Reverse Approximate Queries in Spatial Databases

by

Xinyu Li

Thesis

Submitted by Xinyu Li

for fulfillment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisor: Assoc. Prof. David Taniar,

Assoc. Prof. Muhammad Aamir Cheema

Faculty of Information Technology
Monash University

September, 2021

Copyright notice

©Xinyu Li (2021)

I certify that I have made all reasonable efforts to secure copyright permissions for third-party content included in this thesis and have not knowingly added copyright content to my work without the owner's permission.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

Print Name: Xinyu Li

Date: 10/05/2021

Acknowledgments

I would like to express my very great appreciation for everyone who helped to make this possible. It has been an incredible journey in the past four years. At first, I am particularly grateful for the assistance given by my main supervisor, Associate Professor David Taniar. He was my supervisor since the final year of my Master degree. He offered me huge help for my minor thesis since the Master time and guided me to apply my PhD in Monash University. Even I finished the minor thesis project, I am still new to research. David is always very patient when answering my basic questions.

I would also like to thank my co-supervisor, Associate Professor Muhammad Aamir Cheema. Aamir is very professional and knowledgeable in terms of research. Many research questions that I have totally no idea about how to start are easy for him and he can guide me to the correct direction. All my papers cannot be published without Aamir. I really appreciate his help.

I would like to express my deep gratitude to Dr. Arif Hidayat for his patient guidance, enthusiastic encouragement and useful critiques in my PhD time.

Special thanks to all the nice people in our research group. It was really enjoyable to have group meetings where we can share our ideas. I have been extremely lucky to be in the same group with Zhou Shao, Bojie Shen, Lingxiao Li and Nasser Allheeib. It was always a pleasure to have a chat with you guys.

I would like to extend my thanks to all the staff of Clayton School of Information Technology at Monash University. All of you have been there to support me whenever I needed any help during my PhD program.

Finally, I wish to thank my parents, my brother and his wife for their support and encouragement throughout my study.

Xinyu Li

Monash University

September 2021

Contents

List of Tables

List of Figures

List of Publications

[1] Xinyu Li, Arif Hidayat, David Taniar, and Muhammad Aamir Cheema. Reverse Approximate Nearest Neighbor Queries on Road Network. *World Wide Web* (2020): 1-18.

[2] Xinyu Li, Arif Hidayat, David Taniar, and Muhammad Aamir Cheema. Continuous Monitoring of Reverse Approximate Nearest Neighbour Queries on Road Network. Submitted to *Information Sciences*.

[3] Xinyu Li, David Taniar, and Muhammad Aamir Cheema. Spatial Reverse Approximate Top queries. Prepare to publish to *International Conference on Web Information System Engineering (WISE)* of 2021.

Reverse Approximate Queries in Spatial Databases

Xinyu Li
xinyu.li@monash.edu
Monash University, 2021

Supervisor: Assoc. Prof. David Taniar,
Assoc. Prof. Muhammad Aamir Cheema
david.taniar@monash.edu,
aamir.cheema@monash.edu

Abstract

In spatial database, there are a plethora of queries focusing on various problems. Reverse approximate queries are one kind of query which shows the influence of the query point. This kind of query is similar to reverse queries. The main difference between reverse queries, such as reverse k nearest neighbor (RkNN) queries or spatial reverse top-k (SRTk) queries, and reverse approximate queries is the value k. Because of the value k, many point of interests who are influenced by query point may not be returned as results. Reverse approximate queries relax the strict requirement of k and use a factor x. With this modification, the influence of the query point becomes more precise. Hence, we focus on the reverse approximate queries, including the reverse approximate nearest neighbour (RANN) queries and the spatial reverse approximate top (SRAT) queries.

RkNN queries, which can retrieve all the objects that consider the query as their k most influential objects, have been studied extensively for many years. Given a set of users U, a set of facilities F and a value k, a facility f is said to be influential to a user u if f is one of the k closest facilities to u. An interesting phenomenon has been found that sometimes even if f is not a k closest facility regarding to u, it is very close to u and should also be included in the results. This shows that RkNN does not return the best results. As a complement to the RkNN query, RANN queries consider a relaxed definition of influence, where a user u is influenced not only by its closest facility, but also by every other facility that is almost as close to u as its closest facility. In this situation, if there is a cluster of facilities near u, they can all influence u. In this thesis, we study the RANN query on road network. Existing RANN techniques and algorithms only work for queries on *Euclidean* space and are not directly applicable for RANN queries on road network. We propose pruning techniques that utilize a *Network Voronoi Diagram* (NVD) to efficiently solve

RANN queries on road network. We conduct extensive experiments on different real-world data sets and demonstrate that our algorithm is significantly better than the competitor.

In addition to the snapshot RANN query, we extend the query to a moving environment, where the facilities remaining static and some users move continuously. Our topic is called continuous reverse approximate nearest neighbor (RANN) query on road network. The state-of-the-art techniques about continuous RANN only focus on *Euclidean* space and cannot be used to perform queries on the road network. We propose two efficient ways to utilize a safe zone and influence zone to monitor moving RANN. Finally, we conduct an extensive experiments on different real data sets and demonstrate that our algorithm is significantly better than the competitor.

Finally, a new query called the spatial reverse approximate top (SRAT) query is proposed, which is a totally new concept on which there is no existing research. For a user u, the top-k facilities are the facilities where the scores are computed using different criteria such as distance from the user, rating, price etc. The facilities, which are in the top-k results of different users, have influence on these users. The spatial reverse top-k query returns all the users that can be influenced by the query point. The SRTk query considers more conditions and is more useful in the real world compared to the RkNN query. As the existence of k value, some unexpected results will inevitably be generated. To avoid unreasonable SRTk result, the spatial reverse approximate top (SRAT) query is introduced. We propose a novel way to process the SRAT query efficiently using R*-tree and a *Voronoi* diagram index. At last, a systematic experimental study that adopts both synthetic and real-world data sets is conducted and the results prove our algorithm is considerably faster than the competitor.

Chapter 1

Introduction

1.1 Overview

With advancements in technology, online map applications have become an essential part of daily life. Many companies such as Google, Apple provide map services or navigation services. Behind the advancement of map applications, various spatial queries play a major role. For example, if a user is traveling to attend a friend's wedding and he wants to have a short break and buy a coffee, using the map application makes it easy to find the nearest coffee shop in relation to his current location and it returns the shortest path to it. So, the user is able to reach his destination with the least cost not only in terms of time. The background mechanism on which a map application relies on to search for a target that has the shortest distance to his position is a kind of spatial query called k nearest neighbor query (kNN) [26, 84, 5, 8]. In terms of the kNN query, it returns k closest point of interests from a user's location. Nevertheless, only retrieving the k nearest targets is not good enough in some complicated situations. For example, the government plans to build a new park in which nearby residents can exercise. The new location requires all adjacent users to consider this park as their nearest one and this park should not have intersecting influence areas with other parks so that more residents can benefit from this plan. Under this circumstance, finding the influential area for the existing park is of great significance. A query that can handle this task is called the reverse k nearest neighbour (RkNN) [106, 57, 81, 52] query.

The RkNN query and its variations [75, 7, 82, 99, 47, 19] have been studied extensively as it can has many applications in decision support, location-based services, profile-based marketing etc. Unlike the traditional kNN queries which find the k nearest facilities or points of interest with respect to users, RkNN changes the query perspective from users to facilities and is more helpful in conducting information analysis. Formally, given a set of users U, a set of facilities F, an integer k and a query facility q, an RkNN query returns every user $u \in U$ which considers q as one of its k closest facilities. Consider an example of a supermarket that is launching a promotional deal as shown in Fig 1.1. The manager of this supermarket wants to know which potential customers are possibly interested in the deal. Assuming that distance is the only factor that affects customers' decisions, the supermarket may issue an RkNN query to find all the potential customers. Potential customers are those who consider the supermarket as one of their k closest supermarkets. Assuming $k = 2$, $R2NN$ of f_1 are u_1, u_2 because f_2 is among the two closest facilities for these users, i.e. $R2NN(f_1) = \{u_1, u_2\}$. Similarly, $R2NN(f_2) = \{u_2\}$, $R2NN(f_3) = \{u_1, u_3, u_4, u_5\}$ and $R2NN(f_4) = \{u_3, u_4, u_5\}$.

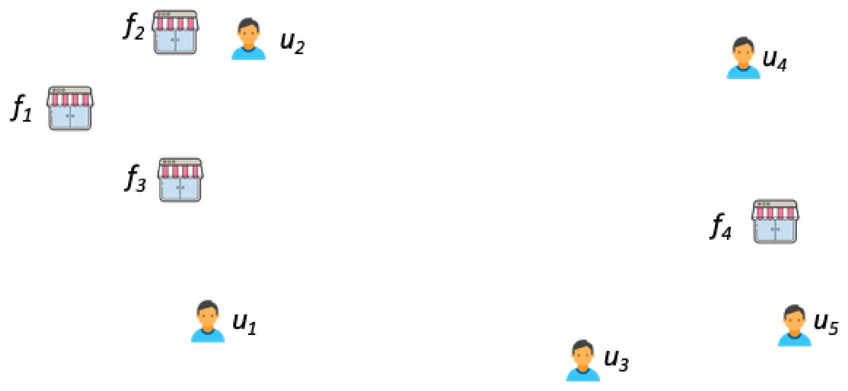


Figure 1.1: Example of RkNN

Even though RkNN is an excellent query and has a plethora of applications, it has some drawbacks. As argued in [37, 38], RkNN has a strict k value to limit the influence of facilities which may cause some users to fall out of the final query outcome set, whereas these users should be influenced by the query point. As shown in Fig. 1.1, the RkNN query may fail to retrieve all potential customers of the supermarket issuing the query. For example, assuming f_1 issues a RkNN query with $k = 2$. Customers u_4 and u_5 are not part of the query answers, since their two closest supermarkets are f_4 and f_3. Hence, based on the RkNN query, u_4 and u_5 are not potential customers of f_1. However, this may be unreasonable. Assume the distance between u_5 to f_4, f_3 and f_1 are 5, 30 and 31 kms respectively. It is believed that u_5 who is travelling 30 kms to reach f_3 normally does not mind travelling 1 or 2 kms further to reach the next closest supermarket (i.e f_1).

Such a customer may not be captured by the RkNN query and thus is not considered as a potential customer of f_1. In such situations, the reverse approximate nearest neighbour (RANN) query can be used as a complement for the RkNN query.

The RANN query was firstly studied on *Euclidean* space [37] (from this point onward, we refer to this as *Euclidean* RANN). This study was extended in [38] to continuously monitor the RANNs of queries. The use of the *Euclidean* RANN query will not be appropriate in a real-life data set, since it uses *Euclidean* distance which corresponds to the straight-line distance between vertices in the graph, which cannot reflect the actual distance between customers and the supermarket. The actual distance between them is reflected by the shortest path distance on the road network. Because of the natural difference between *Euclidean* space and the road network space, a new algorithm which has a better adaptability in the real world such as helping the supermarket to identify its potential customers accurately, needs to be proposed to answer the RANN query. For this reason, we study the RANN query on the road network space.

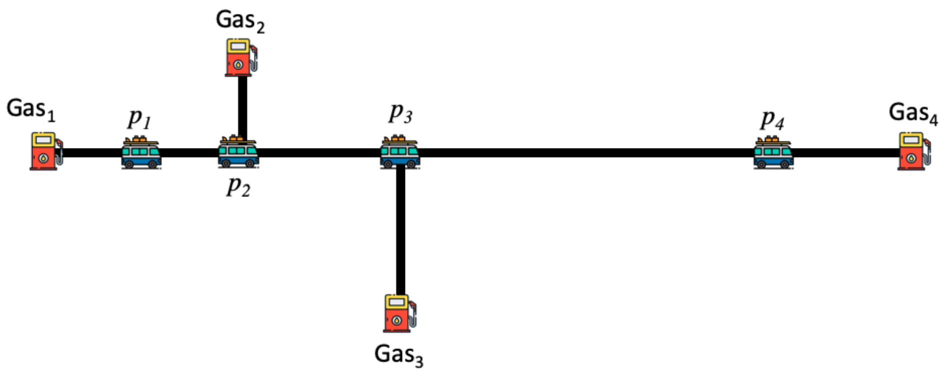


Figure 1.2: Example of Moving queries

With the rapid development of technology, the position locators are becoming cheaper and have been installed in a variety of devices in our daily life, such as mobile phones, cars, computers, etc. With the increased use of position locators, the popularity of location-based services is significantly increasing. Consequently, the algorithms for processing geo-tagged data that support location-based services are receiving significant attention from researchers. Many people extend the spatial query from the stationary environment to the moving environment [41, 105, 93, 94] as the moving queries can have more convenient functional usage in the real world. For instance, a user is allowed to drive with the navigator or cellphone which provides current location information and the distance to destination. All the devices would perform spatial queries continuously

[70], with position data updated at each timestamp. This algorithm usage in practical life is of great importance and meaning.

In real-world applications, the moving queries are monitored continuously when a map or location-based application is running. Consider an example where a person who is traveling to a destination in a car needs to find a gas station, the traveller needs to be continuously informed as to the nearest gas station position and the minimum travel cost. The use of a static query is not helpful since it only gives a snapshot of the results for a particular time when the query is issued. Fig. 1.2 illustrates this situation. In this figure, a driver moves along the road from p_1 to p_4. Four gas stations denoted as GAS_1, GAS_2, GAS_3 and GAS_4 are depicted in the figure. p_1, p_2, p_3 and p_4 show the position of the driver in four different timestamps. At the beginning, when the driver is at p_1 and they issues a nearest neighbor query to find the nearest gas station, the query will return GAS_1. After the driver moves to the next position p_2, its nearest gas station becomes GAS_2. With the driver's movement, the distance to the nearest gas station is changing. If the driver still follows the initial result after moving to p_4, they will need to travel a long distance back to reach GAS_1 which is not the nearest gas station to the driver at the current timestamp. The query result needs to be continuously monitored and updated so that it always returns a correct result to the user rather than legacy ones.

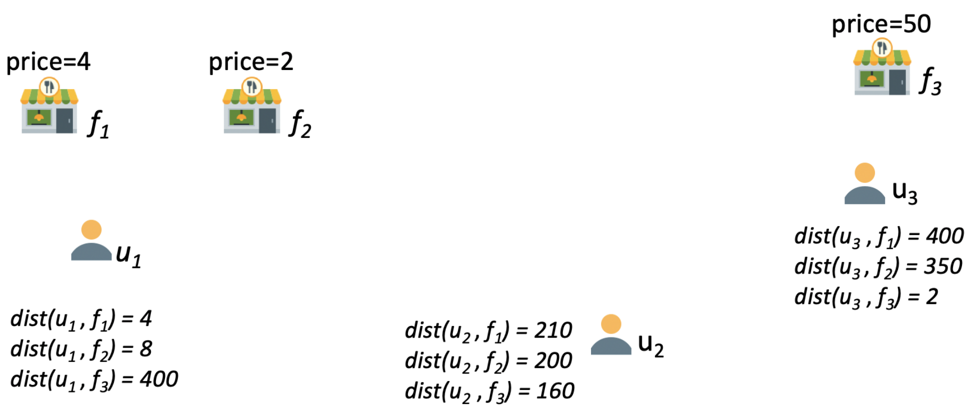


Figure 1.3: Example of SRTk queries

All the previously mentioned queries only take advantage of the distance as the determined conditions for the importance of a result point, such as finding the nearest supermarket or restaurant and looking for potentially influenced customers. Nevertheless, for different users, the requirement of examining a point of interest could be various. Sometimes, distance may not be the first priority for users. For example, a user may wish to find a restaurant that has cheap food and good ratings, while they accept they may need to drive a long way. In this context, distance-based queries are not suitable and some alternative queries such as top-k query [42, 61, 76, 77] can be used. For the top-k query, a score function is used to calculate a value for each point of interest i.e. restaurant. All the criteria are adopted in the calculation process of the score function. In the score function, each attribute is computed with a weight and the total weight of all attributes is 1. For example, in Fig 1.3, three restaurants and three users are given. Assuming each restaurant has a price attribute and the weight of price is same to the weight of distance from users to facilities, which is denoted as $w[price] = w[distance] = 0.5$ and $w[price] + w[distance] = 1$. The distance between a facility f_i and a user u_i is denoted as $dist(u_i, f_i)$. The score of a user u_i regarding a facility f_i is denoted as $score(u_i, f_i)$ and $score(u_i, f_i) = w[price] \times f_i(price) + w[distance] \times dist(u_i, f_i)$. The facilities that have the k smallest scores will be returned as the result. Hence, in Fig 1.3, according to the distance between users and facilities and the price value, the top-1 result of each user can be obtained, i.e top-1$(u_1) = \{f_1\}$, top-1$(u_2) = \{f_2\}$ and top-1$(u_3) = \{f_3\}$. Similarly, the top-2 result of each user can also be obtained, i.e top-2$(u_1) = \{f_1, f_2\}$, top-2$(u_2) = \{f_2, f_3\}$ and top-2$(u_3) = \{f_2, f_3\}$.

Since many conditions can be considered together to decide the ultimate result, a score function-based query is much more accurate and practical than a distance-based query. The same as the RkNN query coming from the kNN query, the top-k query has also been extended to the reverse top-k (RTk) [21, 25, 34, 46, 89, 90, 91, 92, 111] query for a better information analysis of facility influence. However, in traditional RTk queries, the attributes of each facility for all users are the same. In our research, distance is also an important condition that needs to be involved. Because distance is different for each user with respect to the same facility, traditional RTk algorithms cannot work properly. In [109], the spatial reverse top-k (SRTk) queries have been studied and this query can be used to analyse the overall influence (including distance) of a facility with respect to the surrounding users. Considering the example in Fig. 1.3, if the restaurants want to know the influence of both food price and distance for potential customers, they can issue a SRTk query. In this case, two conditions including the distance to a restaurant and price of a restaurant are used in the score function. Assuming the weight of price and distance are same and the sum of all weights is 1, then $w[price] = w[distance] = 0.5$. If $k = 2$, then we can find the spatial reverse top-2 result of f_2 is u_1, u_2 and u_3 i.e. $SRT2(f_2) = u_1, u_2, u_3$. To be more specific, all users can be the potential customers of restaurant f_2. Similarly, we can get $SRT2(f_1) = u_1$ and $SRT2(f_3) = u_2, u_3$.

However, we found an interesting phenomenon that is unreasonable. Seeing u_2, it can only be influenced by both f_2 and f_3, whereas the scores for all three facilities regarding u_2 are very close - $score(u_2, f_1) = 0.5 \times 4 + 0.5 \times 210 = 107$, $score(u_2, f_2) = 0.5 \times 2 + 0.5 \times 200 = 101$ and $score(u_2, f_3) = 0.5 \times 50 + 0.5 \times 160 = 105$. Specifically, the similar score means the overall importance of all criteria is almost identical to u_2. Thus, u_2 should also be influenced by f_1. In addition to this, u_3 also needs attention. In the spatial reverse top-2 result, u_3 belongs to the set of $SRT2(f_2)$ and $SRT2(f_3)$. If we calculate the score of u_3 for all facilities, we can see $score(u_3, f_1) = 0.5 \times 4 + 0.5 \times 400 = 202$, $score(u_3, f_2) = 0.5 \times 2 + 0.5 \times 350 = 176$ and $score(u_3, f_3) = 0.5 \times 50 + 0.5 \times 2 = 26$, which shows a big gap between $score(u_3, f_3)$ and $score(u_3, f_2)$. In this case, it is believed that u_3 would not think f_2 as a potential option. As we can see, with the strict use of k value, some illogical results could be generated but these are meaningless to the end users when issuing spatial reverse top-k query. Therefore, we propose a new query called the spatial reverse approximate top (SRAT) query, with the k value being relaxed. By using the SRAT query in the same scenario, the result will become $SRAT(f_1) = u_1, u_2$, $SRAT(f_2) = u_1, u_2$ and $SRAT(f_3) = u_2, u_3$. In this result, all the irrational results can be removed and the users will not be misled in some cases.

This chapter is organized as follows. Section 1.2 gives a short introduction of all the reverse approximate queries in spatial databases. The next section briefly illustrates the major challenges in this PhD project followed by the objectives in section 1.3. The contributions that we made to meet our objectives are listed in section 1.4. Finally, the organization of this PhD thesis is outlined in section 1.5.

1.2 Reverse Approximate Queries In Spatial Databases

In this section, we introduce some reverse approximate queries in spatial databases, including the snapshot and continuous RANN queries on road network and SRAT queries in *Euclidean* space.

1.2.1 Snapshot RANN Queries On Road Network

Given a graph G representing a road network that contains a set of facilities F and a set of users U, a query facility q and a value of $x > 1$, RANN query returns every user $u \in U$ for which

$dist(u, q) < x \cdot NNdist(u, f')$ where $dist(u, q)$ denotes the shortest path distance between u and q and $NNdist(u, f')$ denotes the shortest path distance between u and its closest facility f'.

1.2.2 Continuous RANN Queries On Road Network

In terms of moving RANN queries, we have a client-server system that maintains many queries. In this system, all the facilities and query points remain stationary and users may get to different locations at each timestamp. The goal of our algorithm is to continuously update RANNs of a given query as the users continuously change their locations. At the beginning, all RANN queries will be processed to get the initial result based on the users' positions. For the influence zone method, influence zones will be generated for all facilities. If a user moves out of a queries' influential area, the result set of that query will be updated in the system. In relation to the safe zone method, safe zones will be created for each user. If users only move inside their safe zone, all the query results remain unchanged. Otherwise, the query result will be updated and new safe zones will be generated accordingly.

1.2.3 SRAT Queries In *Eulicdean* Space

Given a graph G representing a space that contains a set of users U, a set of facilities F, a query facility $q \in F$ and a factor value of $x > 1$. Each $f \in F$ had d attributes (e.g. price, rating) and the value of i-th attribute is denoted as $f[i]$. All the $f[i]$ are static attributes because they remain the same for all the users. Nevertheless, the distance between the facility and different users $dist(u, f)$ is different. Hence, $dist(u, f)$ is called a dynamic attribute for each facility f. We assume all the attributes including both the static and dynamic ones are normalised to values between 0 and 1. For each attribute, a weight $w[i]$ is used to calculate the score and the sum of all $w[i]$ is 1. Considering a $(d + 1)$-dimension linear scoring function where each $w[i] > 0$ and $\sum_{i=1}^{d+1} w[i] = 1$. With these settings, the dynamic attribute (i.e. $dist(u, f)$) has a weight $w[d + 1]$ and the weights for different static attributes $f[i]$ are denoted as $w[i], 1 \leq i \leq d$. The total score of a facility f with respect to a user u is noted as $score(u, f)$. With all the given information, we can calculate the scores using expression $score(u, f) = w[d + 1] \cdot dist(u, f) + \sum_{i=1}^{d} w[i] \cdot f[i], 1 \leq i \leq d$. For any user u, their top-1 facility is the facility f that has smallest $score(u, f)$ and we can denote it as $Top1score(u)$. The SRAT query returns every user $u \in U$ for which $score(u, q) < x \cdot Top1score(u)$.

1.3 Objectives

Because of the natural difference between *Euclidean* space and road network, the existing method of the RANN query is not applicable for the real-world data set. At the same time, even though the SRT*k* query is extensively studied, no research has been conducted on the SRAT query. There are many challenges to be faced. In this section, we outline the objectives of this thesis.

1.3.1 Propose An Effective Method For RANN Query On Road Network

No research has been conducted on the RANN query using road network data and the existing RANN query algorithm on *Euclidean* space cannot be used on mesh graph. Hence, in this thesis, we create a new way to answer RANN with road network data. In the proposed method, a novel pruning and verification approach is used to optimise the performance of the RANN query in the road network space.

1.3.2 Propose An Efficient Algorithm To Monitoring RANN Query On Road Network

To the best of our knowledge, there is no existing techniques that are designed to monitor RANN queries on the road network. The monitoring pattern we use is the client-server mode with all facilities remaining stationary and all users moving at different timestamps. The users' movements cause the RANN result to be updated at every timestamp. We propose an efficient algorithm to resolve the moving RANN query on the road network with quick result set updating techniques.

1.3.3 Propose An Approach For SRAT Query With Good Performance

Finally, even though the SRT*k* query has been well studied and many useful techniques have been devised, no research has investigated the interesting variation query, namely the spatial reverse approximate top (SRAT) query. This is the first time that the SRAT query has been proposed and no previous work can be referenced. In this thesis, a new definition of the SRAT query is given. Based on this definition, we present an innovative method with good efficiency to answer the SRAT query .

1.4 Contributions

In this section, all the contributions made in this thesis are described. To solve all the RANN queries on road network, we adopt the *NVD* to divide the graph into small pieces that can be pruned easily. For the SRAT query, some efficient techniques using R*tree and the *Voronoi* diagram are introduced.

1.4.1 Snapshot RANN Query On Road Network

To handle stationary RANN queries on road network, we exploit the special features of the road network and analyse the inapplicability of the existing RANN methods. To improve the efficiency of query processing, a *NVD* index is used to separate the whole road network graph into small pieces. When performing snapshot RANN queries, some *NVD* cells can be pruned entirely according to the pruning rules. Hence, the users inside the pruned *NVD* cell do not need to be verified, saving a huge amount of query processing time. Furthermore, a systematic experiment is conducted using real data sets to show the effectiveness of our techniques and algorithms.

This work was published in the *World Wide Web Journal* 2020.

1.4.2 Continuous Monitoring RANN Query On Road Network

Next, we extend our research on the RANN query on road networks to the moving environment. Under moving settings, the position of all facilities and query points in our system remain unchanged and at different timestamps, users can move to different locations. Therefore, at each timestamp, the RANN result of each query could be different depending on the users' locations. Two approaches based one influence zone and safe zone are proposed to handle the users' movements. If users stay in their safe zone or in a facilities' influence zone, the RANN result will remain unaffected, otherwise we need to update the RANN result and update the moving user's safe zone. By using the safe zone and influence zone, the continuous queries can be executed efficiently. Last, we design an experiment to prove that our work is much faster than the competitors.

This research has been submitted to *Information Sciences* 2021.

1.4.3 SRAT Query

In terms of the SRAT query, this work is the first to propose and conduct research on this query. As it is a new query, no previous work can be referred to. We define the details of the SRAT query and use a facility R*-tree and a *Voronoi* diagram to process it. Because the SRAT query uses a score function to take many conditions into consideration, using the pruning method from RANN which only examines the distance is clearly insufficient. Firstly, we use a facility R*-tree to index the coordinates and static attribute values for all facilities. Based on the SRAT definition and *Voronoi* diagram, we can easily prune some facilities and find the facilities that cannot be pruned. For all the cannot-pruned facilities, we can find their *Voronoi* cells and all the users in these cells can be further verified to generate the final result set. Finally, our experiment outcome shows that our algorithm works on both synthetic and real-world data sets, which is up to two orders of magnitude better than the competitors and also scales significantly better.

This work is finished and it is planned to be submitted to *International conference on web information systems engineering (WISE)* in 2021.

1.5 Organizations Of The Thesis

The structure of this thesis is organised as follows.

- Chapter 2 describes all the works and techniques that are related to reverse approximate queries in spatial databases, including snapshot and continuous queries
- Chapter 3 describes our work on the snapshot reverse approximate nearest neighbour query in the road network space, which details the new techniques and explains the experiments
- Chapter 4 studies the continuous RANN query on the road network with two novel methods using the safe zone and influence zone.
- Chapter 5 clarifies a newly proposed query called the spatial reverse approximate top (SRAT) query. The SRAT query uses a score function and considers not only the distance attribute but also many other conditions such as rating, price etc. Hence, a novel method that can handle all criteria is proposed and demonstrated comprehensively

- Chapter 6 summarises the outcomes of our research and provides several possible directions for future work.

Chapter 2

Literature Review

2.1 Overview

In this chapter, a detailed literature review of all the relevant studies is presented, consisting of a series of spatial queries. Since our topic is reverse approximate queries in spatial databases, we examine how this concept was devised and why this kind of query is useful. In our projects, we analyse the snapshot and continuous reverse approximate nearest neighbour (RANN) query on road network and spatial reverse approximate top (SRAT) queries. So, we inspect the RANN-related snapshot and continuous queries and the ranking-related queries.

To begin with, we discuss the background information of some snapshot queries like k nearest neighbour (kNN) queries, reverse k nearest neighbour (RkNN) queries and RANN queries in section 2.2. Subsequently, we analyse some continuous queries working on both *Euclidean* space and road network space in section 2.3. Furthermore, several ranking-related queries are clarified and evaluated in section 2.4. Various state-of-the-art algorithms are investigated in this chapter to draw inspiration for our projects.

2.2 RANN-Related Snapshot Queries

In this section, we discuss the snapshot queries related to our research. We start with the k nearest neighbour (kNN) queries in section 2.2.1. Next, considering a different view of kNN, RkNN queries are examined in section 2.2.2. Lastly, the RANN query is examined in section 2.2.3, which is more accurate in computing the influence of a facility than R*k*NN.

2.2.1 *k* Nearest Neighbour Queries

In past decades, because of the rapid development of location-based technologies like the global positioning system (GPS), undertaking analysis using these techniques and geo-location data is becoming increasingly valuable. Thus, finding the nearest neighbour has gained substantial attention from researchers. The concept of the *k*NN query is a way to discover the *k* closest points of interest regarding a query point [69, 39]. Various researchers studied this query in diverse environment settings. With *Euclidean* space data, there are no obstacles between any two vertices and studies on this can be found in [69, 39, 72]. Some researchers conducted investigations using road network data [45, 50, 73, 27, 40, 51, 63], where the distance between any two objects cannot be calculated by coordinates directly but needs to consider the exact path distance. Hence, it is possible that two points are very close in *Euclidean* distance but very far in road network distance. *k*NN has also been examined in indoor space and obstructed space [32, 66, 100, 114, 49, 112, 102, 58], which adopts the variant data from both *Euclidean* space and road network space. We focus on the *k*NN in *Euclidean* space and road network space, as our research only uses data in these environments.

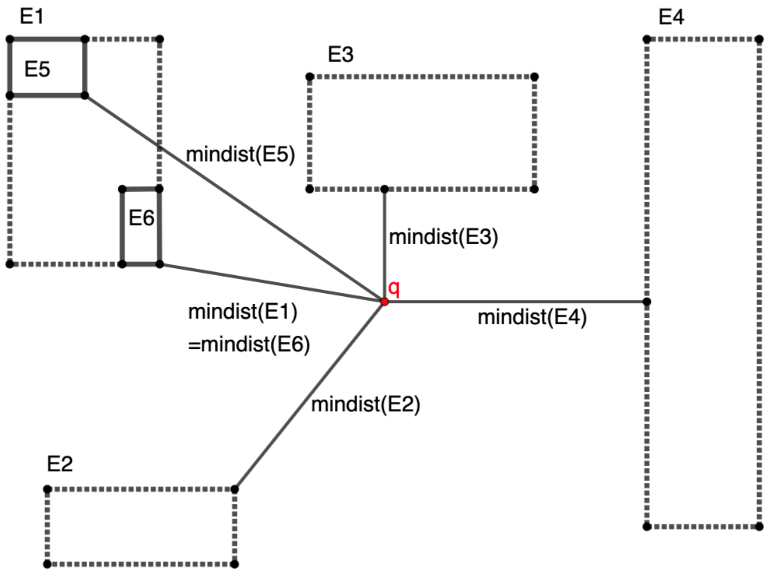


Figure 2.1: Example of Minimum Bounding Rectangles (MBR)

For the *k*NN query, the most commonly used index structure is the R-tree [69, 39]. With the R-tree index, all the points of interest are confined in different minimum bounding rectangles

(MBR). Because of the MBR, we can prune many nodes which definitely cannot be the result. For example, in Fig 2.1, there is a query point q and many points of interest in different MBRs. For each MBR, we can find the minimum distance to the query point. Based on all the minimum distances, a min-heap can be used to store all the MBRs. Every time, the top object in the min-heap will be opened to calculate the minimum distance between q and its children MBRs or target points. The same as the previous steps, all MBRs or target points will be added into the min-heap again. Finally, when an object is get out from the min-heap and it is not a MBR, it is the nearest neighbour result of q and kNN can also be generated by continuing this process.

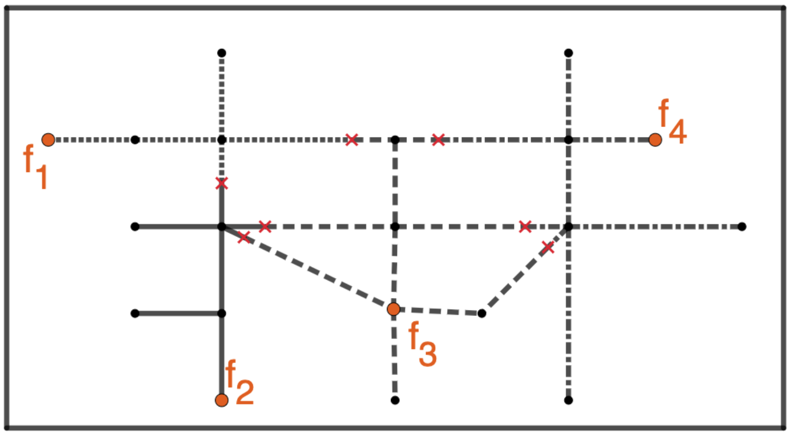


Figure 2.2: Example of Network Voronoi Diagram (*NVD*)

However, when considering the road network data, the *Euclidean* space kNN algorithm is unable to work properly. As the natural difference between these two data sets, the distance calculation between any two points needs to deal with the actual network path. The kNN query on the road network was first studied by [67], using an innovative method called Incremental Euclidean Restriction (IER). The IER method considers a *Euclidean* distance heuristic and retrieves the initial kNN candidates based on this distance. Then, the shortest path distance algorithm can be used to compute how far the query point is from each candidate. In the comprehensive experiment survey conducted by [5], IER also has very good performance in relation to the kNN query and the combination of IER and Pruned Highway Labeling (PHL) [8] can outperform most of the existing algorithms.

