Range and Region Query Processing in Spatial Databases

by

Kefeng Xuan, MInforTech

Thesis

Submitted by Kefeng Xuan

for fulfillment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisor: A/Professor David Taniar

Associate Supervisor: Professor Bala Srinivasan

Clayton School of Information Technology
Monash University

April, 2013

Copyright Notices

Notice 1

Under the Copyright Act 1968, this thesis must be used only under the normal conditions of scholarly fair dealing. In particular no results or conclusions should be extracted from it, nor should it be copied or closely paraphrased in whole or in part without the written consent of the author. Proper written acknowledgement should be made for any assistance obtained from this thesis.

Notice 2

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.

© Copyright

by

Kefeng Xuan

2013

To the people I love

Contents

List of Tables

List of Figures

Range and Region Query Processing in Spatial Databases

Kefeng Xuan, MInforTech
[redacted]
Monash University, 2013

Supervisor: A/Professor David Taniar
[redacted]
Associate Supervisor: Professor Bala Srinivasan
[redacted]

Abstract

With the boom of spatial databases, more and more spatial queries, which play a significant role in many academic and industrial areas, are proposed and studied extensively in last decade. One of the most fundamental queries among these is range search which returns all objects of interest within the pre-defined area. Because of the importance of the spatial queries, a mass of researches concentrated on processing various queries in spatial databases, especially, for k nearest neighbors (kNN) queries and its variations. However, as the fundamental query in spatial databases, range search queries have received far less attention. The existing works cannot process range queries efficiently, especially, in *non-Euclidean space* or on *moving objects*. Furthermore, the existing works for spatial queries retrieve point object only, none of them can find non-point objects, due to the difficulties of representing and indexing such objects in spatial databases. Motivated by above outstanding problems, we discuss several novel range and region queries and provide efficient solutions in spatial databases in this thesis. The following paragraphs describe our contribution.

In the first part, we present several algorithms to process ***point-expected range queries*** that retrieve spatial objects within a specific distance from a query point. We are the first to investigate range queries under many different *practical constraints*. We conduct theoretical analysis to show the precise and effectiveness of

our algorithms. The extensive experimental results provide the practical evidence for our theoretical analysis. Then we discuss point-expected range queries in a *dynamic circumstance*, where the query or the objects of interest are moving continuously. Our experiment results demonstrate that our approach outperforms the existing techniques in most instances.. Thereinto, our algorithms of constrained range queries are base-on Voronoi expansion rather than incremental expansion methods, thus the *response time and I/O access* of range query is much faster than using exiting works. Meanwhile they queries diversify the range queries in spatial databases and solved many novel queries in spatial databases. Our algorithms designed for monitoring range queries involving any moving objects *reduce the computation and communication* cost significantly comparing with others.

In the second part, we propose a new class of range queries, named, ***region-expected range queries***, which find an (some) area(s) according to the location of the given set of objects. Because with the extremely progress of geographic information system, the typical queries in spatial databases cannot fulfill the users' requirements. In this part, we focus on two queries in this class, namely, k*NN region queries* and *optimum region queries.* We are the first to study this sort of range queries in spatial databases. We provide two algorithms for each query, and analyze their performances based on abundant theoretical illation and extensive experiment results. We are the first to investigate retrieving non-point objects in spatial databases. This sorts of spatial queries provide rich functionality in industrial and commercial areas, including, geographic information systems, decision support systems and so forth.

Range and Region Query Processing in Spatial Databases

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Kefeng Xuan
April 22, 2013

Acknowledgments

I would like to thank everyone who helped to make this possible. It has been an incredible journey of self-discovery, and I love everyone of you...

First of all, I would like to express my gratitude to my supervisor, Associate Professor David Taniar, for his visionary guidance, insightful supervision and inspiriting encouragement. It is not often that a supervisor always finds the time for listening to little problems that unavoidably crop up in the course of performing research and whenever I have the inspiration. As an archaism saying: "A teacher for even one day; My mentor of all my life". Not only is he a spiritual father, he is also support my life like a family member. Words cannot express how indebted I am. He deserves a special appreciation for allowing me to conduct research at home in China when I needed to be with my family, and the appreciation also for that, David gave a brilliant name for my son.

I am also very thankful to my joint-supervisor Prof. Bala Srinivasan especially for teaching me numerable lessons and insights on the workings of academic research in general and his valuable editorial and technical advices are the essential to the completion of my thesis.

I would like sent my special appreciation to my parents. I cannot find any proper word to appreciate their love that is always surround me, whenever and wherever I am depressed, sad, delighted and excited. Parents' love, likes a mountain, great and is generous; Parents' love, likes that tree root, firm and is deep; Parents' love, likes the honey which that hundred flowers breeds, but is happy. Thanks Dad; Thanks Mum.

Lastly but the most importantly, I am thankful to my love, Geng. I cannot imagined that what sort of my life will be. She gave me many self-giving assistances, not only on my daily life, but also on my research. I am always so proud that we can inspirit, enlighten and assist each other incessantly.

Publications resulting from this thesis: Below is a list of publications resulting from this thesis. I am very appreciate all of the people who collaborated with me for these publications. Their comments and suggestions are very valuable and insightful.

1. Xuan, K., Zhao, G., Taniar, D., Safar, M. and Srinivasan, B. and Gavrilova, M.L. (2009), Network Voronoi Diagram Based Range Search. **In The 23rd International Conference on Advanced Information Networking and Applications**, pages 741-748, Bradford, United Kingdom, May 2009. (**Best Paper Award**)

2. Xuan, K., Taniar, D., Safar, M. and Srinivasan, B. (2010), Time constrained range search queries over moving objects in road networks. **In The 8th International Conference on Advances in Mobile Computing and Multimedia**, pages 329C336, Paris, France, November 2010.

3. Xuan, K., Zhao, G., Taniar, D., Rahayu, J.W., Safar, M. and Srinivasan, B., Voronoi-based range and continuous range query processing in mobile databases. **J. Comput. Syst. Sci. (JCSS)**, 77(4):637-651, 2011. (**A***)

4. Xuan, K., Zhao, G., Taniar, D., Safar, M. and Srinivasan, B., Constrained range search query processing on road networks. **Concurrency and Computation: Practice and Experience (CONCURRENCY)**, 23(5):491-04, 2011. (**A**)

5. Xuan, K., Zhao, G., Taniar, D., Safar, M. and Srinivasan, B., Voronoi-based multi-level range search in mobile navigation. **Multimedia Tools Appl.**, 53(2):459-479, 2011. (**B**)

Kefeng Xuan

Monash University

April 2013

Chapter 1

Introduction

1.1 Overview

Along with the rapid development of mobile devices and the blossom of Location Based Services (LBS) , it has the pressing need for modeling, storing and querying spatial data. Therefore most of the commercial database software service providers, such as, Oracle, PostGIS, IBM DB2, start a new round of competition on spatial database. In general, a spatial database system is a database that offers spatial data types in its data model and query language, and also provides spatial indexing and efficient algorithms for spatial query processing [Gut94]. In various fields, there is a need for spatial data, such as, Computer Aided Design (CAD), Very-Large-Scale Integration (VLSI), satellite image system, spatial measurement, and multimedia system.

The objects in a spatial database are representations of real-world entities with associated spatial attributes and they are structured by one or more basic spatial data types, including, point data, line data, and polygon data. Fig. 1.1 displays a spatial query that it is finding petrol stations near Monash University Clayton Campus, which utilizes these three data types in Google Maps, where all the petrol stations, $A{\sim}E$ are abstracted as point objects, which is the simplest type of spatial object. A point object is used to represent the real-world entity whose location is the only important spatial attributes. Normally, the location of a point object is a

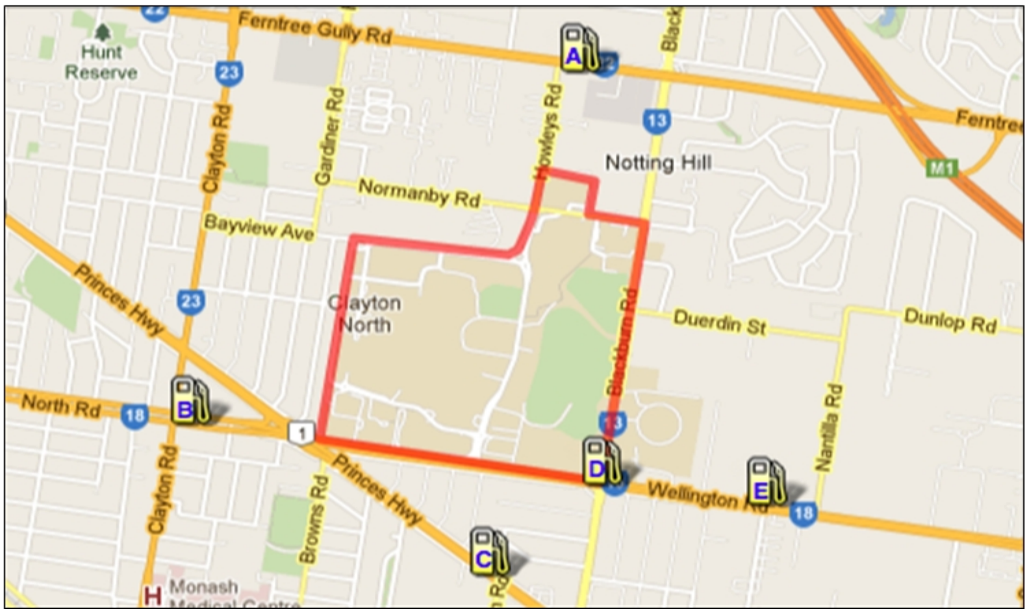


Figure 1.1: Spatial objects in Google Maps

pair of coordinates and each pair is stored as one row alone or with some extended non-spatial information. Therefore each point object is independent of every other point object, represented as a separate row in the database model. In Fig. 1.1, the road networks are represented as line data type. Similarly, a line data can also represent other infrastructure and nature networks, e.g., railways, river channels and airline routes. A line object is composed of two point objects, two ends of this line and a metrical attribute for its length. The last data type, polygon data, represents an entity for which its extent is also very important. As shown in Fig. 1.1, Monash University Clayton Campus is represented by a polygon. There are many other entities can be represented by one or more joint or disjoint polygons, such as, natural resource zones, socio-economic zones and land records. It needs to be clarified that a entity can be represented as a point or a polygon depends on the scale of the map.

In order to manipulate, retrieve and query the spatial data, special indexes and query processing algorithms need to be designed. In Section 1.2, we clarity the objectives and motivation of this thesis. In Section 1.3, we explain why traditional indexes cannot be used for spatial data and list the outstanding problems of the spatial indexes and existing approaches. In Section 1.4, we summarize the contributions of this thesis towards spatial databases. In Section 1.5, we present the organization of this thesis.

1.2 Objectives and Motivations

Range queries can be beneficial in the context of location based services , spatial data mining and decision support system, by finding objects of interest within a specific distance or discovering areas having particular spatial features in a metric space. Because of its importance, it has been studied in many different spaces for various of objects. Before we start to discuss the objective of this thesis. The scopes of this thesis need to be clarified.

The two most common used spaces in real applications are, *Euclidean space*, such as, in the air and *network spaces*, including road networks, waterway and so forth. In Euclidean space, such as, the distance between any two points is measured by the length of their straight connection; While the distances in a network space are calculated by the shortest connections through the underlying networks. The status of spatial objects is also various, they can be classified as *static* or *moving* in term of state of motion, or classified into *points*, *lines* or *polygons]*in term of shape. This thesis covers processing both static and moving objects or query, retrieving point polygon objects in either Euclidean or network spaces.

The main objective of this research is to investigate how to reduce the *computation cost* on the database server and the *communication cost* between spatial objects and the server. Moreover, this research will also investigate the representation of the spatial objects to handle complex range queries.

This research is mainly motivated by the following two facts. The first fact is that the state-of-arts of range query processing cannot process efficiently in two circumstances, namely, processing range queries in a non-Euclidean space and processing range queries on moving queries or objects. In the first circumstance, the existing works require very high computation time to expand the underlying networks incrementally; In the other circumstance, the traditional query processing algorithms will cause frequent communication in order to ensure the consistency of the spatial data and the precise of the query result. Meanwhile, the prompt development of mobile devices also demands that the spatial database can handle moving queries and objects efficiently.

The second fact is the deficient of the spatial queries and algorithms. Although spatial queries has been studied extensively in last two decades, most works mainly deal with a few types of traditional queries, such as kNN, range, closest pairs and skylines. Thus it is not difficult to picture that the existing works cannot fulfill the requirements, due to the increasing popularity of the spatial data, especially, in online maps, (including, Google Maps, Bing Maps and Apple Maps) and location-based services (e.g., GPS). Moreover, spatial queries are still restricted to retrieve point objects. Far less attention has been given to other spatial objects in the existing works, due to the difficulty of the representation of non-point objects.

1.3 Major Problems

1.3.1 Limitations of Traditional Database Indexes

A traditional database index, which can be created using one or more columns of a database table, is a data structure that improves the efficiency of retrieving data. But a main problem is that either one column or multiple column indexes cannot answer spatial queries efficiently. Using the following range search query as an example, we explain this main problem. A user wants to find all the petrol stations within 10km from the current location. A straightforward method is to calculate the distances of all the petrol stations to the current location of the user and report those petrol stations whose distance is smaller than 10km. But this non-index method needs to access all the objects (petrol stations) stored in the spatial database. Alternatively, we can create a one-column index or a two-columns index on petrol station table, for example, construct an index on x and y coordinates of all the petrol stations, and then the objects having $x \leq$10km or $y \leq$10km need to be accessed. However, this method can be very inefficient when a petrol station that is closest to the user in x-dimension, may be the farthest one in y-dimension.

Another solution of this query is to create an index on the distances of all the petrol stations to the current location of the user. The distances are calculated by using their x, y coordinates. Then we can simply return all the petrol stains whose distance is less than 10km by using the sorted index. Whereas, when the user changed its location or some other users issue queries in different places, this index has to be reconstructed to answer them, which means all the distances have to be recalculated. The difficulty lies in the fact that there is no mapping from multidimensional space into one-dimensional space so that the objects that are close in multidimensional space are also close in the one-dimensional sorted index [GG98].

In the light of of above reasons, traditional indexes, e.g., the family of binary tree, B-Tree [BM72], B$^+$-Tree, B*-Tree [Com79] are not appropriate for spatial queries. Consequently, many novel indexes have been proposed in the last two decades, including, QuadTree [FB74], R-Tree [Gut84], KD-Tree [BER85], UB-Tree [Bay97], Antipol-Tree [CFP$^+$05], and so forth. To efficiently answer the spatial queries, most of the existing techniques need to identify interesting problem specific properties and to effectively traverse the existing spatial indexes by exploiting these properties.

1.3.2 Outstanding Problems of Spatial Indexes to Process Range Query

Meanwhile, abundant spatial queries are proposed. Some of them can be efficiently processed by spatial indexes solely; While the others require assistances of designed underlying algorithms. Considering the above range search query as an example, if the user does not consider the restriction of the moving path; in other words, the distances from the user to all the petrol stations are measured by the length of their straight connections. Then it can be processed by any spatial index, as the distances can be calculated straight forward. But if all the objects (user and petrol stations) are restricted by the pattern of the road networks, the result retrieved by spatial index is mostly wrong. Because the length of the tortuous road path connecting user and a petrol station can be very different with the straight line, and the distances cannot be calculated straightforward. In general, the first outstanding problem of spatial indexes is their incapability in non-Euclidean spaces. Thus the auxiliary algorithms need to be designed to calculate these distances.

