editors and the anonymous reviewers for their very helpful comments that have significantly improved this paper. The research of Xuemin Lin was partially supported by ARCDP110102937, ARCDP0987557, ARCDP0881035, NSFC61021004, and NICTA.

References

1. R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, pages 44–53, 2002.
2. T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
3. M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In *ICDE*, pages 189–200, 2010.
4. M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Continuous monitoring of distance based range queries. *TKDE*, 2011.
5. M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. *ICDE*, 2011.

6. M. A. Cheema, X. Lin, Y. Zhang, and W. Wang. Lazy updates: An efficient technique to continuously monitoring reverse knn. In *UNSW Technical Report, 2009. Available at ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0905.pdf.*
7. M. A. Cheema, X. Lin, Y. Zhang, W. Wang, and W. Zhang. Lazy updates: An efficient technique to continuously monitoring reverse knn. *PVLDB*, 2(1):1138–1149, 2009.
8. Y. Chen and J. M. Patel. Efficient evaluation of all-nearest-neighbor queries. In *ICDE*, 2007.
9. Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. In *SIGMOD Conference*, pages 591–602, 2009.
10. H.-J. Cho and C.-W. Chung. An efficient and scalable approach to cnn queries in a road network. In *VLDB*, pages 865–876, 2005.
11. B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.
12. J. Goldstein, R. Ramakrishnan, U. Shaft, and J.-B. Yu. Processing queries by linear constraints. In *PODS*, 1997.
13. M. Hasan, M. A. Cheema, X. Lin, and Y. Zhang. Efficient construction of safe regions for moving knn queries over dynamic datasets. In *SSTD*, pages 373–379, 2009.
14. M. Hasan, M. A. Cheema, W. Qu, and X. Lin. Efficient algorithms to monitor continuous constrained nearest neighbor queries. In *DASFAA (1)*, pages 233–249, 2010.
15. H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD Conference*, pages 479–490, 2005.
16. G. S. Iwerks, H. Samet, and K. P. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, pages 512–523, 2003.
17. C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *GIS*, pages 1–8, 2003.
18. J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *ICDE*, 2007.
19. M. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, pages 840–851, 2004.
20. M. R. Kolahdouzan and C. Shahabi. Continuous k-nearest neighbor queries in spatial network databases. In *STDBM*, pages 33–40, 2004.
21. F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, 2000.
22. I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. In *EDBT*, pages 269–286, 2002.
23. K.-I. Lin, M. Nolen, and C. Yang. Applying bulk insertion techniques for dynamic reverse nearest neighbor problems. *ideas*, 00:290, 2003.
24. F. Liu, T. T. Do, and K. A. Hua. Dynamic range query in spatial network environments. In *DEXA*, pages 254–265, 2006.
25. K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.
26. K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *TKDE*, pages 1451–1464, 2005.
27. K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, pages 43–54, 2006.
28. S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB*, 1(1):1095–1106, 2008.
29. A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* Wiley, 1999.
30. D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.
31. S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Computers*, 51(10):1124–1140, 2002.
32. M. Safar, D. Ebrahimi, and D. Taniar. Voronoi-based reverse nearest neighbor query processing on spatial networks. *Multimedia Syst.*, 15(5):295–308, 2009.
33. H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD Conference*, pages 43–54, 2008.
34. J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.
35. C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *ACM-GIS*, pages 94–10, 2002.
36. S. Shekhar and J. S. Yoo. Processing in-route nearest neighbor queries: a comparison of alternative approaches. In *GIS*, pages 9–16, 2003.
37. A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High dimensional reverse nearest neighbor queries. In *CIKM*, 2003.
38. Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, pages 79–96, 2001.
39. I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.
40. D. Stojanovic, A. N. Papadopoulos, B. Predic, S. Djordjevic-Kajan, and A. Nanopoulos. Continuous range monitoring of mobile objects in road networks. *Data Knowl. Eng.*, 64(1):77–100, 2008.
41. H.-L. Sun, C. Jiang, J.-L. Liu, and L. Sun. Continuous reverse nearest neighbor queries on moving objects in road networks. In *WAIM*, pages 238–245, 2008.
42. Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, 2004.
43. Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, 2002.
44. Y. Tao, M. L. Yiu, and N. Mamoulis. Reverse nearest neighbor search in metric spaces. *TKDE*, 18(9), 2006.
45. Q. T. Tran, D. Taniar, and M. Safar. Reverse k nearest neighbor and reverse farthest neighbor search on spatial networks. *T. Large-Scale Data- and Knowledge-Centered Systems*, 1:353–372, 2009.
46. H. Wang and R. Zimmermann. Snapshot location-based query processing on moving objects in road networks. In *GIS*, page 50, 2008.
47. W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan. Continuous reverse k-nearest-neighbor monitoring. In *MDM*, 2008.
48. W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan. Finch: Evaluating reverse k-nearest-neighbor queries on location data. In *VLDB*, 2008.
49. T. Xia and D. Zhang. Continuous reverse nearest neighbor monitoring. In *ICDE*, page 77, 2006.
50. X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.
51. C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, 2001.
52. M. L. Yiu and N. Mamoulis. Reverse nearest neighbors search in ad hoc subspaces. *TKDE*, 19(3):412–426, 2007.
53. M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. In *ICDE*, 2005.
54. X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.
55. J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD Conference*, pages 443–454, 2003.