In addition to IER, another outstanding kNN algorithm was introduced in [4] which is state-of-the-art. In this paper, a Network Voronoi diagram (*NVD*) [29, 28, 65] is adopted to break the whole graph into small pieces. Fig 2.2 shows an example of the *NVD* diagram, which contains four facility points f_1 - f_4 and the area of each facility is shown with different types of lines.

Same as the *Voronoi* diagram, every node inside a *NVD* cell considers the generator as its nearest neighbour. Hence, when generating a *NVD* diagram, all the points of interest are used as the generators. Specifically, for each query, the 1NN result can be obtained immediately after locating the *NVD* cell to which the query point belongs. For instance, in Fig 2.2, if the query point resides in the solid line area, it will consider f_2 as its 1NN result. For further *k*NN results, all the neighbour generators of the initial generator which contains the query point can be obtained and added into a priority queue in accordance with the distance to query point. Finally, the *k*NN result can be retrieved from the priority queue.

2.2.2 Reverse *k* Nearest Neighbour Queries

Before discussing the R*k*NN, an important fact is that R*k*NN query is an extension of the the *k*NN query. The *k* nearest neighbour query is to find targets that are *k* closest to the query point straightforwardly [12, 84], whereas R*k*NN is different. An example is shown in Fig 2.3, with four facilities and 14 user points given. Each user can find their nearest facility using a *k*NN algorithm. For example, $1NN(u_1) = f_1$, $1NN(u_2) = f_1$, $1NN(u_3) = f_1$, $1NN(u_8) = f_2$ and so forth. It is found that many users consider the same facility as their nearest neighbour. If this facility is a restaurant or a shopping center, the users who consider this facility as their nearest one will be willing to visit it. This means we can change the perspective reversely and analyse how many users refer to a facility as their nearest neighbour, which can reveal the influence of a facility. Then, a R*k*NN query can be issued.

R*k*NN queries reflect the influence of the query point, which searches all the objects that deem the query point as their nearest neighbour [88, 85, 74, 104, 78, 82, 56, 18, 11]. It can be used in many situations such as finding the best location for a store in an area, so that all the residents in this area are closest to that store. The R*k*NN query has attracted significant interest since it was introduced in [52]. In [52], the query is answered by pre-computing a circle for each object with the radius of the distance between the object and its nearest neighbour and the center is itself. For a given query q, all circles that contain q are retrieved and their center points are returned as the query answer. An improvement was proposed in [106], however it still relies on pre-computation which is not applicable in dynamic environments. Significant algorithms that do not rely on pre-computation have been introduced, such as Six-Regions [81], TPL [86], TPL++

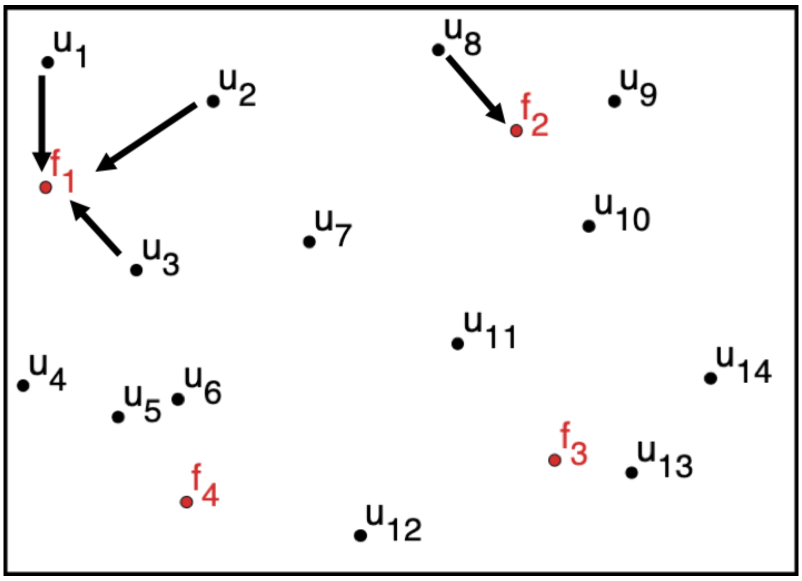


Figure 2.3: Example of *k*NN and R*k*NN

[107], FINCH [99], InfZone [19, 23] and SLICE [108]. All these algorithms use R-tree and various ways of pruning to reduce the number of targets that need to be processed which enhances the query performance.

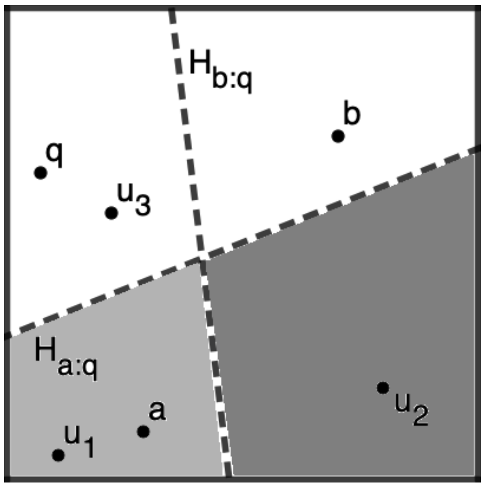


Figure 2.4: Example of half-space pruning

In TPL [86], TPL++ [107], FINCH [99] and InfZone [19, 23], a half-spacing pruning technique is used to filter the search space. A perpendicular bisector between two facilities separates the whole space into two halves. The users who are located in different half spaces are influenced by different facilities. When performing the R*k*NN query, more bisectors can be generated to prune more space and obtain the final result set. An example is illustrated in Fig 2.4. In Fig 2.4, three users u_1 - u_3 and three facilities q, a and b are given. Two perpendiculars of all three facilities are shown using dashed lines. Let $H_{a:b}$ denotes the half space that contains facility a and $H_{b:a}$ denotes the half space that contains facility b. Any user lying in $H_{a:b}$ considers a as its nearest neighbour since the half space is created by drawing a perpendicular line. According to this, the RNN result of q will be u_3 as this user is not in the area of $H_{a:q}$ and $H_{b:q}$. When issuing RkNN, the users which lie in at least the k intersection half space of different facilities cannot be the result of query q. In Fig 2.4, assuming $k = 2$, both u_1 and u_3 are the RkNN result of q, whereas u_2 is not as it belongs to both $H_{a:q}$ and $H_{b:q}$ (the dark shaded area). In other words, a pruning area can be forged during the half space creation process, which is the $H_{a:q} \cap H_{b:q}$.

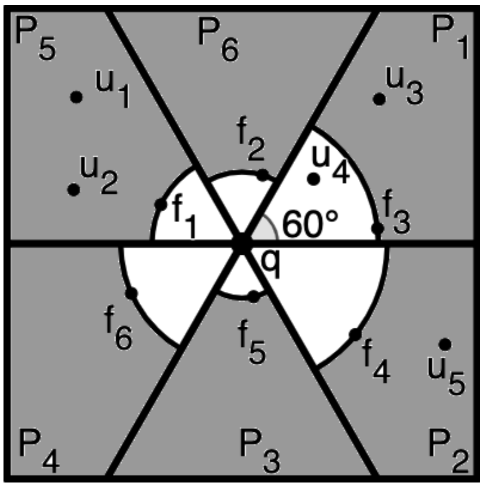


Figure 2.5: Example of Six-region pruning

Another technique, called six-regions [81], can also prune a large space. With six-regions-based method, the whole space is segregated into six equal areas using three lines at 60° and the center is the query point, such as regions P_1 to P_6 in Fig 2.5. In each region, the kth nearest facility f_i with respect to q can generate an arc based on the distance $dist(f_i, q)$. Any users who have $dist(u, q) > dist(f_i, q)$ cannot be the RkNN result of q. As shown in Fig 2.5, for a query point q, six facilities f_1 to f_6 and five users u_1 to u_5 are given. For each region, the nearest facility f_i is found and arcs are created based on the $dist(f_i, q)$. Hence, assuming $k = 1$, any users in the shaded area cannot be the RNN of q. Only u_4 is in the final result set, because u_4 is in section P_4 and $dist(u_4, q) < dist(f_3, q)$.

It has been proven that the half-space-based approach can prune more space than the six-region method [86], however the six-region approach is computationally faster. Since the six-region algorithm always divides the graph into 6 pieces and some space can be further pruned, a novel method called SLICE, which changes the graph partition size to filter more space, is proposed in [108]. In this method, the pruning area is calculated based on the vertex q and the subtended angle between facilities and the partitioned regions. Thus, a facility not only can prune some space in

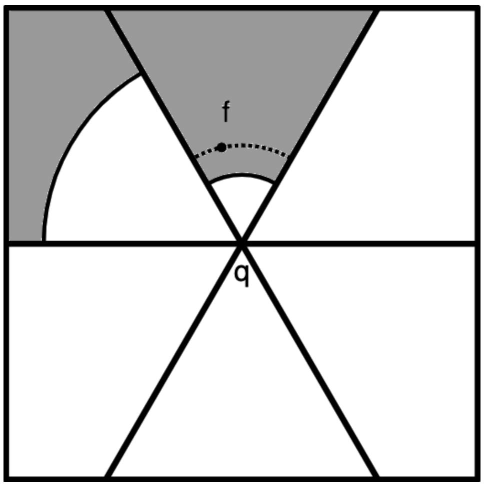


Figure 2.6: SLICE using 6 partitions

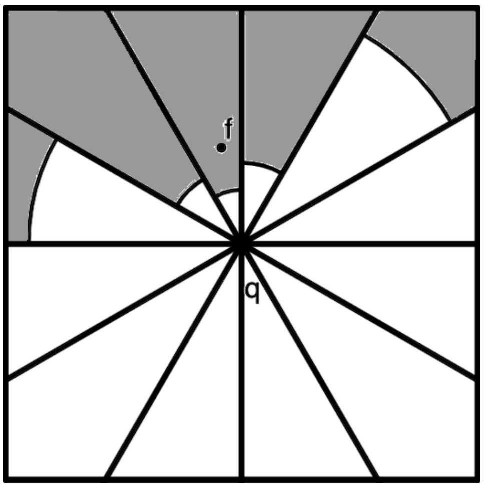


Figure 2.7: SLICE using 12 partitions

which it is located, it can also prune some extra space in other regions. As the partition number increases, the pruning area becomes larger. For example, in Fig 2.6, the graph is separated into 6 parts. The dotted line indicates the pruning area of f using the six-region method and the shaded area is the pruning space generated by SLICE which is bigger. Fig 2.7 demonstrates that more partitions can produce more pruning space for the same facility f.

Several algorithms for the R*k*NN query on the road network have been proposed in [110, 83, 36, 6]. The most common technique adopted to solve the R*k*NN query is the Network Voronoi Diagram (*NVD*). Different from the application in *k*NN query, all the generators of *NVD* will be the facilities and the query point is chosen from the facility set. Using the *NVD* index, the RNN result can easily be found by locating the *NVD* cell to which the query point belongs. For the R*k*NN query, a further retrieval needs to be conducted via reaching the adjacent *NVD* cells.

2.2.3 Reverse Approximate Nearest Neighbour Query

The RANN query [37, 38] was introduced as the complement of the R*k*NN query which considers the relative distance between objects. In the RANN query, the k value is relaxed and users can be influenced by a cluster of facilities. Hence, the RANN query is able to generate more rational results than the R*k*NN query.

The existing RANN algorithm [37] was proposed for queries on *Euclidean* space. It consists of three main phases namely pruning, filtering and verification. In the pruning phase, the areas that cannot contain query answers are identified. The authors proposed a point-based and R*-tree entry-based pruning method to prune these areas. In the point-based pruning, given a query q, a multiplication factor x and an object f, a circle C_f centered at c is created with radius r where $r = \frac{x \cdot dist(q,f)}{x^2-1}$ and c is on the line passing through q and f such that $dist(q,c) > dist(f,c)$ and $dist(q,c) = \frac{x^2 \cdot dist(q,f)}{x^2-1}$. It is guaranteed that any object inside this circle is not the query result. Fig 2.8 illustrates point-based pruning. The shaded area in the circle cannot contain RANNs of q and therefore can be pruned.

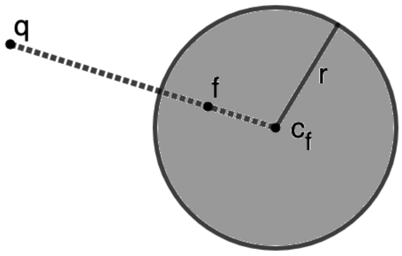


Figure 2.8: Point based pruning

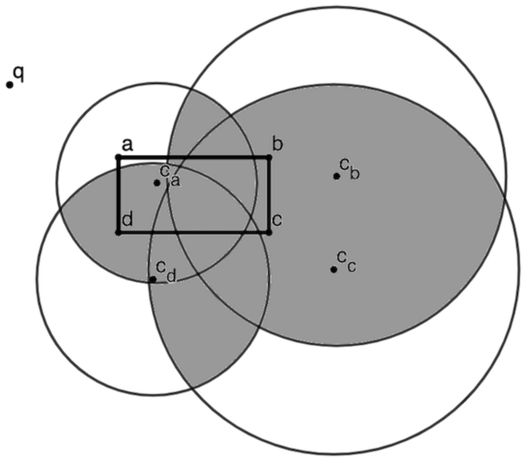


Figure 2.9: MBR based pruning

In the second pruning technique, a Minimum Bounding Rectangle (MBR) which represents a facility R*-tree entry is used. Since we only know that the MBR contains facilities and no information on the exact location of a facility (as MBR is not opened), four pruning circles using the four corners of the MBR are created respectively. Because of the features of MBR, no matter where a facility is located, its pruning circle regarding q will be covered by corner based pruning circles. Therefore, the intersection area between these four MBR corner-based pruning circles cannot contain any RANN query result. Fig 2.9 shows an example of the MBR pruning area which is the shaded area.

In the filtering phase, objects that reside inside the pruning regions are filtered out. The data space is partitioned into six as in the Six-Region [81] and an interval tree is used to locate the pruning regions that overlap the partition. Users that are not filtered are stored in a list and verified in the next phase. A circular *boolean* range query is used in the verification phase to identify if a user is RANN of the query, which is based on the pruning circles presented in Fig 2.8.

2.3 RANN-Related Continuous Queries

In the spatial database, moving queries are very popular since the work in [43]. There are a plethora of different types of continuous queries research, including *k*NN query [44, 87, 115, 60, 103, 62, 113, 22], range query [16, 53, 14, 33, 97, 95] and R*k*NN query [19, 71, 10, 98, 101, 47, 24]. Because our project is a continuous RANN query on road network, in this section, we only focus on various related continuous R*k*NN and RANN queries, using both *Euclidean* and road network data sets.

2.3.1 Continuous Reverse *k* Nearest Neighbour Queries

The first continuous RNN query algorithm was proposed in [9]. This work is based on an assumption where the position of all points changes in a linear function of time. To be more specific, the velocity of all nodes is known in advance. Hence, the coordinates of the moving objects can be predicted at every time stamp. A Time Parameterized R-tree (TPR) [71] is adopted as the underlying index structure. Later, Xia *et al.* [101] and Kang *et al.* [48] introduced continuous RNN query algorithms without any assumption on the pattern of moving objects. These two algorithms utilise the six-region and half-space approach respectively to prune the searching space. Only the unpruned area is monitored which can significantly reduce the processing time. Even though these algorithms do not rely on the assumption of the objects' movement, they are designed for *Euclidean* space and are not applicable in the road network environment.

Commonly, continuous RNN queries continuously update and report the query result to the server and the process is significantly time-consuming. Cheema *et al.* [20] proposed a method called *lazy updates* which reduces the execution times of the pruning phases. In this study, each moving object is assigned an area called a safe region [68]. When a moving object is inside its safe region, the algorithm will not be called and the query results remain unchanged. Similar to the safe region method, another algorithm named the *influence zone* [19, 23] was developed. It provides another perspective on monitoring query results by generating safe areas for query points. Given a graph G and a query q, an *influence zone* can be constructed such that all nodes inside this area are the query result of q. Hence, when a user is moving inside this *influence zone* of q, it will always be a query result of q and no update is required. When a user moves into or leaves the *influence* *zone* of q, the query result of q and its *influence zone* will be recomputed. The *lazy updates* and *influence zone* methods have only been studied on *Euclidean* space. However, the concept of *safe zone* and *influence zone* provide a new path for research on queries on the road network data.

Using the notion of the safe zone, Cheema *et al.* [24] proposed a novel method to solve a continuous R*k*NN query on the road network. They use several special nodes in the network graph called the dead vertex to prune the nodes and construct a safe region for users. Given a node v, a query point q and a facility point f, v is a dead vertex if the shortest distance from v to f is less than the shortest distance from v to q. So, v cannot be the RNN result of q. An example is shown in Fig 2.10 where a query point q, two facility points f_1 and f_2 and many user points are illustrated. Point E is a dead vertex, as the distance from E to f_1 is 1, while the distance from E to q is 3. So, E cannot be the query result of q. All other user nodes that are farther from q than the dead vertex cannot be the RNN result of q. Using this concept, the safe region of each moving object can be generated.

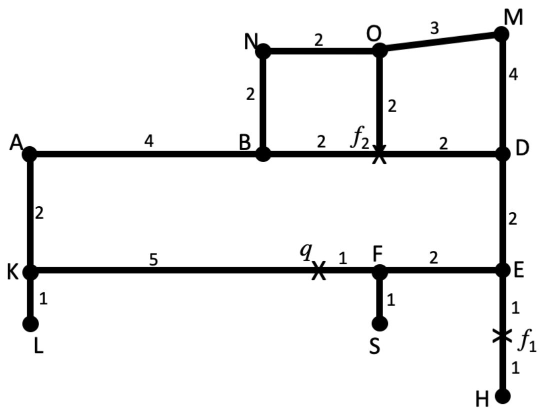


Figure 2.10: Dead vertex in R*k*NN query [24]

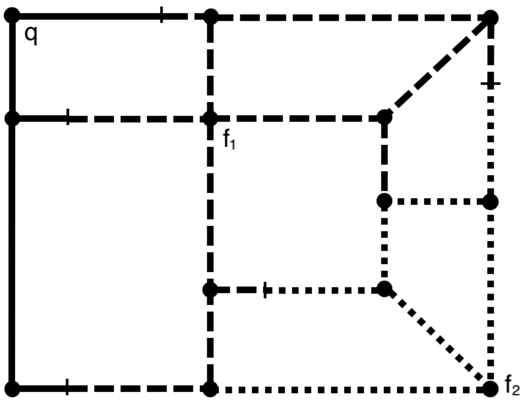


Figure 2.11: Influence zone [19]

Wang *et al.* [96] present an *influence zone* method to handle a moving R*k*NN query on the road network. For all objects, an influence zone will be generated so that if a user moves inside a query's influence zone, it will always have the same result. As shown in the example in Fig 2.11, there are three facilities q, f_1 and f_2 with different types of lines indicating their *influence zone*. If a user point moves along the solid line, it will always be influenced by q. The movement of the user will not change the result unless it moves out of the solid line area. A similar *influence zone* based algorithm has been studied in other moving queries, such as answering *k*NN [54].

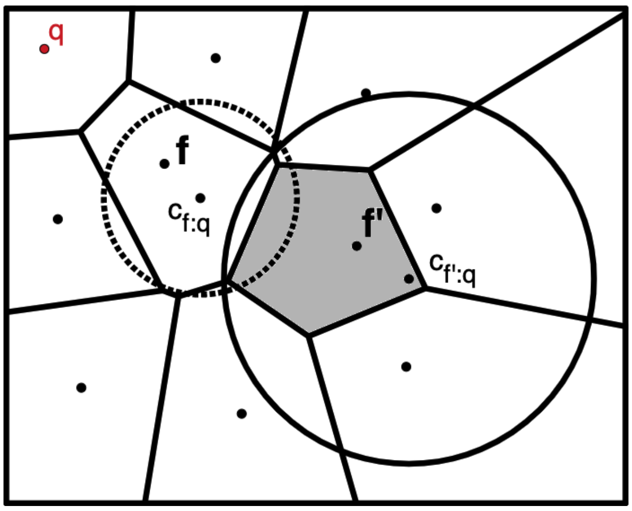


Figure 2.12: Prune a single *Voronoi* cell

2.3.2 Continuous Reverse Approximate Nearest Neighbour Query

In the paper of Hidayat *et al.* [37], the concept of RANN is proposed and a novel method based on R*-tree is used to answer the snapshot RANN query on *Euclidean* space. Subsequently, they offered another innovative approach to handle a moving RANN query on *Euclidean* space [38]. This method uses all facilities as generators to create a *Voronoi* diagram that can divide the whole graph into small segments called *Voronoi* cells. If each *Voronoi* cell can be covered by the pruning circle produced by the query point and its generator facility, any user belonging to this *Voronoi* cell cannot be the result of the query point. Fig 2.12 clarifies the relations between pruning circles and *Voronoi* cells. In the graph, there is a query point q and two facilities f and f'. According to the RANN definition $dist(u, q) \leq x \cdot NNdist(u, f)$, pruning circles can be generated (see as $c_{f:q}$ and $c_{f':q}$). The *Voronoi* cell of f' which is the shaded area is entirely enclosed by $c_{f':q}$. In other words, this cell can be pruned and users belonging to this region cannot be the RANN result of q. In contrast, the pruning circle of f has an intersection with its *Voronoi* cell. Thus, RANN result users could exist in this cell so this cell cannot be pruned.

A facility R*-tree is also adopted for the pruning process. As using the pruning circle to prune all *Voronoi* cells one by one is very slow, a MBR based pruning method is used to increase pruning efficiency. For example, in Fig 2.13, given a query point q, a facility f and a *Voronoi* diagram uses all facilities as generators. The *Voronoi* cell of f has several corner points v_1 to v_5. Using the location of q and f, a pruning circle can be generated and is centered at c_f. We can see in the graph that if the distance from f to F (the intersection point of a pruning circle and the line between q and

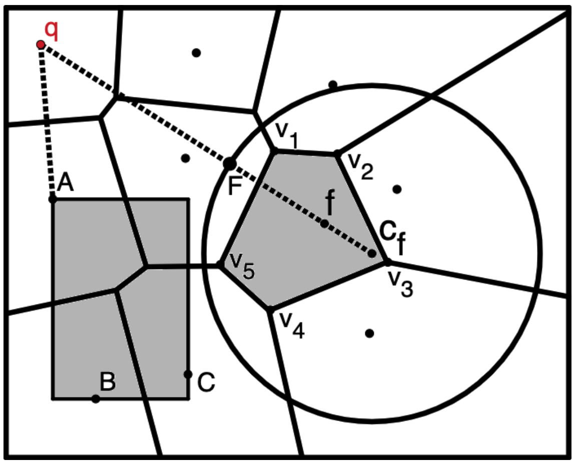


Figure 2.13: Prune a facility MBR

f) is greater than the maximum distance among f to its *Voronoi* cell corners v_i, this *Voronoi* cell can be pruned. To be more specific, if $dist(f, F) > maxdist(f, v_i)$, the *Voronoi* cell can be pruned. The value $dist(f, F)$ can be calculated as $\frac{dist(q,f)}{x+1}$. So, for any facility f, if $\frac{dist(q,f)}{x+1} > maxdist(f, v_i)$, the *Voronoi* cell of f is pruned. Because a R*-tree is used to index all facilities, for each MBR entry e, we can compute its minimum distance to query q which is denoted as $mindist(q, e)$. For facility f_i in a MBR entry e, $dist(q, f_i) > mindist(q, e)$ and the maximum value of $maxdist(f_i, v_i)$ can be found and denoted as $maxMaxdist(v)$. Therefore, for an entry, if $\frac{mindist(q,e)}{x+1} > maxMaxdist(v)$, any facility f_i in the MBR will have $\frac{dist(q,f_i)}{x+1} > maxdist(f, v_i)$, which means all facilities in the MBR can be pruned.

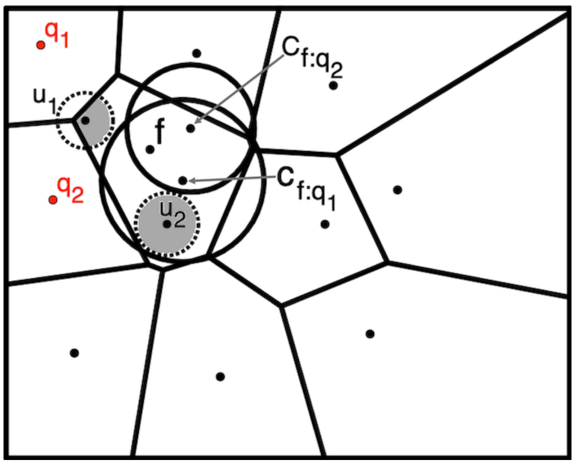


Figure 2.14: Example of user safe zone in RANN

After generating the initial RANN result, a safe zone technique is used to handle the movement of all users. As shown in Fig 2.14, users could have a different range of safe zone, depending on their positions. For each user, the safe zone radius is the distance from itself to the nearest pruning circle like u_2 and the shaded circle. If the safe zone intersects intersection with other *Voronoi* cells, only the part belonging to the cell that a user locates in is valid, such as u_1 and its shaded area. If a user moves out of its safe zone, this region will be re-computed and the query result will be updated. Otherwise, all the queries remain unaffected.

2.4 Ranking Related Queries

In this section, a brief analysis of all the related score function based queries is given. Ranking related queries has been studied extensively and different research has various settings such as divergent score function definitions [17, 80, 79, 59, 35, 31].

2.4.1 Top-*k* Queries

The Top-*k* query retrieves the *k* objects that have minimum scores which is computed using a score function [15, 61, 30, 64]. A wide-ranging survey is given in [42] and clarifies the commonly used techniques for top-*k* query processing. The top-*k* algorithms assume the objects in a source can be accessed either randomly or orderly. For a different source, the top-*k* algorithm may not work if the expected access way is not allowed.

Three well-known algorithms, namely Fagin's algorithm (FA) [31], threshold algorithm (TA) [31, 64, 35] and no-random access(NRA) respectively [31], are used to answer top-*k* queries. With the FA [31] method, the sources need to support both sorted and random access. It performs parallel accessing on each source and returns *k* objects at each source. Then, the scores of all the returned objects need to be calculated again based on random access on other sources. Finally, the *k* objects with the smallest scores are returned. The TA [31, 64, 35] method also accepts both sorted and random access. It maintains threshold values for each source and terminates the processing when there are *k* objects whose scores are at most equal to the threshold values. The NRA [31] algorithm only supports sorted access as it needs to access each source in an orderly way to compute two score boundaries, these being the best possible score and worst possible score.

2.4.2 Spatial Reverse Top-*k* Queries

Similar to the relation between *k*NN and R*k*NN queries, based on the top-*k* query, reverse top-*k* query is introduced. In the past decade, the reverse top-*k* query has gained significant attention from many researchers [21, 25, 34, 46, 111, 89, 90, 91, 92]. In these reverse top-*k* algorithms, an assumption is made that each facility has a series of attributes such as rating and price and the values of these attributes are the same for all users. However, if distance is taken into consideration when creating the score function, the previously mentioned algorithms cannot work. Because the distance from each user to a facility could be different, depending on their locations.

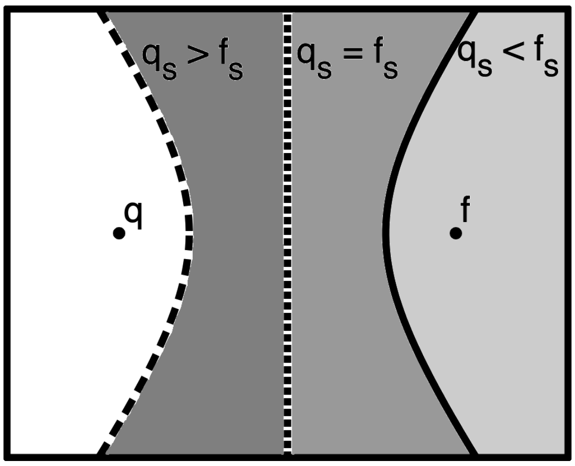


Figure 2.15: Example of hyperbola pruning

Since our project is in a spatial database, we discuss the spatial reverse top-*k* queries [109] which use distance as a dynamic attribute for each facility. In [109], two methods are used to answer spatial reverse top-*k* queries. One is based on the half-space pruning technique. Nevertheless, with a score function, the perpendicular bisector used in the half-space pruning technique is replaced by a half hyperbola line. Given a facility f with d static attributes which is recorded as f_i, the weight of each attribute is recorded as w_i and the total weights of all attributes including both static and dynamic ones are 1. For every user and facility pair, a score function $score(u, f) = w[d+1] \cdot dist(u, f) + \sum_{i=1}^{d} f_i \cdot w[i]$ is used to calculate the total score. Because the static attributes are the same with respect to each user, we can use f_s to denote the total static attributes' value and $f_s = \sum_{i=1}^{d} f_i \cdot w[i]$. So, the score function can be simplified to $score(u, f) = w[d+1] \cdot dist(u, f) + f_s$. By using this score function, a hyperbola pruning line can be generated as shown in Fig 2.15. In this graph, a query point q and a facility f are given. If $q_s > f_s$, the dashed line indicates the pruning area including all shaded areas. If $q_s = f_s$, a perpendicular bisector of q and f separate the whole area, with the section containing f pruned. If $q_s < f_s$, the solid hyperbola line is generated and the light grey area is pruned by f. The greater the value of f_s than q_s, the smaller the pruning area.

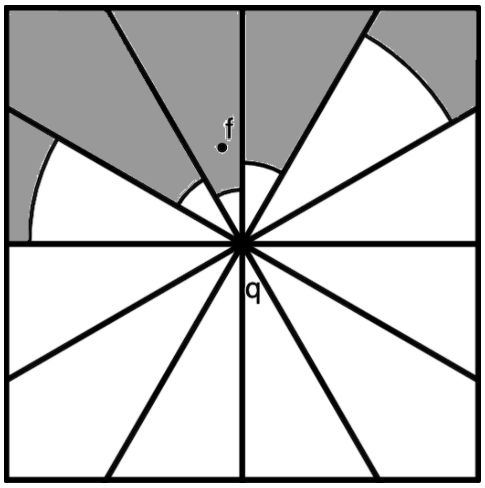


Figure 2.16: Example of *SLICE* pruning

Another method using the *SLICE* technique is also discussed in [109]. The key point in this approach is also to find the critical point and distance which can be used to find the pruning boundary in each sliced region. In the spatial reverse top-k query, using of score function, the static attributes' value of facility and query point need to be considered in computing the upper and lower bound. The final pruning area example is shown in Fig 2.16. With this technique, two phases are used in the algorithm, namely the filtering phase and the verification phase. In the filtering phase, the users who are not in the pruning space will be returned for further checking. In the verification phase, the candidate users are examined one by one to obtain the final result.

2.5 Conclusion

In this chapter, we presented a detailed study for the state-of-the-art algorithms for different kinds of spatial queries. First, we discussed the kNN and RkNN algorithm because the idea of RANN comes from RkNN and is needed to perform the kNN according to its definition. Then, we review the existing RANN query in *Euclidean* space, which can generate a more reasonable result set than the RkNN query. However, the *Euclidean* space RANN algorithm is not applicable for road network data as *Euclidean* space has no connections between all the vertices. Apart from the snapshot queries, we also examine the continuous queries algorithms on *Euclidean* space and road networks. Since with road network data, RANN is of great difference when comparing with the RkNN and RANN on *Euclidean* space, new methods need to be proposed to continuously monitor this query. Finally, the ranking related top-k and reverse top-k queries are analyzed comprehensively. Since our project is in spatial databases and trying to relax the value k, we only focus on the spatial reverse top-k query which adopts the dynamic attribute distance as one criterion when calculating the final scores.

Chapter 3

Snapshot RANN Queries On Road Network

3.1 Overview

With the increase of internet speed, more affordable geo-position locator and cheaper bandwidth cost, online map applications have become an essential part in today's human life. Many leading and start up companies provide map or navigation service as one of their main services, such as Google, Uber, Grab etc. Online map application service allows user to submit various spatial queries, like shortest path query, nearest neighbor query, range query etc. All those online map applications use road network distance which give more accurate distance information compared to the *Euclidean* distance. As discussed in previous section, the RANN query has been well studied in *Euclidean* space. However, the *Euclidean* algorithm is not applicable on road network data. In this chapter, we are going to study the Snapshot RANN query on road network.

The rest of the chapter is structured as follows. We present problem definition and our contribution in Section 3.2. Our proposed technique and algorithm are detailed in Section 3.3. An experimental study is presented in Section 3.4 followed by conclusion in Section 3.5.

3.2 Background Information

In this section, we present the background information of reverse approximate nearest neighbor (RANN) query. This query was introduced in [37, 38] as a complement of reverse k nearest neighbor (RkNN) query. Formally, given a set of users U, a set of facilities F, an integer k and a query facility q, an RkNN query returns every user $u \in U$ that considers q as one of its k closest facilities. Consider the example of Fig. 3.1 and assume $k = 2$, $R2NN$ of f_2 are u_1, u_2, u_3, u_4 and u_5 because f_2 is among the 2 closest facilities for these users, i.e. $R2NN(f_2) = \{u_1, u_2, u_3, u_4, u_5\}$. Similarly, $R2NN(f_1) = \{u_2\}$, $R2NN(f_3) = \{u_1\}$ and $R2NN(f_4) = \{u_3, u_4, u_5\}$.