The spatial index is constructed bottom-up on point objects by grouping a bunch of objects into some nodes iteratively according to their spatial features. But There is not a spatial index designed for non-point objects. Even though some of indexes use minimum bounding rectangle (MBR) as nodes in their tree-structure, the representation of MBR for non-point objects is very inaccurate. Consequently, spatial indexes are not able to process spatial queries involving non-point objects.

1.3.3 Flaws of Existing Spatial Algorithms of Range Query

To solve the outstanding problems of spatial indexes, many researches are conducted on designing algorithms of range queries in spatial databases. Such algorithms are feasible to process spatial queries in many non-Euclidean space, such as road networks, obstructed spaces or land surfaces, etc. However, there are many performance issues if the algorithm is implemented in non-Euclidean space, due to the high computation cost of the distances among spatial objects. The state-of-arts are using incremental expansion methods to calculate the distances in the underlying framework, which is inefficient in complex frameworks. Moreover, if the query point or objects of interest are moving they need to report their new locations to the server, and the server need to update the corresponding information to ensure the consistency of the data and the preciseness of the query result. Consequently, the computation and communication cost will increased significantly.

Although, spatial queries has been studied extensively in the last decades, most of the traditional spatial queries only focus on finding point objects. The results of such queries retrieve a set of discrete points stored in the database. Whereas due to the blossom of LBS, spatial database users may require spatial queries to retrieve some regions or areas (polygon objects) rather than point objects, due to the feasibility and the privacy issues. For example, to find crime hot-regions, wild fire areas, the regions where hot water meets cold water, the polluted hydrologic system, etc. Unfortunately, neither spatial indexes and existing spatial algorithms are able to process those spatial queries expecting regions.

Meanwhile, processing spatial queries which inquire regions is a real pressing and challenging problem because of two main reasons. The first one is that sometimes the expected region is not stored in the database, as the region could be an area with some spatial or not-spatial features rather than a real-entity. For example, to identify crime hot-regions. Such problems cannot be answered by spatial indexes and existing algorithms, because of the massive computation. The second difficulty is the representation of a polygon. The shape of a spatial range is diverse, which leads to describing a polygon object correctly and precisely to be tough. Normally a polygon object is composed by a set of points and lines, organizing an ordered set makes the search result much more feasible and implementable. The outstanding problems and contributions of this thesis are illustrated in Fig. 1.2

1.4 Contributions

In this section, we summarize the contributions of this thesis. We proposed several novel region-related queries, especially region-expected queries to enrich the diversity of spatial queries. Moreover, we proposed some corresponding efficient techniques for each queries. The contributions are briefly described as follow:

1.4.1 Point-Expected Range Query Processing

- ***Processing range queries in constrained circumstances***: The traditional range search processing in spatial databases is performed by measuring the length of the distances that present the relative position of objects in the

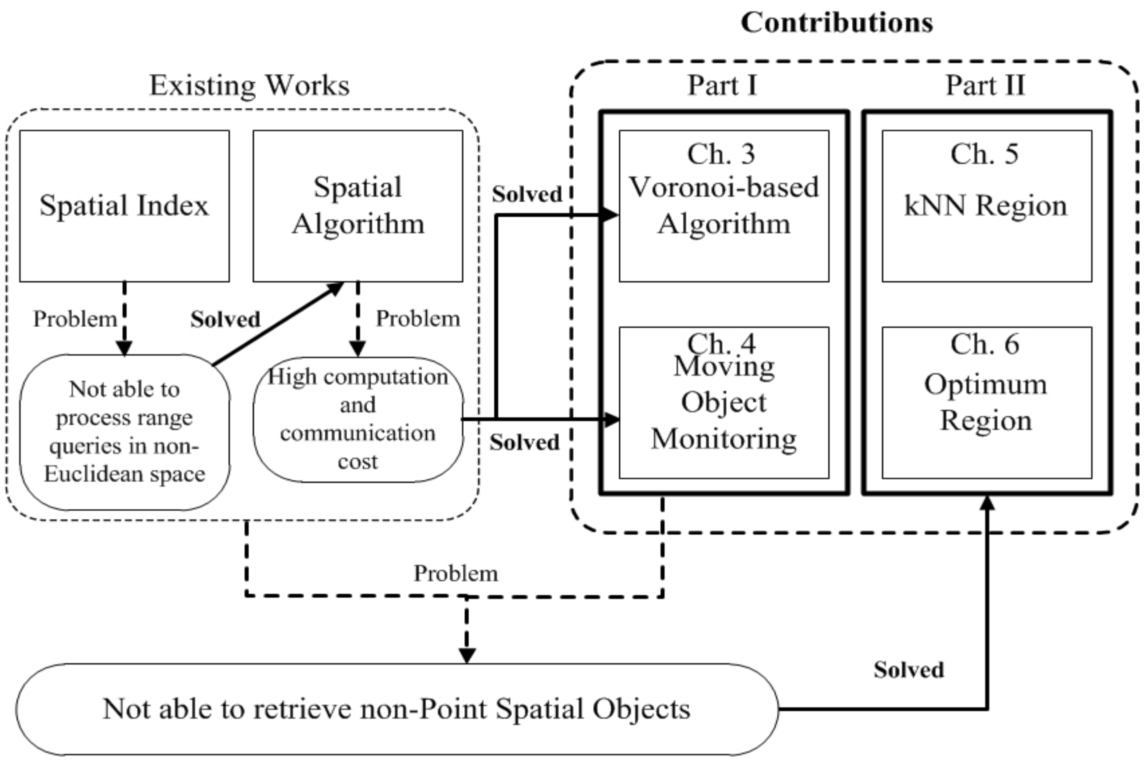

Contributions
Existing Works
Part I
Part II
Spatial Index
Spatial Algorithm
Ch. 3 Voronoi-based Algorithm
Ch. 5 kNN Region
Problem
Solved
Problem
Solved
Not able to process range queries in non-Euclidean space
High computation and communication cost
Ch. 4 Moving Object Monitoring
Ch. 6 Optimum Region
Solved
Problem
Solved
Not able to retrieve non-Point Spatial Objects

Figure 1.2: Major problems and contributions

Cartesian space or road networks. But, in reality, the expected result is normally constrained by other factors, such as, the traveling time, the number of objects, pre-defined areas, and so forth, rather than the distances alone. Therefore range search should be comprehensively discussed in different circumstances. In this thesis, we propose three novel constrained range search queries and an approach for each query based on network Voronoi diagram, which makes the range search query processing more flexible to satisfy various requirements in different circumstances. The performances of these these algorithms are analyzed theoretically and evaluated experimentally. The results show that our approaches can process constrained range search queries very efficiently.

- ***Processing continuous range queries***: We study the problem of continuously monitoring moving range queries on a set of data objects that do not change their locations. Consider the example of a person driving a car who is interested in petrol stations within 10km. In this example, the query is continuously moving whereas the objects are static. We present techniques to answer the continuous range queries in spatial networks. Our proposed technique is based on network Voronoi diagram, which is a spatial decomposition of a metric space. Our algorithm reduces the computation and communication cost because it does not require to recompute the results as long as the query does not move out from a Voronoi cell. We conduct extensive experimental analysis to study the effectiveness of our network Voronoi based approach. Moreover, the experimental results demonstrate that the proposed approach outperforms its competitors.

- ***Processing range queries on moving objects***:On the other hand, the spatial queries of moving objects monitoring is also restricted by frequent updates, as a result, processing spatial queries over moving objects becomes a tough job, especially in road networks. For example, a police wants to monitoring cars traveling through a specific area or within a certain distance to the current location. In this example, the query is static, while the objects of interest (cars) are moving. We store the motion of the moving objects as a function of time instead of its position to achieve equilibrium between the updating and communication cost, and the accuracy of the location of moving objects. By experimental studies, we show that our proposed range monitoring algorithm can process moving objects monitoring range queries efficiently.

1.4.2 Region-Expected Query Processing

- ***kNN Region Query Processing***: None of existing spatial queries can find or retrieve regions closer to a set of specific objects than to any other objects, even though this is an important problem in spatial databases and practical applications. In this thesis, we propose a novel query, k Nearest Neighbor region searching, which retrieves a region where every point considers specified k objects as the k nearest neighbors. In addition, we propose two algorithms, $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR based on high-order (kth-order) Voronoi diagram and

Delaunay triangulation respectively for it. We discuss the kNN region searching in a 2D Cartesian space, but it can extend to a higher dimensional space. An extensive theoretical and empirical study was conducted to compare and evaluate the performance of these two algorithms. The study showed that $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR outperforms the other ones in different scenarios with respect to k, the number of objects in the entire set U, and the number of queries to be processed.

- ***Optimum Region Query Processing***: Another region-expected problem is that database users might be interested in find a region that can cover a maximum number of objects among a set of objects with a specific radius r. To our knowledge, no existing work addresses this sort of quires. So we propose such a query in this thesis, named optimum region. In addition, we developed an algorithm, *Circle Partition and Arcs Superposition* (*CPAS*), to solve this problem. An extensive empirical study was conducted to evaluate the performance of *CPAS*. The results showed that *CPAS* can process the optimum region queries efficiently in most of the circumstances.

1.5 Thesis Organization

This thesis is organized as follows (Refer to Fig. 1.3).

- Chapter 2 provides a survey of the related works.
- Chapters 3 and 4 investigate traditional range queries focusing on retrieving point objects. More specific descriptions are:
  - Chapter 3 proposes three range search queries in road networks with various restrictions as well as corresponding approaches based on the variations of the network Voronoi diagram.
  - Chapter 4 explain range search queries for moving objects (either moving queries or moving objects of interest). The proposed algorithms predict the results at the beginning or reduce the communication costs.

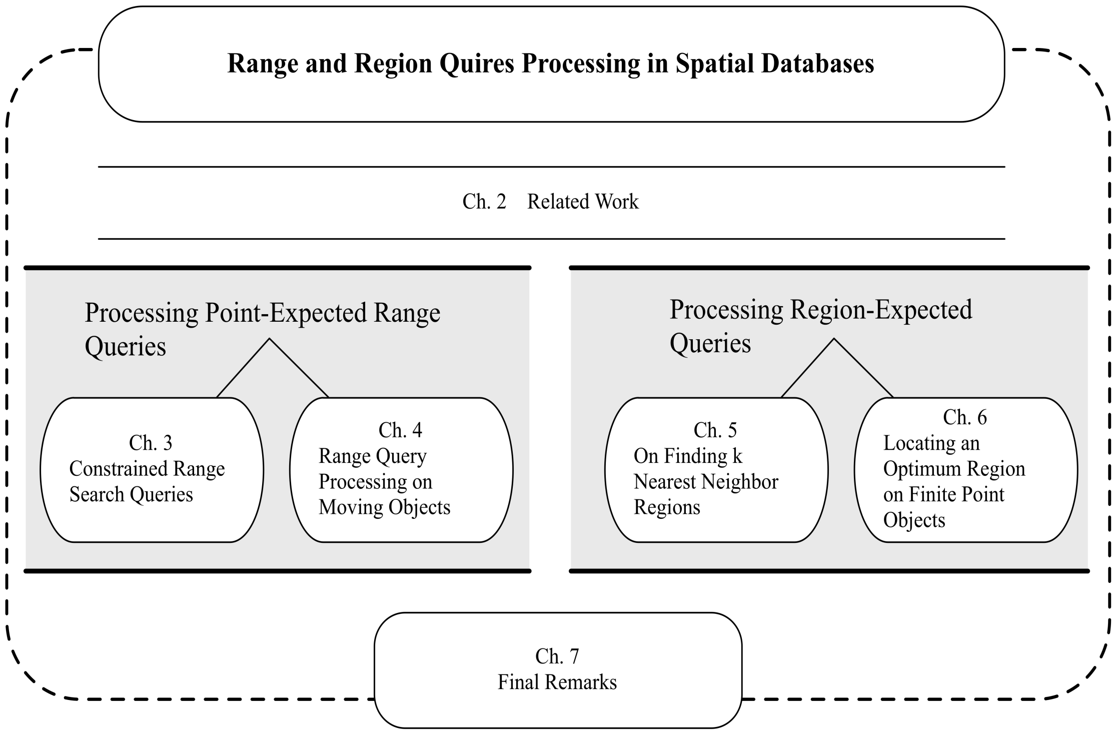

Range and Region Quires Processing in Spatial Databases
Ch. 2 Related Work
Processing Point-Expected Range Queries
Ch. 3 Constrained Range Search Queries
Ch. 4 Range Query Processing on Moving Objects
Processing Region-Expected Queries
Ch. 5 On Finding k Nearest Neighbor Regions
Ch. 6 Locating an Optimum Region on Finite Point Objects
Ch. 7 Final Remarks

Figure 1.3: Thesis structure

- Chapters 5 and 6 propose a original sort of region-expected queries which concentrate on finding or defining a particular region with specific spatial features. Below is the details:
  - Chapter 5 presents a region-expected queries, which finds an area for k specified objects in Cartesian space.
  - Chapter 6 presents a region-expected queries that finds an area to cover maximum number of objects of interest in a finite objects set.
- Chapter 7 concludes our research, describes some of the open problems and provides several possible directions for future work.

Chapter 2

Related Work

2.1 Overview

In this chapter, we provide a brief overview of the related work for spatial indexes, spatial algorithms, especially on two sorts of spatial queries, range and k nearest neighbors (kNN), and a relevant problem which has been studied in computational geometric. We provide the related work on spatial indexes, introduce the two most popular indexes, R-Tree and Quad-Tree in detail in Section 2.2. And then we have an overview of the related techniques for range queries, and explain two algorithms designed for rang query in road networks in Section 2.3. In Section 2.4, we briefly describe the existing algorithms for kNN queries and classify its variations. Finally, we present the smallest circle problem being studied in computational geometric area in Section 2.5, and at the end of each section, we summarize their limitations and disadvantages .

2.2 Spatial Indexes

A typical spatial index is a data structure created on a table in spatial databases to optimize the processing of spatial queries. Many conventional index types do not efficiently query the data with spatial features, such as finding the closest pairs among a set of points, or whether points fall within a spatial area of interest; While spatial indexes are constructed based on the spatial features of a set of spatial objects, by grouping them into the nodes. Common spatial index structures include: QuadTree [FB74], R-Tree [Gut84], KD-Tree [BER85], UB-Tree [Bay97], Antipol-Tree [CFP^{+}05], and so forth. However, spatial indexes can process range query only in Euclidean space in common.

To clarify the procedure of spatial indexes for range queries, we introduce two most popular methods, QuadTree and R-Tree as examples.

2.2.1 QuadTree

A QuadTree is a spatial partitioning strategy used to make queries on relationships between 2D spatial data such as coordinates in spatial databases, or the location of objects in a video game. For instance, to know all of the objects within a region on a map, or detect whether objects are visible by a camera.

The general strategy of the QuadTree is to construct a tree structure that partitions a region recursively into four parts, or Quads which may be squares or rectangles. Each Quad can do the further partition as necessary. A pre-requisite is that the bounds of the area has to be defined; the basic algorithm does not lend itself to the addition or removal of areas under consideration without rebuilding the index. Fig. 2.1 shows an example of a QuadTree constructed on objects 1 to 17. When a user issue a range query, e.g., find all the ATMs within 1km, the searching starts from the root node and checks whether the searching area, which can be a circle or a rectangle depends on the query intersect with the children node, until reach to the leaf nodes. As Fig. 2.1 illustrates, the shaded area are the nodes including retrieved objects and the bold lines indicates the searching path in the QuadTree.

Because of the QuadTree subdivides Euclidean spaces into at least four equal child nodes, if the distribution of the objects of interest is asymmetrical, the structure of the QuadTree will be very unbalanced and it might contain many nodes that do not contain any objects, as shown in Fig. 2.1. Additionally, QuadTree is unable to constructed on non-point objects, if so, one objects will appear in more than one

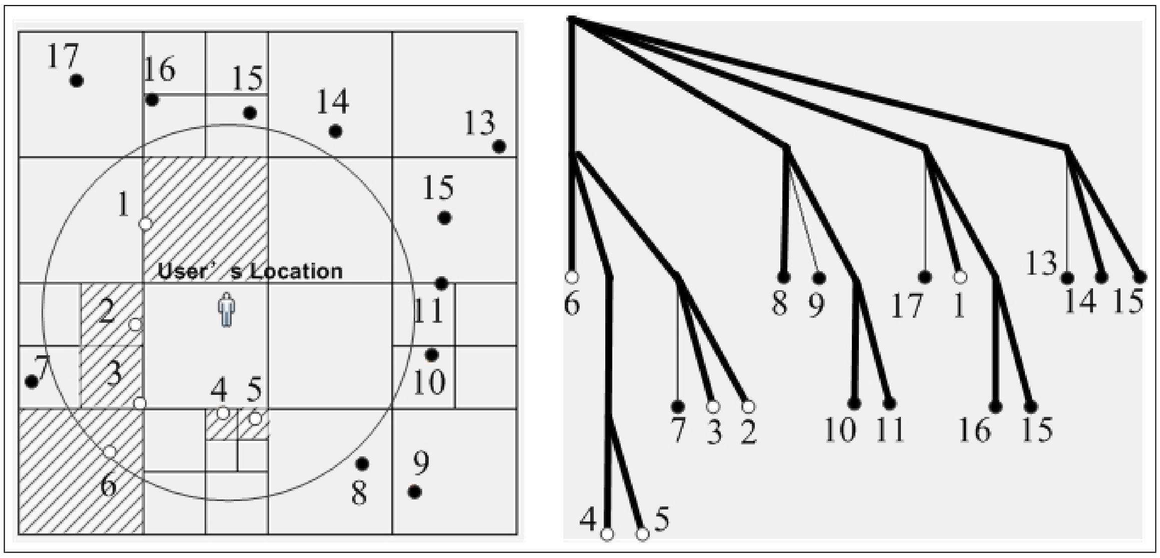


Figure 2.1: QuadTree index on a set of point objects

nodes and also requires massive I/O access to retrieve the data. Thus, a sort of spatial indexes based on minimum boundary rectangles (MBR) are proposed, such as R-Tree.

2.2.2 R-Tree

A R-tree is a tree data structure used to efficiently access the spatial data, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. The R-tree was proposed by Guttman in 1984 [Gut84] and has be extensively used in many research and practical applications, including storing spatial objects such as locations of all the primary schools, or the segment or polygon objects that typical maps are made of: streets, buildings, etc. and then find answers quickly to queries such as "Find all ATMs within 1km of user's current location" or "retrieve all road segments within 10km of a traffic office"

The main strategy of R-Tree is to group nearby objects and represent them with a minimum bounding rectangle (MBR) in the next higher level of the tree. Since all objects lie within a set of bounding rectangles, a query that does not intersect the higher level rectangle also cannot intersect any of the objects contained by this rectangle. At the leaf level, each rectangle describes a single object.

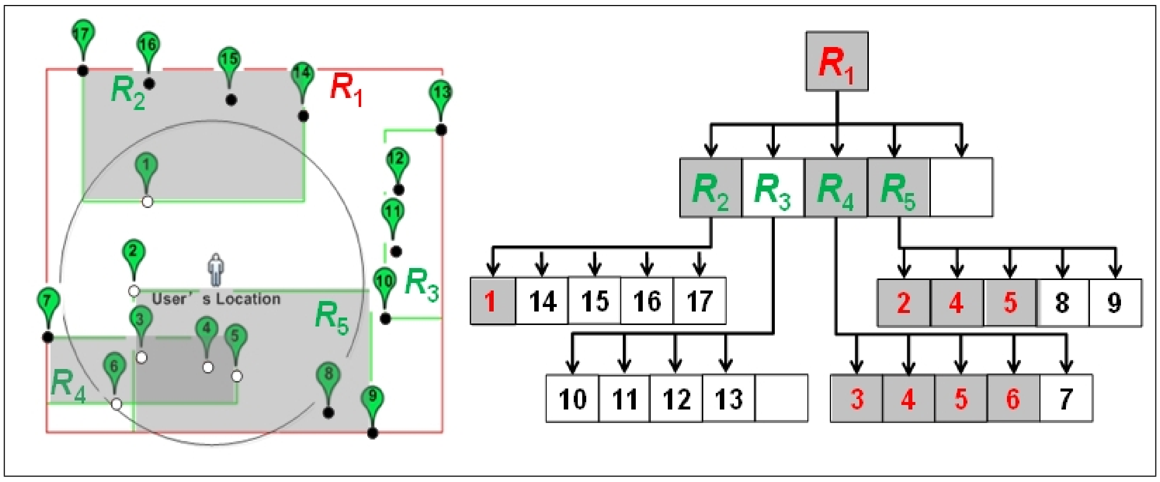


Figure 2.2: R-Tree index on the same point objects in Fig. 2.1

Fig. 2.2 illustrates an example of R-Tree indexing for a range search query. The searching derives at the root, runs through each minimum bounding rectangle intersects with the query circle until reaches the leaf nodes, whereby, this query retrieves objects 1 to 6. Fig. 2.2 also shows the visited nodes in the R-Tree.

Although MBR can represent a non-point object approximately, which may cause the retrieval of incorrect result. See Fig. 2.3 as an example, a leaf node is the minimum boundary rectangle which contains a non-point objects o_i. A range query is represented by a shaded circle intersecting with the MBR. Obviously, the object o_i does not intersect with searching range; while by using R-Tree, o_i will be retrieved. R-Tree also retrieves false result in many non-Euclidean space, such as road networks. As Fig. 2.4 illustrates, object 6 is within the searching area in Euclidean space, but the factual travel distance is out of this restriction.

Abundant of spatial indexes are proposed to optimize the performance, including, R$^+$-Tree [SRF87], R*-Tree [BKSS90], etc., but none of them are able to process spatial queries in non-Euclidean spaces, such as range, kNN queries. Moreover, because of the difficulty of representing non-point objects in spatial indexes, all the spatial indexes are unable to retrieve non-point objects.

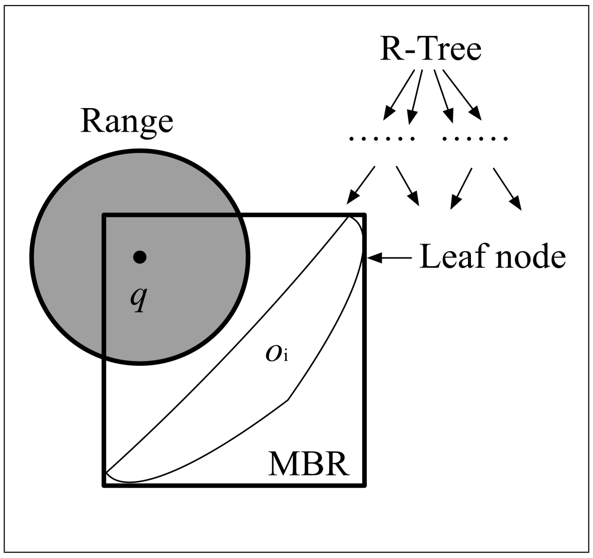


Figure 2.3: An incorrect non-point object retrieved by R-Tree in Euclidean space

2.3 Range Queries

A traditional range search query retrieves all objects of interest within a certain distance from the current location of the query point in Euclidean space. The traditional range queries have been studied extensively and it can be answered by using spatial indexes discussed above. These spatial index methods can address range search query only in Euclidean space, while many applications use different metrics, such as road network distance in a traffic control system. Therefore, many works investigate range search problems in various spaces. Thereinto, two novel approaches, named Range Euclidean Restriction (RER) and Range Network Expansion (RNE) [PZMT03] study range search query processing in road networks. These two algorithms calculate the network distances by expanding the road networks incrementally. Afterward, a Voronoi-based range search approach VRS [XZT$^+$11a] is proposed in 2011. Some other works [CHC04a, XZT$^+$11a] discuss the range search for a moving query, and [XZT$^+$11b] proposed some constrained range queries as well as the corresponding solutions. The range search query is also discussed for uncertain data [TXC07] and in peer-to-peer systems [LCLC09].

The problems of above algorithms have three facts. The first fact is that the range queries are processed according to only the distance metrics. In fact, the range queries may also be restricted by other spatial or non-spatial features, e.g., the number of retrieved objects, specific regions. The second fact is their inefficiency, especially, in complex spaces or the query involving one or more moving objects. The last fact is that they are spatial index-based. Consequently, they concentrate on retrieving point objects only.

To explain this, we introduce two popular range search algorithms in road networks in the following two sections.

2.3.1 Range Euclidean Restriction (RER)

RER applies a range search on Euclidean distance e from query point q to retrieve all possible candidates, as the Euclidean distance can be seen as the lower boundary for network distances $d_e(q,p) \leqslant d_{net}(q,p)$, which insures that all objects of interest within the searching range will not be missed. But a large number of false hits, which have $d_e(q,p) < e$, and $d_{net}(q,p) > e$, are involved in this procedure, these false hits have to be filtered from the candidates list by performing network expansion in the next step until all the candidates are tested or all the segments in the range are exhausted.

Fig. 2.4 demonstrates an example, a user wants to find all bus stops with in 1km around Clayton Campus of Monash University. By the Euclidean searching (e.g., R-Tree index), objects 1 to 6 are retrieved as candidates; While after verifying these objects, object 6 is deleted, as its network distance exceed the searching range.

According to the algorithm of RER, if only few objects locate in complex networks, the processing time will increase significantly, because most of the operation will be wasted on the network expansion. In another scenario that the network distances is quite different with corresponding Euclidean distance, the candidates sets will include massive false hits which still need to be verified to be deleted in the filter step. Consequently, [PZMT03] proposes an alternative solution, Range Network Expansion (RNE).

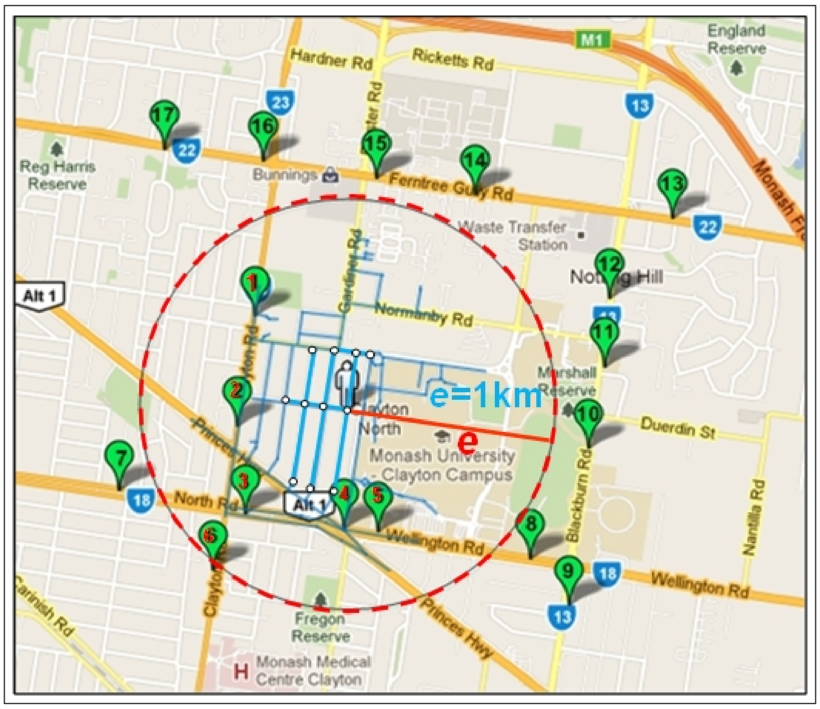


Figure 2.4: Range Euclidean Restriction (RER)

2.3.2 Range Network Expansion (RNE)

RNE outperforms RER by using an expansion method which qualities a set of segments within the network range e to filter the false hits first, and then using R-tree to find the objects of interest falling on the valid segments, which intersect with minimum boundary rectangles (MBRs) [Gut84]. Finally, when R-tree indexing is finished, the results are sorted to remove the duplicates.

Fig. 2.5 illustrates the same example. We can see that since RNE first performs road network expension, which guarantee each segment is within the searching range. Hence, no false hit will be retrieved by using R-Tree index the the next step.

However, RNE still cannot solve the problem of performing some unnecessary expansions. If the range area is enormous, RNE will need to check every segment incrementally, even there are only few objects locating in the searching range.

Intuitively, the performances of RER and RNE will decrease significantly in a complex networks or on the dense objects. As explained above, both RER and RNE invoke R-Tree as a sub-function. Thus, they cannot process range queries involving non-point objects. In the light of these reasons, it is very pressing to design

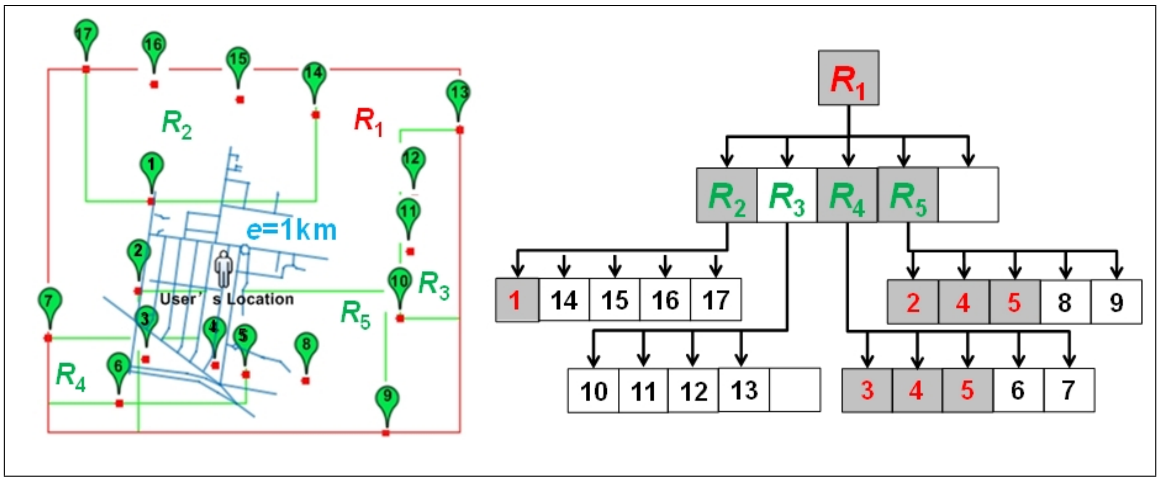


Figure 2.5: Range Network Expansion (RNE)

approaches which can get rid of or reduce the number of false hits, optimizing the performance and flexible enough in different scenarios .

2.4 k Nearest Neighbor Queries

In this section, we browse through k nearest neighbor (kNN) queries and its variations in spatial databases. The the last decades, kNN queries were paid more attention. It focuses on the ranking of a set of discrete points in term of the distances to the query point differing with range queries, but for the query processing, they have many methodologies in common.

2.4.1 Typical kNN Queries

kNN queries have received more attention over the past two decades. The kNN queries were first introduced in [RKV95] in Euclidean space. Thereafter, kNN queries were studied in many other metric spaces, such as in road networks [PZMT03], [KS04], [CC,DKS], land surfaces [DZS+06], [STX08], [XSP09], arbitrary dimensionality [TPL04,TYSK09], the presence of obstructed space [NTZ10], [GZ09], [GZC+09b] and weighted regions [LGYL11]. All of these kNN queries assess the relative position of two points in metrics other than Euclidean distance. kNN queries were also discussed for moving queries [TPS02], [MYPM06], [KS05], [ZXR+08], [CSZY], [NZTK08], [HXL05], [HCLZ09], [DKS09] and moving objects [TP03,LZZ06,ZJDR10], which maintain the kNN result in a dynamic circumstance.