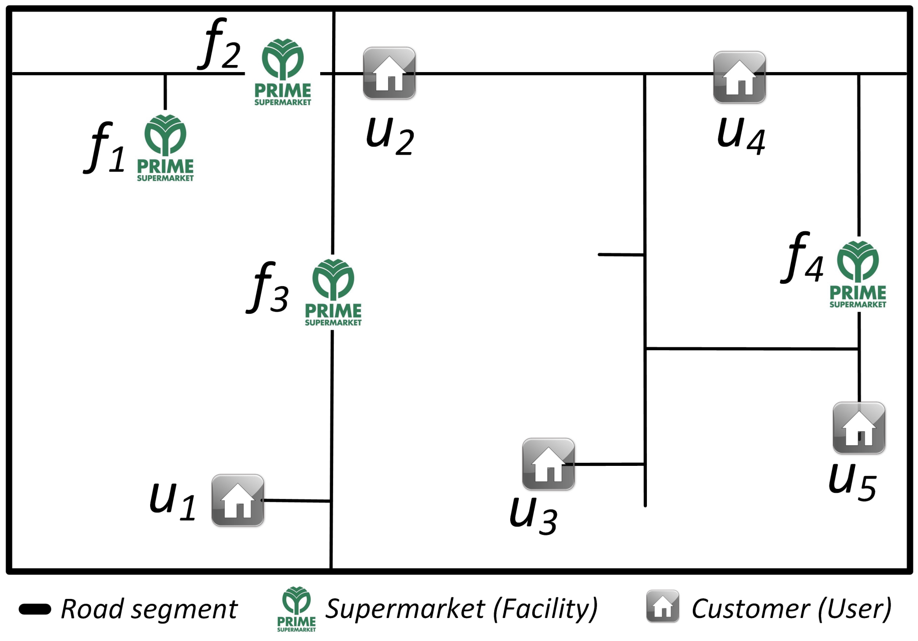


Figure 3.1: Example of RkNN

The RkNN query has many applications in decision support, location-based service, profile-based marketing etc. Consider an example of a supermarket that is launching a promotion deal as shown in Fig 3.1. The supermarket wants to know which potential customers that are possibly interested in the deal. Assuming that distance is the only factor that affects customers' decision, the supermarket may issue an RkNN query to find all those potential customers. Potential customers are those who consider the supermarket as one of their k closest supermarkets.

As argued in [37, 38], RkNN query may fail to retrieve all potential customers of the query issuing supermarket. For example, assume f_1 issues a RkNN query with $k = 2$. Customers u_4 and

u_5 are not part of the query answers, since their 2 closest supermarkets are f_4 and f_2. Hence, based on R*k*NN query, u_4 and u_5 are not potential customers of f_1.

However, this may not be true. Assume the distance between u_5 to f_4, f_2 and f_1 are 5, 30 and 31 Kms respectively. It is believed that u_5 that is travelling 30 Kms to reach f_2 normally does not mind to travel 1 or 2 Kms farther to reach the next closest supermarket (i.e f_1). Such customer may not be captured by R*k*NN query and thus is not considered as a potential customer of f_1. In such circumstances, RANN query can be used as a complement for R*k*NN query. RANN query was firstly studied on *Euclidean* space [37] (from this point onward, we refer this as *Euclidean* RANN). This study was extended in [38] to continuously monitor RANNs of queries.

In the example above, the use of *Euclidean* RANN query will not be appropriate, since it uses *Euclidean* distance which corresponds to the straight-line distance between vertices in the graph. In the example above, it does not reflect the actual distance between customers and the supermarket. The actual distance between them is reflected by the shortest path distance on road network. In this case, RANN query on road network is required to help the supermarket to identify its potential customers.

To the best of our knowledge, there is no study on RANN query on road network. The existing *Euclidean* RANN algorithm works only on *Euclidean* space. The algorithm uses data indexing, space pruning, user filtering and verification techniques which are not applicable on road network. In this chapter, we propose an algorithm to efficiently answer RANN queries on road network. In our proposed algorithm, road network is represented as a graph that is partitioned into smaller regions according to a structure called *Network Voronoi Diagram* [29, 28, 65].

3.2.1 Problem Definition

Similar to *Euclidean* RANN query, RANN query on road network can be classified into *monochromatic* and *bichromatic* RANN queries. However, in this chapter we focus on *bichromatic* version since it has more applications in real world scenario. In this version, objects are divided into two categories namely facility and user. Facilities are those that provide services and users are the ones who use the services. In the rest of this chapter, whenever it is used, RANN query corresponds to *bichromatic* RANN query.

RANN query. Given a graph G representing a road network that contains a set of facilities F and a set of users U, a query facility q and a value of $x > 1$, RANN query returns every user $u \in U$ for which $dist(u, q) < x \times dist(u, f')$ where $dist(u, q)$ denotes the shortest path distance between u and q and $dist(u, f')$ denotes the shortest path distance between u and its closest facility f'.

Due to the different characteristics between *Euclidean* distance and network distance, existing *Euclidean* RANN algorithm cannot be directly applied for RANN query on road network. Firstly, *Euclidean* RANN algorithm uses branch and bound property of R*-tree. In R*-tree, objects are grouped into rectangles based on their closeness according to their *Euclidean* distances. Since *Euclidean* distance is not relevant in road network environment then branch and bound property of R*tree cannot be utilized to solve RANN query on road network.

Secondly, *Euclidean* RANN algorithm uses pruning, filtering and verification techniques that only work on *Euclidean* space. For example, RANN algorithm in [37] utilizes pruning circles generated by facility points to prune all users that cannot be RANN of the query. Consider a query q and a facility f in Fig 3.2, the red circle centered at C (C_f) is the pruning circle generated by f. Assume $x = 1.5$, any user inside C_f (e.g., u_1) has *Euclidean* distance to q greater than x times its *Euclidean* distance to its nearest facility f. In *Euclidean* space, it meets $dist(u_1, q) > x \times dist(u_1, f)$ and hence u_1 can be pruned.

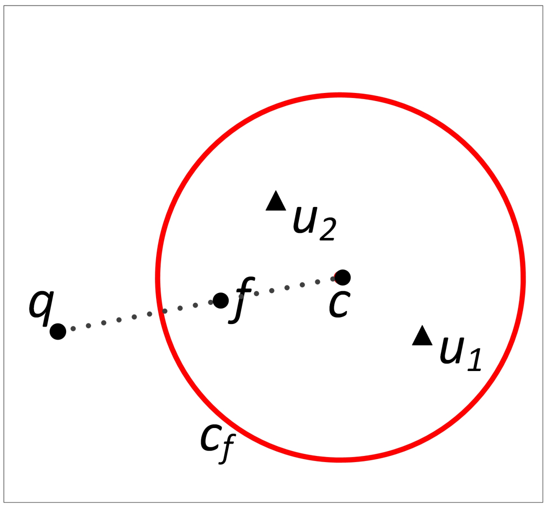


Figure 3.2: Pruning circle in *Euclidean* RANN algorithm [37, 38]

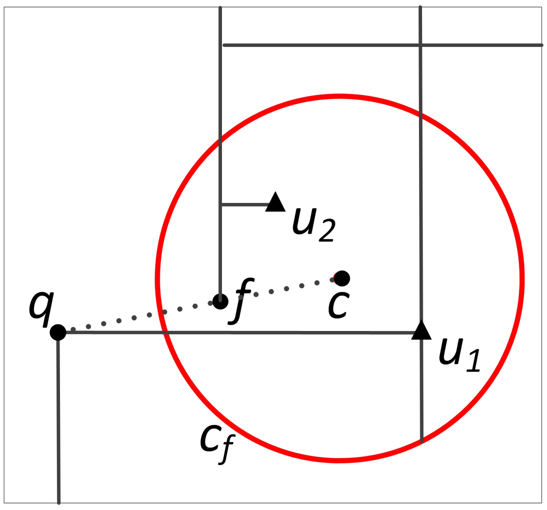


Figure 3.3: In-applicablity in RANN query on road network

This pruning rule does not always hold in road network environment. Consider the same pruning circle C_f on a road network shown in Fig 3.3. User u_1 is inside C_f, however its shortest path distance to q is smaller than x times its shortest path distance to its nearest facility f. In road network environment, it meets $dist(u_1, q) \leq x \times dist(u_1, f)$. According to the RANN definition, u_1 is RANN of q and should not be pruned even though it is inside pruning circle C_f. Hence, it is obvious that existing *Euclidean* RANN pruning technique is not applicable for query on road network.

3.2.2 Contributions

Below, we summarize our contributions in this research.

- We introduce pruning techniques based on *Network Voronoi Diagram* to prune road segments that do not contain query answer.
- We propose an efficient algorithm that utilizes our pruning and verification techniques to efficiently solve RANN query on road network.
- We conduct an extensive experimental study on real data sets to show the effectiveness of our techniques and algorithms.

3.3 Answering RANN query on road network

We study RANN query in road network environment. Road network is usually represented as a graph consist of edges that represent road segments and vertices that represent intersections and other point of interests (POI). In this chapter, POIs include objects that provide services, such as restaurant, school, etc (called as facility). RANN query retrieves all users that consider the query facility as almost as close as their nearest facility.

A naive solution is to check if the shortest path distance of each user to the query point is less than x times of the shortest path distance to its nearest facility. However, checking the shortest path distance of every single user to the query point is computationally expensive. We propose our solution to efficiently find RANNs of a query. Our solution consists of three phases namely pre-processing, pruning and verification. Details of each phase is presented in the following sections. From this point onward, any *distance* between two points a and b, denoted as $dist(a, b)$, refers to the shortest path distance between vertex a and vertex b in the observed road network graph.

3.3.1 Pre-processing

In pre-processing phase, we partition the road network graph into small regions, each of which containing a set of road segments that share one same nearest facility. Such structure is called *Network Voronoi Diagram (NVD)* [29, 28, 65]. In *Network Voronoi Diagram*, the small region is called *Voronoi* cell and the common shared facility point inside the cell is called the *generator* point.

Given a *Voronoi* cell V_f generated by a facility point f, all users inside V_f consider f as their closest facility. Specifically, for any user u inside V_f, $dist(u, f) < dist(u, f')$ where f' is any other facility in the graph. In our solution, due to the presence of *Network Voronoi Diagram*, a nearest neighbor query is not required to find the nearest facility of a given user. Consider four facilities q, f_1, f_2, f_3 in a *Network Voronoi Diagram* in Fig 3.4. The nearest facility for u_1, u_3 and u_4 is q as they are inside the *Voronoi* cell generated by q. Similarly, the nearest facility of u_5 is f_3. In the rest of this chapter, we use *NVD* to refer the discussed *Network Voronoi Diagram.*

For each *Voronoi* cell V_f generated by facility point f in a *NVD*, we record the distances between f to every user inside V_f. All users inside V_f are stored in a list sorted in descending order according to their distances to f. The maximum user distance in each *Voronoi* cell is maintained and will be used in the pruning phase. Consider the cell generated by f_2 (V_{f_2}) in Fig. 3.4. In pre-processing phase, when (V_{f_2}) is constructed, distances between u_6, u_7 and u_8 to f_2 are recorded. These users are stored in descending order of their distances to f_2. In addition, the maximum distance is recorded as internal distance of V_{f_2} (details will be discussed in Section 3.3.2).

3.3.2 Pruning

A straightforward approach to find RANNs of a query is to check the distance of every user in the graph to the query point, and compare the computed distance with its distance to the generator point of *Voronoi* cell where it resides. However, this approach takes $O(U)$ where U is the total number of users in the graph. We present our technique to prune road segments that can not contain the query answer. Specifically, our technique ignores every *Voronoi* cells containing road segments without RANNs of the given query.

Before we present our Lemma to prune a *Voronoi* cell, we present the definition of *internal distance*.

Definition 3.3.1. Internal distance. Let V_f be a *Voronoi* cell generated by facility point f and U_f be a set of user points inside V_f, internal distance of V_f (denoted as $indist(V_f)$) is the maximum distance between user $u \in U_f$ and f, i.e., $indist(V_f) = max(dist(u, f)), \forall u \in U_f$.

Consider a *NVD* consist of a set of facilities and users in Fig. 3.4. Internal distance of V_{f_3} is the distance between f_3 and u_5 since u_5 is the only user in *Voronoi* cell generated by f_3, i.e., $indist(V_{f_3}) = dist(u_5, f_3)$. In V_{f_2}, there are three users u_6, u_7 and u_8. Assume $dist(u_7, f_2) = 4$, $dist(u_6, f_2) = 5$ and $dist(u_8, f_2) = 10$, then $indist(V_{f_2}) = max(dist(u_7, f_2), dist(u_6, f_2), dist(u_8, f_2)) = dist(u_8, f_2) = 10$.

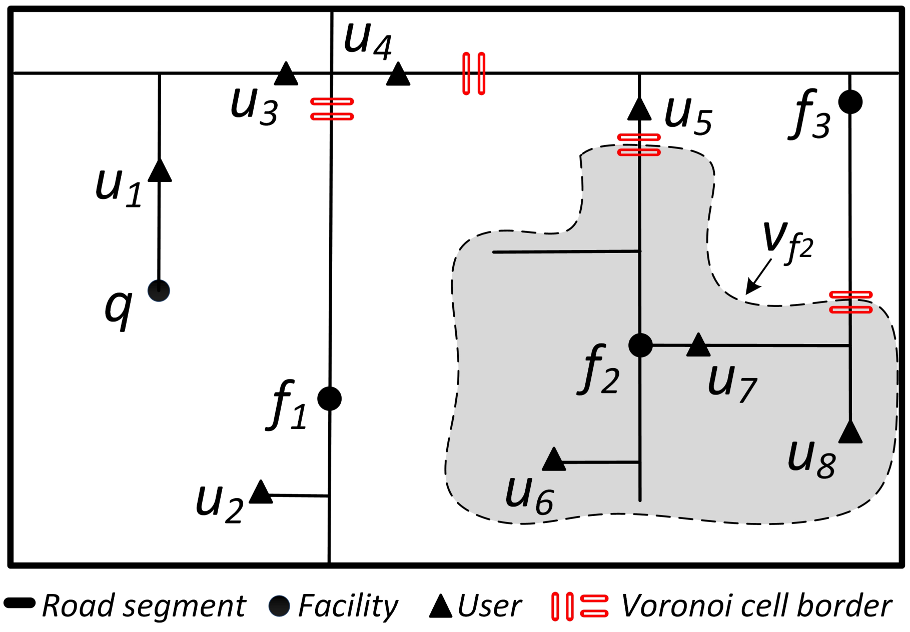


Figure 3.4: *NVD* of four facilities (q, f_1, f_2, f_3)

Recall that internal distance is query independent and is computed in pre-processing stage. Hereafter, we use *Voronoi* cell of f to refer the *Voronoi* cell generated by facility point f. Now we present our first pruning rule in Lemma 3.3.1.

Lemma 3.3.1. *Given a query q, a multiplication factor $x > 1$ and a Voronoi cell of f V_f, all users inside V_f cannot be RANN of q if $dist(q, f) - indist(V_f) > x \times indist(V_f)$, i.e., V_f can be pruned.*

Proof: For any user u inside V_f, f is its closest facility and $dist(u, f) \leq indist(V_f)$. Since $dist(u, q) \geq dist(q, f) - indist(V_f)$ and $dist(q, f) - indist(V_f) > x \times indist(V_f)$, then $dist(u, q) \geq x \times dist(u, f)$. Hence u cannot be RANN of q and consequently V_f can be safely pruned.

Consider *Voronoi* cell of f_2 in Fig. 3.4. Let $indist(V_{f_2}) = 10$, $x = 1.5$ and $dist(q, f_2) = 30$. All users in V_{f_2} (u_6, u_7 and u_8) cannot be RANN of q since $dist(q, f_2) - indist(V_{f_2}) = 20$ is greater than $x \times indist(V_{f_2}) = 15$. Hence V_{f_2} can be safely pruned.

Lemma 3.3.1 prunes every *Voronoi* cell that does not contain RANNs of q. However, opening all *Voronoi* cells one by one to check if it can be pruned is costly. It takes $O(F)$ where F is the number of facilities in the graph. We present a technique to prune all remaining *Voronoi* cells once a pruned *Voronoi* cell is encountered. Before we present our second pruning rule, we present the definition of *maximum internal distance*.

Definition 3.3.2. Maximum internal distance. Let G be a road network graph containing a set of facilities (F) and a set of users (U), and $NVD(G)$ be the Network *Voronoi* Diagram of F in G, maximum internal distance (*maxIndist*) of $NVD(G)$ is the biggest internal distance among all *Voronoi* cells in $NVD(G)$, i.e., $maxIndist = max(indist(V_f)), \forall f \in F$.

Consider the example in Fig. 3.4. Assume $indist(V_q)$, $indist(V_{f_1})$, $indist(V_{f_2})$ and $indist(V_{f_3})$ are 7, 5, 10 and 7 respectively. The maximum internal distance of this *Voronoi* diagram is $maxIndist = max(7, 5, 10, 7) = 10$. Now, we present our pruning rule in Lemma 3.3.2 to efficiently prune *Voronoi* cells in that do not contain RANN of q.

Lemma 3.3.2. *Given a query q, a multiplication factor $x > 1$ and Voronoi cell of a facility f, if $dist(q, f) - maxIndist > x \times maxIndist$ then Voronoi cell of any other facility f' for which $dist(q, f') > dist(q, f)$ can be pruned.*

Proof: Since $dist(q, f) - maxIndist > x \times maxIndist$ and $dist(q, f') > dist(q, f)$ then it is obvious that $dist(q, f') - maxIndist > x \times maxIndist$ and hence, according to Lemma 3.3.1, V'_f of any other facility f' can be pruned.

Consider f_1, f_3, f_2 in Fig. 3.4 above. Let $x = 1.5$, $dist(q, f_1) = 22$, $dist(q, f_3) = 26$, and $dist(q, f_2) = 28$. *Voronoi* cells are accessed in ascending order of distances between their generator point to q. For f_3, $dist(q, f_3) - maxIndist = 16$ which is greater than $x \times maxIndist = 15$. Hence any other *Voronoi* cell V'_f for which $dist(q, f') > dist(q, f_3)$ can be pruned (i.e. no user in this

Voronoi cell can be RANN of q). In this example, all remaining *Voronoi* cells (i.e., V_{f_2}) can be pruned.

We use Lemma 3.3.1 and 3.3.2 to prune the road segments that do not contain RANN of the query. Algorithm 1 presents the details of the pruning process. The algorithm initializes a priority queue *vQueue* with all neighboring facilities of the query facility q (line 2). The entries in *vQueue* are sorted according to their distance to q and are de-queued iteratively from the queue. If the processed entry f is not pruned (line 6), its corresponding *Voronoi* cell V_f is inserted into an output set $\mathcal{A}$. The algorithm terminates if the stopping condition in Lemma 3.3.2 (line 8) is met or if *vQueue* becomes empty.

Algorithm 1: Pruning

```
1 Input: Road network graph $G$, corresponding Voronoi diagram $V(G)$ and a query $q$
2 Output: The set of un-pruned Voronoi cells $\mathcal{A}$
 1: $\mathcal{A} \leftarrow \phi$
 2: insert each neighbor of $q$ in vQueue
 3: while vQueue is not empty do
 4:    de-queue a facility $f$
 5:    insert each neighbor of $f$ in vQueue
 6:    if $V_f$ is not pruned then
 7:       insert $V_f$ in $\mathcal{A}$
 8:    if $dist(q, f) - maxIndist > x \times maxIndist$ then
 9:       break
10: return $\mathcal{A}$
```

3.3.3 Verification

Algorithm 1 returns a set of unpruned *Voronoi* cells. For each unpruned cell v, each user u inside v is verified whether u is RANN of q. Note that evaluating every single user in all unpruned cells is highly inefficient. We present our technique in Lemma 3.3.3 to immediately identify if users inside a *Voronoi* cell need not to be verified.

Lemma 3.3.3. *Given a query q, a multiplication factor $x > 1$, Voronoi cell of a facility f (V_f) and a set of users inside V_f (U_f), for any user $u \in U_f$, if $dist(q, f) - dist(u, f) \geq x \times dist(u, f)$, then any other user $u' \in U_f$ for which $dist(u', f) \leq dist(u, f)$ cannot be RANN of q.*

Proof: Based on triangle inequality, dist(q,f)-dist(u,f) $\geq dist(q, u)$ and similarly dist(q,f)-dist(u',f) $\geq dist(q, u')$. If $dist(u', f) \leq dist(u, f)$, then $x \times dist(u, f) \geq x \times dist(u', f)$ and it holds $dist(q, u') \geq$

dist(q,f)-dist(u',f) $\geq$ dist(q,f)-dist(u,f) $\geq x \times dist(u, f) \geq x \times dist(u', f)$. Hence $dist(q, u') \geq x \times dist(u', f)$ which proves that u' cannot be RANN of q.

Consider u_6, u_7 and u_8 inside V_{f_2} in Fig. 3.4. Let $x = 1.5$, $dist(q, f_2) = 28$, $dist(u_7, f_2) = 4$, $dist(u_6, f_2) = 5$ and $dist(u_8, f_2) = 10$. For u_7, $dist(q, f_2) - dist(u_7, f_2) = 24$ is greater than $x \times dist(u_7, f_2) = 6$. In this case, u_6 and u_8 whose distances to f_2 are greater than $dist(u_7, f_2)$ can be ignored (they do not need to be verified as RANN of q).

Note that in the preprocessing, users in *Voronoi* cell of f are stored in descending order of their distances to f. When a user in V_f is verified, the algorithm will skip verifying the rest of the users in V_f if condition in Lemma 3.3.3 is fulfilled. Algorithm 2 present the details of verification process. For every un-pruned *Voronoi* cell in *unpruned* list, a loop is initiated to iteratively verify every user inside it (line 4). The loop stops if the algorithm encounters a user for which Lemma 3.3.3 is applied (line 7). Algorithm 2 terminates when un-pruned list becomes empty.

Algorithm 2: Verification

```
1 Input: A set of un-pruned Voronoi cells $\mathcal{A}$ and a query $q$
2 Output: The query result $RANN(q)$
 1: $RANN(q) \leftarrow \phi$
 2: for each cell $v \in \mathcal{A}$ do
 3:    $f \leftarrow$ generator facility of $v$
 4:    for each user $u$ inside $v$ do
 5:       if $dist(q, u) < x \times dist(u, f)$ then
 6:          insert $u$ in $RANN(q)$
 7:       if $dist(q, f) - dist(u, f) \geq x \times dist(u, f)$ then
 8:          break
 9: return $RANN(q)$
```

3.4 Experiments

To the best of our knowledge, there is no algorithm to solve RANN queries on road network. We consider a naive algorithm (**NAIVE**) as the competitor for our NVD based algorithm (**NVD**). In **NAIVE**, each user is verified if it is a RANN of the query. If a user u satisfies $dist(u, q) \leq x \times dist(u, f_u)$ where f_u is the nearest facility to u, then u is a RANN of q. In both algorithms, NVD is constructed and is specifically used in **NAIVE** to find the nearest facility of a user. In addition, we implement *Pruned Highway Labelling (PHL)* [8] in both algorithms to get the shortest path distance between two vertices in the graph.

All algorithms were implemented in C++ and experiments were conducted on Nectar Elastic Cloud Computing(EC2) with 16 AMD EPYC Processor (with IBPB) and 64GB memory running Debian Linux. We use real road networks with different network size as shown in Table 3.1 [3]. We randomly choose vertices from the graph and set them as the users. We use 8 sets of POI extracted from Open Street Map (OSM) and set them as the facilities, as shown in Table 3.2

Table 3.1: Road Networks for snapshot RANN

Name	**Region**	**No. of Vertices**	**No. of Edges**
NW-US	Northwestern United States	1,089,933	2,545,544
W-US	Western United States	6,262,104	15,119,284
US	United States	23,947,347	57,708,624

Table 3.2: Facility Sets for snapshot RANN

Facility type	**Size (W-US)**
Schools	27,613
Parks	20,900
Fast Food	7,547
Post Offices	5,198
Hotels	2,843
Hospitals	2,503
Universities	633
Courthouses	354

We vary the network size, number of facilities, number of users and the value of x (x is a multiplication factor) and evaluate their effect to the performance of both algorithms. The parameters used in the experiments are shown in Table 3.3 and the default values are shown in bold. We run 100 queries and report the average CPU cost for every experiment. Each query point is selected randomly from the facility set.

Table 3.3: Parameters of snapshot RANN

Parameter	**Ranges**
Network size	NW-US (1M), **W-US (6M)**, US (23M)
Number of facilities	Refer to Table 3.2
Number of users	150K, 200K, 300K, **500K**, 1M, 2M
x	1.1, **1.5**, 2, 3, 4

Effect of network size. Fig. 3.5 studies the effect of network size on both algorithms. The cost of both algorithms increases as the network size increases. This is because in higher network size there are more edges involved in the computation of the shortest path distance between facilities and users. In all data sets, our algorithm significantly outperforms the competitor. Its CPU cost is at least 10 times better than that of **NAIVE**.

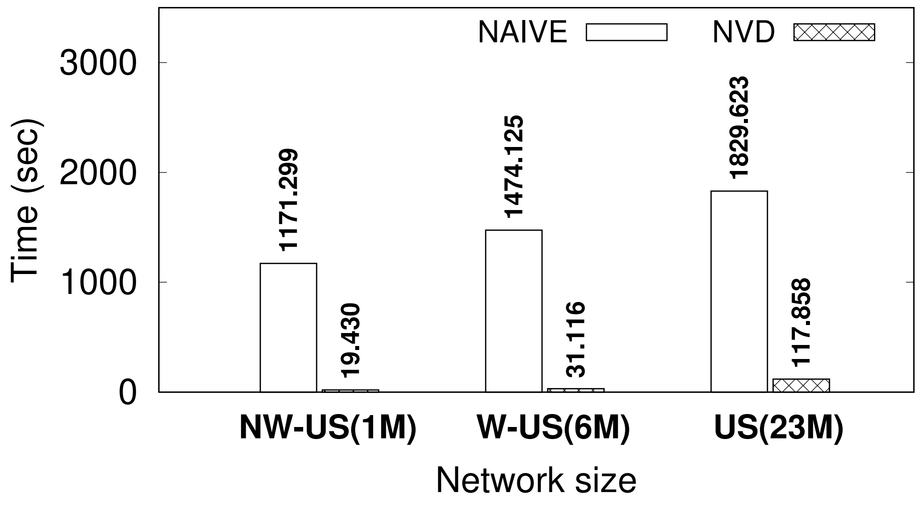


Figure 3.5: Effect of network size

Effect of number of facilities. In Fig. 3.6, we study the effect of the number of facilities on both algorithms. The cost of **NAIVE** is relatively stable on different number of facilities. This is because **NAIVE** checks the distance of every user to one facility only (the generator facility where the user resides) and compare it with the distance to the query point. Similarly, the cost of **NVD** is also relatively stable. We believe this is due to the fact the the observed ranges are quite low (0.3K-27.6K) so that the cost difference is not noticeable on different observed number of facilities. In all data sets, our algorithm is at least 30 times better than **NAIVE**.

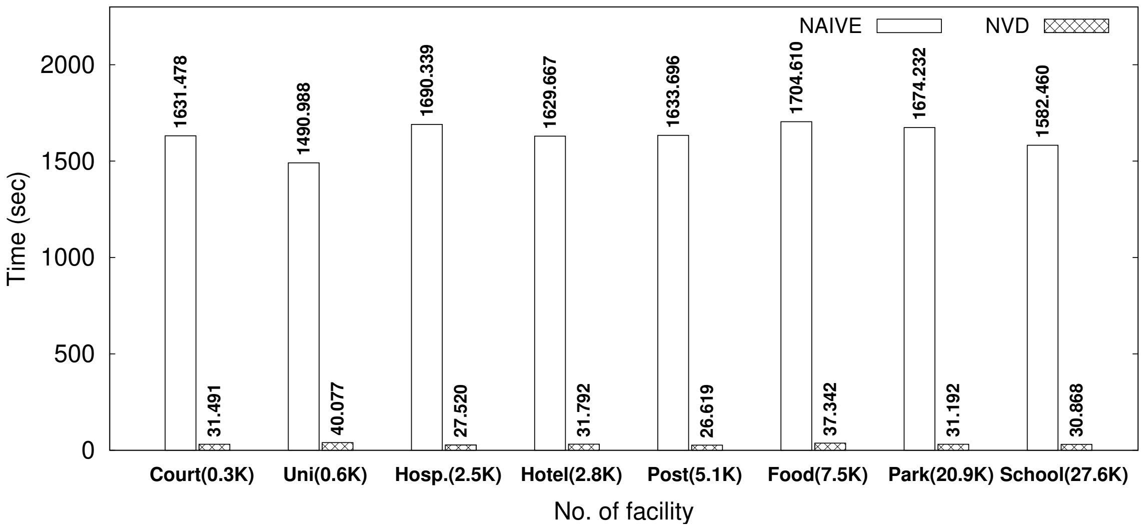


Figure 3.6: Effect of number of facilities

Effect of number of users. Fig. 3.7 shows the effect of number of users on both algorithms. The cost of both algorithms increases with the increase of the number of users. In **NAIVE**, more users in the graph causes the algorithm checks more users to verify if they are RANNs of the queries. Similarly in **NVD**, the higher number of users creates more dense *Voronoi* cells. Thus, in the verification phase, there are more users in the un-pruned cells that need to be evaluated. Our algorithm is up to two orders of magnitude better than the competitor and it also scales significantly better.

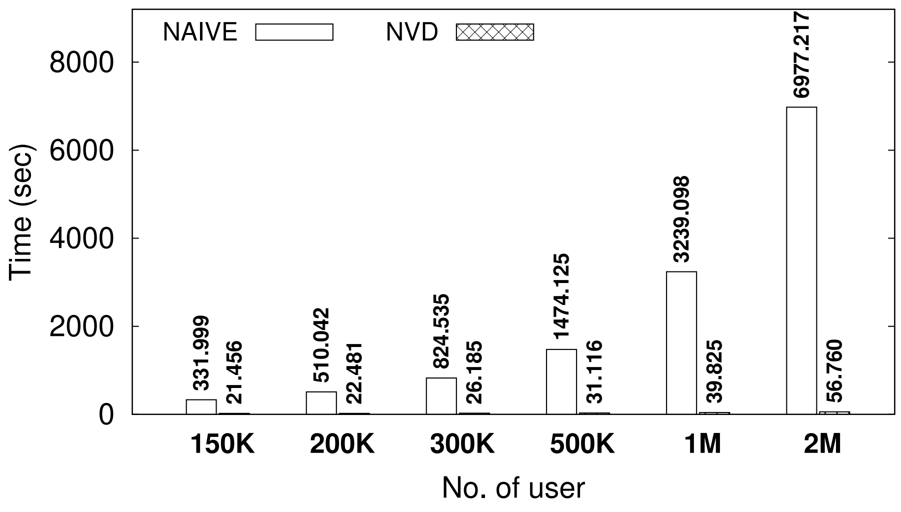


Figure 3.7: Effect of number of users

Effect of value of *x*. Fig. 3.8 studies the effect of x factor on the cost of both algorithms. The cost of **NAIVE** is relatively stable on different values of x. This is because **NAIVE** evaluates every user in the graph regardless of the value of x. On the other hand, the cost of **NVD** increases with the increase of x factor because less *Voronoi* cells are pruned when the value of x is bigger. The higher value of x, the higher number of un-pruned cells and hence the higher number of users to be verified. In all values of x, our algorithm significantly outperforms the competitor.

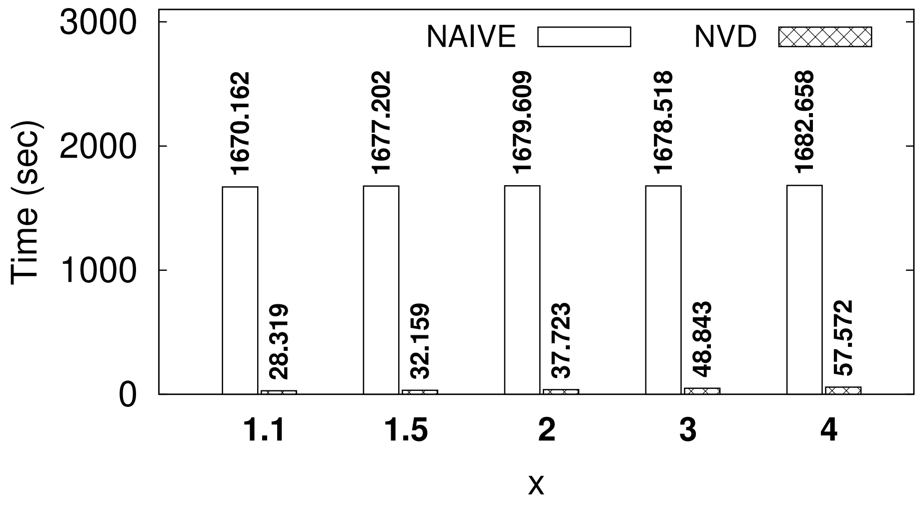


Figure 3.8: Effect of x value

3.5 Conclusion

In this project, we propose techniques and algorithm to solve RANN queries on road network. We show that existing algorithm for RANN queries on *Euclidean* space is not applicable for queries on road network. We introduce our NVD based pruning technique to reduce the computation cost of RANN queries. Our extensive experimental study on real data sets show that our algorithm is significantly better than the competitor.

Chapter 4

Continuous RANN Queries On Road Network

4.1 Overview

In this chapter, we are going to discuss the Moving RANN query on road network. As happening in contemporary world, more and more devices in our daily life have been installed with position locators such as cellphones, cars, computers, etc. When people use their electronic devices, they could stay at one spot or keep moving like a person is calling some while he is driving. For the moving cases, the position locating services offered by network provider will update the users current coordinates at different timestamp. Since there are billions mobile devices, handling the geo-location data updating is essential. Hence, the moving queries can play a vital role to improve the data processing efficiency and moderate the server workload. In this chapter, we study the continuous RANN query on road network.

This chapter is organised as follows. In section 4.2, we clarify the limitation and definition of monitoring moving RANN query on road network. In the subsequent section 4.3, two efficient algorithms are illustrated. The first one adopts a safe zone technique while the second method uses the influence zone technique. Both algorithms are analysed and examined in the following experiment evaluation in section 4.4. The final section is the conclusion.

4.2 Background Information

In this section, we introduce some background information of continuous monitoring RANN queries on road network. RANN queries on *Euclidean* space have been studied in [37, 38] as a complement for reverse k nearest neighbour (RkNN) query. Formally, given a set of facilities F, a set of users U, a query facility $q \in F$ and a number $x > 1$, every user $u \in U$ is a RANN of q if $dist(u, q) \le x * dist(u, f)$, where $dist(u, q)$ and $dist(u, f)$ are the shortest road network distance from u to q and from u to f respectively.