Thereinto, Voronoi-based kNN approaches are the most efficient. Voronoi diagram is first adopted in kNN by Kolahdouzan and Shahabi [KS04] in which the performance results show their approach VN^3 outperforms its main competitor INE [PZMT03] by up to one order of magnitude. One year later they proposed some new approaches for CkNN query, named IE and UBA [KS05]. They use these approaches to find the different variation trends of the network distance to the query point between adjacent objects in the candidates set when the query point is moving. The only difference between IE and UBA is that the former compares kNN results of the two ends of the road segment, while UBA just extends the kNN results at one end to enhance the execution performance.

2.4.2 Variations of kNN queries

Meanwhile, abundant variations of kNN queries were proposed, including: aggregate kNN (also known as group kNN) [PSTM04, YMP05, PTMH05], reverse kNN [TPL04, TST$^+$11, CLZ$^+$09], range kNN [HL06, CMNN09], constrained kNN [FSAA] and reverse furthest neighbor query [YLK09]. All of these kNN queries change the threshold to meet diverse demands. Descriptions of the above variations are listed in detail:

- **Aggregate kNN/Group kNN:** a novel form of kNN query. The output contains the k point(s) with the minimum sum of distances to the query point.
- **Reverse kNN:** Given a query point, a reverse k nearest neighbor (RkNN) query retrieves all the objects of interest that consider the query point as one of their k nearest neighbors
- **Range kNN:** Given a region as the query, a range nearest-neighbor query retrieves the nearest neighbors for every point in a query region.

- **Constrained kNN:** nearest neighbor queries are constrained to a specified region. It retrieves k nearest neighbors in a bounded region rather than in the entire space.
- **Reverse furthest neighbor:** Specifying a query point in a set of objects of interest, the reverse furthest neighbor (RFN) query fetches the objects considering the query point as their furthest neighbor among all objects of interest.

As the essential objectives of kNN queries and range queries are different, all the algorithms designed for kNN queries cannot be implemented for range queries straightforward, even they have some similar fundamentals sometime. Moreover, kNN queries also retrieve point objects rather than non-point objects.

Based on our explanation above, neither range search queries nor k nearest neighbors queries, which are basic spatial queries, cannot retrieve or identify a region, which is our main motivation to conduct the researches presented in the second part of this thesis.

2.5 Smallest Circle Problem

The smallest enclosing circle problem, also known as minimum covering circle problem is a historical mathematical problem of computing a smallest circle covering all the given points. Given a set of points, U=$\{o_1, o_2, ..., o_n\} \in \mathbb{R}^2$, the smallest circle is one and only one $\odot \supset U$ with radius r, and $\nexists \odot' \supset U$ with r'<r. It has been extensively used in planning the location of a shared facility that minimizes the farthest distance from a point to the facility. Such as, locate a post office which minimizes the farthest distance of all the residents. Fig. 2.6 shows the comparison of the smallest circle with a too big circle and a too small circle. The circle C_1 is too big because we can find a circle covering all the black points with a smaller radius; While circle C_2 is too small as it cannot cover all the black points.

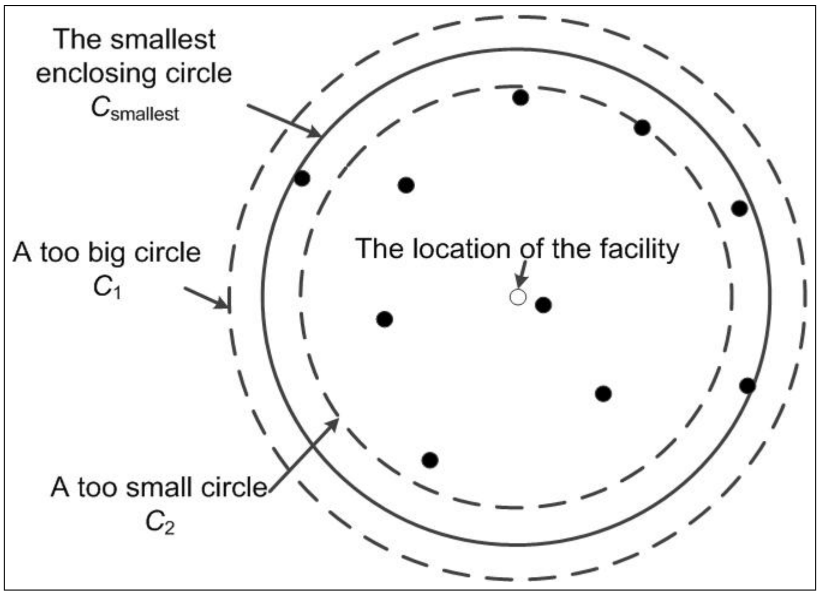

The smallest enclosing circle C_{smallest}
A too big circle C_1
The location of the facility
A too small circle C_2

Figure 2.6: An example of the smallest circle comparing with big and small circles on a set of points.

The smallest enclosing circle problem was first proposed by Sylvester in 1857 [Syl75], where the author suggested a naive solution that has a $O(n^4)$ time complexity, where n is the cardinality of the set U. This algorithm is based on the fact that the smallest enclosing circle of the given set U is determined by either a pair or a triplet of points in U. Bass and Schubert proposed an improved algorithm [BS67] relying on the convex hull constructed on the given set, whereby the time complexity is reduced to $O(h^4{+}n\log n)$, in which h is the vertices of the convex hull of U. In 1972, Elzinga and Hearn [EH72] proposed a quadratic algorithm that runs in $O(n^2)$. Shamos [Sha75], Shamos and Hoey [SH75] were the first to improve the computation of the smallest enclosing circle considerably to an approximate linear time complexity, $O(n\log n)$. Shamos and Hoey's algorithm is based on constructing a furthest Voronoi diagram that is a significant structure in geometric computation and its requires $O(n\log n)$ to construct. This leads the computation of smallest enclosing circle to have a lower bound of $O(n\log n)$. Finally, in 1983 Nimrod Megiddo [Meg83] showed that the minimal enclosing circle problem can be solve in $O(n)$ time using the prune-and-search techniques for linear programming, which is the most efficiency algorithm to our knowledge. Afterwards, Martin Dyer [Dye86] and Nimrod Megiddo [Meg84] extended this linear solution in any fix high-dimension.

However, there are two main disadvantages of the smallest enclosing circle during the implementation in many real applications. One is that the smallest enclosing circle does not take the coverage capacity of the shared facility into its account, which results in incapable coverage of all given points sometime. But many facilities have the limited coverage capacity, such as, wireless base station, explosion range of a bomb. The other is that finding the smallest enclosing circle is a 1-center problem in $\mathbb{R}^{\nvDash}$, in other words, the location of the facility found by the smallest enclosing circle is a unique point that is the center of the smallest circle. But in many circumstances, especially, on dense objects the found location, the centre of the smallest circle, to construct the shared facility is mostly unavailable, e.g. being occupied by other objects or blocked, which leads finding the smallest enclosing circle useless.

In our proposed optimum region, we can find the best location to cover most given point according to the coverage capacity of the facility. Moreover, the optimum region is very flexible in locating a facility because the result includes infinite points.

2.6 Summary

In this section, we briefly introduce many spatial indexes, range queries, kNN queries and smallest circle problem with explaining two representative existing works for each. We found that none of the existing works are adequate and suitable to solve the problems described in Chapter 1. Their limitations and disadvantages are summarized as follow.

- **Spatial Indexes**: none of spatial indexes are able to process spatial queries in non-Euclidean spaces. Furthermore, because of the difficulty of representing non-point objects in spatial indexes, all the spatial indexes are unable to retrieve non-point objects.

- **Algorithms for Range Queries**: the first limitation is that the range queries are processed according to only the distance metrics. In fact, the range queries may also be restricted by other spatial or non-spatial features. The second limitation is their inefficiency, especially, in complex spaces or the query involving one or more moving objects, as they require an incremental expansion method to calculate distances among spatial objects. The last one is that they are spatial index-based, thus they cannot retrieve non-point objects, but point objects only.

- **Algorithms for Range *k*NN**: the essential objectives of kNN queries is different with range queries. Consequently, the algorithms designed for *k*NN queries cannot be implemented for range queries straightforward. Moreover, kNN queries also retrieve point objects rather than non-point objects.

- **Smallest Circle**: it assumes that the shared facility having infinite coverage capacity, does not consider the limitation of coverage capacity of the shared facility in fact. The location of the expected facility is unique. But in many cases, it availability of this location is not guaranteed, because of the occupation or the blockage of other objects.

Part I

Processing Point-Expected Range Queries

Chapter 3

Constrained Range Search Query Processing

In this chapter, we present our techniques to process range search queries under some restrictions in road networks. We propose three novel constrained range queries, namely, *Time constrained Range*, *Region Constrained Range* and *k Nearest Neighbors Constrained Range*. Our algorithms called *TCR*, *RCR* and *kCR* for these three queries are based on network Voronoi diagrams and all of them can process corresponding queries efficiently. The research presented in this chapter was published in [XZT$^+$11b] and [XTSS10]

3.1 Overview

Range search is one of the most common and fundamental queries in spatial and mobile databases [WST03, WST04]. It has been also extensively studied in many other areas, including data structures, information retrieval, computational geometry, wireless communication [Muh09] and so forth in the past decades. The research results are implemented in many practical applications, such as search engine, geographic information system (GIS), global positioning system (GPS) and digital map. A qualified application (Web applications: Google Maps, Bing Maps and

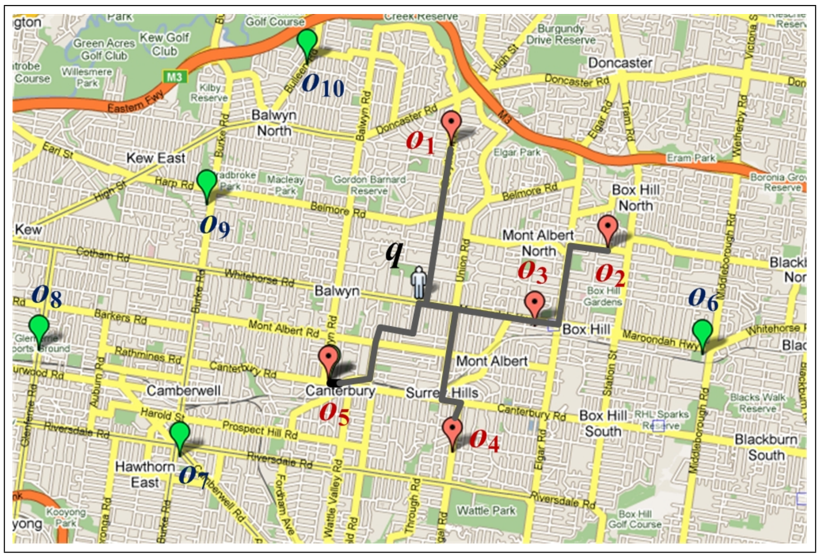


Figure 3.1: An example of range search

Mobile devices: PDAs, cellular phones, car navigation systems) for range searching must be competent for a variety of queries in any complex environment and can respond queries accurately and efficiently. Fig.3.1 illustrates an example of range search in a digital map. The user (query point q) wants to get all petrol stations within 3km from current location. In Fig.3.1, the object of interest (petrol station) is represented by a sequence of numbers. The distance from query point to each object is the shortest road network connection rather than the straight line. Then the objects within 3km to query point are highlighted with red while the green objects indicate the petrol stations are out of 3km to the query point. This simple question has been answered by many existing work, by using spatial indexing [Gut84, BER85, Bay97, CFP$^+$05, TR02, TR04] and various spatial query algorithms [PZMT03, XZT$^+$09b].

A typical range search on road networks is defined as: Given a set of discrete objects of interest (OOIs) on road networks, a query location, and a network distance e indicating the searching range, find all objects of interest within range e from the query point.

However in many applications, the objects of interest may be constrained by many other factors or spatial objects, such as, other metrics (e.g. traveling time), a polygon spatial object and the required number of objects. For example, a user

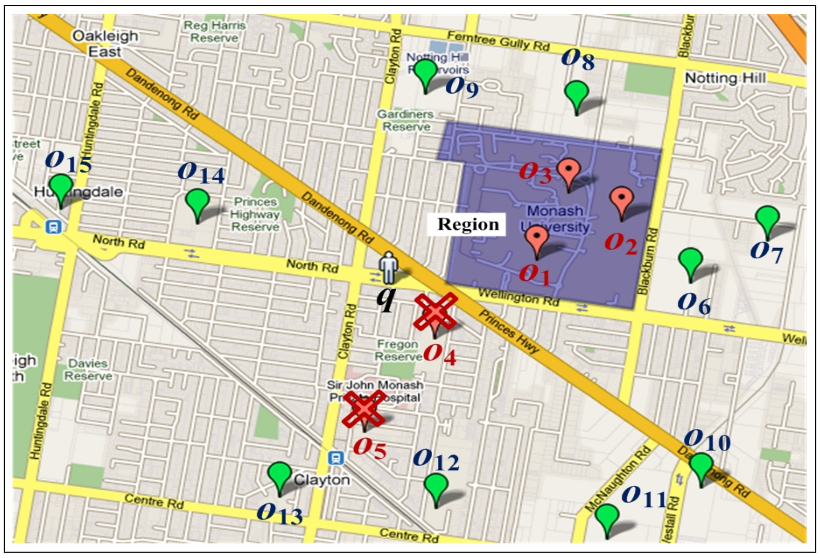


Figure 3.2: An example of region constrained range search

may want to find all the petrol stations within 5km, which also need to be reached in 10min. It is very common that a object is closer to the current location but takes longer time to reach, due to the traffic jam, road works or topographic conditions. We also need to have the distance restriction because of the capacity of the objects, such as, the remaining petrol can only travel 20km. Therefore, we have to use several metrics to answer a query.

Consider another instance, a user may want to find car parks in a university campus within 1.5km. See Fig.3.2, the car parks from query point within 1.5km are o_1, o_2, o_3, o_4, o_5, but only o_1, o_2, o_3 are located in the pre-defined region (university campus). So object o_4 and o_5 have to be removed from the result list. In such a range search query, car parks which might be closer to the query point but are not in a specific region, such as, a polygon object, will not be of interest to the user. Hence, the requested object of interest have to be in a specified region. The ordinary range search approaches are not concerned about objects of interest in a large region, and obviously, these approaches can not directly answer region constrained range queries.

Another problem whereby the existing spatial range search algorithms will not work is where the user who invokes the query, apart from having a certain radius of the range search, also specifies a limit to the number of objects retrieved. This is different from the traditional kNN queries – retrieve k nearest objects of interest, regardless their distance limit. It is also different from the basic range search, because in the basic range search, the number of objects retrieved is of no concern, as long as all of them are within the specified radius. An example of k constrained range search is to retrieve a maximum of 10 car parks within 5 km. If the number of car parks within a radius of 5 km is less than 10, then a decision has to be made whether it is feasible to extend the radius search in order to cover more objects. The existing range and kNN search algorithms do not deal with this type of range search.

This chapter concentrates on processing constraint range search queries, especially when the range search has (*i*) time constraint, like in the first example above, (*ii*) region constraint, like in the second example above, or (*iii*) k number of objects constraint, like in the last example above. We next summarize our contributions.

- *TimeConstrainedRange* (TCR): It is a network range search approach to process the range queries, which looks for all objects of interest within the range are also reachable in a specified time quantum. We construct a double weighted (time metric and space metric) road networks that appends the traveling time as the other weight for each segment in the road networks to reflect the time events, and then propose the approach (TCR) based on it. As a result, the range search queries relevant with the traveling time which could not be served by the traditional range search approaches are solved by TCR properly.
- *RegionConstrainedRange* (RCR): It satisfies the users who want to get all objects of interest being located in some particular regions within the range in the road networks. We broaden the concept of the objects from a point object to a weighted region (polygon object), which can involve some objects of interest inside. We found a novel weighted network Voronoi diagram to present the road networks including some weighted objects. Then the proposed

RCR can handle the range search queries that demands the objects of interest within some particular regions.

- *kNNConstrainedRange* (kCR): It is an approach process a kind of range search queries that try to find k nearest neighbors of the query point within the range in the road networks. Since the number of the objects of interest within the range could less than required k, which means not enough objects in the searching range, kCR also need to estimate the cost and to determine expanding the searching range or decreasing of number k according to the deviation ratio of e and k (defined in section 3.5), whereby we can make a equilibrium between the searching range and the quantity of retrieved objects

The remainder of this chapter is structured as follow. The underlying frameworks and the basis method for our approaches are introduced in section 3.2. In the next three sections 3.3, 3.4 and 3.5, we describe our proposed solutions for three different types of constrained range search queries respectively. Performance evaluation is explained in section 3.6. Finally, section 3.7 summaries this chapter.

3.2 Preliminary

3.2.1 Voronoi Diagram

Voronoi Diagram partition the Euclidean plane into a set of convex polygons, named *Voronoi polygon* or *Voronoi region.* Each Voronoi polygon involves a generator (point) to which the Euclidean distance d_e from any point in its polygon is smaller than to any other generator. Voronoi polygon is composed by several Voronoi edges, also called borders, which is always shared by a pair of adjacent polygons. Any point on the Voronoi edge has the same distance to the pair of generators that associate with this edge. The formal definition of Voronoi Diagram is given as follow:

Definition 3.2.1. *Given a set of discrete objects $\wp$ = (o_1, o_2,..., o_n), $n \in$ Integer I_n, in Cartesian space, the **Voronoi polygon** of o_i, $VP(o_i)$, is a universal of points* *that satisfy:*

$$\{\forall p | d_e(p, o_i) \leq d_e(p, o_j)\}, (i, j \in I_n, i \neq j, o_i, o_j \in \wp)$$

The Voronoi diagram for $\wp$ is:

$$VD(\wp) = \bigcup_{i=1}^{n} VP(o_i)$$

The function used to calculate the distance from point p to o_i, $DISTANCE(p, o_i)$ is Euclidean distance:

$$DISTANCE(p, o_i) = d_e(p, o_i) = \sqrt{(x_p^2 - x_{o_i}^2) + (y_p^2 - y_{o_i}^2)}$$

x, y are the aces in a coordinate system. In term of the definition of VP, for any point p inside $VP(o_i)$, the distance from it to o_i must be the minimum one comparing to the distances to other objects. The equality in the definition of Voronoi polygon holds for the points on the borders of VP_{o_i} and VP_{o_j}

Fig.5.4 shows an example of VD, in which, any point p in VP_{o_2}, whose distance to generators o_1, o_2 are $d_e(p, o_1)$ and $d_e(p, o_2)$ satisfying $d_e(p, o_1) < d_e(p, o_2)$. Since point b is on the shared edge E of VP_{o_1} and VP_{o_2}, then $d_e(b, o_1) = d_e(b, o_2)$.

3.2.2 Network Voronoi Diagram

Network Voronoi Diagram (NVD) is a special Voronoi diagram constructed on road networks involving a set of nodes and links rather than Euclidean plane. In NVD, the partitions are a set of road segments termed network Voronoi polygon (NVP) instead of Voronoi polygon. For each NVP, there is only one generator. Any point p in $NVP(o_i)$ also satisfies:

$$\{\forall p | d_{net}(p, o_i) \leq d_{net}(p, o_j)\}, (i, j \in I_n, i \neq j, o_i, o_j \in \wp)$$

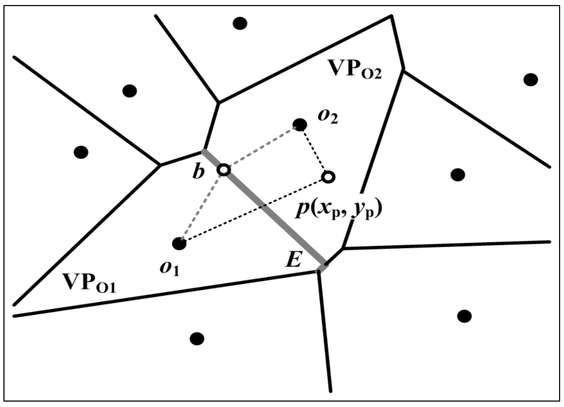


Figure 3.3: Voronoi Diagram (VD)

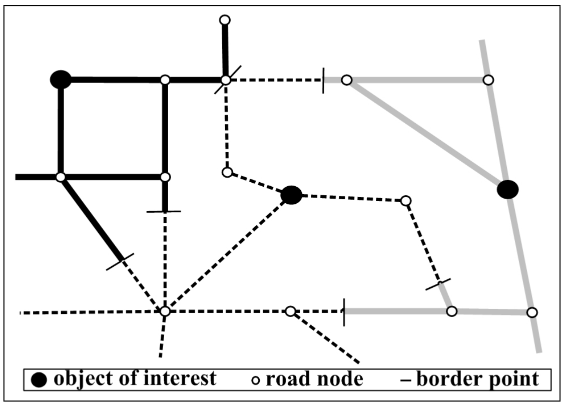


Figure 3.4: Network Voronoi Diagram (NVD)

The network distance is calculated by Dijkstra's algorithm [Dij59], which find the shortest path between two points in the road networks.

$$\mathit{DISTANCE}(p, o_i) = d_{net}(p, o_i) = \sum f(p, n),\ n\ is\ a\ road\ node$$

The edges of Voronoi polygons shrink to the midpoints of the connection through road networks between two objects. Beside the object of interest, the intersections (white point) of the networks are also presented in the NVD. Referring to Fig. 3.4, the objects of interest (dark points) act as the generators of NVPs that are distinguished by different line styles.

With NVD, a spatial network can be modeled as a weighted graph. All road network intersections and objects can be represented as nodes in the weighted graph. The link connecting these nodes represents road networks while the distances are the weight for these links. NVD stores the relative position of the objects in the road networks, therefore NVD-based approaches can load more information from spatial database if required, which improves the performance of spatial queries processing.

3.2.3 Weighted Voronoi Diagram

In ordinary Voronoi diagrams all generators are identical points and do not have any extent, the Voronoi edges are straight lines and the Voronoi polygons are contiguous. The weighted Voronoi diagram (multiplicatively, additively, compound) differs from ordinary Voronoi diagram in that the generators have different weight which denotes as ω [OBSC00a]. In this section, we introduce additively weighted (AW) Voronoi diagram. The definition of AW Voronoi diagram is as follow:

Definition 3.2.2. *Given a set of discrete objects $\wp = (o_1, o_2, ..., o_n)$, $n \in$ Integer I_n, in Cartesian space. Each point o is assigned a weight ω_o. the weighted Voronoi polygon of $o_i \in \wp$, $VP(o_i)$, includes all points that satisfy:*

$$\{\forall p | d_e(p, o_i) - \omega_{o_i} \leq d_e(p, o_j) - \omega_{o_j}\}, (i, j \in I_n, i \neq j, o_i, o_j \in \wp)$$

The additively weighted Voronoi diagram for $\wp$ is:

$$WVD(\wp) = \bigcup_{i=1}^{n} WVP(o_i)$$

In a 2D space, AW-Voronoi diagram can be seen as a standard Voronoi diagram of a set of rotundities each centered at a point o with ω_o as a radius. According to the definition of weighted Voronoi diagram, each weighted Voronoi polygon is assigned to a unique rotundity which is the closest area of all points inside that polygon. A point b on the edge of weighted Voronoi polygons o_i and o_j satisfies:

$$d_e(b, o_i) - \omega_{o_i} = d_e(p, o_j) - \omega_{o_j}$$

Fig.3.5 illustrates an example of AW-Voronoi diagram of five weighted objects (R) centered at Voronoi generators (o) with different weights (ω_o). It is obvious that for any point p in weighted Voronoi polygon of R_1 has: $d_e(p, R_1) \leq d_e(p, R_2)$ and for any point b on the border of R_1 and R_2 has: $d_e(b, R_1) = d_e(b, R_2)$

3.2.4 Voronoi Range Search: our previous work

Voronoi Range Search, VRS, is a Voronoi-based range search approach [XZT+09b] to retrieve all objects of interest within the expected searching range in road networks. VRS involves *contain*(q) to retrieve the NVP containing the query point first

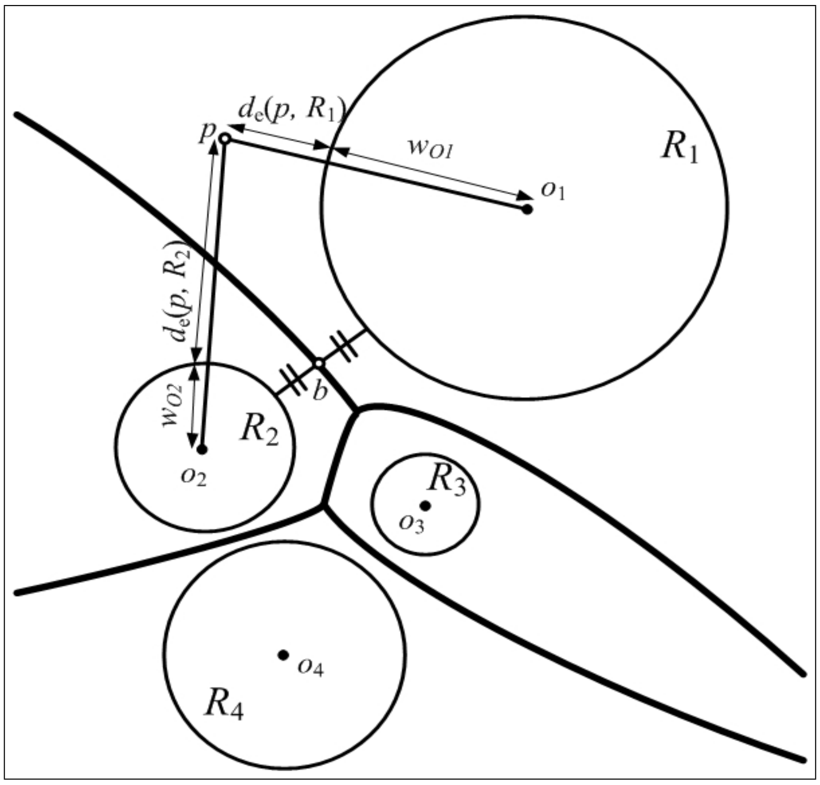


Figure 3.5: Additively weighted Voronoi diagram

and check whether its generator is in the range e or not. If it is in the range, then we bulk load a set of adjacent NVPs into $Q_{pre}()$ and compare the network distance from furthest generator o to query point with e. If $dis(o, q) < e$, then we repeat expansion by loading the next round of NVPs. Otherwise we discard o and select the next furthest object in Q_{pre}.

Because typical range search approaches are based on network expansion method, it can only retrieve a small number of road segments from the database at each step. While VRS utilizes the properties of NVD and changes the expansion method from road segments expansion to NVPs expansion, which improves the processing time greatly. It outperforms all other existing range search approaches. Even though VRS requires some pre-computed values (e.g. border to border, generator to border) to be retrieved from the database, VRS can retrieve the required value only once to improve the processing time.

Since our proposed method in this chapter is based on NVD and deal with range search queries in road networks, an NVD-based range search method is preferred as our foundation. VRS is the only NVD-based range search method that provides a better solution than other network range search methods by reducing the expansion area and declining the rate of false hits based on NVD, so VRS is utilized as the foundation for all our two proposed methods.

```
Algorithm 1: VRS($q$, $e$)
    Input: query point:$q$, searching range:$e$
    Output: $Q_{pre}()$
1   Contain(q) finds the generator $o_q$;
2   $Q_{pre}() \leftarrow Q_{pre}() \cup o_q$;
3   if $e < dis_{net}(q, o_q)$ then
4       return $o$ with true property in $Q_{pre}()$;
5   else
6       NVD_Expansion();
7   end
8   return $o$ with true property in $Q_{pre}()$;
```

The general process of the VRS can be summarized into three steps:

- Step 1: Locating the Query Point (LQP): first involves contain(q) function, a common spatial index structure to find the NVP(q) containing the query point q, and put its generator o_q into $Q_{pre}()$ that is a sorted candidates queue used to store the object of interests whose validity has been checked. $Q_{pre}()$ does not only involve the valid objects which in the expected searching range, but also a few of false hits just out of the searching range. Then a network expansion method will be applied in the NVP(q) to get the $dis(q, o_q)$ and $dis(q, Borders)$.

- Step 2: Current Searching Range Expansion (CSRE): is a NVP expansion method that adds all neighbors of objects which are in $Q_{pre}()$ in the range, which can expedite the searching range expansion, until the size of current searching range is comparable with the expected searching range and simultaneously, all met objects will be inserted into $Q_{pre}()$.

- Step 3: Validating Objects (VO) and Gradually Shrinking (GS): compares the searching range e and dis_{max} in $Q_{pre}()$ (VO), if $e < dis_{max}$, set the value of inside/outside property of o_{max} to false and set dis_{max} to the next largest value in $Q_{pre}()$(GS); if $e > dis_{max}$, do step 2 until $e < dis_{max}$

When the algorithm of VRS terminates, the structure of $Q_{pre}()$ must be as follow:

$$Q_{pre}() = (\ldots, (o_x, \ldots, False), (o_y, \ldots, True), \ldots)$$

The object with false value indicates the object out of the range (e.g. o_{out}), while the one with true value is the expected object in the range(e.g. o_y). The algorithm of VRS is shown below:

The NVD_Expansion() function includes step 2 and step 3, to get all objects of interest in the searching range e.

3.3 Time Constrained Range

In this section, we present Time Constrained Range (TCR) to process range search query in road networks based on a double-weighted network diagram. TCR retrieves the objects of interest relying on the parameters of the searching range e and the traveling time t.

Example 3.3.1. *A businessman wants to find a bank branch within 10km and 20min traveling time due to his busy schedule.*

3.3.1 Motivation

Despite network distance is widely used to represent the shortest links between two points (road nodes or objects) and to describe the pattern of the road networks, it still has lots of limitation and flaws. It is well-known that traffic congestion have to be changed in different period of time, but due to the immutability of network distance, it cannot reflect the changes of the traffic condition, which requires a new variable to detect the changes in the road networks. Further more, because of variety of the traveling methods, sometimes the users may require a best solution from

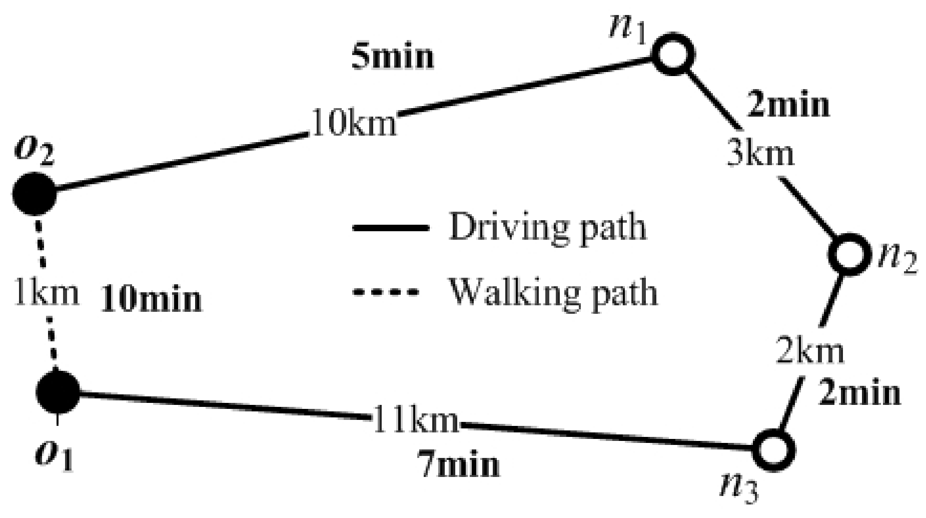


Figure 3.6: The variety of traveling methods

these traveling methods. Since network distance cannot distinct these methods, the comparison of different traveling methods becomes an impossible task. In Fig. 3.6, the driving traveling time is longer than the walking time from o_1 to o_2. When the user wants to get a path which costs the shortest time from o_1 to o_2, the conventional approaches can only give a result for only one particular traveling method, which may be the unexpected result. Consequently a new metric which can compare the efficiency of different traveling methods is demanded urgently.