Consider four restaurants ($f_1 - f_4$) and five users ($u_1 - u_5$) in Fig. 4.1. If R2NN query is issued from f_3, it returns u_2 and u_3, which means in the context of R2NN, only those two users that are influenced by f_3. However, we argue that u_3 is actually influenced by f_3 as well, because u_3 that has to travel $82Km$ to reach its second closest facility (f_2) may be willing to travel slightly farther ($3Km$) to reach its next closest facility (f_3). This example shows that influence definition used in RkNN query only considers relative ordering of the facilities according to their distances to the user. It ignores the actual distance between facilities and users. RANN queries was introduced in [37, 38] as the complement of RkNN query by taking into consideration the actual distance between facilities and users.

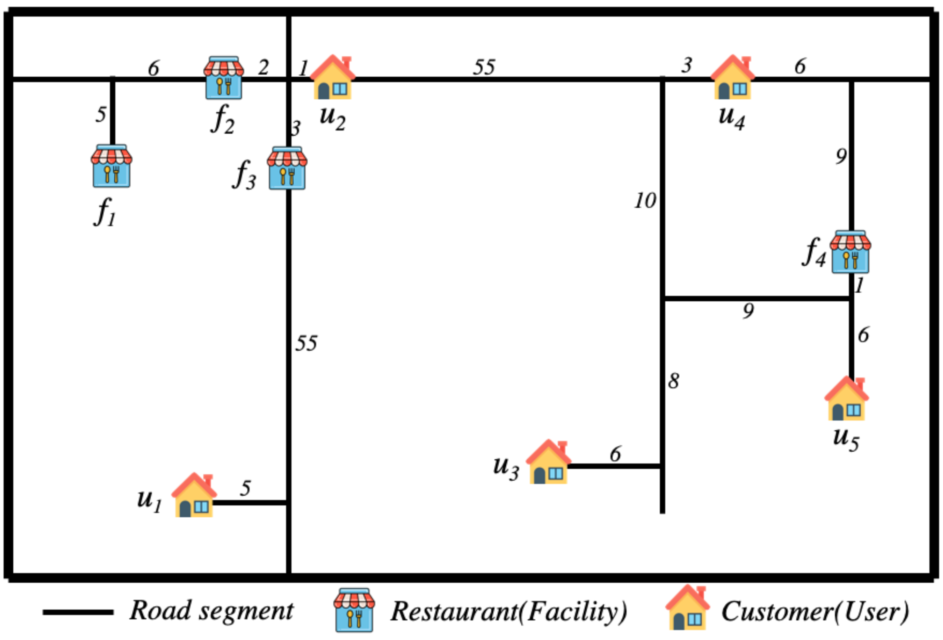


Figure 4.1: Example of RkNN and RANN

To the best of our knowledge, there was no study on continuous RANN queries on road network. Due to the different characteristics between *Euclidean* space and road network, the existing algorithm for continuous RANN queries on *Euclidean* space [37, 38], cannot be directly extended to road network environment. In this chapter, we propose two novel methods to efficiently monitor RANN queries on road network. The first method uses *safe zone* approach whereas the second method is based on *influence zone*.

4.2.1 Problem Definition

RANN queries can be classified into *monochromatic* and *bi-chromatic* queries. This research focuses on *bi-chromatic* RANN queries, as this version has more real-world applications. Nevertheless, our techniques can be easily applied to *monochromatic* RANN queries. In *bi-chromatic* queries setting, objects are categorized into two, namely facility and user. Facilities are the objects that provide services such as gas stations, restaurants, supermarket etc while users are customers that use those services. In the rest of this chapter, the RANN query always refer to *bi-chromatic* RANN query whenever this terminology is mentioned.

We study continuous RANN queries in road network environment. Road network is represented as a graph consists of edges and vertices. Edges represent road segments whereas vertices denote all intersections, dead-ends, users and points of interest (POI). POIs include all facility objects that provide services. Below we present the formal definition of continuous RANN queries.

Continuous RANN Queries. RANN query returns all users who consider the query facility as their closest facility or almost as close to the users as their closest facility. Given a graph G representing road network, a set of facilities F, a set of users U, a set of queries Q and a value of $x > 1$, the problem of continuous RANN query is to continuously monitor the query results for every $q \in Q$ when one or more users move along G.

As mentioned in [37, 38], a user u is considered as a RANN of q if $dist(u, q) < x * dist(u, f)$, where $dist(u, q)$ is the distance between user u and query facility q and $dist(u, f)$ is the distance between user u and its nearest facility f. In road network graph, the distance between two points a and b is measured as the length of the shortest path from vertex a to vertex b. From this point onward, we use $dist(a, b)$ to denote the shortest path distance between vertex a and b on the corresponding graph.

Due to the characteristics of road network which is inherently different from the *Euclidean* space, the existing algorithm [38] for continuous RANN queries on *Euclidean* space cannot be applied for queries on road network. For example, pruning and safe zone computation in [38] are based on *Euclidean* distance which measures the distance as the straight distance between two points. This is different from road network distance which uses the shortest path distance between two vertices in the graph. Hence, the existing algorithm for continuous RANN queries on *Euclidean* space is not applicable for continuous RANN queries on road network.

4.2.2 Contributions

Below, we summarize our contributions in this chapter.

- We propose *safe zone* based method and *influence zone* based method to efficiently monitor continuous RANN queries on road network
- We design and implement a comprehensive experiment using real data set to demonstrate the effectiveness of our algorithms.

4.3 Algorithms

In this section, we present our solution to efficiently monitor RANN queries on road network. Our solution uses a client-server approach to handle moving users. The clients send queries to the server and the server computes the initial query results and send them back to the clients. The server maintains the location of moving users and report to the client if the RANN of its query change.

A straightforward approach to continuously monitor RANN queries is to recompute them when there is at least one user that changes its location. However, this approach is costly since the server has to continuously recompute RANN of all queries. In addition, it will also incur considerable communication overhead because the server constantly sends the query results to all clients.

We propose two methods to significantly reduce the computation and communication costs of continuous RANN queries. The first method uses a *safe zone* that is assigned to every moving user. *Safe zone* of a user u is a set of road segments such that if u moves along these segments, the RANN of all queries remain unchanged. The second method is based on *influence zone* as proposed in [19]. *Influence zone* is a set of road segments that is assigned to each query such that if no user enters or leaves these road segments, the RANN of this query does not change. Both methods utilize *Network Voronoi Diagram* to efficiently locate the moving users and to efficiently compute the *safe zone* and the *influence zone*. The *NVD* is constructed in a pre-processing phase and is subsequently used to compute the initial query result. Both phases are detailed in the following sections.

4.3.1 Pre-processing

In this phase, we compute the *Network Voronoi Diagram(NVD)* of the observed graph to partition it into smaller sets of road segments. Each set is called a *Voronoi* cell which has a facility vertex as the generator. We use V_f to denote the *Voronoi* cell generated by facility vertex f. Computing NVD is independent from the queries and hence it will not affect the computation cost of our algorithm to answer the queries.

Given a *Voronoi* cell V_f, all users inside V_f consider f as their closest facility. Specifically, for any user u inside V_f, $dist(u, f) < dist(u, f')$ where f' is any other facility in the graph. Fig 4.2 shows an example of *Network Voronoi Diagram*. It consists of four *Voronoi* cells namely V_q, V_{f_1}, V_{f_2} and V_{f_3}, which are generated by four facilities q, f_1, f_2 and f_3 respectively. *Voronoi* cell borders are indicated with two red dashes. The nearest facility for u_1, u_3 and u_4 is q as they are inside the *Voronoi* cell generated by q (V_q). Similarly, the nearest facility of u_5 is f_3. In the rest of this chapter, we use *NVD* to denote the observed *Network Voronoi Diagram*.

For each *Voronoi* cell V_f generated by facility point f, we create a list to record all nodes inside V_f and their distance to f. This list is sorted in descending order according to the distance. Using this list, we can efficiently locate the nearest facility for each moving user. In addition, it is also useful to efficiently check if a given vertex belongs to a *Voronoi* cell. The maximum distance within each *Voronoi* cell is also maintained and will be used in the subsequent pruning phase.

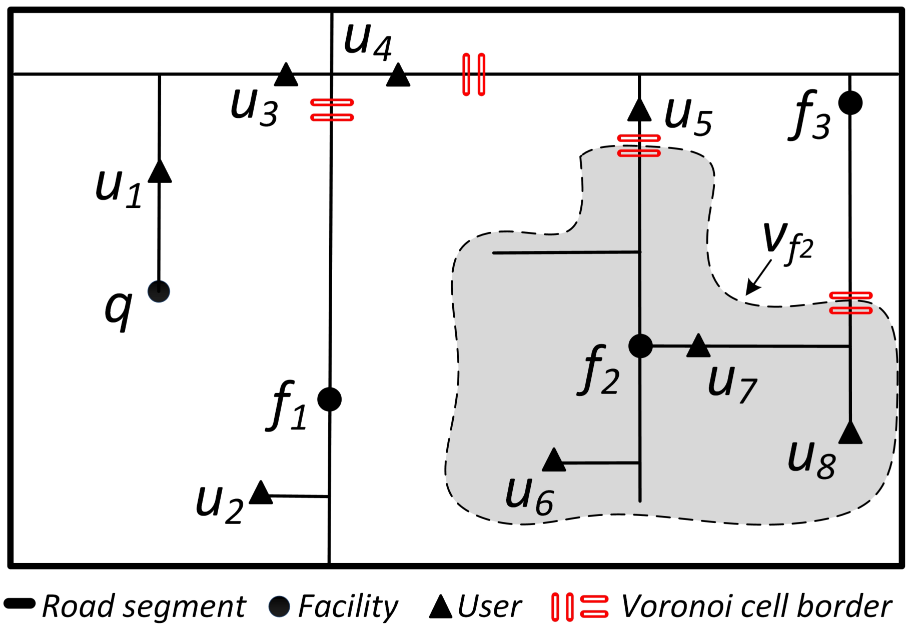


Figure 4.2: *NVD* of four facilities (q, f_1, f_2, f_3)

Once the *NVD* is constructed, it is used to efficiently compute the query result and to continuously monitor it. In our experiment, we use algorithms presented in [55] to utilize the *NVD* to compute initial RANN of queries. The algorithms consist of pruning and verification phases to significantly reduce the computation cost. Some pruning rules in [55] are relevant to compute the *safe zone* and hence will be re-used here.

4.3.2 Safe zone based method

In this section we present our first solution which is based on safe zone approach. In this method, a safe zone will be assigned to each user. The safe zone will be updated only when the user leaves it.

Computing safe zone

The safe zone of a user consists of a set of road segments such that if the user moves on those segments, it does not affect the result of all queries. A straight forward approach to compute the safe zone for a user u is to perform road network expansion and check if each road segment on the graph is part of the safe zone of u. A road segment is considered to be part of the safe zone of u if any move from u along this segment does not change the result of the queries.

Performing network expansion for each single user is computationally expensive. If the cost to check whether a road segment is part of a safe zone is C, then the worst case complexity for computing the safe zone is $O(UQEC)$, where U is total users, E is the number of edges in the observed graph and Q is the number of queries in the system. This is not to mention the cost to update the safe zone if a new query is added to the system. We propose our techniques to significantly reduce the cost of computing the safe zone for all users. The techniques are presented below.

Lemma 4.3.1. *Given a query q, a user u with its nearest facility f and a Voronoi cell generated by f (V_f), if V_f can be pruned, then V_f is the safe zone of u with respect to q.*

Proof: If V_f can be pruned (according to the pruning rule in [55]), then all users inside V_f are non-result users, i.e. they are not RANNs of q. Any move from any of these non-result users u will not make u to be the query result of q, as long as u stays in V_f. Therefore, V_f is the safe zone of u with respect to query q.

Consider an example in Fig 4.2. The shaded area is the *Voronoi* cell for the facility f_2 and it can be pruned according to the pruning rule in [55] . Users u_6, u_7 and u_8 can move to any position in this area and the result of query q remains the same. The shaded area is the safe zone for u_6, u_7 and u_8 with respect to query q.

Given a user u, we use *NVD* to locate the cell that contains it and use Lemma 1 in [55] to check if the cell is pruned. Both operations can be done in O(1). With Lemma 4.3.1, when a user inside a pruned cell is moving, as long as it stays inside its *Voronoi* cell, we do not need to recompute its safe zone. This will significantly reduce the monitoring cost since computing safe zone requires the algorithm to perform road network expansion which is computationally expensive.

Lemma 4.3.1 defines the safe zone for all users inside pruned cells. Next, we present our technique to compute the safe zone for users that do not reside in pruned cells. A straightforward approach to get the safe zone for such user is to access all road segments surrounding it and check if they are safe for u. A road segment is an edge on road network graph which connects two vertices. We use $l(a,b)$ to denote a road segment that is represented as an edge connecting vertex a and vertex b on the observed road network graph. Given a user u and a road segment $l(a,b)$,

$l(a, b)$ is safe for u with respect to a query q if u moves along between a and b and the RANN of q remains the same.

A simple method to check if a road segment $l(a, b)$ is safe with respect to a query q is to do trial and error where we repeatedly slide a user point u between a and b and check if the result of q changes. However, this method is considerably expensive as we have to do infinite number of trials due to infinite number of possible points between a and b, We present Lemma 4.3.2 and Lemma 4.3.3 to efficiently check if a road segment is safe with respect to a given query.

Lemma 4.3.2. *Given a query q, a facility point f, a road segment $l(a, b)$ where a and b consider f as their nearest facility, if a and b are RANNs of q, any point u on $l(a, b)$ is the RANN of q and therefore $l(a, b)$ is a safe road segment with respect to q*

Proof: If a and b are RANN of q, $dist(a, q) \leq x \cdot dist(a, f)$ and $dist(b, q) \leq x \cdot dist(b, f)$. Since u is on $l(a, b)$, the shortest path from u to q and from u to f must go through a or b. If it passes through a, then $dist(u, q) = dist(u, a) + dist(a, q)$ and $dist(u, f) = dist(u, a) + dist(a, f)$. Since $x > 1$ and a is RANN of q, then $dist(u, a) + dist(a, q) \leq x \cdot (dist(u, a) + dist(a, f))$ which completes the proof. If the shortest path passes through b, it can be proved in the similar way

Consider a query q and a facility f in Fig 4.3. The area that can be pruned by f is shown in dotted lines, i.e any user on these road segments cannot be RANN of q. Two vertices a and b on $l(a, b)$ are both RANNs of q. According to Lemma 4.3.2, any user on $l(a, b)$ is a RANN of q. Lemma 4.3.2 defines if a road segment is safe if both end vertices of the segment are RANNs of q. Next, we present Lemma 4.3.3 to check if a road segment is safe if both end vertices of the segment are not RANNs of the query.

Lemma 4.3.3. *Given a query q, a facility point f, a road segment $l(a, b)$ where a and b consider f as their nearest facility, if a and b are not RANNs of q and it holds $dist(a, f) = dist(a, b) + dist(b, f)$ or $dist(b, f) = dist(b, a) + dist(a, f)$, any user u on $l(a, b)$ is not a RANN of q and hence $l(a, b)$ is a safe road segment with respect to q.*

Proof: If a and b are not RANNs of q, then $dist(a, q) > x \cdot dist(a, f)$ and $dist(b, q) > x \cdot dist(b, f)$. Since u on $l(a, b)$, the shortest path from u to q and from u to f must go through a or b. If it passes through a, then $dist(u, q) > dist(u, a) + dist(a, q)$ and $dist(u, f) = dist(u, a) + dist(a, f)$. Since $x > 1$ and a is not a RANN of q, then $dist(u, a) + dist(a, q) > x \cdot (dist(u, a) + dist(a, f))$. It shows that u is not a RANN of q. If the shortest path passes through b, the lemma can be proved in the same way.

Lemma 4.3.2 and 4.3.3 check if a road segment is safe with respect to a query q. Given a query q and a road segment $l(a, b)$, we check if a and b are either both RANNs or both are not RANNs of q, and accordingly we can define if $l(a, b)$ is a safe road segment with respect to q. To better improve the efficiency of the method, we present Lemma 4.3.4 to quickly check if both end vertices of a road segment are RANNs or not RANNs of a given query.

Lemma 4.3.4. *Given a query point q, a facility point f and two non-facility vertices a and b which consider f as their nearest facility, if a is RANN of q and $dist(b, f) = dist(b, a) + dist(a, f)$, then b is RANN of q. Similarly, if a is not RANN of q and $dist(b, f) = dist(b, a) + dist(a, f)$, then b is not RANN of q.*

Proof: If a is RANN of q, $dist(a, q) \leq x \cdot dist(a, f)$. Since $dist(b, f) = dist(b, a) + dist(a, f)$, then $dist(a, q) + dist(a, b) \leq x \cdot (dist(b, a) + dist(a, f))$ or $dist(a, q) + dist(a, b) \leq x \cdot dist(b, f)$ because $x > 1$. According to the triangle inequality, $dist(b, q) < dist(a, b) + dist(a, q)$. Hence, $dist(b, q) < x \cdot dist(b, f)$, which proves that b is also a RANN of q. In the similar way, we can prove that if a is not a RANN of q and $dist(a, f) = dist(a, b) + dist(b, f)$, then b is not a RANN of q either.

An example is shown in Fig 4.4. Given a query point q, a facility f and two users u_1 and u_2, both u_1 and u_2 consider f as their nearest facility. The dotted lines show the area that can be pruned by f, i.e. any user on these lines cannot be RANN of q. For u_1, $dist(u_1, q) < x \cdot dist(u_1, f)$ and therefore it is a RANN of q. Since it also holds that $dist(u_2, f) = dist(u_2, u_1) + dist(u_1, f)$, then u_2 is a RANN of q as well. However, for the other two users u_3 and u_4, $dist(u_3, f) = dist(u_3, C) + dist(C, f)$ and $dist(u_4, f) = dist(u_4, M) + dist(M, f)$. In this example, even though u_4 is RANN of q, we cannot guarantee that u_3 is also RANN of q.

Lemma 4.3.1, 4.3.2 and 4.3.4 define the safe zone for a user with respect to a given query. Let U be the number of moving users in the system and C be the cost to compute a user's safe zone with respect to a query, it requires $O(UC)$ to compute the safe zone for all users in the system. Our observation shows that many users can actually share the same safe zone on certain conditions.

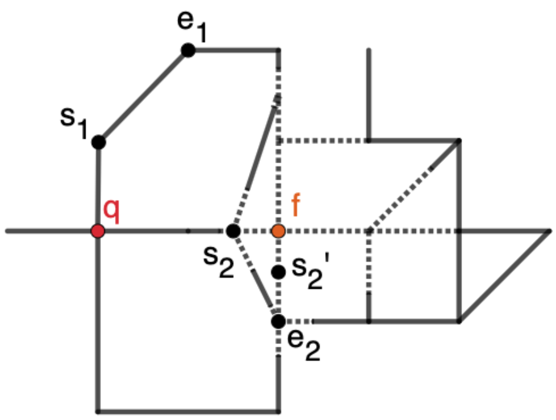


Figure 4.3: Lemma 4.3.2

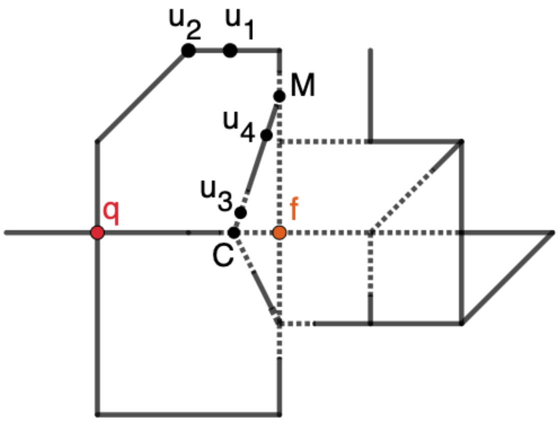


Figure 4.4: Lemma 4.3.4

We formally present this observation in Lemma 4.3.5 to significantly reduce the cost of computing safe zone for all users in the system.

Lemma 4.3.5. *Given a query point q and two users u_1 and u_2 inside the Voronoi cell of facility f (V_f), if both u_1 and u_2 are RANNs of q or both u_1 and u_2 are not RANNs of q, then u_1 and u_2 have a same safe zone with respect to q.*

Proof: Let P_f be a set of road segments in V_f that can be pruned by f and $\mathcal{A}_u$ be the safe zone of u containing a set of safe road segments for u in V_f. If u_1 and u_2 are both RANNs of q, then u_1 and u_2 are not in P_f. Hence, the safe zone of u_1 and u_2 are $\mathcal{A}_{u_1} = \mathcal{A}_{u_2} = V_f - P_f$. On the other side, if u_1 and u_2 are not RANNs of q then both u_1 and u_2 are in P_f and their safe zone are $\mathcal{A}_{u_1} = \mathcal{A}_{u_2} = P_f$.

Given the NVD of a graph and a query q, Lemma 4.3.1 - 4.3.4 can be used to compute the safe zone of a user u with respect to q. The details of how these Lemmas are used to get the safe zone is presented in Algorithm 3. The algorithm initializes an empty safe zone for u and gets the *Voronoi* cell (V_f) that contains u. It uses NVD index built in pre-processing phase to efficiently locate the cell. It then uses *internal distance* [55] to check if V_f is pruned. Note that *internal distance* is query independent and is computed together with NVD in the pre-processing phase. If V_f is pruned then V_f is returned as the safe zone of u (line 3).

If V_f is not pruned, the algorithm inserts u in a stack S. All connected vertices to u are then retrieved and those which are inside V_f are inserted to S. For each connected vertex v, the algorithm checks if e and v are either both RANNs of q or both are not RANNs of q (line 10 and 13). Then it checks if the road segment that connects e and v ($l(e,v)$) is safe with respect to q. If it is safe, road segment $l(e, v)$ is added to the safe zone of u (line 12). The algorithm stops when S becomes empty.

Algorithm 3: getSafeZone(u,q)

```
1 Input: $NVD$ of given graph, a query $q$, a user $u$ and its closest facility $f$
2 Output: safe zone of $u$ with respect to $q$ ($\mathcal{A}_{u:q}$)
 1: $\mathcal{A}_{u:q} \leftarrow \phi$
 2: if $V_f$ is pruned then
 3:     $\mathcal{A}_{u:q} \leftarrow V_f$                         ▷ Lemma 4.3.1
 4: else
 5:     insert $u$ in a stack $S$
 6:     while $S$ is not empty do
 7:         pop an entry $e$
 8:         for each vertex ($v$) connected to $e$ do
 9:             if $v$ is inside $V_f$ then
10:                 if $e$ and $v$ are RANNs of $q$ then      ▷ Lemma 4.3.4
11:                     if $l(e, v)$ is safe then              ▷ Lemma 4.3.2
12:                         insert $l(e, v)$ to $\mathcal{A}_{u:q}$
13:                 else if $e$ and $v$ are not RANNs of $q$ then   ▷ Lemma 4.3.4
14:                     if $l(e, v)$ is safe then              ▷ Lemma 4.3.3
15:                         insert $l(e, v)$ to $\mathcal{A}_{u:q}$
16:                 insert $v$ in $S$
17: return $\mathcal{A}_{u:q}$
```

Algorithm 3 summarizes our technique to compute a safe zone for a moving user. Now, we are ready to present our algorithms to continuously monitor the RANNs of all queries. First, we show how to handle a new issued query (section 2). Then, we present our algorithm to handle the case when a new user is added to the system (Section 2). In Section 2, we present our algorithm to handle the case when a query or a user is removed from the system. Finally, we present our algorithm to handle the case when one or more users change their locations (Section 2).

Add query

We use the term in [38], where a facility f is marked *insignificant* for a query q if all nodes in the *Voronoi* cell of f are not the RANN of q. Otherwise, this facility is called a *significant* facility for q. For each facility f, a list called $siglist_f$ will be maintained to record all the query points that consider f as a significant facility. In addition, for each user we also maintain a list called *qlist* to record all queries for which the user is their RANN. Specifically, for each user u, $qlist_u$ stores all queries that have u as one of their RANNs.

We use a map to store all computed safe zone in the system. The key for entries in this map is a pair (q, f) where q and f are the query and facility numbers respectively. The value that is mapped to the key is the safe zone for corresponding query and facility. Given a query q and a facility f, an entry of the map $(q, f) - Z_{(q,f)}$ is a key-value pair that maps the key (q, f) to the safe zone $Z_{(q,f)}$. We use (q, f) as the key to quickly find the safe zone for all users inside V_f that become RANNs of q. We denote the map that store these entries as *safemap*. With *safemap*, Lemma 4.3.5 can be implemented efficiently.

Algorithm 4 details our technique to handle a new issued query. When a new query point q is added into the system, the algorithm will compute its initial RANN set. For each user u in this set, q is added to $qlist_u$. The algorithm locates the nearest facility (f) to u and insert q into $siglist_f$. It then checks if key (q, f) exists in the $safemap$. If the key exists, its value $Z_{(q,f)}$ is assigned to be the safe zone of u. Otherwise, the key (q, f) is created and $Z_{(q,f)}$ is computed using Algorithm 4.3.1. The pair of $(q, f) - Z_{(q,f)}$ is then added into the *safemap*.

Algorithm 4: addQuery(q)

```
1 Input: $NVD$ of given graph and a query $q$
2 Output: updated $safemap$
   1: compute $RANN_q$
   2: for each $u \in RANN_q$ do
   3:    add $q$ to $qlist_u$
   4:    get nearest facility $f$ to $u$
   5:    add $q$ to $siglist_f$
   6:    if $(q, f) \in safemap$ then
   7:       $\mathcal{A}_{u:q} \leftarrow Z_{(q,f)}$                    ▷ Lemma 4.3.5
   8:    else
   9:       $Z_{(q,f)} \leftarrow getSafeZone(u, q)$                    ▷ Alg 3
  10:       insert $(q, f) - Z_{(q,f)}$ to $safemap$
```

Add user

Algorithm 5 presents our technique to handle the case when a new user is added to the system. When a user u is added, the algorithm uses NVD to find its nearest facility f. It then opens $siglist_f$ to check if for each $q \in siglist_f$, u is a RANN of q. If u is a RANN of q, the algorithm checks if $Z_{(q,f)}$ is in the *safemap*. If it is, $Z_{(q,f)}$ is assigned as the safe zone of u, otherwise it calls Algorithm 3 to compute the safe zone of u and insert it in the $safemap$ for future reference.

If u is not a RANN of q, the *Voronoi* cell of f (V_f) is assigned as the safe zone of u. Likewise, if $siglist_f$ is empty, all queries in the system consider f as insignificant facility and hence V_f can be set as the safe zone of u.

Algorithm 5: AddUser(u)

```
1 Input: $NVD$ of given graph and a user $u$
2 Output: updated $safemap$
 1: get nearest facility $f$ to $u$
 2: if $siglist_f \neq \phi$ then
 3:     for each $q \in siglist_f$ do
 4:         if $q$ is RANN of $q$ then
 5:             if $(q, f) \notin safemap$ then
 6:                 $Z_{(q,f)} = \mathcal{A}_{u:q} \leftarrow getSafeZone(u, q)$
 7:                 insert $(q, f) - Z_{(q,f)}$ to $safemap$
 8:             else
 9:                 $\mathcal{A}_{u:q} \leftarrow Z_{(q,f)}$
10:         else
11:             $\mathcal{A}_{u:q} \leftarrow V_f$
12: else
13:     $\mathcal{A}_{u:q} \leftarrow V_f$
```

Delete a query or a user

When a query q is deleted, the algorithm checks the *siglist* of each facility and remove q from the list. It also iterates over each user u and deletes q from $qlist_u$. Finally, all safe zones in the *safemap* that involves q will be deleted. When a user u is deleted from the system, the algorithm opens $qlist_u$ and for each q in this list, u will be removed from $RANN_q$.

Handling user movement

If a user u stays in its safe zone, its movement will not affect the query result. In our solution, u only sends its location to the server when it moves out of its safe zone. When this happens, the algorithms will first deletes u from the system, then it calls $addUser(u)$ to add it back with its new location.

4.3.3 Influence zone based method

In this section, we present our second approach to continuously monitor RANN queries. This approach is based on *influence zone* presented in [19]. Given a query q, *influence zone* of q is a set of road segments such that every user on this segments is guaranteed to be a RANN of q. We can continuously monitor RANNs of a query by checking its *influence zone*. The RANN of a query q will not change if there is no user enters or leaves the *influence zone* of q. The location of facilities in the data set does not change, and hence the *influence zone* of queries also remain the same. Once *influence zone* of a query is set, there is no requirement to update it regardless of the movement of the users.

Computing influence zone

The influence zone of a query q (denoted as $\mathcal{B}_q$) can be computed with the similar method as used in computing safe zone. Given a query q, we create a list ($unpruned_q$) to store all facilities whose $Voronoi$ cell cannot be pruned (refer to pruning rule in [55]). For each facility f in $unpruned_q$, we perform road network expansion started from f. Using Lemma 4.3.2, each road segment connected to f is iteratively checked if it can be inserted to $\mathcal{B}_q$. The steps to compute *influence zone* of a query are presented in Algorithm 6.

During road network expansion, it may happen that only some parts of a road segment that can be inserted to the *influence zone*. In this case, we still insert the whole road segment together with the length of the parts that belong to the *influence zone*. Consider an example of a query q, a road segment $l(a,b)$ and a point p on $l(a,b)$. If only segment $l(a,p)$ that belongs to $\mathcal{B}_q$, then we insert $l(a,b)$ together with $dist(a,p)$ to $\mathcal{B}_q$. This way, when a user u moves on $l(a,b)$, we check $dist(a,u)$ and compare it with $dist(a,p)$ to verify if u is inside the $\mathcal{B}_q$.

Add query or user

An influence zone is computed and assigned once during the life time of a query. Adding a new query to the system will not affect existing queries' influence zone. When a new query q is added

Algorithm 6: getInfluenceZone(q)

```
1 Input: $NVD$ of given graph and a query $q$
2 Output: influence zone of $q$ $\mathcal{B}_q$
 1: $\mathcal{B}_q \leftarrow \phi$
 2: for each facility $f \in unpruned_q$ do
 3:     insert $f$ in a stack $S$
 4:     while $S$ is not empty do
 5:         pop an entry $e$
 6:         for each user vertex $(v)$ connected to $e$ do
 7:             if $v$ consider $f$ as its nearest facility then
 8:                 if $q \in qlist_e$ and $q \in qlist_v$ then        ▹ Lemma 4.3.2
 9:                     insert $l(e, v)$ to $\mathcal{B}_q$
10:                 else if $q \in qlist_e$ and $q \notin qlist_v$ then
11:                     let $l(e, p)$ be the segment belongs to $\mathcal{B}_q$
12:                     insert $l(e, v)$ to $\mathcal{B}_q$ and store $dist(e, p)$
13:                 insert $v$ to $S$
14: return $\mathcal{B}_q$
```

into the system, its RANNs are computed and a list $unpruned_q$ is populated. Then, Algorithm 6 is called to compute the influence zone of q.

When a new user u is added, the algorithm locates the closest facility f to u. It then opens $siglist_f$ and for each query q in $siglist_f$, it checks if u is inside the influence zone of q ($\mathcal{B}_q$). If u is inside $\mathcal{B}_q$, $RANN_q$ will be updated to include u.

Delete query or user

Deleting a query or a user in the influence zone based method is similar to the one in the safe zone based method. When a query q is deleted from the system, $qlist$ of each user that contains q will be updated to remove q. Similarly, q will be deleted from $siglist$ of facilities and the influence zone of q will also be deleted. When a user u is deleted, the algorithm will iterate over RANN set of all queries and remove u from them.

Handling user movement

If a user stays in the influence zone of a query q, its movement will not change the RANN of q. An update to RANN of q is only required when there is a user that enters or leaves the influence zone of q ($\mathcal{B}_q$). When a user u leaves $\mathcal{B}_q$, the algorithm will remove u from $RANN_q$. It then checks if u enters the influence zone of other queries. If it does, RANN set of the corresponding queries will be updated to include u. The influence zone of all queries remain unchanged during their lifetime in the system.

4.4 Experiment

To the best of our knowledge, there was no work on continuous RANN queries on road network. We use naive algorithm (**NAIVE**) as the competitor for our safe zone based algorithm (**SAFEZONE**) and influence zone based algorithm (**INFZONE**). In **NAIVE**, at each timestamp, RANN of all queries are re-computed. Any addition or deletion of query or user will also call the algorithm to update the RANN of affected queries. In **SAFEZONE** and **INFZONE**, NVD is constructed and is used to efficiently locate the nearest facility to a given user. We implement *Pruned Highway Labelling (PHL)* [8] in both methods to get the shortest path distance between two vertices in the graph.

Table 4.1: Road Networks for moving RANN

Name	Region	No. of Vertices	No. of Edges
ME-US	Maine United States	187,315	412,352
COL-US	Colorado United States	435,666	1,042,400
NW-US	Northwestern United States	1,089,933	2,545,544

Table 4.2: Facility Sets for moving RANN

Facility type	Size (COL-US)
Parks	1,392
Fast Foods	717
Cafes	381
Banks	331
Hotels	252
Post Offices	186
Hospitals	148
Information	67

All algorithms are implemented in C++ running on Debian Linux on Nectar Elastic Cloud Computing (EC2) with 16 AMD EPYC processor (with IBPB) and 64 GB memory. The real dataset with different data size are shown in Table 4.1 [3]. Users are selected randomly from the graph. In addition, ten sets of POIs extracted from Open Street Map (OSM) are used as the facilities as shown in Table 4.2.

Table 4.3: Parameters of moving RANN

Parameter	**Ranges**
Network size	ME-US (190K), **COL-US (450K)**, NW-US (1M)
Number of facilities	Refer to Table 4.2
Number of users	5K, 10K, **50K**, 100K, 150K
Number of moving users (% of users)	20, 40, **60**, 80, 100
Speed (km/h)	40, 60, **80**, 100, 120
x (a multiplication factor)	1.1, **1.5**, 2, 3, 4

We vary the network size, number of facilities, number of users, the value of x (x is a multiplication factor), number of moving users and the speed of moving users and evaluate their effects on the performance of all algorithms. The parameters used in this experiment are shown in Table 4.3, with all default values set to bold. In each experiment unit, 100 query points are randomly selected from the facility set.