Traveling time provides a nice solution to monitor the real-time events and the changes of the traffic congestion in the road networks, and it can integrate the different traveling paths in the travel time networks and evaluate them in one dimension. On the other hand, users may also want to get objects of interest restricted by both distance and traveling time.

3.3.2 Double Weighted Networks

Since traveling time is tightly associated with the moving speed of objects and and the traffic congestion, frequently updating and keeping tracking the traveling time will depress the efficiency of query processing seriously (the updating strategy of the moving objects will be illustrated in the chapter 4). So here, we assume the

Table 3.1: Time profile

Congestion	extra time (et)	Time Period
Heavy	2 min/km	7:00-9:00 and 16:00-19:00
Medium	1 min/km	9:00-16:00 and 19:00-22:00
Light	0.5 min/km	22:00-7:00

traveling time of each segment from one end to the other has been calculated off-line as another weight of road networks. The weight of traveling network diagram can be generated from the distance network diagram directly if all traffic events are ignored. Assume network distance (space weight sw) for segment n_1n_2 is $dis(n_1, n_2)$ and the speed limitation of n_1n_2 is sl, then the traveling time tt for n_1n_2 can be calculated by equation (3.1).

$$tt = \frac{dis(n_1, n_2)}{sl} \tag{3.1}$$

$$tt = \frac{dis(n_1, n_2)}{sl} + dis(n_1, n_2) * et \tag{3.2}$$

But since the perfect state is rare during traveling, the time profile should be attached if the user expects accurate results considering traffic congestion. Table 1 is one instance of time profile, which records the extra cost of the traveling time et that can be set using statistical data according to traffic condition and time periods. In this case, tt for n_1n_2 should be extended after calculating by equation (3.2).

tt is set as an additional weight (time weight tw) for each segments in the road networks, and then the double weighted networks can be drawn from space weight and time weight, thereby we can generate NVDs based on both weights. Fig. 3.7 demonstrates how the double weighted networks (refer to Fig. 3.7(a)) can be used to generate these two NVDs with et in a medium congestion road networks. After observation, though road network and objects are the same, the NVDs constructed on space weight and time weight are slightly different. In other words, we can only store one set of objects and road segments to get more than one NVDs according to different weights, which saves storage space dramatically. To explain our approach more clearly, we denote the space weighted NVD as NVD_s (refer to Fig. 3.7(b))and the time weighted NVD as NVD_t (refer to Fig. 3.7(c)).

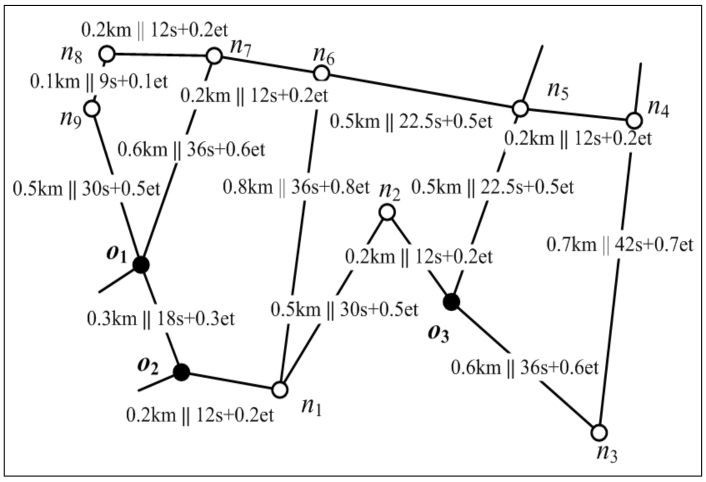
3.3.3 Algorithm of TCR

Time Constrained Range (TCR) approach adopts the double weighted NVD as its underlying framework and implements a Voronoi-based Range Search method (e.g. VRS) on both time-NVD and space-NVD. Since the searching on both dimensions does not effect each other until finalizing the result, the VRS method can be carried out synchronously on each dimension. After retrieving the data sets of these two dimensions, the final result can be obtained from their intersection. The algorithm for TCR is shown in Algorithm 2.

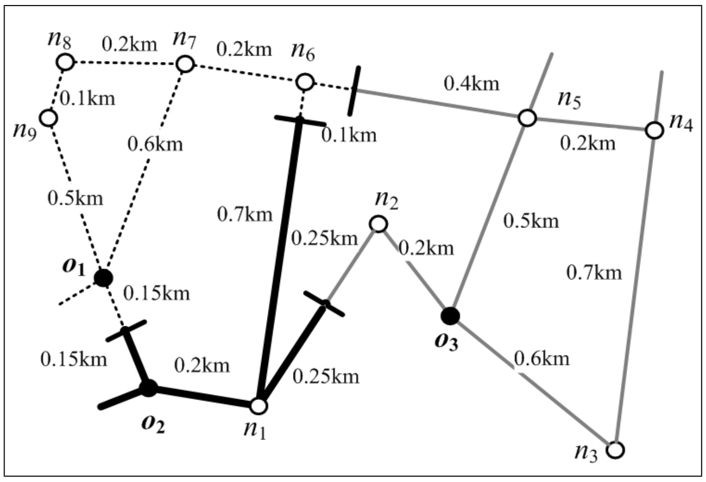
Algorithm 2: $TCR(q, e, t)$

```
Input: query point:$q$, searching range:$e$, traveling time:$t$
Output: $Q_{e\cap t}()$
1  $o_{q\_e} \leftarrow Contain_e(q)$; /* find $NVP_s(o_{q\_e})$ including $q$ in $NVD_s$ */
2  $o_{q\_t} \leftarrow Contain_t(q)$; /* find $NVP_t(o_{q\_t})$ including $q$ in $NVD_t$ */
3  $dis(q, o_{q\_e}) \leftarrow$ Dijkstra's algorithm($NVP_s(o_{q\_e})$);
4  $tt(q, o_{q\_t}) \leftarrow$ Dijkstra's algorithm($NVP_t(o_{q\_t})$);
5  if $e < dis_{net}(q, o_{q\_e})$ or $t < tt(q, o_{q\_t})$ then
6      Return $Q_{e\cap t}() \leftarrow \emptyset$
7  else
8      Insert $o_{q\_e}$ into $Q_e()$;
9      Insert $o_{q\_t}$ into $Q_t()$;
10     $Q_e() \leftarrow (Q_e() \cup NVD\_Expansion(NVD_s))$;
11     $Q_t() \leftarrow (Q_t() \cup NVD\_Expansion(NVD_t))$;
12 end
13 return $Q_{e\cap t}() \leftarrow (Q_e() \cup Q_t())$;
```

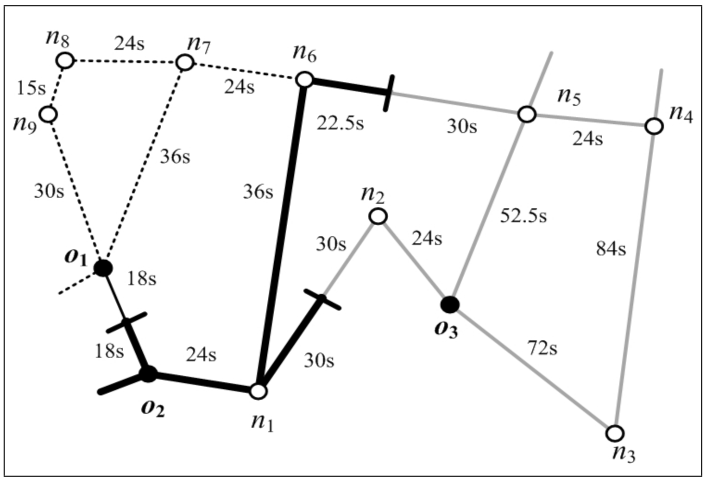
TCR first invokes function $contain(q)$ to get the position of the query point on NVD_s and NVD_t, and then implements Dijkstra's algorithm within $NVP_s(o_{q\_e})$, $NVP_t(o_{q\_t})$ to get the values of network distance $dis_{net}(q,\ o_{q\_e})$ and traveling time $tt(q, o_{q\_t})$ respectively. If the nearest neighbor o_{q_e} is within range e and the traveling time of the fastest reachable object o_{q_t} is less than t, we do the NVD expansion on both NVD_s and NVD_t to get the interim result $Q_e()$, $Q_t()$ respectively, whose



(a) Road networks



(b) S-NVD_s



(c) T-NVD_t

Figure 3.7: Generating of space-NVD and time-NVD

intersection will be the final result. Otherwise, we return a empty set, which means that no qualified objects of interest satisfy the input query.

3.4 Region Constrained Range

n this section, we propose Region Constrained Range (RCR) constructed on Weighted Network Voronoi Diagram (WHNVD) to deal with the range search queries that ask for the objects of interest locating in some pre-defined region that can be seen as a weighted object, such as, shopping center, national park and so on. See Example 3.4.1, which is a typical region constrained range search query. The expected searching range e is 500km and the objects of interest (small water areas) are constrained within some pre-defined region $\Re$ (national park) strictly. Apparently, the traditional range search approaches cannot give the proper result for this query.

Example 3.4.1. *A biologist wants to do a survey for of small water areas in the national park within 500km to find out the influences of water pollution to the wild animals.*

3.4.1 Motivation

When users intend to find the objects of interest within a searching range, sometimes they expect these objects belong to some region, such as, the university campus, the suburb in a city, the shopping center and so on. Fig.3.8 shows all parking lots (stars) in a university campus. The traditional range search methods only focus on the objects of interest needed to be found, hence they cannot recognize whether the objects of interest belong to the expected region or not. That is the main motivation for proposing a new approach, RCR, dealing with these type of range search queries.

3.4.2 Weighted Network Voronoi Diagram

We designed a novel Voronoi diagram, named weighted network Voronoi diagram (WNVD), as the underlying framework for our proposed range search algorithm.

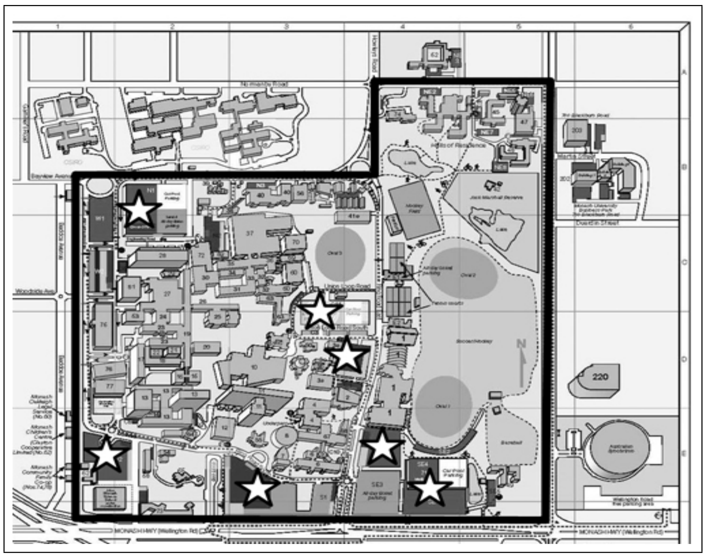


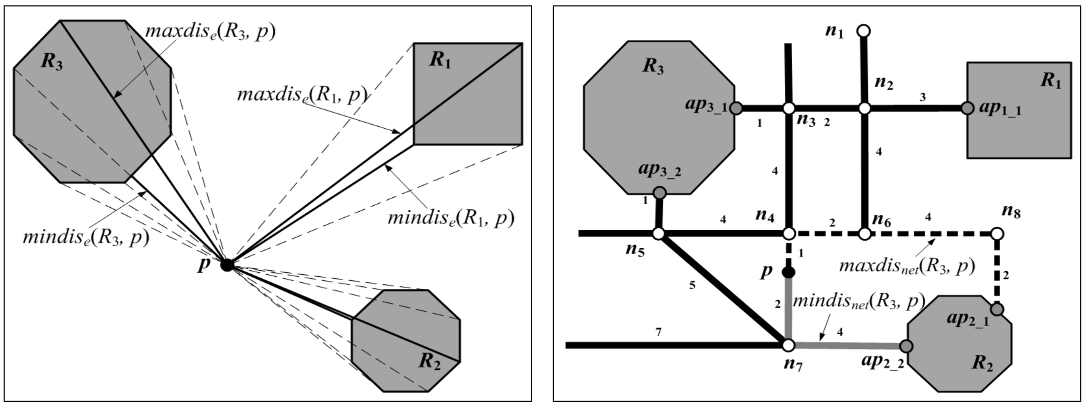
Figure 3.8: Objects of interest within a weighted object

Before that, we first introduce what is a weighted object and illustrate the distance calculation between a polygon object and a point in road networks.

A weighted objects is abstracted as a polygon rather than a point when generating network Voronoi diagrams, as their extents should not be ignored. Normally, weighted objects are the real entities occupied a large area, such as, national parks, shopping centers and school campuses.

There are two important distances from a polygon object to a point in Cartesian space, namely, the minimum distance $mindis_e$ and the maximum distance $maxdis_e$, indicating the shortest and longest distance among the distances from this point to the vertices and to the edges. As Fig. 3.9(a) shows, $mindis_e$ and $maxdis_e$ can be from a vertex of a polygon (e.g., $mindis_e(R_1,\ p)$ and $maxdis_e(R_3,\ p)$) or the intersection of a perpendicular (e.g., $mindis_e(R_3,\ p)$).

Unlike Euclidean space, in the road networks, all the objects are restricted by road pathes. Therefore a polygon object has to intersect with the road networks, which means that a path from any point in road networks to a polygon object have to pass through one of the entries or exits of this polygon object. We define the entries and the exits of weighted objects as access points and denote as ap (Refer to Fig. 3.9(b)).



(a) $maxdis_e(R, p)$ and $mindis_e(R, p)$ (b) $maxdis_{net}(R_2, p)$ and $mindis_{net}(R_2, p)$

Figure 3.9: Distance calculation from polygons to a point in Cartesian space and road networks

Definition 3.4.1. *Access points AP_k are the set of intersections between edges Ed of weighted object R_k and road networks RN.*

$$AP_k = \bigcup_{m=1}^{n} ap_{k\_m} = Ed(R_k) \cap RN$$

According to the definition of access point, a weighted object can have multiple access point *ap*, and an *ap* can only belong to a particular weighted object. The two subscript k, m of *ap* indicate which weighted object it belongs to and the indexing number of this access point respectively.