Effect of network size. Fig 4.5 studies the effect of network size on all algorithms. The initial cost of **SAFEZONE** and **INFZONE** are higher than the initial cost of **NAIVE** because both algorithm compute RANN of queries as well as the initial safe zone of all users or influence zone of all queries. On the other hand, **NAIVE** only computes the RANN of queries. The CPU cost of continuous RANN monitoring on our algorithms are at least 15 times lower than the CPU cost of **NAIVE** on all network size. This shows the effectiveness of both safe zone and influence zone methods. From this point onward, we only display the monitoring cost of all algorithms.

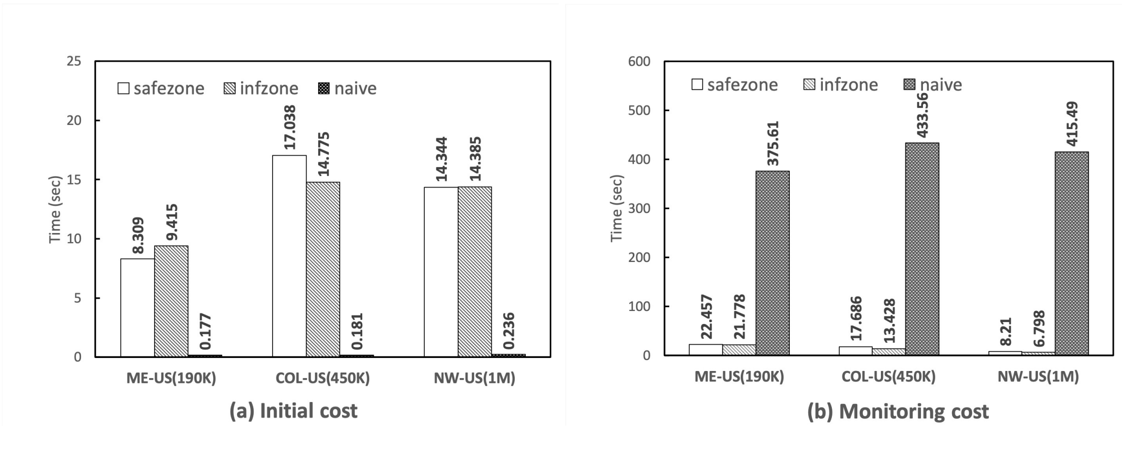


Figure 4.5: Effect of network size

Effect of number of facilities. In Fig 4.6, we study the effect of the number of facilities on all algorithms. Both **SAFEZONE** and **INFZONE** significantly outperform **NAIVE**. The monitoring cost of our algorithms is up to 45 times better than that on **NAIVE**. The monitoring cost of all algorithms are relatively stable on all facility size.

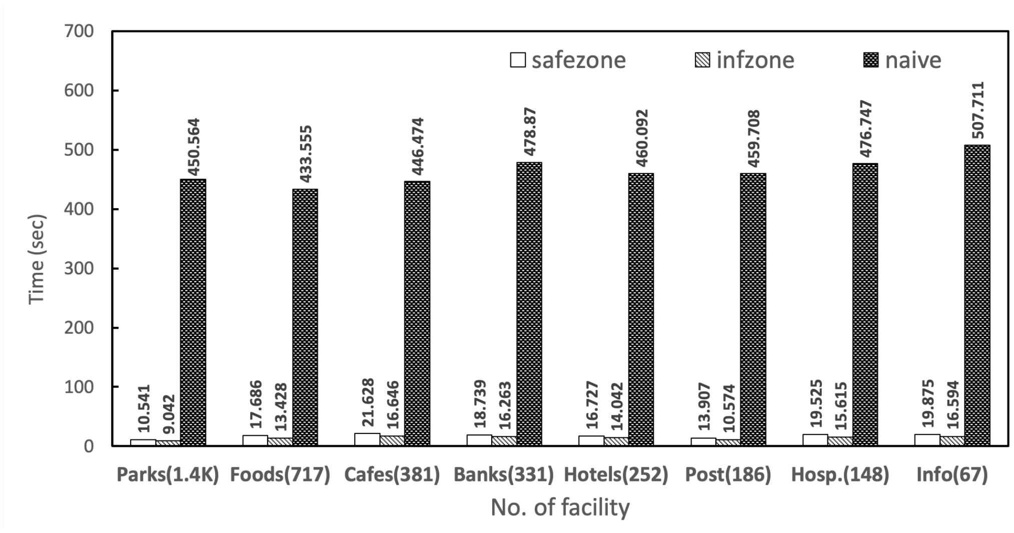


Figure 4.6: Effect of number of facilities

Effect of number of users. Fig 4.7 shows the effect of the number of users on the performance of all algorithms. The monitoring cost of all three algorithms increases as the increase of the number of users. In **NAIVE**, the higher number of users causes the algorithm to perform more checks to verify if the users are RANN of queries at each timestamp. Similarly, in **SAFEZONE** and **INFZONE**, the higher number of users leads to the more dense users in each *Voronoi* cell which will increase the cost of continuous monitoring. Our algorithm is up to two orders of magnitude better than the competitor and it also scales significantly better.

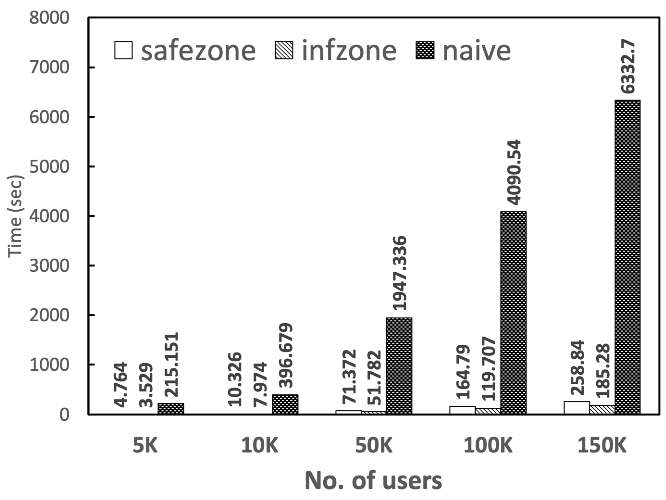


Figure 4.7: Effect of number of users

Effect of mobility. Fig 4.8 studies the effect of mobility, which corresponds to the percentage of the users that move between two timestamps. For example, 80 percent mobility refer to the data set where 80 percent of the total users change their locations between two timestamps. As expected, the monitoring cost of all algorithms increases with the increase of users' mobility.

In **NAIVE**, the higher number of users that are moving causes the higher number of RANN verification at each time stamp. In **SAFEZONE** and **INFZONE**, the more moving users causes the more users leaving their safe zone or the queries' influence zone. Both **SAFEZONE** and **INFZONE** significantly outperform **NAIVE**.

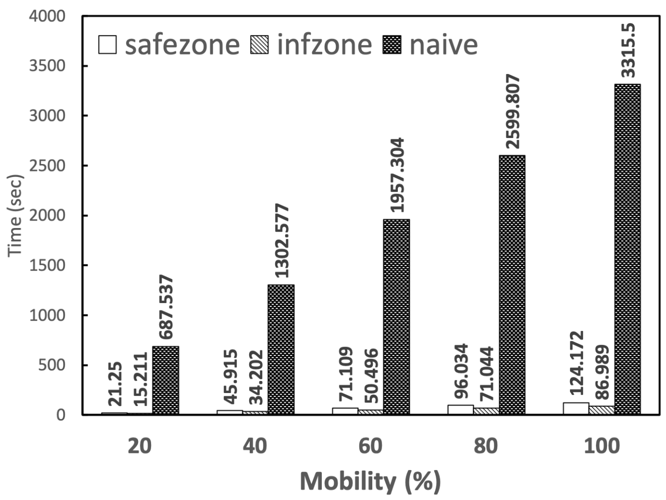


Figure 4.8: Effect of mobility

Effect of users' speed. Fig 4.9 demonstrates the effect of users' speed on the monitoring cost of all algorithms. When users move faster, they will leave the safe zone and the influence zone faster and will cause more frequent updates in the system. Hence, the cost of **SAFEZONE** and **INFZONE** increases with the increase of users' speed. Nevertheless, the change of speed does not affect the cost of **NAIVE**, as this method computes RANN of queries at each timestamp regardless of the users' speed. Both our algorithms are significantly better than **NAIVE**.

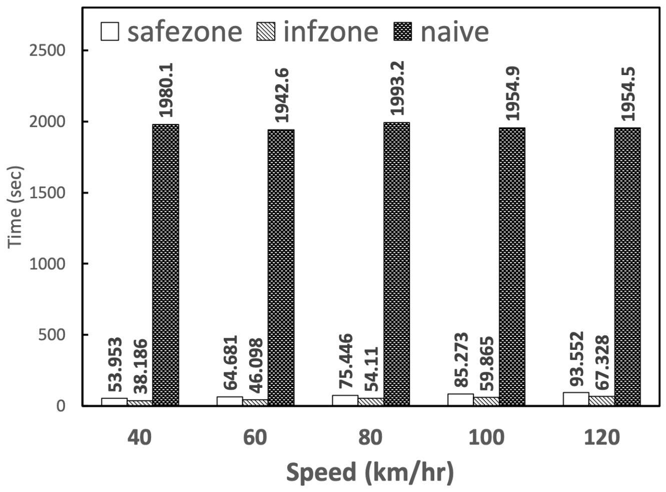


Figure 4.9: Effect of users' speed

Effect of *x* value. Fig 4.10 studies the effect of *x* on the monitoring cost of all algorithms. When the value of *x* increases, the size of the pruning area of a facility is getting smaller. It causes the less number of *Voronoi* cells that can be pruned. For **SAFEZONE** and **INFZONE**, increasing *x* value causes the increase of the number of significant facilities to be recorded for each query. During the monitoring of RANN queries, there will be more operation to check the users inside the *Voronoi* cell of those significant facilities. In all values of *x*, our algorithm significantly outperforms the competitor.

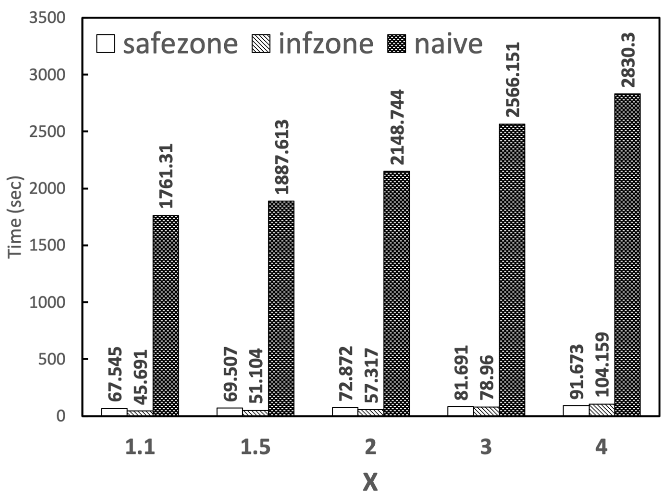


Figure 4.10: Effect of *x*

4.5 Conclusion

In this chapter, we propose techniques to efficiently monitor RANN of queries on road network. We also show that existing algorithms cannot be extended for continuous RANN queries on road network. We devise two algorithms based on the safe zone and influence zone concepts. Our extensive experiments on real data set show the effectiveness of our algorithms.

Chapter 5

Snapshot SRAT Queries

5.1 Overview

With the rapid development of technology, an increasing number of amenities are emerging to make life more convenient and enjoyable. With a reverse k nearest neighbour (RkNN) query [12, 84], all the nearby users that can be influenced by a query facility q will be returned. All the returned users consider the query facility q to be an important facility. Hence, this query can also be used to find the appropriate location for a new facility. Nevertheless, in RkNN, the concept of importance is only decided by the distance between users and facilities. In general, when people choose to access a facility, distance is not always the most significant criteria. Some other conditions like price, food rating, environment rating and so forth can also be meaningful. In this context, a spatial reverse top-k (SRTk) query is more advantageous. By using a given linear scoring function, the SRTk query returns users who consider the query q as one of the top-k facilities.

For example, given a person who is searching for restaurants, the factors that could influence his final decision could be the distance between his location and the restaurants, the price and the quality of the food. With these criteria, a top-k query can be issued and a scoring function which consists of these conditions can be used. The k restaurants whose scores are smaller than other objects will be returned as the result. If facility q is in the top-k result set, this person is influenced by q. Fig 5.1 illustrates an example of the SRTk query. In this graph, there are three

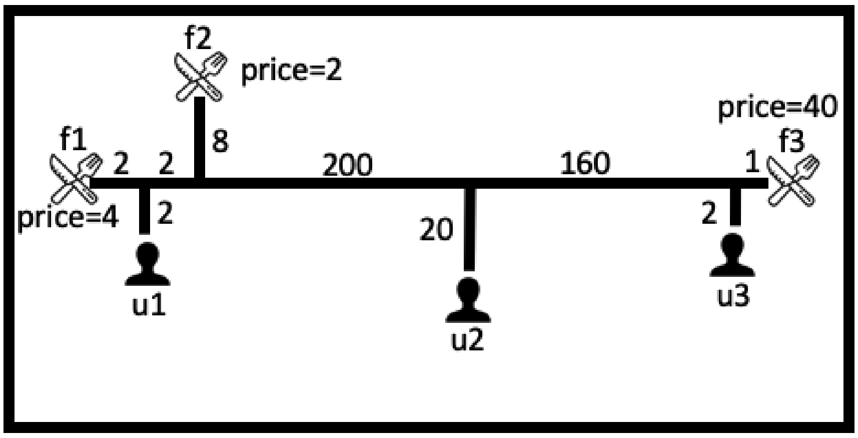


Figure 5.1: Example of SRTk query

restaurant facilities f_1 to f_3 and three users u_1 to u_3. Each facility has a price attribute. The distance between facilities and users is shown near the solid line. Assume the weight for price and distance are the same and denoted as $w[price] = w[distance] = 0.5$. Next, a score function $score(u, f) = w[distance] \times distance + w[price] \times price$ can be used to calculate the scores for all users with respect to different facilities. For instance, $score(u_1, f_1) = 4 \times 0.5 + 4 \times 0.5 = 4$, $score(u_1, f_2) = 12 \times 0.5 + 2 \times 0.5 = 7$, $score(u_1, f_3) = 365 \times 0.5 + 40 \times 0.5 = 202.5$. Hence, the top-1 facility for u_1 is f_1. Using a similar process, the top-k facilities for u_1, u_2 and u_3 can also be found.

After obtaining the top-k results, it is easy to identify the users who can be influenced by the same facility which is the SRTk query result. Referring to Fig 5.1, assuming $k = 1$, the SRT1 query results are $srtop - 1(f_1) = \{u_1, u_2\}$, $srtop - 1(f_2) = \{\}$ and $srtop - 1(f_3) = \{u_2, u_3\}$. There is an interesting phenomenon in the SRT1 result. The score of u_1 regarding f_1 and f_2 are 4 and 7 which are very close in value, whereas the SRT1 query result of f_2 does not have u_1. If $k = 2$, the SRT2 query results are $srtop - 2(f_1) = \{u_1, u_2, u_3\}$, $srtop - 2(f_2) = \{u_1\}$ and $srtop - 2(f_3) = \{u_2, u_3\}$. In this case, u_3 can be influenced by both f_1 and f_3. Nevertheless, there is a huge gap in the scores of u_3 with respect to f_1 and f_3 i.e. $score(u_3, f_1) = 185$ and $score(u_3, f_3) = 21.5$. This means result f_1 is redundant for u_3 as this user will not consider f_1 if f_3 is much better.

Usually, people prefer to go to the local main street or downtown where there is a cluster of facilities. Thus, we argue that all the restaurants that have similar scores could all be of interest to a user and the significantly unacceptable results should be abandoned. Therefore, we propose a spatial reverse approximate top (SRAT) query that relaxes the definition of influence, where a parameter x (called the x factor in this thesis) is adopted. This query considers the relative gap between the scores of users and facilities to obtain the result. To be more specific, a SRAT query returns every user *u* for which the query facility is in his approximate top facility set.

This chapter is organised as follows. In section 5.2, we analyse the limitations of SRT*k* query and formally define the SRAT query. In section 5.3, a novel algorithm is proposed to handle the SRAT query, with an optimised pruning phase and verification phase. A detailed experimental evaluation is illustrated in section 5.4. The final section presents the conclusion.

5.2 Background Information

In this section, the motivation for conducting this project is discussed. A formal definition of the SRAT query is given, based on a different definition of influence.

5.2.1 Limitations of the SRT*k* query and RANN query

To the best of our knowledge, this is the first time the SRAT query has been proposed and studied. Thus, no algorithm is able to solve this query directly. Several similar queries have been studied such as the SRT*k* query and RANN query. We examine the unique features of the SRAT query and discuss why the solutions for SRT*k* query and RANN query cannot be used to answer the SRAT query.

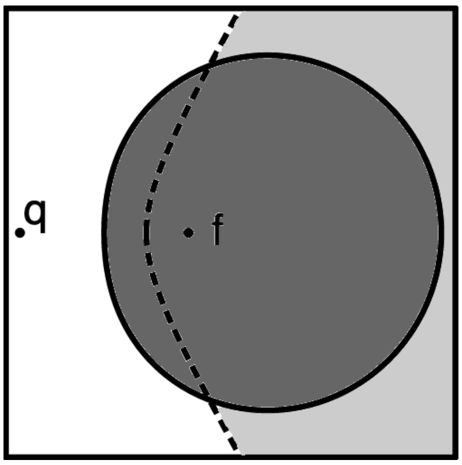


Figure 5.2: Hyperbola pruning technique limitation

In terms of the SRT*k* query, two approaches are used to prune some space to improve the query performance. One is hyperbola-based pruning and the other is SLICE region-based pruning.

However, according to our observations, when changing the k value to factor x, the pruning area will be changed to a closed curve. Unlike the SRTk query pruning techniques that can prune most of the space, the pruning for the SRAT query is much more challenging. Considering the example shown in Fig 5.2, the dashed line shows the pruning area in the SRTk query. All users belonging to the region in which facility f is located can be pruned. By using the definition of the SRAT query, the dark shaded area can be pruned by f, which is a closed curve but not a circle. In the SRAT query, the area that a facility can prune is much smaller than the SRTk query.

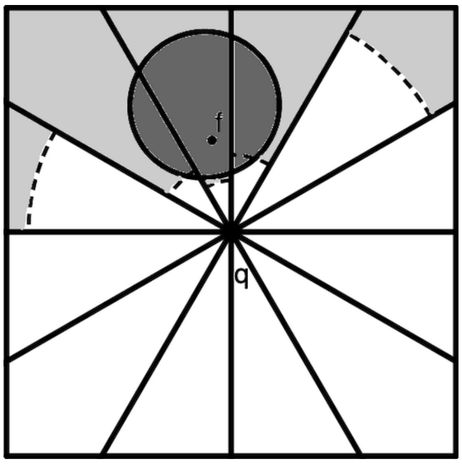


Figure 5.3: SLICE pruning technique limitation

When using the SLICE method, the graph is divided into small regions and more space can be pruned. As we can see from Fig 5.3, any user in the light shaded area can be pruned in the SRTk query. However, only a small area can be pruned using the SRAT query.

Because the RANN query only considers the distance between users and facilities, it can generate a pruning circle to decrease the searching space. Since the SRAT query is based on a score function and the pruning area of a facility f is not a circle, the RANN algorithm query cannot be used directly and the computation process of the SRAT query is far more complex. The RANN algorithm uses a R*-tree and a Voronoi diagram to index all the facilities. Based on the minimum distance from query q to a MBR and the maximum distance from a facility to its Voronoi cell corner points, many Voronoi cells can be pruned straightforwardly. Regarding the Voronoi diagram feature, all the users in a Voronoi cell consider this cell's generator facility as the nearest neighbour. However, in the SRAT query, not only distance is taken into consideration, but also other attributes, which requires the top-1 facility. So, the users in a Voronoi cell may not consider the generator facility as the top-1 facility. In short, the existing RANN method also has issues in dealing with the SRAT query.

5.2.2 Problem Definition

The SRAT query can be classified into *monochromatic* and *bichromatic* SRAT queries. In this chapter, we focus on the *bichromatic* version since it has more applications in real-world scenarios. In this version, objects consists of two categories of points, namely facility and user. Facilities are amenities or places that offer services, while users are the consumers who would like to use these services. In the rest of this chapter, the SRAT query refers to a *bichromatic* SRAT query.

Given a set of users U and a set of facilities F, each facility $f \in F$ has d attributes, i.e., price, ranking, and two coordinates to mark its location. Since the d attributes of each facility is same for every user, they can be called static attributes and the i-th dimension is noted as $f[i]$. The distance between a user $u \in U$ and a facility $f \in F$ is denoted as $dist(u, f)$. Because every user could have various positions, the distance to the same facility will be different. Thus, distance is recorded as a dynamic attribute of a facility. All the attribute values are normalised to within the range from 0 to 1. Furthermore, for each attribute, a weight value is given and denoted as $w[i]$. Assuming the sum of all dynamic and static attributes weight is 1, we have $\sum_{i=1}^{d+1} w[i] = 1, w[i] \geq 0$. The score function with $d + 1$ dimensions for a user u with respect to f can be shown as follows.

$$score(u, f) = w[d + 1] \cdot dist(u, f) + \sum_{i=1}^{d} w[i] \cdot f[i] \tag{5.1}$$

.

Definition 5.2.1. Spatial Approximate Top Facility. Let $Top1score(u)$ denote the score of the top-1 facility regarding u. Given a value of $x > 1$, a facility f is called a spatial approximate top facility of u if $score(u, f) < x \times Top1score(u)$.

Spatial Reverse Approximate Top (SRAT) query. Given a value of $x > 1$ and a query point q, a SRAT query returns every user u for which $score(u, q) < x \times Top1score(u)$, i.e., returns every user u who considers q as its spatial approximate top facility. The set of SRAT results for a query q is denoted as $SRAT_x(q)$. Note that a SRAT query is the same as a SRT1 query if $x = 1$. For instance, in Fig 5.1, assuming $x = 2$, the SRAT of f_2 are u_1 and u_2, i.e., $SRAT_2(f_2) = \{u_1, u_2\}$. Similarly, $SRAT_2(f_1) = \{u_1, u_2\}$ and $SRAT_2(f_3) = \{u_2, u_3\}$.

Due to the particular characteristics of the SRAT query, the existing RANN and SRTk algorithms cannot be applied directly. RANN algorithms are only based on distance, while many other attributes of the facilities also need to be considered. For SRTk techniques, even though the score function is utilised, factor x which will change the pruning space to a closed area cannot be handled properly.

5.2.3 Contributions

Below, we summarise our contributions in this project.

- We complement the SRTk query by proposing a new definition that a user u is influenced by a query q if q is one of the approximate top neighbours.
- The pruning techniques used to the solve SRTk query cannot be applied or extended to answer SRAT query. The main reason for this is, in our problem settings, the k value is relaxed to a x factor which means a user u cannot be influenced by two facilities that have a big difference in terms of scores. In accordance with several non-trivial observations, we propose an efficient algorithm that utilizes our pruning and verification techniques to efficiently solve the SRAT query.
- We conduct an extensive experimental study on both synthetic and real data sets to show the effectiveness of our algorithms. As this is the first time the SRAT query has been proposed, we compare our algorithm with a naive algorithm. The experiment results show that our approach is several orders of magnitude better than its competitor.

5.3 Answering Spatial Reverse Approximate Top (SRAT) queries

In this part, our algorithms to solve SRAT queries are elaborated. In section 5.3.1, we introduce and explain all the terminologies and notations used in this project. Next, our pruning techniques are demonstrated in section 5.3.2, followed by a detailed clarification of the algorithm in section 5.3.3.

5.3.1 Terms and notations

Before we discuss the algorithm, some terminologies and notations are clarified. In the linear score function we used, the sum of the static attribute values is the same for all users and only related to facilities. Hence, the Equation5.1 can be rewritten as the following one.

$$score(u, f) = w[d + 1] \cdot dist(u, f) + f_s \tag{5.2}$$

.

Table 5.1: Notations used for the SRAT query algorithm

Notation	**Definition**
$f[i]$	i-th attribute of a facility
$w[i]$	weight of i-th attribute
f_s	the static score of f ($\sum_{i=1}^{d} f_i \cdot w[i]$)
$score(u, f)$	$w[d + 1] \cdot dist(u, f) + f_s$
$Top1score(u)$	the $score(u, f)$ where f is the top-1 facility of u
Δ_f	$(x \cdot f_s - q_s)/w[d + 1]$
e	an entry of the facility R*-tree
e_s^{min} (e_s^{max})	minimum (maximum) static score of f in e
Δ_e^{min}	$(x \cdot e_s^{min} - q_s)/w[d + 1]$
Δ_e^{max}	$(x \cdot e_s^{max} - q_s)/w[d + 1]$
$maxdist(f, v)$	maximum distance between generator and Voronoi cell corners

In our algorithm, a R*-tree with $(d + 2)$ dimensions is used to index the facilities' coordinates and static attributes. Within the R*-tree, let e be an entry (intermediate or leaf node). The minimum (resp. maximum) static value of f in e is denoted as e_s^{min} (resp. e_s^{max}). Given a point p, the $maxdist(p, e)$ (resp. $mindist(p, e)$) refers to the maximum (resp. minimum) Euclidean distance between the entry e and the point p (only position coordinates are used in the distance computation). Except for R*-tree, another index called the Voronoi diagram is used simultaneously for pruning the space in the SRAT query. The generators of the Voronoi diagram are all the facilities. Each Voronoi cell whose generator is f has several corner points denoted as v_i. The maximum distance between f and its Voronoi cell corner vertex is denoted as $maxdist(f, v)$. Table 5.1 summarises the terms and notations used throughout this chapter.

5.3.2 Pruning techniques

Recall that a SRAT query returns every user u for which $score(u, q) < x \times Top1score(u), x > 1$. Let f be the top-1 facility of u and $Top1dist(u, f)$ denotes the distance between u and f. According to equation 5.2, the SRAT query definition can be re-written as the following expression.

$$w[d+1] \cdot dist(u, q) + q_s < x \cdot (w[d+1] \cdot Top1dist(u, f) + f_s) \tag{5.3}$$

.

This expression can be further converted to:

$$dist(u, q) - x \cdot Top1dist(u, f) < \frac{x \cdot f_s - q_s}{w[d+1]} \tag{5.4}$$

.

Let $\Delta_f = \frac{x \cdot f_s - q_s}{w[d+1]}$, a simplified expression can be generated.

$$dist(u, q) - x \cdot Top1dist(u, f) < \Delta_f \tag{5.5}$$

.

For each user, if it meets eq 5.5, it is the SRAT of q. Otherwise, it can be pruned by f. When the facility f is clear by context, the Δ_f will be simplified to Δ.

Corollary 5.3.1. A user u can be pruned by a facility f, if $dist(u, q) - x \cdot Top1dist(u, f) \geq \Delta$.

Since corollary 5.3.1 cna only check users who consider f as its top-1 facility, some users who are close to f but consider other ones as their top-1 facility are unsure. We need to find a way to also check these users. The subsequent lemma shows how to prune a user u with facility f even though f is not the top-1 facility of u.

Lemma 5.3.1. *If a user u is in the pruning area of f and u considers another facility f' as its top-1 facility, u cannot be the result of query q.*

Proof: Given a user u in the pruning area of f, according to the SRAT query definition, $score(u, q) > x \cdot score(u, f)$. Meanwhile, u considers f' as its top-1 facility, $score(u, f) > score(u, f')$. Then, we can have $score(u, q) > x \cdot score(u, f')$, which means u is not the result of q. Proved.

Based on corollary 5.3.1 and lemma 5.3.1, the following corollary can be written.

Corollary 5.3.2. A user u can be pruned by a facility f, if $dist(u, q) - x \cdot dist(u, f) \geq \Delta$.

In summary, as stated in the SRAT definition, a user u cannot be the SRAT of q, if it can be pruned by a facility f, which means $dist(u, q) - x \cdot dist(u, f) \geq \Delta$. By contrary, if and only if $dist(u, q) - x \cdot Top1dist(u, f) < \Delta$, a user u is the SRAT of q.

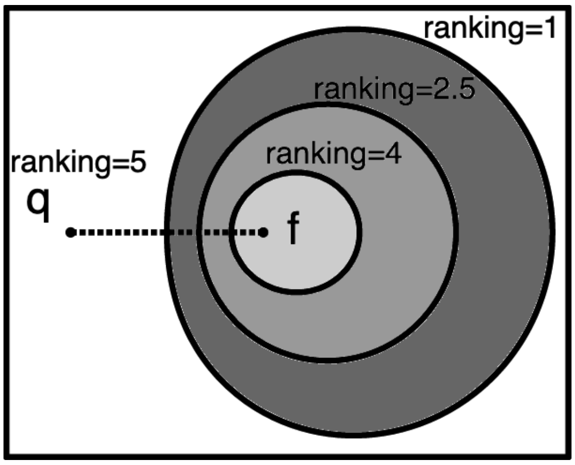


Figure 5.4: Example of pruning area based on the corollary 5.3.2

With inequality 5.5, a space bounded by a closed curve $dist(u, q) - x \cdot Top1dist(u, f) = \Delta_f$ can be generated. Any user inside the area of this closed curve can be pruned. An example can be found in Fig 5.4. A query point q and a facility f are given. Assuming factor $x = 2$, $q[ranking] = 5$, $f[ranking] = 4$ and $w[ranking] = w[distance] = 0.5$. Then $\Delta_f = \frac{2*4*0.5-5*0.5}{0.5} = 3$. The lightest shaded area in Fig 5.4 is the space which can be pruned by f when the $f[ranking] = 4$. If the ranking value of f is changed to 2.5, the $\Delta_f = 0$, generating a larger pruning area. By further decreasing the ranking value to 1, the $\Delta_f = -3$ and the pruning area becomes the darkest and largest curve enclosed space. Hence, we can identify that if $\Delta > 0$, the pruning area is an ellipse-like shape. If $\Delta = 0$, the pruning area is a circle. If $\Delta < 0$, the pruning area is a heart-like shape. Let q_s remain fixed, and with the increment of the static value, the pruning area of f will keep shrinking until $\Delta = dist(q, f)$ and no space can be pruned by f. In this case, all the users in the graph are the SRAT of q. This kind of facility is called a futile facility and the next lemma identifies the futile facilities.

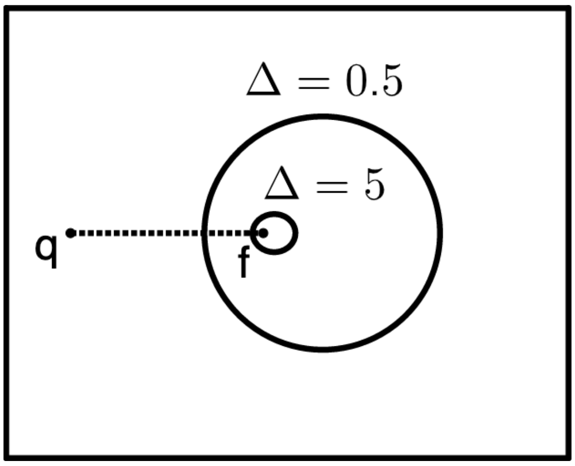


Figure 5.5: Example of futile facility

Lemma 5.3.2. *A facility f is a futile facility if $\Delta \geq dist(f,q)$.*

Proof: Given a user u can locate anywhere in the data space. According to the triangular inequality, $dist(u,q) - dist(u,f) < dist(f,q)$. Because $\Delta \geq dist(f,q)$, we can get $\Delta \geq dist(u,q) - dist(u,f)$. The given x factor is always greater than 1. Hence, $\Delta \geq dist(u,q) - x \cdot dist(u,f)$, which means any user u in the data space is the result of q. Proved.

Fig 5.5 shows an example of a futile facility. Given a query q and a facility f, the pruning area of f is shrinking when the value of Δ is becoming bigger. The pruning area of $\Delta = 0.5$ is much larger than the pruning space of $\Delta = 5$. When the value of $\Delta = dist(f,q)$, the pruning area will decrease to nothing. To be more specific, any user u in the space is the SRAT result of q. Since a facility R*-tree is used, according to lemma 5.3.2, more futile facilities can be identified in the R*-tree, which generates the following lemma.

Lemma 5.3.3. *Every facility $f \in e$ is a futile facility if $\Delta_e^{min} \geq maxdist(e,q)$.*

Proof: Because a R*-tree is used to index all facilities and $f \in e$, $dist(f,q) \leq maxdist(e,q)$ and $\Delta \geq \Delta_e^{min}$. Regarding lemma 5.3.2, if $\Delta_e^{min} \geq maxdist(e,q)$, each facility $f \in e$ has $\Delta \geq dist(f,q)$. Therefore, all facilities of e are futile facilities. Proved.

In our algorithm, all futile facilities are identified and hence all the users whose top1 facility is a futile facility are the SRAT result of q. Furthermore, a *Voronoi* diagram is used to separate the whole space into small pieces, with all generators being the facilities. All the users in a *Voronoi* cell consider this cell's generator as the nearest facility but maybe not the top1 facility. As presented in lemma 5.3.1, if a user u is in the pruning area of facility f, it cannot be the SRAT result of q.

If a *Voronoi* cell is covered by the pruning area of f, any users in this *Voronoi* cell can be pruned. Next, two techniques used to prune a single *Voronoi* cell are clarified.

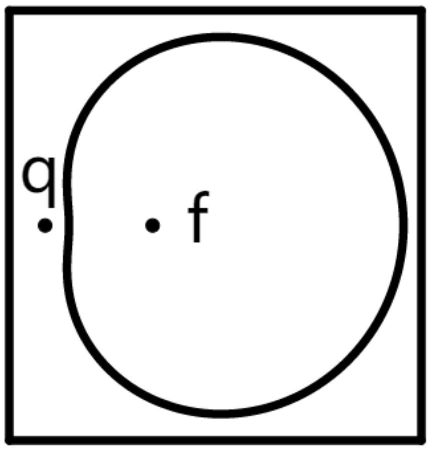


Figure 5.6: Pruning area of f when $\Delta < 0$

Lemma 5.3.4. *If* $\Delta > 0$ *and every corner* v_i *of a* Voronoi *cell can be pruned, this* Voronoi *cell can be pruned.*

Proof: Given $\Delta > 0$, the generated pruning curve is a convex curve. For any user u in a Voronoi cell with a generator f, there exist $dist(v_i, f) > dist(u, f)$ and $dist(u, q) > dist(v_i, q)$. Assuming v_i can be pruned, so $dist(v_i, q) - x \cdot dist(v_i, f) > \Delta$. Thus, $dist(u, q) - x \cdot dist(u, f) > \Delta$. Proved.

As shown in Fig 5.6, when $\Delta < 0$, the pruning area is enclosed by a non-convex curve. With this pruning area, even all the *Voronoi* cell corners can be pruned, but it is still possible to have some vertices remaining outside the pruning area. Except for checking all the corner points, the maximum distance from the corner points to generator f can be found and denoted as $maxdist(f, v)$. Then, we have the following lemma.

Lemma 5.3.5. *A whole Voronoi cell space whose generators is f can be pruned, if* $dist(f, q) - (x + 1) \cdot maxdist(f, v) > \Delta$.

Proof: Given a Voronoi cell has a generator f and the maximum distance to its corner points is $maxdist(f, v)$. For any user u in this Voronoi cell, $dist(u, f) \leq maxdist(f, v)$ and $dist(u, q) > dist(f, q) - maxdist(f, v)$. Assuming $dist(f, q) - (x + 1) \cdot maxdist(f, v) > \Delta$, we can get $dist(u, q) - x \cdot dist(u, f) > \Delta$. Proved.

As all the facilities are indexed in a R*-tree, the maximum value $maxdist(f, v)$ for all facilities in an R*-tree entry e can be computed and denoted as $maxMaxdist(e)$. Hence, in the facility traversal process, not only can a single facility be pruned but also a R*-tree entry, which significantly improves query performance.

Lemma 5.3.6. *The Voronoi cell whose generators is $f \in e$ can be pruned, if $mindist(e, q) - (x + 1) \cdot maxMaxdist(e) > \Delta_e^{max}$.*

Proof: Since a R*-tree is used to index all facilities and $f \in e$, $dist(f, q) \geq mindist(e, q)$ and $\Delta_e^{max} \geq \Delta$. Based on lemma 5.3.5, if $mindist(e, q) - (x+1) \cdot maxMaxdist(e) > \Delta_e^{max}$, for every f in e, we can get $dist(f, q) - (x+1) \cdot maxdist(f, v) > \Delta$, indicating every $f \in e$ can be pruned. Proved.

After pruning the facilities, all users in the *Voronoi* cells which cannot be pruned need to be verified to find the final SRAT query result. The next lemma explains how to examine and prune a single user.

Lemma 5.3.7. *For every user u in a Voronoi cell whose generator is f, if $dist(f, q) - maxdist(f, v) - x \cdot dist(u, f) \geq \Delta$, u can be pruned.*

Proof: Given a user u located in a Voronoi cell whose generator is f. Because u is inside the Voronoi cell, $dist(u, q) > dist(f, q) - maxdist(f, v)$. Therefore, $dist(u, q) - x \cdot dist(u, f) \geq \Delta$. Proved.

For all the users inside a *Voronoi* cell, the maximum distance to the generator facility can be obtained and denoted as $maxdist(u, f)$.

Lemma 5.3.8. *For all the users in a Voronoi cell whose generator is f, if $dist(f, q) - maxdist(f, v) - x \cdot maxdist(u, f) \geq \Delta$, all the users can be pruned.*

Proof: Given a user u' located in a Voronoi cell whose generator is f. Because $maxdist(u, f) > dist(u', f)$. Similar to the proof process of lemma 5.3.7, for any user u', $dist(u', q) - dist(u', f) \geq \Delta$. Proved.

5.3.3 Algorithms

With all the pruning techniques, our algorithm is formulated and explained in this section. We start with the pre-processing part of our algorithm. The subsequent two sections are the main phases, including pruning phase and verification phase.

Pre-processing

In the pre-processing phase, we partition the whole space into small regions with a *Voronoi* diagram. Each *Voronoi* cell has a generator point which is a facility point. As the computation to find a user's nearest neighbour and top-1 facility is very time-consuming compared to our main algorithm, we deal with this part in the pre-processing stage. For each user u, we use the facility R*-tree to search its nearest and top-1 result, and for the users in the same *Voronoi* cell, the distance from the user to the generator is saved in a decreasing order. Besides, in the R*-tree, we also index the $maxMaxdist(e)$, e_s^{min} and e_s^{max} for each entry e. This is because after generating the *Voronoi* diagram, the maximum distance between each generator and their corner points $maxdist(f, v)$ is fixed, which can be further indexed into the facility R*-tree. Static attributes are only related to each facility. Thus, all the f_s can be calculated and recorded for future usage.

Algorithm 7: Pruning

```
1 Input: A facility R*-tree and a query $q$
2 Output: The set of cannot be pruned areas $\mathcal{A}$, the set of futile facilities $\mathcal{B}$

 1: $\mathcal{A} \leftarrow \phi, \mathcal{B} \leftarrow \phi$
 2: insert root of facility R*-tree in a min-heap $h$
 3: while $h \neq \phi$ do
 4:     de-heap an entry $e$
 5:     if $e$ is an intermediate node then
 6:         if $mindist(e, q) - (x + 1) \cdot maxMaxdist(e) > \Delta_e^{max}$ then
 7:             continue                                          ▷ Lemma 5.3.6
 8:         else if $\Delta_e^{min} \geq maxdist(e, q)$ then
 9:             insert all $f \in e$ into $\mathcal{B}$           ▷ Lemma 5.3.3
10:         else
11:             insert children nodes of $e$ into $h$
12:     else
13:         convert $e$ to $f$
14:         if $dist(f, q) - (x + 1) \cdot maxdist(f, v) > \Delta$ then
15:             continue                                          ▷ Lemma 5.3.5
16:         else if $\Delta \geq dist(f, q)$ then
17:             insert $f$ into $\mathcal{B}$                     ▷ Lemma 5.3.2
18:         else
19:             insert $f$ into $\mathcal{A}$
20: return $\mathcal{A}, \mathcal{B}$
```

Pruning

In accordance with the SRAT query definition, a straightforward way to obtain the result of a query is to check the score of every user regarding query points and compare this with their top-1 facility score. The time complexity of this approach is $O(n)$ where n is the total number of users in the data set. Our technique is able to prune *Voronoi* regions that do not have a query answer and identify all the futile facilities.

In the pruning phase, we traverse the facility R*-tree to prune the facility entries in the search space, significantly optimising SRAT query performance. Algorithm 7 presents the details of our pruning techniques. Before starting the algorithm, a min-heap h is initialised with the root of the facility R*-tree. Entries of R*-tree are de-heaped from the heap one by one.

If e is an intermediate node of R*-tree, the minimum distance from e to query q can be computed and denoted as $mindist(e, q)$. The maximum distance generator and *Voronoi* cell corners are retrieved from the R*-tree. Hence, if condition (line 6) is met, the whole entry e can be pruned and a new entry will be de-heaped from the h. If $\Delta_e^{min} \geq maxdist(e, q)$ (line 8), every $f \in e$ is recognised as a futile facility and added into set $\mathcal{B}$ for further processing. This is because we use a *Voronoi* diagram and there may be some users who consider f as the nearest neighbour but not the top-1 facility. These users need to be verified to confirm they belong to the SRAT of q or not. If entry e can neither be pruned nor is futile, its children entries will be added into heap h for future looping.

If e is not a MBR but a facility, we convert e to f. For facility f, we first check whether the *Voronoi* cell in which it is located can be pruned or not (line 14). If it can be pruned, the loop continues to de-heap a new entry. Next, it is examined for a futile facility (line 16). Otherwise, this facility f is inserted into the temporary cannot-pruned facility set. The algorithm terminates when the heap becomes empty.

Verification

After the execution of algorithm 7, two sets of facilities are returned, namely the cannot-pruned facility set and the futile facility set respectively. The verification of these two sets is similar, the

Algorithm 8: Verification

```
1 Input: The set of cannot pruned facilities $\mathcal{A}$
2 Output: The query result $SRAT(q)$
1:  $SRAT(q) \leftarrow \phi$
2:  for each $f \in \mathcal{A}$ do
3:      if $dist(f, q) - maxdist(f, v) - x \cdot maxdist(u, f) \geq \Delta$ then
4:          break                                              ▹ Lemma 5.3.8
5:      for each user $u$ who considers $f$ as nearest neighbour do
6:          if $dist(f, q) - maxdist(f, v) - x \cdot dist(u, f) \geq \Delta$ then
7:              break                                          ▹ Lemma 5.3.7
8:          else if $dist(u, q) - x \cdot dist(u, f) < \Delta$ then
9:              insert $u$ into $SRAT(q)$
10: return $SRAT(q)$
```

main difference being the number of users who need to be examined. For the cannot-pruned facility set $\mathcal{A}$, each f in $\mathcal{A}$ will be retrieved iteratively. Since f is a generator of a *Voronoi* cell, every user u inside this *Voronoi* cell considers f to be their nearest neighbour and the distance is denoted as $dist(u, f)$. The maximum distance from u to f can be found and denoted as $maxdist(u, f)$. If $dist(f, q) - maxdist(f, v) - x \cdot maxdist(u, f) \geq \Delta$ (line 3), all users in this *Voronoi* cell can be pruned. Otherwise, each user u in the *Voronoi* cell in which f is located will be verified one by one. If the condition meets (line 6), the rest of the users can be pruned as the distance between the users and the generator facility is sorted in descending order. The definition of the SRAT query is used for the final verification. As for the futile facility set, all the users who consider these facilities as the top-1 facility are the SRAT results of q. However, some users may consider a futile facility as the closest one but not the top-1 facility. For these users, the same verification process can be undertaken as in algorithm 8. Algorithm 8 terminates when the facility set becomes empty.

5.4 Experiments

To the best of our knowledge, there is no existing algorithm to answer SRAT queries. We consider to use a naive algorithm (**NAIVE**) as the competitor for our Voronoi based algorithm (**VORONOI**). In **NAIVE**, we examine every user to check it is a SRAT result of the query. If a user u satisfies $score(u, q) \leq x \cdot score(u, f)$ where f is the top-1 facility of u, u is a SRAT result of q.

All algorithms are implemented using the C++ programming language, with experiments conducted on Nectar Elastic Cloud Computing(EC2) with a 16 AMD EPYC Processor (with IBPB) and 64GB memory running Debian Linux. We use both synthetic and real data sets. The real facility data set consists of 1,000,000 locations in California [2]. Each facility location is assigned up to 5 static attributes that are obtained from the House data set [1]. All the static attributes are normalised into a range between 0 and 1, with an assumption that smaller values are preferred. The real users' locations are randomly chosen from the California [2] graph. In terms of the synthetic facility data set, the facility and user locations are generated according to a uniform distribution. The static attributes for the synthetic facilities are also produced following a uniform distribution.

Table 5.2: Experiment settings of the SRAT queries

Parameter	**Range**
Number of facilities	1K, 5K, **10K**, 50K, 100K, 200K
Number of users	1M, 2M, **5M**, 8M, 10M
Number of static attributes	1, **2**, 3, 4, 5
x factor	1.1, **1.5**, 2, 3, 4
Distance weight	0.1, 0.3, 0.5, 0.7, 0.9
Location data distribution	California, Uniform
Static data distribution	House, Uniform

In our experiments, the number of facilities, the number of users, the number of static attributes, the value of *x* and the distance weight are varied to evaluate their effect on the performance of both algorithms. All the parameters in our experiments are displayed in Table 5.2, with the default values labelled in bold text. We run 100 queries and report the average CPU cost for every experiment. All the query points are selected randomly from the facility sets.

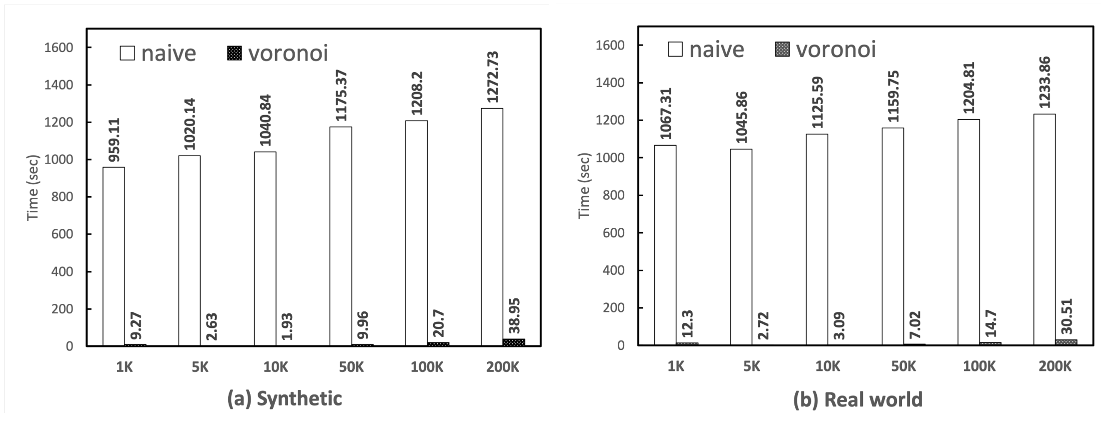


Figure 5.7: Effect of number of facilities

Effect of number of facilities. Fig 5.7 studies the effect of facility size on both algorithms. With both synthetic and real-world data sets, the cost of the **NAIVE** method increases as the number of facilities increases. The cost of the **VORONOI** approach keeps decreasing until the number of facilities is 10,000 and grows gradually after that point. This is because with the increase in facility size, no matter whether it is the **NAIVE** or **VORONOI** algorithm, more computation will be involved, eventually leading to more time consumption. The slightly decline of the **VORONOI**

method is caused by the facility pruning problem. When the number of facilities is small, each *Vornoi* cell is relatively big and will induce only very few facilities to be pruned. Specifically, the more facilities which are examined, the longer the processing time. In all data sets, our algorithm significantly outperforms the competitor.

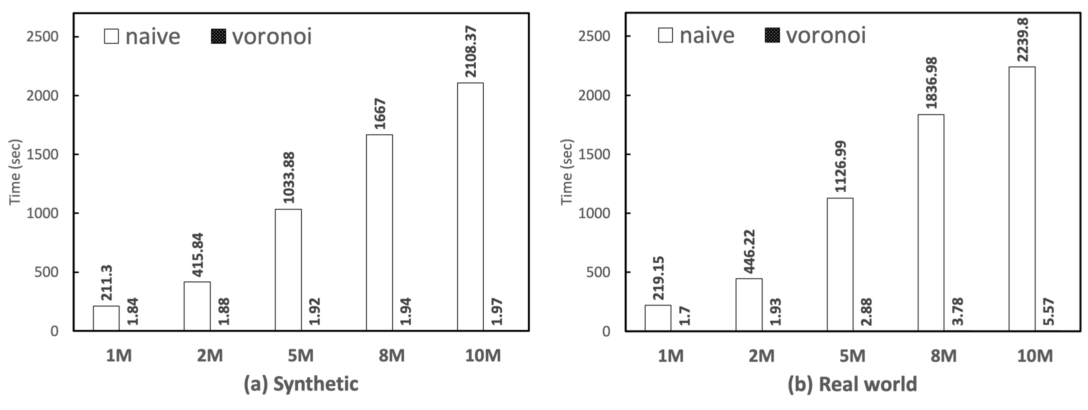


Figure 5.8: Effect of number of users

Effect of number of users. Fig 5.8 shows the effect of changing user data size on both algorithms. Regarding the **NAIVE** method, the CPU time cost surges dramatically with the increment of users. However, the cost of the **VORONOI** approach grows moderately at a very low level. This is because a higher number of users enhances the density of the users in a *Voronoi* cell and more users can be pruned in the **VORONOI** method. Our algorithm is up to two orders of magnitude better than the competitor and it also scales significantly better.

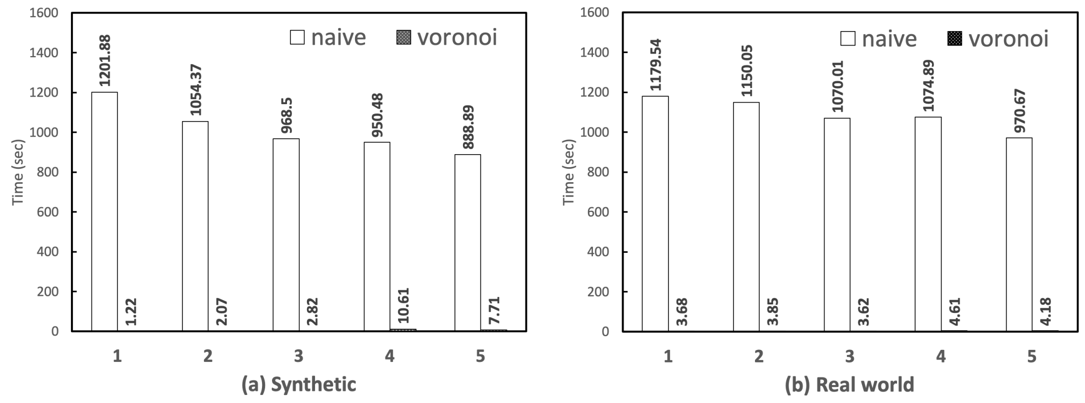


Figure 5.9: Effect of number of static attributes

Effect of number of static attributes. As shown in Fig 5.9, the effect of the number of static attributes on both the synthetic and real-world data is examined. Since in our experimental settings, all the static attributes and dynamic attributes have the same weight, increasing the number of static attributes will cause the total static score to rise, which facilitates the addition of the Δ value.

According to the SRAT definition, if $\Delta \geq dist(f, q)$, this f is a futile facility and has no pruning area. Therefore, more futile facilities means more users need to be processed in the **VORONOI** method. In the **NAIVE** approach, users need to be examined one by one, whereas the users who consider a futile facility as the top-1 facility are identified as the SRAT result directly, decreasing the processing time. Even though the increment of the static attributes restrains the performance of our algorithm, **VORONOI** is still up to two orders of magnitude better than its competitor.

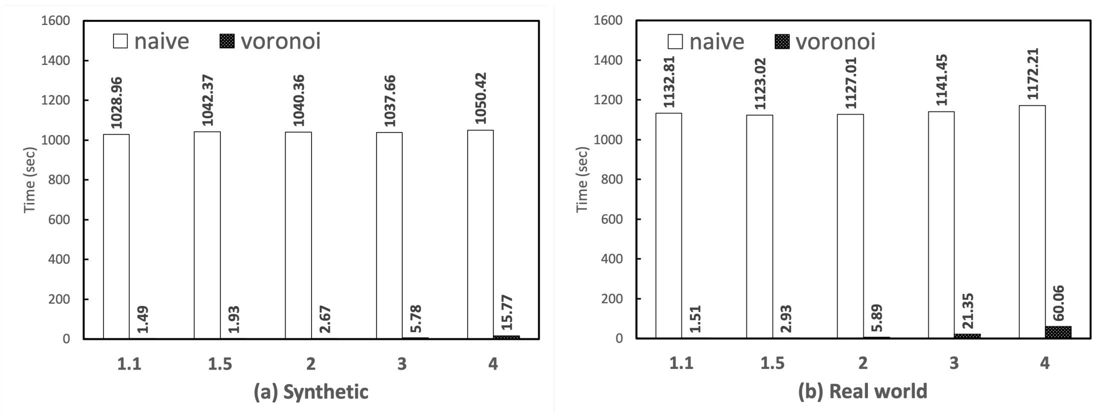


Figure 5.10: Effect of value of x

Effect of value of x. Fig 5.10 studies the effect of the x factor on the cost of both algorithms with synthetic and real-world data sets. The cost of **NAIVE** is relatively stable for different values of x because it only evaluates every user in the graph and is not sensible for the changing of x factor. On the contrary, the cost of **VORONOI** increases with the increment of the x value. This is because the bigger the x factor, the bigger the Δ value, meaning more facilities cannot be pruned and hence a higher number of users need to be verified. Our algorithm remarkably outperforms the competitor with various x factor values.

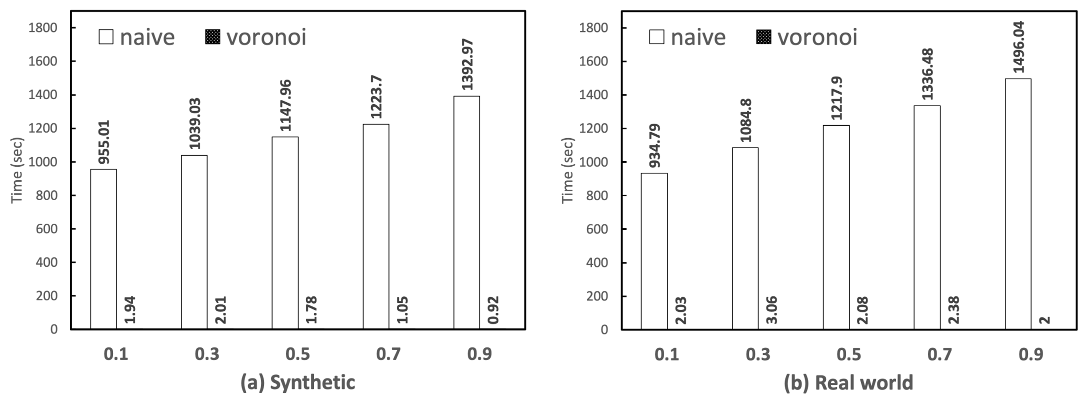


Figure 5.11: Effect of distance weight

Effect of distance weight. Fig 5.11 illustrates the effect of distance weight on the cost of both algorithms. Because we need to change the weight of the distance, we use the default number of static attributes which is 2 and the weight for all static attributes is $\frac{(1-w[distance])}{2}$. For example, if $w[distance] = 0.1$, all the static attribute weights are 0.45. As an increase in the distance weight would amortize the influence of the static score, more facilities can be pruned and **VORONOI** will have better performance. However, **NAIVE** needs to process more users. For all the values of the distance weight, our algorithm performs significantly better than the competitor and also scales better.

5.5 Conclusion

In this chapter, we studied the spatial reverse approximate top (SRAT) queries. This query is proposed to supplement the spatial reverse top-k queries by relaxing the k value to a x factor. With the SRAT query, the influence of a cluster of facilities can be investigated accurately. As SRAT is a totally new query and no existing algorithm can answer it, we propose a novel approach based on a *Voronoi* diagram and facility R*-tree to reduce the computation cost of SRAT queries. Our extensive experimental study on both synthetic and real-world data sets demonstrates that our algorithm is considerably better than the rival.

Chapter 6

Final Remarks

6.1 Overview

In this thesis, we propose several efficient algorithms for reverse approximate queries in spatial databases, including reverse approximate nearest neighbor (RANN) query on road network, continuous reverse approximate nearest neighbor query on road network and spatial reverse approximate top (SRAT) query. Chapter 2 lists all the works that are related to our algorithms. We comprehensively analyse all of them to find the reasons why they are ineffective and obtain inspiration to solve our problems. In Chapters 3 and 4, the snapshot and continuous RANN is studied with real world road network data. The following chapter illustrates our research on the snapshot SRAT query.

This chapter is organized as follows. The following section reiterates our detailed contributions. Section 6.3 presents possible directions for future works.

6.2 Contributions

This section summarises the contributions of the thesis. We studied the snapshot and continuous RANN on road network and the SRAT queries. All our algorithms were evaluated with real-world and synthetic data sets. It is proven that our approaches significantly outperform their competitors.

6.2.1 Snapshot RANN query on Road Network

In Chapter 3, we carefully exploit the special property of the road network and the RANN query. The existing RANN algorithm only works on *Euclidean* space. Since processing the users one by one in accordance with the RANN definition is extremely time-consuming, we propose an innovative way to process a RANN query efficiently with road network data. To handle the RANN on road network, an index called network *Voronoi* diagram (*NVD*) is used to partition the whole graph into small pieces. With the *NVD*, we adopt the state-of-the-art *k*NN algorithm proposed in [4] to find the *k* nearest *NVD* cells' generator facilities and use the distance between facilities to determine whether this cell can be pruned. With this pruning technique, a large space can be eliminated in query processing. Only the users who belong to the *NVD* cells which cannot be pruned need to be examined in the verification phase, which saves a huge amount of time. In the extensive experiment evaluation, we test our algorithm on real-world data that has up to 20 million nodes. It is proven that our method significantly outperforms the competitors and is easily scalable.

6.2.2 Continuous Monitoring RANN query on Road Network

In Chapter 4, we monitor the moving RANN queries on the road network. We use a client-server mode for the whole system. In this system, all users can move at each timestamp and all facilities remain stationary. Query points are selected from the facility vertices. The RANN query result of all queries could be different at each timestamp because the coordinates of users could change at any time.

In the traditional mode, users will report their locations to the server if they move at each timestamp, which is dramatically time-consuming and can easily cause the server to reach a bottleneck. We proposed two approaches using the safe zone and influence zone to handle user movements. For the safe zone method, a safe region is computed for each user in the system. When users move inside their safe region, the RANN query result is not changed and there is no need to report the users' new locations to the server at each timestamp. Only when users move out of their safe zone, their new positions will be updated in the server and new safe zones will be computed for them. For the influence zone method, users have no safe regions but an area of influence will be computed for the query point. Because all the facilities remain static, after calculating the influence zone, this area is fixed and there is no need for re-creation even if users move in or out of this influence area. If a user is in the query's influence zone and only moves inside it, the query result is unchanged and no updates occur. If a user moves out of the query's influence zone, only the query result will be updated and the influence zone does not need to be recomputed. From a different perspective, these two approaches can significantly shorten the updating time when the user moves. Last, we design a comprehensive experiment to prove that our work is much faster than the competitors.

6.2.3 SRAT query

In Chapter 5, we proposed a new query called the spatial reverse approximate top (SRAT) query, which supplement the spatial reverse top-*k* (SRT*k*) query. By relaxing the restrictions of the *k* value, the SRAT query can return a more reasonable result than the SRT*k* query. As this is the first time the SRAT concept has been proposed, no research has been done on this query. Recall that the SRAT query returns every user $u \in U$ for which $score(u, q) < x \cdot Top1score(u, f)$ where $Top1score(u, f)$ is the score between a user u and its top1 facility f.

In our algorithm, we use the R*-tree to index all the facilities. Then, we use all the facilities as generators to create a *Voronoi* diagram. The *Voronoi* diagram is used to separate the whole searching space into small segments. Since a score function is used in the SRAT definition and the score function contains both the static and dynamic attributes, a pruning corollary can be generated as $dist(u, q) - x \cdot dist(u, f) > \Delta$. With this corollary, the R*-tree index and the *Voronoi* diagram, many facilities can be pruned or identified as futile facilities. When a facility is pruned, this means the *Voronoi* cell in which this facility is located is pruned. If a facility is a futile facility, all the users who consider this facility as the top-1 facility will be the result of q. Hence, only the users who reside in the *Voronoi* cells which cannot be pruned need to be verified using the SRAT definition. Lastly, our experiment outcome shows that our algorithm works on both synthetic and real-world data sets, which is up to two orders of magnitude better than the competitor and also scales significantly better.

6.3 Directions for Future Work

On the basis of our research, there are many aspects of reverse approximate queries in spatial databases which have not been studied, and some ranking related queries are very interesting but as yet, have received no attention. In this section, several possible directions for future works have been discussed.

6.3.1 Spatial Reverse Approximate Top Queries On Road Network

As far as we know, R*k*NN queries have been studied extensively, whereas reverse approximate queries in spatial databases have not received much attention. In this thesis, we have studied several reverse approximate queries, including the reverse approximate nearest neighbour (RANN) query and spatial reverse approximate top (SRAT) query. We proposed innovative approaches to answer snapshot and continuous RANN queries on the road network. However, our SRAT query research only focuses on *Euclidean* space. No work has been done to process SRAT queries on the road network. Due to the natural difference between *Euclidean* data and road network data, how to process a SRAT query on road network is worth considering.

6.3.2 Continuous Reverse Approximate Top Queries

Continuous queries is another main direction. As for the SRAT query, a score function is used to compute the influence of the query point, which considers various criteria. Because this query is score function based, the generated pruning area in a graph is not a straight line (similar to the distance-based queries). Hence, the pruning area is an irregular shape and the computation speed is slower. When issuing this query in the moving environment, the updating cost could be very high and cause a bottleneck for the server. How to monitor the continuous SRAT query is a possible direction and is worth investigating.

Meanwhile, changing the data sets to road network data could also cause the existing SRAT algorithm to fail. When performing a continuous SRAT query on the road network, both the pruning techniques and server information updating techniques need to be devised. Conducting a continuous SRAT query on the road network is also difficult and needs more thought.

6.3.3 Spatial Reverse Top-*k* Queries On Road Network

Spatial reverse top-*k* (SRT*k*) queries is proposed in [109] and they are well studied in *Euclidean* space. Different from traditional reverse top-*k* queries, SRT*k* queries include a dynamic attribute distance. The distance value from each user to the same facility is different. As the distance calculation process in a road network and *Euclidean* space is different, conducting a SRT*k* query on a road network is becoming more complex. With road network data, the existing algorithm may not work properly and thus processing a SRT*k* query on the road network could be an interesting research direction.

6.3.4 Continuous Spatial Reverse Top-*k* Queries

Similar to other queries, the moving environment is more applicable in the real world and processing moving queries is more meaningful. Currently, only snapshot SRT*k* queries have been studied. We are interested in monitoring this query in a continuous environment with both *Euclidean* space data and road network data.

References

[1] https://international.ipums.org/ international/.

[2] https://www.census.gov/geo/maps-data/data/ tiger-line.html.

[3] http://users.diag.uniroma1.it/challenge9/.

[4] Tenindra Abeywickrama and Muhammad Aamir Cheema. Efficient landmark-based candidate generation for knn queries on road networks. In *International Conference on Database Systems for Advanced Applications*, pages 425–440. Springer, 2017.

[5] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *arXiv preprint arXiv:1601.01549*, 2016.

[6] Elke Achtert, Christian Böhm, Peer Kröger, Peter Kunath, Alexey Pryakhin, and Matthias Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 515–526, 2006.

[7] Elke Achtert, Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Andreas Züfle. Reverse k-nearest neighbor search in dynamic and general metric databases. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 886–897. ACM, 2009.

[8] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 147–154. SIAM, 2014.

[9] Rimantas Benetis, Christian S Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Database Engineering and Applications Symposium, 2002. Proceedings. International*, pages 44–53. IEEE, 2002.

[10] Rimantas Benetis, Christian S Jensen, Gytis Karĉiauskas, and Simonas Ŝaltenis. Nearest and reverse nearest neighbor queries for moving objects. *The VLDB Journal*, 15(3):229–249, 2006.

[11] Thomas Bernecker, Tobias Emrich, Hans-Peter Kriegel, Matthias Renz, Stefan Zankl, and Andreas Züfle. Efficient probabilistic reverse nearest neighbor query processing on uncertain data. *Proceedings of the VLDB Endowment*, 4(10):669–680, 2011.

[12] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999.

[13] Thomas Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[14] Ying Cai, Kien A Hua, and Guohong Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *IEEE International Conference on Mobile Data Management, 2004. Proceedings. 2004*, pages 27–38. IEEE, 2004.

[15] Kevin Chen-Chuan Chang and Seung-won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 346–357, 2002.

[16] Muhammad Aamir Cheema, Ljiljana Brankovic, Xuemin Lin, Wenjie Zhang, and Wei Wang. Continuous monitoring of distance-based range queries. *IEEE Transactions on Knowledge and Data Engineering*, 23(8):1182–1199, 2010.

[17] Muhammad Aamir Cheema, Xuemin Lin, Haixun Wang, Jianmin Wang, and Wenjie Zhang. A unified approach for computing top-k pairs in multidimensional space. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1031–1042. IEEE, 2011.

[18] Muhammad Aamir Cheema, Xuemin Lin, Wei Wang, Wenjie Zhang, and Jian Pei. Probabilistic reverse nearest neighbor queries on uncertain data. *IEEE Transactions on Knowledge and Data Engineering*, 22(4):550–564, 2010.

[19] Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Ying Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 577–588. IEEE, 2011.

[20] Muhammad Aamir Cheema, Xuemin Lin, Ying Zhang, Wei Wang, and Wenjie Zhang. Lazy updates: An efficient technique to continuously monitoring reverse knn. *Proceedings of the VLDB Endowment*, 2(1):1138–1149, 2009.

[21] Muhammad Aamir Cheema, Zhitao Shen, Xuemin Lin, and Wenjie Zhang. A unified framework for efficiently processing ranking related queries. In *EDBT*, pages 427–438, 2014.

[22] Muhammad Aamir Cheema, Yidong Yuan, and Xuemin Lin. Circulartrip: an effective algorithm for continuous knn queries. In *International Conference on Database Systems for Advanced Applications*, pages 863–869. Springer, 2007.

[23] Muhammad Aamir Cheema, Wenjie Zhang, Xuemin Lin, and Ying Zhang. Efficiently processing snapshot and continuous reverse k nearest neighbors queries. *The VLDB Journal*, 21(5):703–728, 2012.

[24] Muhammad Aamir Cheema, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Xuefei Li. Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(1):69–95, 2012.

[25] Sean Chester, Alex Thomo, S Venkatesh, and Sue Whitesides. Indexing reverse top-k queries in two dimensions. In *International Conference on Database Systems for Advanced Applications*, pages 201–208. Springer, 2013.

[26] King Lum Cheung and Ada Wai-Chee Fu. Enhanced nearest neighbour search on the r-tree. *ACM SIGMOD Record*, 27(3):16–21, 1998.

[27] Hyung-Ju Cho and Chin-Wan Chung. An efficient and scalable approach to cnn queries in a road network. In *International Conference on VLDB*, volume 2, pages 865–876. International Conference on VLDB, 2005.