As a weighted object can have more than one access point, the distance from a random point p to the weighted object can have multiple values, then distances between a polygon object and a point can be expressed as $dis_{net}(p, AP)$, where AP = (ap_1, ap_2, ..., ap_n). See Fig. 3.9(b) as an example. The $mindis_{net}$ and $maxdin_{net}$ can be expressed as:

$$mindis_{net}(p, R_k) = \min(dis_{net}(p, AP_k))$$

$$maxdis_{net}(p, R_k) = \max(dis_{net}(p, AP_k))$$

Now, we introduce weighted network Voronoi diagram, which differs from weighted

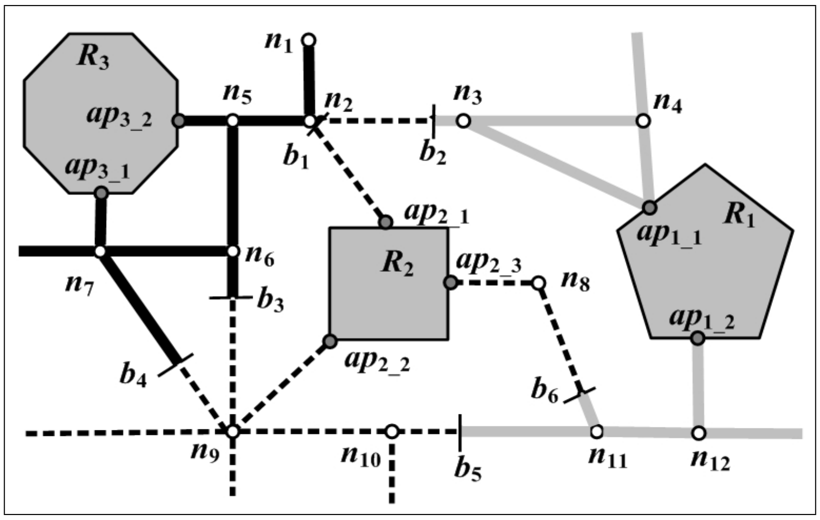


Figure 3.10: Weighted Network Voronoi Diagram (WNVD)

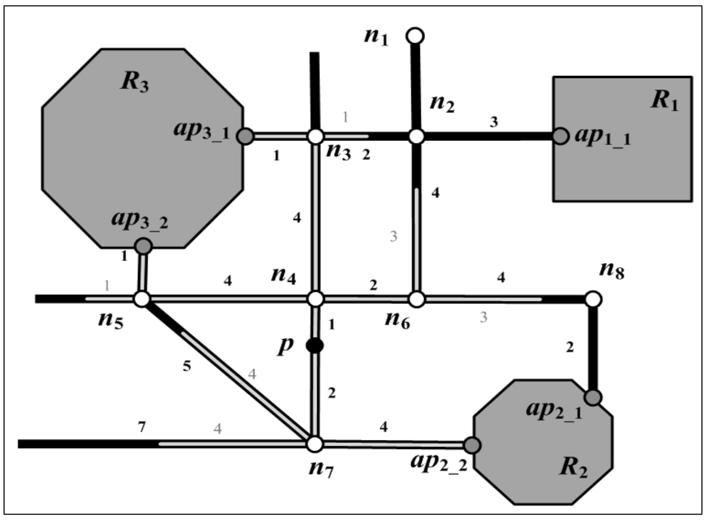


Figure 3.11: Distance calculation in road networks

Voronoi diagram and ordinary network Voronoi diagram (see Fig. 3.10). It can be defined as:

Definition 3.4.2. *Given a set of weighted objects* $\Re = \{R_1, R_2, \ldots, R_n\}$ *and road networks RN. The NVP(R_k) is a set of segments L in the road networks,* $L \in RN$*. Then* $\forall p \in L$*, have* $mindis_{net}(p, R_k) \leq mindis_{net}(p, R_l)$*,* $k,l \in \aleph$*,* $k \neq l$*.*

$$WNVD = \bigcup_{k=1}^{n} NVP(R_k)$$

In term of definition of WNVD, all details inside of weighted objects are ignored during the construction of a WNVD. But when we calculate the distance $dis_{net}(p, ap)$

Table 3.2: Distances from p to weighted objects

To R_1 through ap_{1_1}:
$d_{net}(p, ap_{1_1}) = pn_4 + n_4n_6(n_4n_3) + n_6n_2(n_3n_2) + n_2ap_{1_1} = 10$
To R_2 through ap_{2_1}, ap_{2_2}:
$d_{net}(p, ap_{2_1}) = pn_4 + n_4n_6 + n_6n_8 + n_8ap_{2_1} = 9$
$d_{net}(p, ap_{2_2}) = pn_7 + n_4ap_{2_2} = 6$
To R_3 through ap_{3_1}, ap_{3_2}:
$d_{net}(p, ap_{3_2}) = pn_4 + n_4n_3 + n_3ap_{3_1} = 6$
$d_{net}(p, ap_{3_1}) = pn_4 + n_4n_5 + n_4ap_{3_2} = 6$

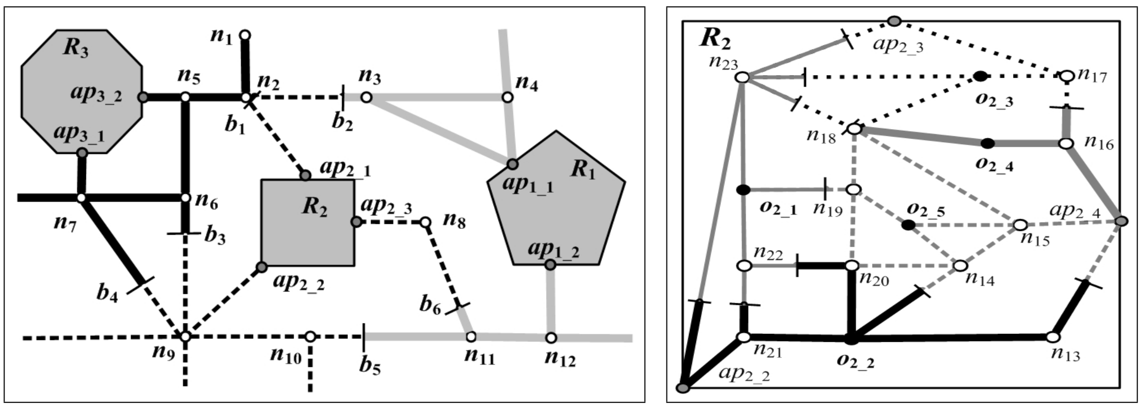
by Dijkstra's algorithm, we have to consider the pathes in WNVD, as the shortest path from p to an ap may go through the path inside.

The $mindis_{net}(p,\ R)$ is the lower boundary and we used it to estimate whether the weighted object is intersect with the searching range.; While the $maxdis_{net}(p,\ R)$ is the upper boundary, if $maxdis_{net}(p,\ R) < e$, then R is fully contained by the range.

For example, there are three weighted objects, namely, R_1, R_2 and R_3 each including several access points in Fig.3.11. After performing Dijkstra's algorithm, the distances from point p to these weighted objects are in Table 3.2. If the searching range is 6kms, for R_1 the $mindis_{net}(p,\ R_1)$=10>6, then we can say R_1 is out of range; for R_2, the $mindis_{net}(p,\ R_1)$=6 and $maxdis_{net}(p,\ R_1)$=9>6, then R_2 intersects with the searching range; for R_3 $maxdis_{net}(p,\ R_1) = 6$, then we say R_3 is fully contained by the searching range.

3.4.3 Algorithm for RCR

We proposed a two-level NVDs structure, the first level NVD is weighted network Voronoi diagram constructed by weighted objects containing a set of point objects, while the second level NVD is built inside each weighted objects. Fig.3.12 shows an example of this two-level NVD structure. Fig.3.12(a) is a WNVD and it includes three weighted objects R_1, R_2, R_3, some road nodes involving the intersections and starting/ending points (white points) and a set of access points (AP, gray points).



(a) First level NVD, WNVD (b) Second level NVD

Figure 3.12: Two-level network Voronoi diagram structure

Fig. 3.12(b) is a NVD constructed on R_2. Beside the access points, ap_{2_2}, ap_{2_3}, $ap_{2\_4} \in AP_2$, inheriting from the first level NVD, it also involves the road nodes (white points) and some objects of interest, o_{2_1}∼o_{2_5} (black points). If an access point within R_k falls into $NVP(o_{k\_n})$, then this access point is denoted as ap_{k_n}. The construction of the second level NVDs are constructed inside of weighted objects (e.g., NVD on R_2) as a normal NVD.

The searching of RCR includes two levels, we apply VRS on the first level of NVD to get all valid regions that include the objects of interest, referring to Algorithm 3. The relative position of the searching range and the weighted objects can be categorized into three conditions:

- (i) If $mindis_{net}(q,\, R_k) > e$, the R_k is out of searching range.
- (ii) If $maxdis_{net}(q,\, R_k) < e$, the R_k is fully or partially contained by searching range.
- (iii) If $mindis_{net}(q,\, R_k)<e<maxdis_{net}(q,\, R_k)$, then R_k intersects with searching range

If the query point q is contained by a weighted object R_k, then the $mindis_{net}(q,$ $R_k)=$ $mindis_{net}(q,\, R_k)=0$. For the first condition, we can simply discard all objects of interest in the weighted objects. Only the weighted objects intersecting with the searching range need to do the second level search, referring to Algorithm 4 Since the shortest distance from a query point q to an point object o_k inside a weighted region R_k has to be through one of the access points, ap_k, then $dis_{net}(q, o_k)$ can be calculated by:

$$dis_{net}(q, o_k) = dis_{net}(q, ap_k) + dis_{net}(ap_k, o_k)$$

We need to clarify that we cannot use $mindis_{net}(q, R_k)$ to calculate $dis_{net}(q, o_k)$. Since the point object in a weighted region may closer to other access point. Assuming that:

$$mindis_{net}(q, R_k) = dis_{net}(q, ap_{k_n}) < dis_{net}(q, ap_{k_m})$$

But if

$$dis_{net}(ap_{k_n}, o_k) > dis_{net}(ap_{k_m}, o_k)$$

and

$$dis_{net}(ap_{k_n}, o_k) - dis_{net}(ap_{k_m}, o_k) < dis_{net}(ap_{k_m}, o_k) - dis_{net}(ap_{k_n}, o_k)$$

then

$$mindis_{net}(q, R_k) + dis_{net}(q, ap_{k_n}) > dis_{net}(q, o_k)$$

Algorithm 3: RCR-1(q, e, $\Re$)–First Level Search

```
   Input: query point:$q$, searching range:$e$, Pre-defined Region:$\Re$
   Output: $RQ()$
1  $R_q \leftarrow$ Contain($q$) on WNVD($\Re$); /* $R_q \in \Re$ */
2  Dijkstra algorithm to get $maxdis_{net}(R_q, q)$ and $mindis_{net}(R_q, q)$;
3  if $e \leq mindis_{net}(R_q, q)$ then
4      Return $PQ() \leftarrow \emptyset$
5  else
6      if $mindis_{net}(R_q, q) < e < maxdis_{net}(R_q, q)$ then
7          Insert $R_q$ into $RQ()$;
8          NVD_Expansion(WNVD) /* retrieve all weighted objects $R$
               may intersect with searching area */
9          ;
10     end
11 end
12 return $RQ()$;
```

Algorithm 4: *RCR*-2($RQ()$, $RQ'()$)–Second Level Search

```
Input: $RQ()$
Output: $OQ()$
1 for $R \in RQ()$ do
2     for $o \in R$ do
3         if $dis_{net}(q, o) < e$ then
4             Insert $o$ into $OQ$
5         end
6         ;
7     end
8 end
9 return $OQ()$;
```

3.5 kNN Constrained Range

We design a new method, called *k*NN Constrained Range (*kCR*) to balance the efficiency of searching range *e* and the number of objects of interest *k* given by the user. Referring to Example 3.5.1, the users may not only expect the objects in the searching range, but also require a certain number of objects of interest.

Example 3.5.1. *A marketing manager would like to do an investigation on the marketing status of their products at all supermarkets within 50 km, and the target should involve 10 supermarkets.*

In this case, the object of interest is supermarket, within range $e = 50$ km, and the target number of objects $k = 10$ (supermarkets)

3.5.1 Motivation

Range search and kNN are the two most popular queries in spatial databases. The former confines all objects of interest on a certain range *e*, whereas the latter need to retrieve *k* nearest objects of interest. Obviously, sometimes these two queries can cooperate to retrieve objects of interest satisfying both requirements. We named this kind of queries as, *k*NN constrained range search. To make this query more applicable, we allow e to make an adjustment to k, the strategies of which are discussed in the next section.

3.5.2 e vs. k

The traditional range search approach (e.g, RNE, VRS) retrieves all objects of interest whose network distance to the query point is smaller than the search range e and puts them into a queue $Q()$. To answer the kNN constrained range search query, we can retrieve the k nearest objects of the query point from $Q()$, if the number of objects in $Q()$ is larger than k. But supposing the number of objects in $Q()$ is smaller than the required k, which means k^{th} object does not exist in the current searching range, then we may need to expand the range e to get more objects to meet the required k if the cost is acceptable, instead of simply returning an empty set to the user. The above statements can be summarized as follows:

- If Count($Q()$) $> k$, apply a kNN approach to $Q()$
- If Count($Q()$) $= k$, return $Q()$ to the user
- If Count($Q()$) $< k$, expand the range e if the cost is acceptable

Here, we define a deviation ratio dr for both e and k to estimate the cost when the searching range needs to be expanded.

Definition 3.5.1. *Given a set of objects of interest o in the road networks, a query point q, a searching range e and a natural number k. Assuming the range search result for e is $Q()$, $Count(Q()) < k$ and e is expanded to the l^{th} nearest object o^l of q, $o^l \notin Q()$, $Count(Q()) < l \leq k$, the deviation ratio of e and k, dr_{el}, dr_{kl}, for object l are:*

$$dr_{el} = \frac{dis(q, o^l) - dis_{max}(q, Q())}{e}$$

$$dr_{kl} = \frac{l - Count(Q())}{k}$$

If and only if $dr_{el} \leq dr_{kl}$, which means small expansion of range e can get a large set of objects belonging to k, the cost is deemed to be acceptable. We should clarify

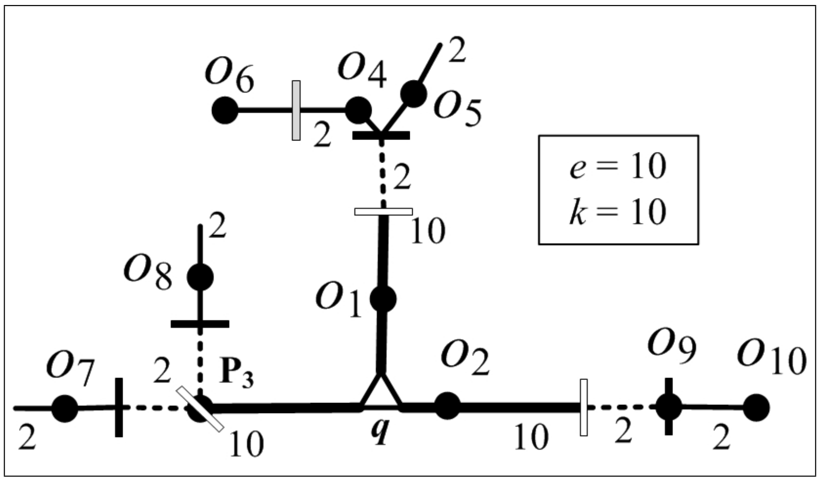


Figure 3.13: Explanation of deviation ratio

that $dr_{el} \leq dr_{kl} \nRightarrow dr_{em} \leq dr_{km}$, Count($Q()$)$<m<l<k$, for the distribution of the objects of interest is random. See Fig. 3.13, e=10, k=10, Q = (o_1, o_2, o_3), for o_{10} that is the 9th nearest neighbor of q (shaded triangle), we have:

$dr_{e9} = \frac{14-10}{10} = \frac{4}{10}$

$dr_{k9} = \frac{9-3}{10} = \frac{6}{10}$

$dr_{e9} < dr_{k9}$

whereas for o_9, the 4th NN of q:

$dr_{e4} = \frac{12-10}{10} = \frac{2}{10}$

$dr_{k4} = \frac{4-3}{10} = \frac{1}{10}$

$dr_{e4} > dr_{k4}$

So when expanding the searching range, we reach to the k^{th} NN out of the range directly, if $dr_{ek} \leq dr_{kk}$, then we consider the cost of this expansion is acceptable, otherwise, we reduce k by 1 repetitively, until $dr_{el} \leq dr_{kl}$, Count($Q()$) $< l < k$.