[28] Ugur Demiryurek and Cyrus Shahabi. Indexing network voronoi diagrams. In *International Conference on Database Systems for Advanced Applications*, pages 526–543. Springer, 2012.

[29] Martin Erwig. The graph voronoi diagram with applications. *Networks: An International Journal*, 36(3):156–163, 2000.

[30] Ronald Fagin. Combining fuzzy information from multiple systems. *Journal of computer and system sciences*, 58(1):83–99, 1999.

[31] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[32] Yunjun Gao, Baihua Zheng, Wang-Chien Lee, and Gencai Chen. Continuous visible nearest neighbor queries. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 144–155, 2009.

[33] Buğra Gedik and Ling Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *International Conference on Extending Database Technology*, pages 67–87. Springer, 2004.

[34] Orestis Gkorgkas, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. Discovering influential data objects over time. In *International Symposium on Spatial and Temporal Databases*, pages 110–127. Springer, 2013.

[35] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. 2000.

[36] Li Guohui, Li Yanhong, Li Jianjun, LihChyun Shu, and Yang Fumin. Continuous reverse k nearest neighbor monitoring on moving objects in road networks. *Information Systems*, 35(8):860–883, 2010.

[37] Arif Hidayat, Muhammad Aamir Cheema, and David Taniar. Relaxed reverse nearest neighbors queries. In *Advances in Spatial and Temporal Databases - 14th International Symposium, SSTD 2015, Hong Kong, China, August 26-28, 2015. Proceedings*, pages 61–79, 2015.

[38] Arif Hidayat, Shiyu Yang, Muhammad Aamir Cheema, and David Taniar. Reverse approximate nearest neighbor queries. *IEEE Transactions on Knowledge and Data Engineering*, 30(2):339–352, 2018.

[39] Gisli R Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.

[40] Haibo Hu, Dik Lun Lee, and Jianliang Xu. Fast nearest neighbor search on road networks. In *International Conference on Extending Database Technology*, pages 186–203. Springer, 2006.

[41] Keum-Sung Hwang and Sung-Bae Cho. A lifelog browser for visualization and search of mobile everyday-life. *Mobile Information Systems*, 10(3):243–258, 2014.

[42] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):1–58, 2008.

[43] Tomasz Imielinski and BR Badrinath. Querying in highly mobile distributed environments. In *VLDB*, volume 92, pages 41–52, 1992.

[44] Glenn S Iwerks, Hanan Samet, and Ken Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *Proceedings 2003 VLDB Conference*, pages 512–523. Elsevier, 2003.

[45] Christian S Jensen, Jan Kolářvr, Torben Bach Pedersen, and Igor Timko. Nearest neighbor queries in road networks. In *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, pages 1–8, 2003.

[46] Cheqing Jin, Rong Zhang, Qiangqiang Kang, Zhao Zhang, and Aoying Zhou. Probabilistic reverse top-k queries. In *International Conference on Database Systems for Advanced Applications*, pages 406–419. Springer, 2014.

[47] James M Kang, Mohamed F Mokbel, Shashi Shekhar, Tian Xia, and Donghui Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 806–815. IEEE, 2007.

[48] James M. Kang, Mohamed F. Mokbel, Shashi Shekhar, Tian Xia, and Donghui Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *ICDE*, pages 806–815, 2007.

[49] Joon-Seok Kim and Ki-Joune Li. Location k-anonymity in indoor spaces. *Geoinformatica*, 20(3):415–451, 2016.

[50] Mohammad Kolahdouzan and Cyrus Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 840–851. VLDB Endowment, 2004.

[51] Mohammad R. Kolahdouzan and Cyrus Shahabi. Continuous k-nearest neighbor queries in spatial network databases. In Jörg Sander and Mario A. Nascimento, editors, *Spatio-Temporal Database Management, 2nd International Workshop STDBM'04, Toronto, Canada, August 30, 2004*, pages 33–40, 2004.

[52] Flip Korn and Suresh Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *ACM Sigmod Record*, volume 29, pages 201–212. ACM, 2000.

[53] Iosif Lazaridis, Kriengkrai Porkaew, and Sharad Mehrotra. Dynamic queries over mobile objects. In *International Conference on Extending Database Technology*, pages 269–286. Springer, 2002.

[54] Chuanwen Li, Yu Gu, Jianzhong Qi, Rui Zhang, and Ge Yu. Moving knn query processing in metric space based on influential sets. *Information Systems*, 83:126–144, 2019.

[55] Xinyu Li, Arif Hidayat, David Taniar, and Muhammad Aamir Cheema. Reverse approximate nearest neighbor queries on road network. *World Wide Web*, 24(1):279–296, 2021.

[56] Xiang Lian and Lei Chen. Efficient processing of probabilistic reverse nearest neighbor queries over uncertain data. *VLDB J.*, 18(3):787–808, 2009.

[57] King-Ip Lin, Michael Nolen, and Congjun Yang. Applying bulk insertion techniques for dynamic reverse nearest neighbor problems. In *Seventh International Database Engineering and Applications Symposium, 2003. Proceedings.*, pages 290–297. IEEE, 2003.

[58] Hua Lu, Xin Cao, and Christian S Jensen. A foundation for efficient indoor distance-aware query processing. In *2012 IEEE 28th International Conference on Data Engineering*, pages 438–449. IEEE, 2012.

[59] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W Cheung. Efficient top-k aggregation of ranked inputs. *ACM Transactions on Database Systems (TODS)*, 32(3):19–es, 2007.

[60] Mohamed F Mokbel, Xiaopeing Xiong, and Walid G Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 623–634, 2004.

[61] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 635–646, 2006.

[62] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 634–645, 2005.

[63] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias, and Nikos Mamoulis. Continuous nearest neighbor monitoring in road networks. 2006.

[64] Surya Nepal and MV Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*, pages 22–29. IEEE, 1999.

[65] Sarana Nutanong, Egemen Tanin, Mohammed Eunus Ali, and Lars Kulik. Local network voronoi diagrams. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 109–118. ACM, 2010.

[66] Sarana Nutanong, Egemen Tanin, and Rui Zhang. Visible nearest neighbor queries. In *International Conference on Database Systems for Advanced Applications*, pages 876–883. Springer, 2007.

[67] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *Proceedings 2003 VLDB Conference*, pages 802–813. Elsevier, 2003.

[68] Jianzhong Qi, Rui Zhang, Christian S Jensen, Kotagiri Ramamohanarao, and Jiayuan He. Continuous spatial query processing: a survey of safe region based techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[69] Nick Roussopoulos, Stephen Kelley, and Frederic Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 71–79, 1995.

[70] Maytham Safar. K nearest neighbor search in navigation systems. *Mobile Information Systems*, 1(3):207–224, 2005.

[71] Simonas Šaltenis, Christian S Jensen, Scott T Leutenegger, and Mario A Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 331–342, 2000.

[72] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 154–165, 1998.

[73] Cyrus Shahabi, Mohammad R Kolahdouzan, and Mehdi Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica*, 7(3):255–273, 2003.

[74] Shuo Shang, Bo Yuan, Ke Deng, Kexin Xie, and Xiaofang Zhou. Finding the most accessible locations: reverse path nearest neighbor query in road networks. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 181–190, 2011.

[75] Mehdi Sharifzadeh and Cyrus Shahabi. Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. *Proceedings of the VLDB Endowment*, 3(1-2):1231–1242, 2010.

[76] Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Haixun Wang. Efficiently monitoring top-k pairs over sliding windows. In *2012 IEEE 28th International Conference on Data Engineering*, pages 798–809. IEEE, 2012.

[77] Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Haixun Wang. A generic framework for top-k pairs and top-k objects queries over sliding windows. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1349–1366, 2012.

[78] Amit Singh, Hakan Ferhatosmanoglu, and Ali Şaman Tosun. High dimensional reverse nearest neighbor queries. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 91–98. ACM, 2003.

[79] Ammar Sohail, Arif Hidayat, Muhammad Aamir Cheema, and David Taniar. Location-aware group preference queries in social-networks. In *Australasian Database Conference*, pages 53–67. Springer, 2018.

[80] Ammar Sohail, Ghulam Murtaza, and David Taniar. Retrieving top-k famous places in location-based social networks. In *Australasian Database Conference*, pages 17–30. Springer, 2016.

[81] Ioana Stanoi, Divyakant Agrawal, and Amr El 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.

[82] Ioana Stanoi, Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*, volume 2001, pages 99–108, 2001.

[83] Huan-Liang Sun, Chao Jiang, Jun-Ling Liu, and Limei Sun. Continuous reverse nearest neighbor queries on moving objects in road networks. In *The Ninth International Conference on Web-Age Information Management*, pages 238–245. IEEE, 2008.

[84] David Taniar and Wenny Rahayu. A taxonomy for nearest neighbour queries in spatial databases. *Journal of Computer and System Sciences*, 79(7):1017–1039, 2013.

[85] David Taniar, Maytham Safar, Quoc Thai Tran, Wenny Rahayu, and Jong Hyuk Park. Spatial network rnn queries in gis. *The Computer Journal*, 54(4):617–627, 2011.

[86] Yufei Tao, Dimitris Papadias, and Xiang Lian. Reverse knn search in arbitrary dimensionality. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 744–755. VLDB Endowment, 2004.

[87] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 287–298. Elsevier, 2002.

[88] Quoc Thai Tran, David Taniar, and Maytham Safar. Reverse k nearest neighbor and reverse farthest neighbor search on spatial networks. In *Transactions on large-scale data-and knowledge-centered systems I*, pages 353–372. Springer, 2009.

[89] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. Reverse top-k queries. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 365–376. IEEE, 2010.

[90] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Norvag. Monochromatic and bichromatic reverse top-k queries. *IEEE Transactions on Knowledge and Data Engineering*, 23(8):1215–1229, 2011.

[91] Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. Monitoring reverse top-k queries over mobile devices. In *Proceedings of the 10th ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 17–24, 2011.

[92] Akrivi Vlachou, Christos Doulkeridis, Kjetil Nørvåg, and Yannis Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, pages 481–492, 2013.

[93] Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar. Research in mobile database query optimization and processing. *Mobile Information Systems*, 1(4):225–252, 2005.

[94] Agustinus Borgy Waluyo, David Taniar, Wenny Rahayu, and Bala Srinivasan. Mobile service oriented architectures for nn-queries. *Journal of Network and Computer Applications*, 32(2):434–447, 2009.

[95] Haojun Wang, Roger Zimmermann, and Wei-Shinn Ku. Distributed continuous range query processing on moving objects. In *International Conference on Database and Expert Systems Applications*, pages 655–665. Springer, 2006.

[96] Shenlu Wang, Muhammad Aamir Cheema, and Xuemin Lin. Efficiently monitoring reverse k-nearest neighbors in spatial networks. *The Computer Journal*, 58(1):40–56, 2015.

[97] Xiaoyuan Wang and Wei Wang. Continuous expansion: Efficient processing of continuous range monitoring in mobile environments. In *International Conference on Database Systems for Advanced Applications*, pages 890–899. Springer, 2006.

[98] Wei Wu, Fei Yang, Chee Yong Chan, and Kian-Lee Tan. Continuous reverse k-nearest-neighbor monitoring. In *The Ninth International Conference on Mobile Data Management (mdm 2008)*, pages 132–139. IEEE, 2008.

[99] Wei Wu, Fei Yang, Chee-Yong Chan, and Kian-Lee Tan. Finch: Evaluating reverse k-nearest-neighbor queries on location data. *Proceedings of the VLDB Endowment*, 1(1):1056–1067, 2008.

[100] Chenyi Xia, David Hsu, and Anthony KH Tung. A fast filter for obstructed nearest neighbor queries. In *British National Conference on Databases*, pages 203–215. Springer, 2004.

[101] Tian Xia and Donghui Zhang. Continuous reverse nearest neighbor monitoring. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 77–77. IEEE, 2006.

[102] Xike Xie, Hua Lu, and Torben Bach Pedersen. Efficient distance-aware query evaluation on indoor moving objects. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 434–445. IEEE, 2013.

[103] Xiaopeng Xiong, Mohamed F Mokbel, and Walid G Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *21st International Conference on Data Engineering (ICDE'05)*, pages 643–654. IEEE, 2005.

[104] Xiao-Jun Xu, Jin-Song Bao, Bin Yao, Jing-Yu Zhou, Fei-Long Tang, Min-Yi Guo, and Jian-Qiu Xu. Reverse furthest neighbors query in road networks. *Journal of Computer Science and Technology*, 32(1):155–167, 2017.

[105] Ikuko Eguchi Yairi and Seiji Igi. Mobility support gis with universal-designed data of barrier/barrier-free terrains and facilities for all pedestrians including the elderly and the disabled. In *Systems, Man and Cybernetics, 2006. SMC'06. IEEE International Conference on*, volume 4, pages 2909–2914. IEEE, 2006.

[106] Congyun Yang and King-Ip Lin. An index structure for efficient reverse nearest neighbor queries. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 485–492. IEEE, 2001.

[107] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, and Wei Wang. Reverse k nearest neighbors query processing: experiments and analysis. *Proceedings of the VLDB Endowment*, 8(5):605–616, 2015.

[108] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, and Ying Zhang. Slice: reviving regions-based pruning for reverse k nearest neighbors queries. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 760–771. IEEE, 2014.

[109] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, Ying Zhang, and Wenjie Zhang. Reverse k nearest neighbors queries and spatial reverse top-k queries. *The VLDB Journal*, 26(2):151–176, 2017.

[110] Man Lung Yiu, Dimitris Papadias, Nikos Mamoulis, and Yufei Tao. Reverse nearest neighbors in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):540–553, 2006.

[111] Adams Wei Yu, Nikos Mamoulis, and Hao Su. Reverse top-k search using random walk with restart. *Proceedings of the VLDB Endowment*, 7(5):401–412, 2014.

[112] Jiao Yu, Wei-Shinn Ku, Min-Te Sun, and Hua Lu. An rfid and particle filter-based indoor spatial query evaluation system. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 263–274, 2013.

[113] Xiaohui Yu, Ken Q Pu, and Nick Koudas. Monitoring k-nearest neighbor queries over moving objects. In *21st International Conference on Data Engineering (ICDE'05)*, pages 631–642. IEEE, 2005.

[114] Jun Zhang, Dimitris Papadias, Kyriakos Mouratidis, and Zhu Manli. Query processing in spatial databases containing obstacles. *International Journal of Geographical Information Science*, 19(10):1091–1111, 2005.

[115] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. Location-based spatial queries. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 443–454, 2003.

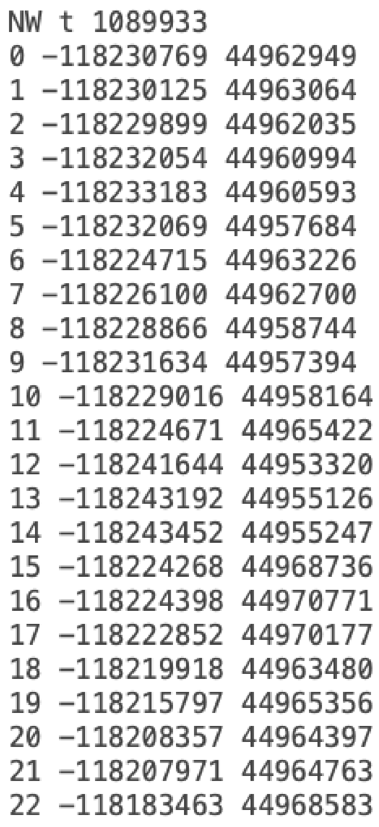
Appendix A

Reverse Approximate Query Data Sets Demo

In this appendix, we shows the details of the data sets used in our algorithms.

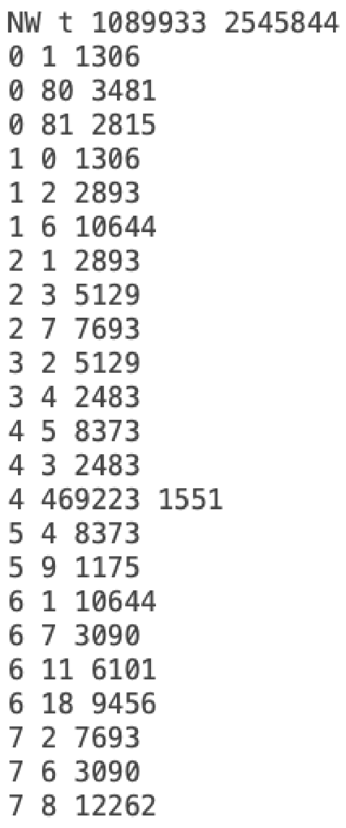
A.1 RANN query on road network data sample

In the snapshot and continuous RANN queries, we only use the real-world data sets. In fig A.1, the sample data of the north west area of the USA displayed. In this data set, we have the id and coordinates of each vertex. As our research is about RANN on road network space, there is a graph data indicating the relations among all the edges. The sample edges data set can be seen in fig A.2. The data contains the id of nodes and the length between these nodes. A single node can connect to multiple points.



```
NW t 1089933
0 -118230769 44962949
1 -118230125 44963064
2 -118229899 44962035
3 -118232054 44960994
4 -118233183 44960593
5 -118232069 44957684
6 -118224715 44963226
7 -118226100 44962700
8 -118228866 44958744
9 -118231634 44957394
10 -118229016 44958164
11 -118224671 44965422
12 -118241644 44953320
13 -118243192 44955126
14 -118243452 44955247
15 -118224268 44968736
16 -118224398 44970771
17 -118222852 44970177
18 -118219918 44963480
19 -118215797 44965356
20 -118208357 44964397
21 -118207971 44964763
22 -118183463 44968583
```

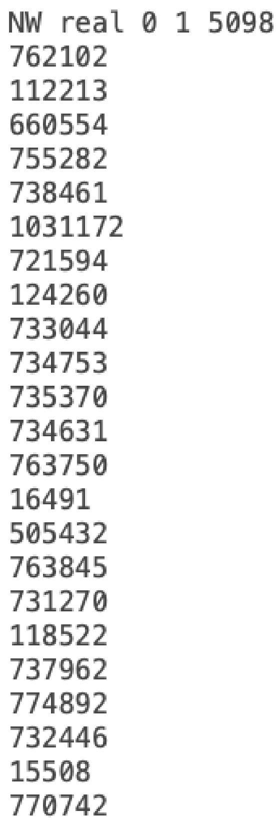
Figure A.1: Sample of point coordinates data



```
NW t 1089933 2545844
0 1 1306
0 80 3481
0 81 2815
1 0 1306
1 2 2893
1 6 10644
2 1 2893
2 3 5129
2 7 7693
3 2 5129
3 4 2483
4 5 8373
4 3 2483
4 469223 1551
5 4 8373
5 9 1175
6 1 10644
6 7 3090
6 11 6101
6 18 9456
7 2 7693
7 6 3090
7 8 12262
```

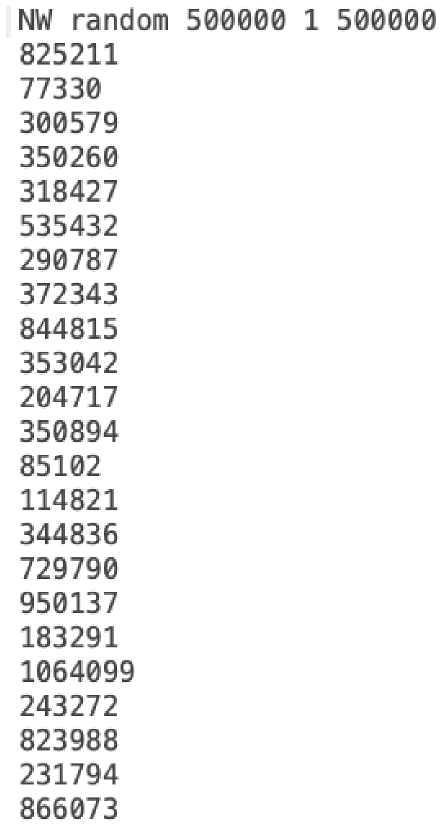
Figure A.2: Sample of edges data

In our experiment, we use real road networks with different network size from website *http* : *//users.diag.uniroma*1*.it/challenge*9*/* [3]. The largest data set contain up to 23,947,347 vertices and 57,708,624 edges. Except the graph data, the facility data is extracted from Open Street Map (OSM). We compare the coordinates and find the corresponding node id in existing graph data. The final facility list data can be found in fig A.3. And, the user data sets are randomly chosen from the graph which has a similar format as the facility in fig A.4.



```
NW real 0 1 5098
762102
112213
660554
755282
738461
1031172
721594
124260
733044
734753
735370
734631
763750
16491
505432
763845
731270
118522
737962
774892
732446
15508
770742
```

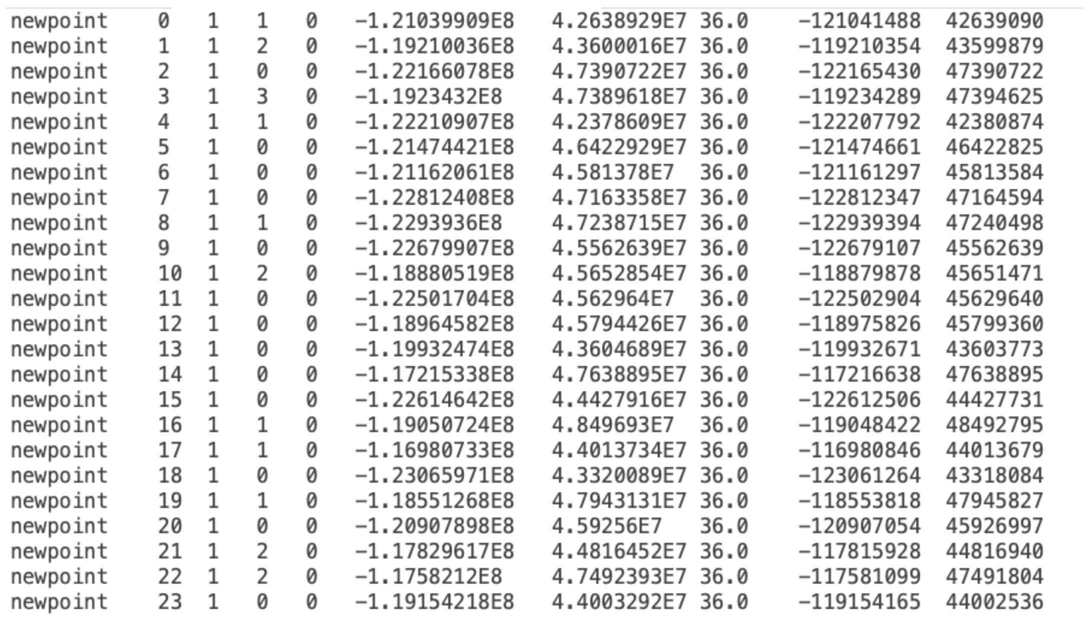
Figure A.3: Sample of real world facilities data



```
NW random 500000 1 500000
825211
77330
300579
350260
318427
535432
290787
372343
844815
353042
204717
350894
85102
114821
344836
729790
950137
183291
1064099
243272
823988
231794
866073
```

Figure A.4: Sample of users data

In our second project, the continuous RANN queries require moving user data. We use the well-know Brinkhoff data generator [13] to generate all the moving users on road network. The sample data is shown in fig A.5. Each line of data contains a timestamp, the current coordinates and the moving destination coordinates. The graph data used in continuous RANN is same as the snapshot queries.

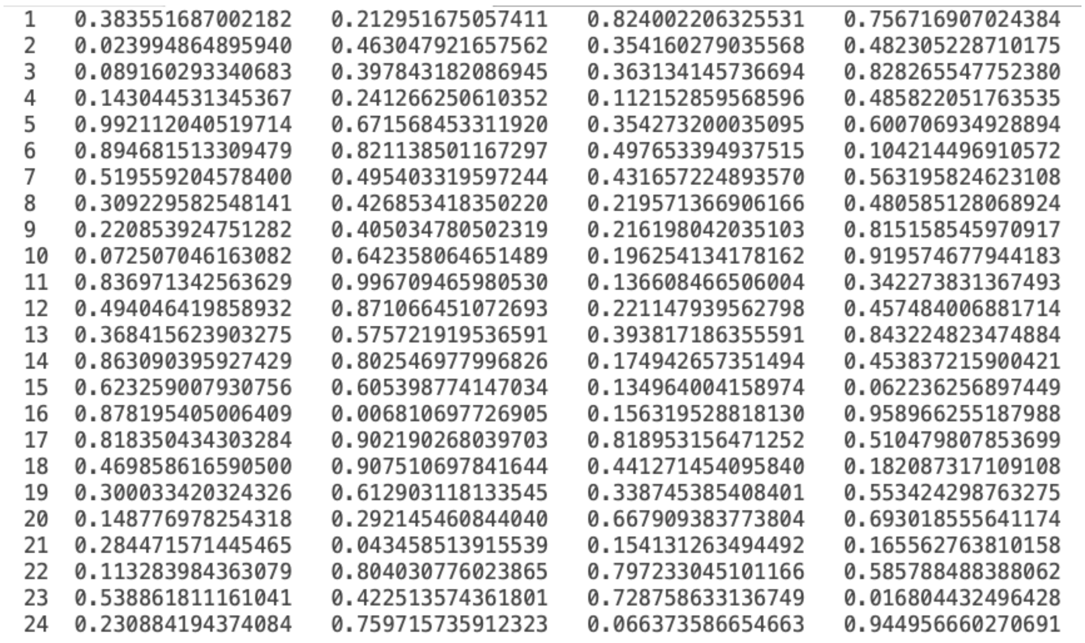


```
newpoint    0   1   1   0   -1.21039909E8   4.2638929E7 36.0    -121041488  42639090
newpoint    1   1   2   0   -1.19210036E8   4.3600016E7 36.0    -119210354  43599879
newpoint    2   1   0   0   -1.22166078E8   4.7390722E7 36.0    -122165430  47390722
newpoint    3   1   3   0   -1.1923432E8    4.7389618E7 36.0    -119234289  47394625
newpoint    4   1   1   0   -1.22210907E8   4.2378609E7 36.0    -122207792  42380874
newpoint    5   1   0   0   -1.21474421E8   4.6422929E7 36.0    -121474661  46422825
newpoint    6   1   0   0   -1.21162061E8   4.581378E7  36.0    -121161297  45813584
newpoint    7   1   0   0   -1.22812408E8   4.7163358E7 36.0    -122812347  47164594
newpoint    8   1   1   0   -1.2293936E8    4.7238715E7 36.0    -122939394  47240498
newpoint    9   1   0   0   -1.22679907E8   4.5562639E7 36.0    -122679107  45562639
newpoint    10  1   2   0   -1.18880519E8   4.5652854E7 36.0    -118879878  45651471
newpoint    11  1   0   0   -1.22501704E8   4.562964E7  36.0    -122502904  45629640
newpoint    12  1   0   0   -1.18964582E8   4.5794426E7 36.0    -118975826  45799360
newpoint    13  1   0   0   -1.19932474E8   4.3604689E7 36.0    -119932671  43603773
newpoint    14  1   0   0   -1.17215338E8   4.7638895E7 36.0    -117216638  47638895
newpoint    15  1   0   0   -1.22614642E8   4.4427916E7 36.0    -122612506  44427731
newpoint    16  1   1   0   -1.19050724E8   4.849693E7  36.0    -119048422  48492795
newpoint    17  1   1   0   -1.16980733E8   4.4013734E7 36.0    -116980846  44013679
newpoint    18  1   0   0   -1.23065971E8   4.3320089E7 36.0    -123061264  43318084
newpoint    19  1   1   0   -1.18551268E8   4.7943131E7 36.0    -118553818  47945827
newpoint    20  1   0   0   -1.20907898E8   4.59256E7   36.0    -120907054  45926997
newpoint    21  1   2   0   -1.17829617E8   4.4816452E7 36.0    -117815928  44816940
newpoint    22  1   2   0   -1.1758212E8    4.7492393E7 36.0    -117581099  47491804
newpoint    23  1   0   0   -1.19154218E8   4.4003292E7 36.0    -119154165  44002536
```

Figure A.5: Sample of moving data

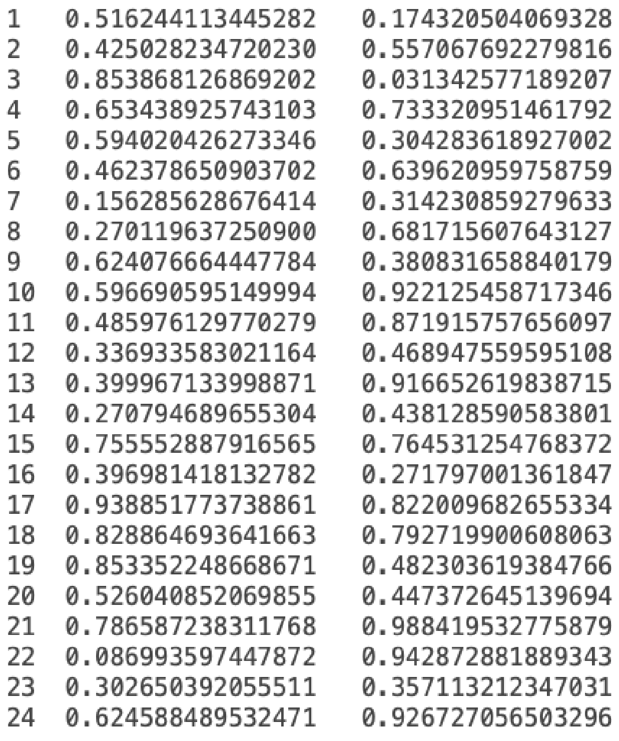
A.2 SRAT query data sample

In the third project, we proposed a new query called Spatial Reverse Approximate Top (SRAT) query. We study this query in *Euclidean* space and hence there is no graph data like road network. In the experiment, all the data is normalised to between 0 and 1. The user data is similar as before and only contains id and coordinates like fig A.7. However, the facility data is different. As in SRAT query, more attributes (i.e. rating, price) are considered when computing the importance or influence of a facility, these extra attributes values are also recorded. In fig A.6, a sample facility data with not only the coordinates but also two static attributes is shown.



```
1   0.383551687002182   0.212951675057411   0.824002206325531   0.756716907024384
2   0.023994864895940   0.463047921657562   0.354160279035568   0.482305228710175
3   0.089160293340683   0.397843182086945   0.363134145736694   0.828265547752380
4   0.143044531345367   0.241266250610352   0.112152859568596   0.485822051763535
5   0.992112040519714   0.671568453311920   0.354273200035095   0.600706934928894
6   0.894681513309479   0.821138501167297   0.497653394937515   0.104214496910572
7   0.519559204578400   0.495403319597244   0.431657224893570   0.563195824623108
8   0.309229582548141   0.426853418350220   0.219571366906166   0.480585128068924
9   0.220853924751282   0.405034780502319   0.216198042035103   0.815158545970917
10  0.072507046163082   0.642358064651489   0.196254134178162   0.919574677944183
11  0.836971342563629   0.996709465980530   0.136608466506004   0.342273831367493
12  0.494046419858932   0.871066451072693   0.221147939562798   0.457484006881714
13  0.368415623903275   0.575721919536591   0.393817186355591   0.843224823474884
14  0.863090395927429   0.802546977996826   0.174942657351494   0.453837215900421
15  0.623259007930756   0.605398774147034   0.134964004158974   0.062236256897449
16  0.878195405006409   0.006810697726905   0.156319528818130   0.958966255187988
17  0.818350434303284   0.902190268039703   0.818953156471252   0.510479807853699
18  0.469858616590500   0.907510697841644   0.441271454095840   0.182087317109108
19  0.300033420324326   0.612903118133545   0.338745385408401   0.553424298763275
20  0.148776978254318   0.292145460844040   0.667909383773804   0.693018555641174
21  0.284471571445465   0.043458513915539   0.154131263494492   0.165562763810158
22  0.113283984363079   0.804030776023865   0.797233045101166   0.585788488388062
23  0.538861811161041   0.422513574361801   0.728758633136749   0.016804432496428
24  0.230884194374084   0.759715735912323   0.066373586654663   0.944956660270691
```

Figure A.6: Sample of facilities with static attributes data



```
1   0.516244113445282   0.174320504069328
2   0.425028234720230   0.557067692279816
3   0.853868126869202   0.031342577189207
4   0.653438925743103   0.733320951461792
5   0.594020426273346   0.304283618927002
6   0.462378650903702   0.639620959758759
7   0.156285628676414   0.314230859279633
8   0.270119637250900   0.681715607643127
9   0.624076664447784   0.380831658840179
10  0.596690595149994   0.922125458717346
11  0.485976129770279   0.871915757656097
12  0.336933583021164   0.468947559595108
13  0.399967133998871   0.916652619838715
14  0.270794689655304   0.438128590583801
15  0.755552887916565   0.764531254768372
16  0.396981418132782   0.271797001361847
17  0.938851773738861   0.822009682655334
18  0.828864693641663   0.792719900608063
19  0.853352248668671   0.482303619384766
20  0.526040852069855   0.447372645139694
21  0.786587238311768   0.988419532775879
22  0.086993597447872   0.942872881889343
23  0.302650392055511   0.357113212347031
24  0.624588489532471   0.926727056503296
```

Figure A.7: Sample of users data

Appendix B

Reverse Approximate Query Main Logic Code Snippet

In this appendix, the main part of our algorithm code is presented.

B.1 Snapshot RANN query on road network

Generate users:

```
void ExperimentsCommand::generateFixNumberUsers(Graph& graph, std::size_t
    numSets, std::vector<double> objDensities, const std::string&
    filePathPrefix, const std::string& poiFileName, const std::string&
    userFolderName, bool syntheticOrRealWorld) {
    StopWatch sw;
    double totalINETime, totalINEMemory;
    std::string localPoiPath;
    std::string localUserPath;
    if (syntheticOrRealWorld) {
        localPoiPath = filePathPrefix + "/obj_indexes/";
        localUserPath = filePathPrefix + "/obj_indexes_users/";
    } else {
        localPoiPath = filePathPrefix + "/real_world_pois/";
        localUserPath = filePathPrefix + "/real_world_users/";
```

```
}



SetGenerator sg;
int numNodes = graph.getNumNodes(), setSize;
std::vector<NodeID> sampleSet;
std::vector<NodeID> queryNodeSet;

std::string objSetOutputFile;
std::string userSetOutputFile;
std::string querySetOutputFile;
int totalSets = numSets;



for (std::size_t j = 0; j < objDensities.size(); ++j) {
    setSize = objDensities[j];

    for (std::size_t k = 0; k < numSets; ++k) {
        if (objDensities[j] != 1) {
            std::string tempPoiFilPath = localPoiPath + poiFileName;
            std::vector<NodeID> tempFacilityNodes =
                utility::getPointSetFromFile(tempPoiFilPath);

            std::cout << "number of nodes:" << numNodes << std::endl;

            std::cout << "temp facilaity nodes" << tempFacilityNodes.size() <<
                std::endl;



            std::cout << tempFacilityNodes.size() << std::endl;

            std::vector<NodeID> tempSet =
                sg.generateRandomSampleSetNotInSpecifiedSet(numNodes, setSize,
                tempFacilityNodes);
            sampleSet = std::vector<NodeID>(tempSet.begin(), tempSet.begin() +
                setSize);

            std::cout << "original sampleset size:" << sampleSet.size() <<
                std::endl;

        } else {
            sampleSet = graph.getNodesIDsVector();
        }



        if (objDensities[j] != 1 || k == 0) {

            userSetOutputFile = localUserPath + userFolderName +
                utility::constructObjsectSetFileName(
            graph.getNetworkName(),"random",objDensities[j],1,k);
            utility::writeSampleSet(userSetOutputFile,
            graph.getNetworkName(),"random",objDensities[j],setSize,1,sampleSet);

        }

        graph.resetAllObjects();
        sw.reset();
        sw.start();
        graph.parseObjectSet(sampleSet);
        sw.stop();
        totalINETime += sw.getTimeUs();
        totalINEMemory +=
            static_cast<double>(sizeof(NodeID)*sampleSet.size())/(1024*1024);

        if (objDensities[j] == 1) {
            totalSets = 1;
            break;
        } else {
            totalSets = numSets;
        }
```

```
        }
        double processingTimeMs = totalINETime/totalSets;
        double memoryUsage = totalINEMemory/totalSets*1024; // in KB

    }
    std::cout << "user and query sets successfully generated for " <<
        graph.getNetworkName() << std::endl;
}
```

Find cannot pruned facilities:

```
void ILBR::getKNNByNVDPHL2(Graph& graph,
                           ALT& alt, NVD& nvd,
                           unsigned int k,
                           NodeID queryNodeID,
                           std::vector<NodeID>& kNNs,
                           std::vector<EdgeWeight>& kNNDistances,
                           PrunedHighwayLabeling& phl,
                           std::map<NodeID, std::vector<std::pair<NodeID,
                               EdgeWeight>>>& maxDistPairMap,
                           std::set<NodeID>& noUserGeneratorSet,
                           int maxMaxDist,
                           std::set<NodeID>& cantPrunedSet,
                           std::map<NodeID, EdgeWeight>& cantPrunedDistMap,
                           NodeID& facilityQueryNodeOut)
{

    int factorX = 2;
    NodeID candidate = queryNodeID;
    google::dense_hash_set<NodeID> neighboursAdded;
    neighboursAdded.set_empty_key(constants::UNUSED_NODE_ID);
    BinaryMaxHeap<EdgeWeight,NodeID> knnCandidates;
    EdgeWeight spDist = 0, Dk = 0, candidateLBDist;
    bool DkInfinity = true;
```

```
BinaryMinHeap<EdgeWeight,NodeID> nvdQueue;
nvd.generateNeighbourCandidates(
alt,nvdQueue,queryNodeID,candidate,spDist,neighboursAdded);

facilityQueryNodeOut = queryNodeID;
while (nvdQueue.size() > 0 && (DkInfinity || nvdQueue.getMinKey() < Dk)) {
    candidateLBDist = nvdQueue.getMinKey();
    candidate = nvdQueue.extractMinElement();

    if (noUserGeneratorSet.find(candidate) == noUserGeneratorSet.end()) {
        EdgeWeight maxDist = maxDistPairMap[candidate][0].second;

        int euclideanFq = graph.getEuclideanDistance(candidate, queryNodeID);

        if (euclideanFq < maxDist) {

            int realFq = phl.Query(candidate, queryNodeID);
            if (realFq < maxDist) {
                cantPrunedSet.insert(candidate);
                cantPrunedDistMap[candidate] = realFq;
            } else {
                EdgeWeight shortestDist = realFq - maxDist;
                if (shortestDist > factorX * maxMaxDist) {

                    if((realFq - maxMaxDist) > factorX * maxMaxDist) {
                        break;
                    }
                } else {
                    if (shortestDist > factorX * maxDist) {
                        //can be prune
                    } else {
                        cantPrunedSet.insert(candidate);
                        cantPrunedDistMap[candidate] = realFq;
                    }
                }
            }
        }
        else
        {
            EdgeWeight shortestDist = euclideanFq - maxDist;
            if (shortestDist > factorX * maxMaxDist) {

                if((euclideanFq - maxMaxDist) > factorX * maxMaxDist) {
                    break;
                } else {

                    int realFq = phl.Query(candidate, queryNodeID);
                    if (realFq - maxMaxDist > factorX * maxMaxDist) {
                        break;
                    }
                }
            } else {
                int realFq = phl.Query(candidate, queryNodeID);
                shortestDist = realFq - maxDist;

                if (shortestDist > factorX * maxDist) {
                    //can be prune
                } else {

                    cantPrunedSet.insert(candidate);
                    cantPrunedDistMap[candidate] = realFq;
                }
            }
        }
    }

    if (DkInfinity) {

        spDist = phl.Query(queryNodeID,candidate);
        knnCandidates.insert(candidate,spDist);
        nvd.generateNeighbourCandidates(
        alt,nvdQueue,queryNodeID,candidate,spDist,neighboursAdded);
```

```
            if (knnCandidates.size() == k) {
                DkInfinity = false;
                Dk = knnCandidates.getMaxKey();
            }
        } else if (candidateLBDist < Dk) {
            spDist = phl.Query(queryNodeID,candidate);
            if (spDist < Dk) {
                knnCandidates.insert(candidate,spDist);
                nvd.generateNeighbourCandidates(
                alt,nvdQueue,queryNodeID,candidate,spDist,neighboursAdded);
                knnCandidates.extractMaxElement();
                Dk = knnCandidates.getMaxKey();
            }
        }
    }

    knnCandidates.populateKNNs(kNNs,kNNDistances);
}
```

User verification:

```
void test(Graph& graph, std::string altIdxFilePath, const std::string& method,
    const std::string& idxFilePath, std::vector<NodeID>& queryNodes, std::size_t
    numSets, std::vector<double> objDensities, std::vector<std::string>
    objTypes, std::vector<int> objVariable, const std::string& filePathPrefix,
    const std::string& statsOutputFile, bool verifyKNN,
    std::vector<std::string>& parameterKeys, std::vector<std::string>&
    parameterValues, const std::string& numLandmarks, const std::string&
    landmarkType, bool queryType, const std::string& poiNvdFileName, const
    std::string& userFolderName, const double factorX, const std::string&
    resultFile) {
    for (auto queryNodeIt = queryNodesFromFile.begin(); queryNodeIt !=
        queryNodesFromFile.begin()+100; ++queryNodeIt) {
        std::set<NodeID> cantPrunedSet;
        std::map<NodeID, EdgeWeight> cantPrunedDistMap;
        std::vector<NodeID> resultList;
```

```
NodeID facilityQueryNode;

ilbr.getKNNByNVDPHL2(
graph, alt,nvd,kValue,*queryNodeIt,kNNs,kNNDistances,phl,
generatorUserDistanceMap, noUserGeneratorSet, maxMaxDist, cantPrunedSet,
cantPrunedDistMap, facilityQueryNode);

resultList = generatorUserMap[facilityQueryNode];

int countCantPrunedUser = 0;

int totalForCounter = 0;
for(auto cantPruneGenerator = cantPrunedSet.begin(); cantPruneGenerator
    != cantPrunedSet.end(); ++cantPruneGenerator) {
   int fq = cantPrunedDistMap[*cantPruneGenerator];
   double bigR = (double)fq / (factorX - 1);
   double smallR = (double)fq / (factorX + 1);

   int forCounter = 0;
   for (auto tempUserDisPair =
       generatorUserDistanceMap[*cantPruneGenerator].begin();
        tempUserDisPair !=
           generatorUserDistanceMap[*cantPruneGenerator].end();
           ++tempUserDisPair) {
      forCounter++;
      NodeID userNode = tempUserDisPair->first;
      EdgeWeight uf = tempUserDisPair->second;

      if (fq > uf && (fq - uf > factorX * uf)) {
         break;
      } else if (uf >= bigR) {
         resultList.push_back(userNode);
      } else if (uf > smallR && uf < bigR) {
         countCantPrunedUser++;
         int uq = phl.Query(facilityQueryNode, userNode);
         if (uq <= factorX * uf) {
```

```
                        resultList.push_back(userNode);
                    }
                }
            }
            totalForCounter += forCounter;


        }
    }
}
```

B.2 Continuous RANN query on road network

Add query:

```
void ExperimentsCommand::addQueryNewRecordDistance(ILBR &ilbr, Graph &graph, ALT
    &alt, NVD &nvd, int &kValue, NodeID &queryNodeIt, std::vector<NodeID> &kNNs,
    std::vector<EdgeWeight> &kNNDistances, PrunedHighwayLabeling &phl,
    std::map<NodeID, std::pair<NodeID, EdgeWeight>> &generatorUserMaxDistMap,
    std::set<NodeID> &noUserGeneratorSet, int &maxMaxDist, std::set<NodeID>
    &cantPrunedSet, std::map<NodeID, EdgeWeight> &cantPrunedDistMap, NodeID
    &facilityQueryNode, std::set<NodeID> &resultSet, std::map<NodeID,
    std::set<NodeID>> &generatorUserMap, const double factorX, std::map<NodeID,
    std::set<NodeID>> &voronoiCellSigSetMap, std::map<NodeID, std::set<NodeID>>
    &userBelongQuerySetMap, std::map<NodeID, std::vector<std::pair<NodeID,
    EdgeWeight>>> &generatorNodeDistanceMap, std::map<NodeID, std::set<NodeID>>
    &generatorNodeMap, std::map<std::pair<NodeID, NodeID>,
    std::map<std::pair<NodeID, NodeID>, double>>
    &queryFacilityInfzoneSegmentsMap, std::map<std::pair<NodeID, NodeID>,
    std::map<std::pair<NodeID, NodeID>, double>>
    &queryFacilityNotInfzoneSegmentsMap
) {



    getSingleNVDPHL2(ilbr, graph, alt, nvd, kValue,
                  queryNodeIt, kNNs, kNNDistances,
                  phl, generatorUserMaxDistMap,
                  noUserGeneratorSet, maxMaxDist, cantPrunedSet,
                  cantPrunedDistMap, facilityQueryNode, resultSet,
                  generatorUserMap, factorX);



  for (auto cantPrunedGenerator : cantPrunedSet) {

       int fq = cantPrunedDistMap[cantPrunedGenerator];
       double bigR = (double)fq / (factorX - 1);
       double smallR = (double)fq / (factorX + 1);

       std::map<std::pair<NodeID, NodeID>, double>
           queryInfzoneSegmentsInOneFacility;
       std::map<std::pair<NodeID, NodeID>, double>
           queryNotInfzoneSegmentsInOneFacility;

       std::set<NodeID> isResultNodes;

       bool pruneAll = false;
       for (auto tempNodeDistancePair :
           generatorNodeDistanceMap[cantPrunedGenerator]) {

           NodeID currentNode = tempNodeDistancePair.first;
           int uf = tempNodeDistancePair.second;

           if (pruneAll) {

              checkConnectedNodeWhenNotResult(graph, nvd, queryNodeIt, phl,
                                         queryInfzoneSegmentsInOneFacility,
                                         queryNotInfzoneSegmentsInOneFacility,
                                         isResultNodes, currentNode,
                                         cantPrunedGenerator,
                                         bigR, smallR, factorX, uf);
```

```
} else if (isResultNodes.find(currentNode) != isResultNodes.end()) {
    checkConnectedNodeWhenIsResult(graph, nvd, queryNodeIt, phl,

                                   queryInfzoneSegmentsInOneFacility,
                                   queryNotInfzoneSegmentsInOneFacility,
                                   isResultNodes, currentNode,
                                   cantPrunedGenerator,
                                   bigR, smallR, factorX, uf);
} else {
    if (fq > uf && (fq - uf > factorX * uf)) {
        pruneAll = true;
        checkConnectedNodeWhenNotResult(graph, nvd, queryNodeIt, phl,
                                        queryInfzoneSegmentsInOneFacility,
                                        queryNotInfzoneSegmentsInOneFacility,
                                        isResultNodes, currentNode,
                                        cantPrunedGenerator,
                                        bigR, smallR, factorX, uf);
    } else if (uf >= bigR) {
        checkConnectedNodeWhenIsResult(graph, nvd, queryNodeIt, phl,
                                       queryInfzoneSegmentsInOneFacility,
                                       queryNotInfzoneSegmentsInOneFacility,
                                       isResultNodes, currentNode,
                                       cantPrunedGenerator,
                                       bigR, smallR, factorX, uf);

        isResultNodes.insert(currentNode);
    } else if (uf > smallR && uf < bigR) {
        int uq = phl.Query(queryNodeIt, currentNode);
        if (uq <= uf * factorX) {
            checkConnectedNodeWhenIsResult(graph, nvd, queryNodeIt, phl,
                                           queryInfzoneSegmentsInOneFacility,
                                           queryNotInfzoneSegmentsInOneFacility,
                                           isResultNodes, currentNode,
                                           cantPrunedGenerator,
                                           bigR, smallR, factorX, uf);
```

```
                isResultNodes.insert(currentNode);
            } else {
                checkConnectedNodeWhenNotResult(graph, nvd, queryNodeIt,
                    phl,
                                                queryInfzoneSegmentsInOneFacility,
                                                queryNotInfzoneSegmentsInOneFacility,
                                                isResultNodes, currentNode,
                                                cantPrunedGenerator,
                                                bigR, smallR, factorX, uf);

            }
        } else {
            checkConnectedNodeWhenNotResult(graph, nvd, queryNodeIt, phl,
                                            queryInfzoneSegmentsInOneFacility,
                                            queryNotInfzoneSegmentsInOneFacility,
                                            isResultNodes, currentNode,
                                            cantPrunedGenerator,
                                            bigR, smallR, factorX, uf);

        }
    }
}

queryFacilityInfzoneSegmentsMap[std::make_pair(queryNodeIt,
    cantPrunedGenerator)] = queryInfzoneSegmentsInOneFacility;
voronoiCellSigSetMap[cantPrunedGenerator].insert(queryNodeIt);

}

voronoiCellSigSetMap[queryNodeIt].insert(queryNodeIt);

std::map<std::pair<NodeID, NodeID>, double> queryInfzoneSegmentsInQuery;


for (auto tempNodeDistancePair : generatorNodeDistanceMap[queryNodeIt]) {
    int currentNode = tempNodeDistancePair.first;
```

```
        int uf = tempNodeDistancePair.second;

        for (int i = graph.firstEdgeIndex[currentNode]; i <
            graph.firstEdgeIndex[currentNode + 1]; i++) {
            NodeID anotherEnd = graph.edges[i].first;
            EdgeWeight anotherEndUserDist = graph.edges[i].second;
            EdgeWeight anotherEndf = nvd.nodeDRIDs[anotherEnd].second;



            if (nvd.drObjectIDs[nvd.nodeDRIDs[anotherEnd].first] != queryNodeIt) {
                double safeDistance = (double)(anotherEndf + anotherEndUserDist -
                    uf) / 2;
                queryInfzoneSegmentsInQuery[std::make_pair(currentNode,
                    anotherEnd)] = safeDistance;
            } else {
                queryInfzoneSegmentsInQuery[std::make_pair(currentNode,
                    anotherEnd)] = anotherEndUserDist;
            }
        }
    }



    queryFacilityInfzoneSegmentsMap[std::make_pair(queryNodeIt, queryNodeIt)] =
        queryInfzoneSegmentsInQuery;




    for (auto result : resultSet) {
        userBelongQuerySetMap[result].insert(queryNodeIt);
    }

}
```

Add user:

```
void ExperimentsCommand::addUser(NVD &nvd, NodeID user,
        std::map<NodeID, std::set<NodeID>> &generatorUserMap,
        std::set<NodeID> &noUserGeneratorSet,
        std::map<NodeID, std::set<NodeID>> &voronoiCellSigSetMap,
        std::map<NodeID, std::set<NodeID>> &userBelongQuerySetMap,
        std::map<std::pair<NodeID, NodeID>,
        std::map<std::pair<NodeID, NodeID>, double>>
            &queryFacilityInfzoneSegmentsMap,
        std::map<std::pair<NodeID, NodeID>,
        std::map<std::pair<NodeID, NodeID>, double>>
            &queryFacilityNotInfzoneSegmentsMap

) {
    NodeID nearestFacility;
    EdgeWeight uf;

    nvd.getNearestNeighbour(user,nearestFacility,uf);

    auto it = noUserGeneratorSet.find(nearestFacility);
    if (it != noUserGeneratorSet.end()) {
        noUserGeneratorSet.erase(it);
    }
    generatorUserMap[nearestFacility].insert(user);



    if (!voronoiCellSigSetMap[nearestFacility].empty()) {
        for (auto queryNode : voronoiCellSigSetMap[nearestFacility]) {
            std::pair<NodeID, NodeID> tempQueryFacility =
                std::make_pair(queryNode, nearestFacility);

            bool userIsResult = false;
            for (auto itSegment =
                queryFacilityInfzoneSegmentsMap[tempQueryFacility].begin();
                itSegment !=
                queryFacilityInfzoneSegmentsMap[tempQueryFacility].end();
                itSegment++) {
                if (itSegment->first.first == user) {
```

```
                userBelongQuerySetMap[user].insert(queryNode);
                userIsResult = true;
                userQueryMinimumSafeZoneSegmentMap[std::make_pair(user,
                    queryNode)] =
                    queryFacilityInfzoneSegmentsMap[tempQueryFacility];
                break;
            }
        }
        if (!userIsResult) {
            userQueryMinimumSafeZoneSegmentMap[std::make_pair(user,
                queryNode)] =
                queryFacilityNotInfzoneSegmentsMap[tempQueryFacility];
        }
    }
  }
}
```

Monitoring user moving:

```
void ExperimentsCommand::monitorMoving() {
    std::vector<std::vector<MovingNode>> allMovingUsers;
    readMovingNodesFromFile(movingSetFile, userNodes, graph, allMovingUsers);
    for (auto &movingUserAtTimestamp : allMovingUsers) {
            for (auto & tmpMovingNode : movingUserAtTimestamp) {

                NodeID startNode = tmpMovingNode.getMovingNodeId();

                NodeID nextNode = tmpMovingNode.getNextNodeId();

                NodeID edgeStartNode;
                if (movingRouteMap[startNode].empty()) {
                    edgeStartNode = startNode;
                    movingRouteMap[startNode].push_back(startNode);
                } else {
                    edgeStartNode = movingRouteMap[startNode].back();
                }
```

```
NodeID edgeEndNode = nextNode;



NodeID edgeStartNearestFacility;
EdgeWeight edgeStartSpdist;
nvd.getNearestNeighbour(edgeStartNode,
    edgeStartNearestFacility, edgeStartSpdist);

NodeID edgeEndNearestFacility;
EdgeWeight edgeEndSpdist;
nvd.getNearestNeighbour(edgeEndNode, edgeEndNearestFacility,
    edgeEndSpdist);

if (nextNode != edgeStartNode) {
    isResultUserQuerySafeDistanceMap.clear();
    notResultUserQuerySafeDistanceMap.clear();

    EdgeWeight startEndEdgeWeight = 0;
    bool oneEdgeFlag = false;
    std::set<NodeID> connectStartSet;
    for (int i = graph.firstEdgeIndex[edgeStartNode]; i <
        graph.firstEdgeIndex[edgeStartNode + 1]; i++) {
        if (graph.edges[i].first == nextNode) {
            oneEdgeFlag = true;
            startEndEdgeWeight = graph.edges[i].second;
            break;
        } else {
            connectStartSet.insert(graph.edges[i].first);
        }

    }
    if (!oneEdgeFlag) {
        for (int i = graph.firstEdgeIndex[nextNode]; i <
            graph.firstEdgeIndex[nextNode + 1]; i++) {
            if
                (!connectStartSet.insert(graph.edges[i].first).second)
                {
                edgeStartNode = graph.edges[i].first;
                startEndEdgeWeight = graph.edges[i].second;
            }
        }

        movingRouteMap[startNode].push_back(edgeStartNode);
    }
    movingRouteMap[startNode].push_back(nextNode);

    if (edgeStartNearestFacility == edgeEndNearestFacility) {
        for (auto tempQuery :
            voronoiCellSigSetMap[edgeStartNearestFacility]) {
            bool edgeStartNodeIsResult =
            userBelongQuerySetMap[edgeStartNode].find(tempQuery)
                !=
            userBelongQuerySetMap[edgeStartNode].end();
            bool edgeEndNodeIsResult =
            userBelongQuerySetMap[edgeEndNode].find(tempQuery) !=
            userBelongQuerySetMap[edgeEndNode].end();
            std::pair<NodeID, NodeID> tempEdgeStartUserQuery =
            std::make_pair(edgeStartNode, tempQuery);
            std::pair<NodeID, NodeID> tempQueryEdgeStartFacility
                =
            std::make_pair(tempQuery, edgeStartNearestFacility);

            if (startEndEdgeWeight == 0) {
                continue;
            }
            int distanceStartEnd =
            queryFacilityInfzoneSegmentsMap[tempQueryEdgeStartFacility]
            [std::make_pair(edgeStartNode, edgeEndNode)];
            int distanceEndStart =
            queryFacilityInfzoneSegmentsMap[tempQueryEdgeStartFacility]
        [std::make_pair(edgeEndNode, edgeStartNode)];

        if ((!edgeStartNodeIsResult &&
            !edgeEndNodeIsResult)) {
            if (distanceStartEnd != startEndEdgeWeight) {
                notResultUserQuerySafeDistanceMap
                [tempEdgeStartUserQuery] = distanceEndStart;
                notResultUserQuerySafeDistanceMap
                [std::make_pair(edgeEndNode, tempQuery)] =
                    distanceEndStart;
            }
        } else if (edgeStartNodeIsResult &&
            !edgeEndNodeIsResult) {
            isResultUserQuerySafeDistanceMap
            [tempEdgeStartUserQuery] = distanceStartEnd;
        } else if (!edgeStartNodeIsResult &&
            edgeEndNodeIsResult) {
            notResultUserQuerySafeDistanceMap
            [tempEdgeStartUserQuery] = distanceEndStart;
        }
    }
} else {

    std::set<NodeID> combinedSet;
    combinedSet.insert(
    userBelongQuerySetMap[edgeStartNode].begin(),
    userBelongQuerySetMap[edgeStartNode].end());
    combinedSet.insert(
    voronoiCellSigSetMap[edgeEndNearestFacility].begin(),
    voronoiCellSigSetMap[edgeEndNearestFacility].end());

    for (auto tempQuery : combinedSet) {
        bool edgeStartNodeIsResult = userBelongQuerySetMap
        [edgeStartNode].find(tempQuery) !=
        userBelongQuerySetMap[edgeStartNode].end();
```

```
std::pair<NodeID, NodeID> tempEdgeStartUserQuery =
std::make_pair(edgeStartNode, tempQuery);
std::pair<NodeID, NodeID> tempQueryEdgeStartFacility
    =
std::make_pair(tempQuery, edgeStartNearestFacility);
std::pair<NodeID, NodeID> tempQueryEdgeEndFacility =
std::make_pair(tempQuery, edgeEndNearestFacility);

if (edgeStartNodeIsResult) {
    bool startHalfSafe =
        queryFacilityInfzoneSegmentsMap
    [tempQueryEdgeStartFacility]
    .find(std::make_pair(edgeStartNode, edgeEndNode))
        !=
    queryFacilityInfzoneSegmentsMap
    [tempQueryEdgeStartFacility].end();
    bool endHalfSafe =
    queryFacilityInfzoneSegmentsMap
    [tempQueryEdgeEndFacility]
    .find(std::make_pair(edgeEndNode, edgeStartNode))
        !=
    queryFacilityInfzoneSegmentsMap
    [tempQueryEdgeEndFacility].end();
        if (startHalfSafe) {
        isResultUserQuerySafeDistanceMap
        [tempEdgeStartUserQuery] =
        queryFacilityInfzoneSegmentsMap
        [tempQueryEdgeStartFacility]
        [std::make_pair(edgeStartNode, edgeEndNode)];
    } else if (endHalfSafe) {
        notResultUserQuerySafeDistanceMap
        [tempEdgeStartUserQuery] =
        queryFacilityInfzoneSegmentsMap
        [tempQueryEdgeStartFacility]
        [std::make_pair(edgeEndNode, edgeStartNode)];
    }
```

```
                    } else {
                        if (queryFacilityInfzoneSegmentsMap
                        [tempQueryEdgeEndFacility]
                        .find(std::make_pair(edgeEndNode, edgeStartNode))
                            !=
                            queryFacilityInfzoneSegmentsMap
                            [tempQueryEdgeEndFacility].end()) {
                            notResultUserQuerySafeDistanceMap
                            [tempEdgeStartUserQuery] =
                            queryFacilityInfzoneSegmentsMap
                            [tempQueryEdgeEndFacility]
                            [std::make_pair(edgeEndNode, edgeStartNode)];
                        } else {
                            //no change
                        }
                    }
                }

            }
        } else {
            int edgeStartX;
            int edgeStartY;
            graph.getCoordinates(edgeStartNode, edgeStartX, edgeStartY);

            double inlineX = tmpMovingNode.getCurrentX();
            double inlineY = tmpMovingNode.getCurrentY();

            int edgeEndX = tmpMovingNode.getNextNodeX();
            int edgeEndY = tmpMovingNode.getNextNodeY();

            double distStart = std::sqrt(
                    (edgeStartX - inlineX) * (edgeStartX - inlineX) +
                        (edgeStartY - inlineY) * (edgeStartY - inlineY));
            double distEnd = std::sqrt(
                    (edgeEndX - inlineX) * (edgeEndX - inlineX) +
                        (edgeEndY - inlineY) * (edgeEndY - inlineY));
                    for (auto temp : isResultUserQuerySafeDistanceMap) {
                        if (distStart < temp.second) {
                            //no change
                        } else {
                            queryResultSetMap[temp.first.second]
                            .erase(edgeStartNode);
                            userBelongQuerySetMap[edgeStartNode]
                            .erase(temp.first.second);
                        }
                    }

                    for (auto temp : notResultUserQuerySafeDistanceMap) {
                        if (distEnd > temp.second) {
                            //no change
                        } else {
                            queryResultSetMap[temp.first.second]
                            .insert(edgeEndNode);
                            userBelongQuerySetMap[edgeEndNode]
                            .insert(temp.first.second);
                        }
                    }
                }
            }
        }
    }
```

B.3 Spatial Reverse Approximate Top (SRAT) query

Prune MBR:

```
bool isMBRPruned(int qID, CNode* &currentNode, double my_xFactor, double*
    &facility_static_score_table, vector<double> &weight_vector, map<int,
    Query*> &queryTable)
```

```
{
    double minDist = currentNode->getRect()->getMinDist2D(queryTable[qID]->loc);
    if (minDist < EPSILON) {
        return false;
    }
    double q_static_score = facility_static_score_table[qID];
    double f_max_static_score = currentNode->m_maxStaticScore;
    double delta = (my_xFactor * f_max_static_score - q_static_score) /
        weight_vector[0];

    double m_maxVDist = currentNode->m_maxDist;

    if((minDist - delta) / (my_xFactor + 1) > m_maxVDist) {
        return true;
    }

    return false;
}
```

Check futile MBR:

```
bool isMBRAllResult(int qID, CNode* &currentNode, double my_xFactor, double*
    &facility_static_score_table, vector<double> &weight_vector, map<int,
    Query*> &queryTable) {
    double min_delta = (my_xFactor * currentNode->m_minStaticScore -
        facility_static_score_table[qID]) / weight_vector[0];
    double max_q_f = currentNode->getRect()->getMaxDist(queryTable[qID]->loc);

    if (min_delta >= max_q_f) {
        return true;
    } else {
        return false;
    }
}
```

Prune a *Voronoi* cell:

```
bool isVoronoiCellPrunedForBasicVoronoi(int qID, int facility_ID, double*
    &facility_static_score_table, double my_xFactor, vector<double>
    &weight_vector, CPoint** &facilityTable, map<int, Query*> &queryTable,
    VCell** &voronoiTable, double delta, double dist_q_f)
{

    double q_static_score = facility_static_score_table[qID];
    double f_static_score = facility_static_score_table[facility_ID];

    if (delta > 0) {
        for (int i = 0; i < voronoiTable[facility_ID]->fpolygon.size(); i++) {
            auto corner = voronoiTable[facility_ID]->fpolygon[i];
            double query_score = corner->loc->getDist2D(queryTable[qID]->loc) *
                weight_vector[0] + q_static_score;
            double f_score = my_xFactor * (corner->distFromQ * weight_vector[0] +
                f_static_score);

            if (query_score <= f_score) {
                return false;
            }
        }
    } else if (delta <= 0) {
        if ((dist_q_f - delta) / (my_xFactor + 1) >=
            voronoiTable[facility_ID]->maxVertex) {
            return true;
        }
    }

    return true;
}
```

Get cannot pruned and futile facilities:

```
void getPrunedAndResultFacilities(int qID, CRTree* &Ftree,
                                  CPoint** &facilityTable,
                                  double* &facility_static_score_table,
                        map<int, Query*> &queryTable, VCell** &voronoiTable,
                        double my_xFactor, vector<double> &weight_vector)
{

   CHeap heap;
   Ftree->getRoot()->getRect()->blowHuge();
   heap.insertNode(0,Ftree->getRoot());

   CNode* curNode;
   CHeapNode* heapNode;
   int numChildren;
   CEntry* child;

   while(heap.getHeapSize() != 0) {
      heapNode = heap.top();
      curNode = (CNode*)heapNode->getData();
      heap.pop();
      if (curNode->m_id >= 0) {

         if (isMBRPruned(qID, curNode, my_xFactor, facility_static_score_table,
             weight_vector, queryTable)) {
            continue;
         } else if (isMBRAllResult(qID, curNode, my_xFactor,
             facility_static_score_table, weight_vector, queryTable)) {
            vector<CNode*> allf;
            Ftree->getAllObjects(curNode, &allf);
            for (auto t_node : allf) {
               int fID = t_node->m_id * -1;
               double dist_q_f =
                   facilityTable[fID]->getDist2D(queryTable[qID]->loc);
               double min_q_score = (dist_q_f - curNode->m_maxDist) *
                   weight_vector[0] + facility_static_score_table[qID];
               queryTable[qID]->all_result_f_my_obj.insert(new myobj(fID,
                   min_q_score));
            }
            }
            else {
                curNode->load();

                numChildren = curNode->getNumChild();
                for(int i = 0; i < numChildren; i++) {
                    child = curNode->getEntry(i);
                    heap.insertNode(child->getRect()->getMinDist(queryTable[qID]->loc),
                        (CNode*) child);
                }
            }
        } else {
            int fID = curNode->m_id * (-1);
            if (fID != qID){
                double dist_q_f =
                    facilityTable[fID]->getDist2D(queryTable[qID]->loc);
                double min_q_score = (dist_q_f - curNode->m_maxDist) *
                    weight_vector[0] + facility_static_score_table[qID];
                double max_f_score = my_xFactor * curNode->m_maxDist *
                    weight_vector[0] + my_xFactor * curNode->m_maxStaticScore;
                double delta = (my_xFactor * facility_static_score_table[fID] -
                    facility_static_score_table[qID]) / weight_vector[0];
                if (min_q_score > max_f_score) {
                    continue;
                } else if (delta >= dist_q_f) {
                    queryTable[qID]->all_result_f_my_obj.insert(new myobj(fID,
                        min_q_score));
                }
                else {
                    if (!isVoronoiCellPrunedForBasicVoronoi(qID, fID,
                        facility_static_score_table, my_xFactor,
                                                        weight_vector,
                                                            facilityTable, queryTable,
                                                        voronoiTable, delta,
                                                            dist_q_f)) {
```

```
                        queryTable[qID]->cant_pruned_f_my_obj.insert(new myobj(fID,
                            min_q_score));
                    }
                }
            }
        }
    }
    heap.clear();
}
```

User verification:

```
void useRtreeMethod(vector<CPoint*> &allUsers, map<int, Query*> &queryTable,
            double my_xFactor, CPoint** &facilityTable,
            double* &facility_static_score_table,
            vector<double> &weight_vector, CRTree* &Ftree,
            int STATIC_DIMENSION, VCell** &voronoiTable,
            double max_user_nearest_not_top1_score,
            double max_user_nearest_score) {

    for (auto query : queryTable) {

        getPrunedAndResultFacilities(query.first, Ftree, facilityTable,
                                     facility_static_score_table,
                                     queryTable, voronoiTable,
                                     my_xFactor, weight_vector);

        for (auto temp_f : queryTable[query.first]->cant_pruned_f_my_obj) {
            int fID = temp_f->facility_id;
            double min_u_q_score = temp_f->min_u_q_score;

            if (min_u_q_score >= max_user_nearest_score) {
                break;
            }
            for (auto temp: facilityTable[fID]->nearest_user_f_dist_vec) {
                if (min_u_q_score >= temp.second) {
```

```
                break;
            }
            int uid = temp.first;
            CPoint *user = allUsers[uid-1];
            double user_query_score = user->getDist2D(query.second->loc) *
                weight_vector[0] + facility_static_score_table[query.first];
            if (user_query_score < temp.second) {
                query.second->new_voronoi_rtk.insert(uid);
            }
        }
    }
  }
}
```