3.5.3 Algorithm for kCR

In the algorithm for kCR, as the NVD is also used as the underling framework, any NVD-based approach for range/kNN is feasible for kCR. We adopt VRS as

Algorithm 5: $kCR(q,\ e,\ k)$

```
Input: query point:$q$, searching range:$e$, Quantity of objects user expected:$k$
Output: $Q_{ek}()$, $fe$
1  $Q_{pre}() \leftarrow$ VRS($q$, $e$);
2  if $Count(Q_{pre}() \geqslant k$ then
3      $Q_{ek}() \leftarrow$ get $k$th object from $Q_{pre}()$;
4      $fe \leftarrow e$;
5  else
6      $Q_k() \leftarrow k$NN(q, k) approach;
7      $fe \leftarrow dis(q, o^k)$;
8      $l \leftarrow k$;
9      Calculate $dr_{el}$;
10     Calculate $dr_{kl}$;
11     while $(dr_{el} > dr_{kl})$ AND $(Q()) < l \leqslant k)$ do
12         $Q_k() \leftarrow Q_k()$ - $o^l()$;
13         $l \leftarrow l$-1;
14         Calculate $dr_{el}$;
15         Calculate $dr_{kl}$;
16         $fe \leftarrow dis(q, o^l)$;
17     end
18     $Q_{ek}() \leftarrow Q_k()$;
19 end
20 return $Q_{ek}()$ and $fe$
```

the sub-function of kCR to retrieve objects of interest within in the searching range and store them in $Q_{pre}()$. Because $Q_{pre}()$ is a sorted queue, if Count($Q_{pre}()$) $\geq k$, then the k nearest objects within e can be obtained easily; if Count($Q_{pre}()$) $< k$, then kCR involves an NVD-based kNN approach to get $dis(q, o^k)$, if $dr_{ek} > dr_{kk}$, we reduce k, until $dr_{el} \leq dr_{kl}$, Count($Q()$) $< l < k$. The algorithm for kCR is shown in Algorithm 5.

Besides k constrained range search result $Q_{ek}()$, kCR also returns the factual searching range fe to indicate the original searching range e is already changed if the number of objects of interest is less than k. The objects in $Q_{ek}()$ can still be smaller than k, if the cost to get the k^{th} object is higher than the expected benefit to get more objects.

3.6 Performance Evaluation

n this section, we evaluate our three proposed algorithms by using low density and high density synthetic data sets with thousands of objects in road networks.

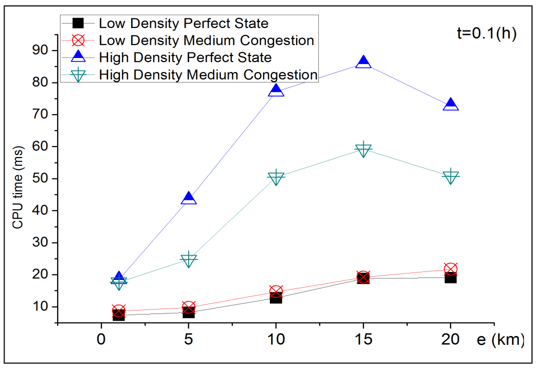
All experiments were programmed in Java and conducted on a IBM Thinkpad T43 laptop running MS Windows XP professional service pack 2, with an Intel Pentium Mobile CPU of 1.86GHz, one gigabyte of RAM and 40GB disk storage. For each algorithm, we generated random road network segments with a set of objects to evaluate its performance in terms of the CPU time and other important factors. The data of all the experiments shown below are collected by averaging the results for 1000 random query points on each experiment to reduce the inaccuracy.

3.6.1 Experimental Results of TCR

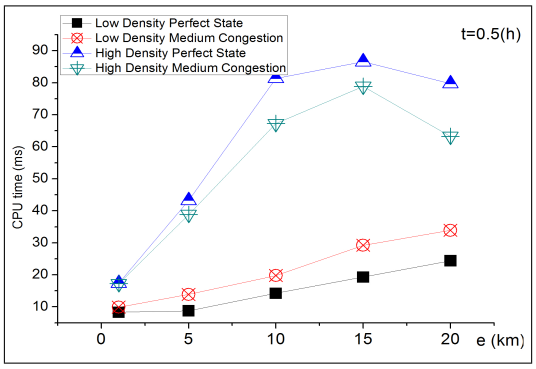
First we estimate the performance of our *TCR* by comparing its CPU time and memory size in the low density and high density environments for both perfect state (no traffic congestion considered) and medium congestion road networks. Because all distance and traveling time between any two objects are pre-calculated, the main factor effecting the CPU time (refer to Fig.3.14) and the memory size of our *TCR* (refer to Fig.3.15) is the density of objects of interest when searching range e and t are increasing. The relationships of e and CPU time and memory size of *TCR* are approximate to the linear functions whose slope for high density objects are sharper than the low density objects. Since the time profile is invoked in traveling time networks, the CPU time and memory size of the medium congestion networks are slightly larger than them in the perfect state in both low density and high density environments.

3.6.2 Experimental Results of RCR

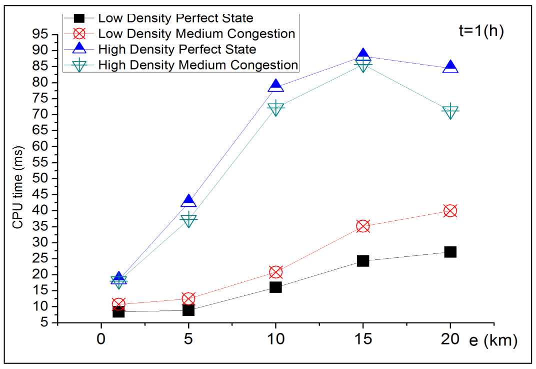
This set of experiments evaluate the performance of RCR in term of CPU time, memory size and number of weighted objects which are within (R) and partially



(a) CPU time t=0.1h

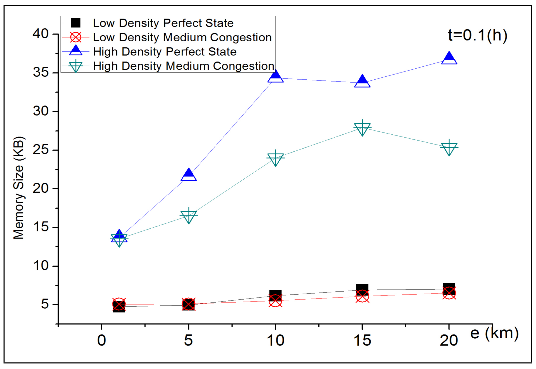


(b) CPU time t=0.5h

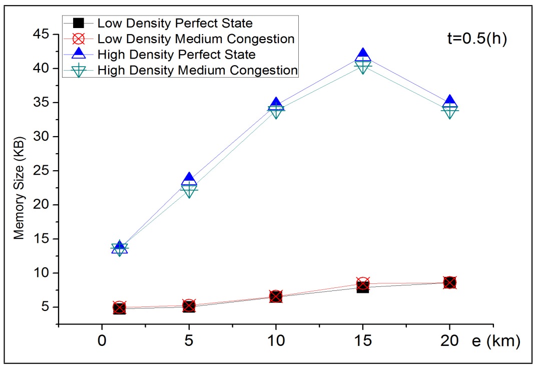


(c) CPU time t=1h

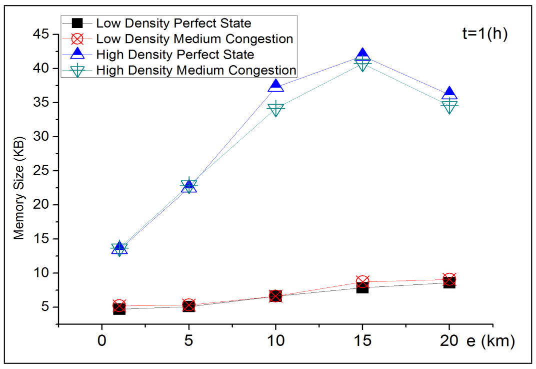
Figure 3.14: CPU time of TCR



(a) Memory size t=0.1h

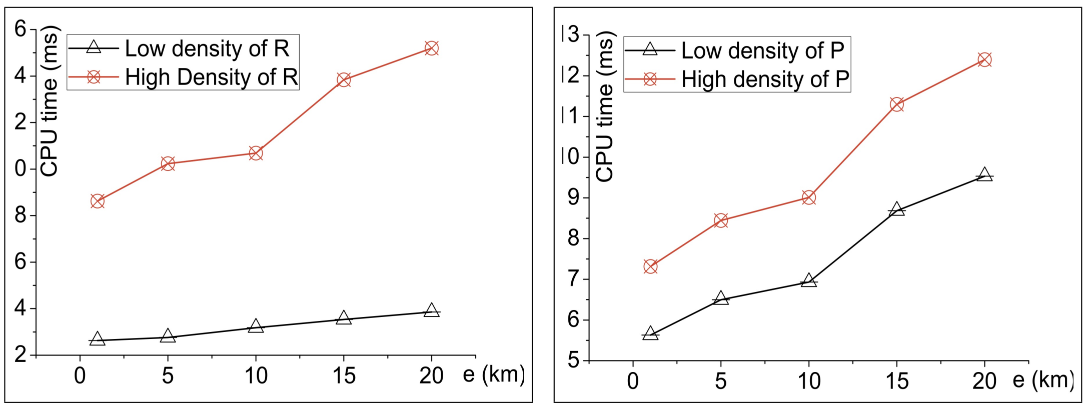


(b) Memory size t=0.5h



(c) Memory size t=1h

Figure 3.15: Memory size of TCR



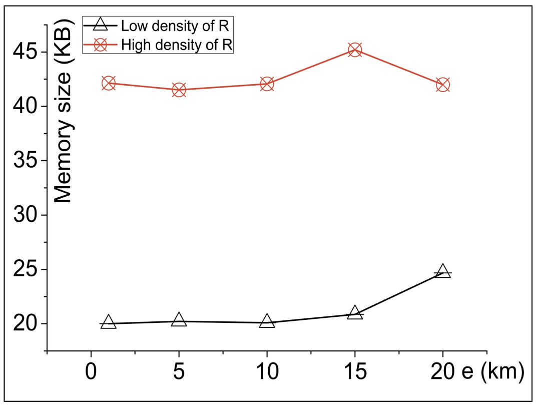
(a) Density of WO (b) Density of OI

Figure 3.16: CPU time of RCR

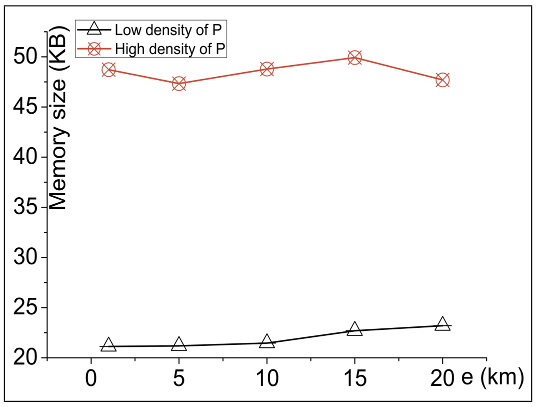
within (R') the searching range, respectively. Fig.3.16 shows the processing time of our RCR in different environments on the density of weighted objects (region R) and objects of interest (o) within the region. Since the essence of RCR is applying VRS on level-1 and level-2 NVD sequentially, the CPU time increases linearly on both NVDs. The increasing density of the corresponding objects also affects the performance of RCR negatively.

Fig.3.17 shows the processing memory size of our RCR in different density environments of weighted objects (region) and objects of interest within the region. Observing from Fig.3.17, the searching range expansion will not affect memory size dramatically, except the objects evolved in the searching range increases extremely, whereas the increasing of objects density becomes the main factor that enlarges the memory size of RCR.

Fig.3.18 displays the number of objects within/partially within the range. As most of processing time of RCR is spent on the weighted objects (R')partially within the range, its percentage should be kept in an extremely low level to guarantee good performance. From Fig.3.18, we can see that the increase of R' is slower than the increase of the objects (R), which means that only few weighted objects needs to be checked when the searching range is large.



(a) Density of WO



(b) Density of OI

Figure 3.17: Memory size of RCR

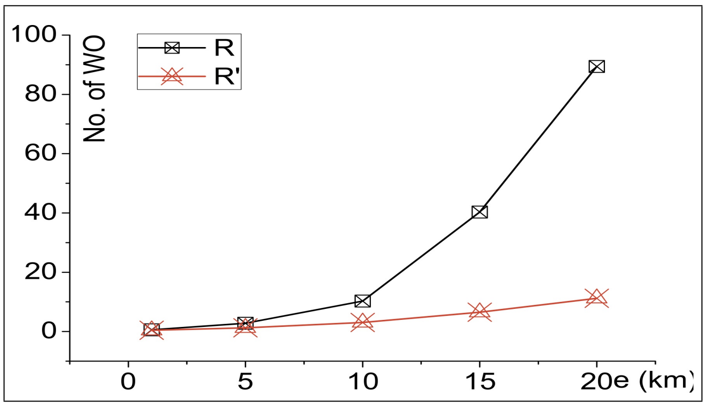
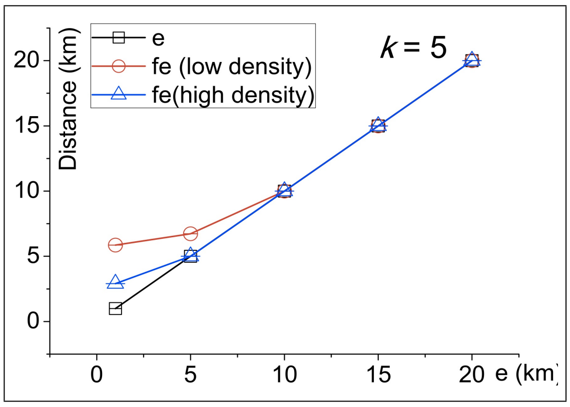


Figure 3.18: No. of R/R' for RCR

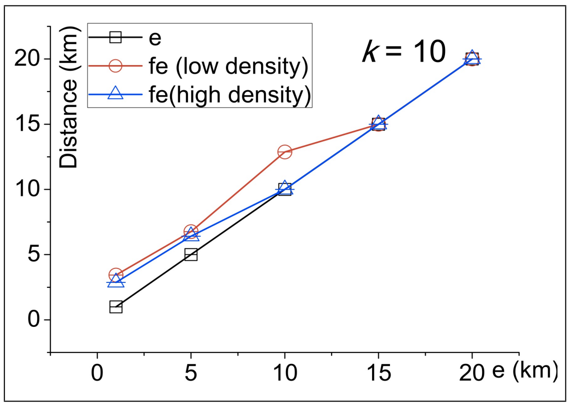
3.6.3 Experimental Results of kCR

Finally, we tested the CPU time of kCR in different scenarios and compared the size of the factual range fe with e. As we explained in section 5, if the number of objects within the searching range is larger or equal to k, the final result can be obtained simply by returning the corresponding objects in Q_{pre}. Otherwise, we need to expand the current searching range e. This is the reason why the processing time of kCR is in direct proportion to the number of nearest neighbors k in Fig.3.20.

Observing from Fig.3.19, the searching range e is the lower boundary of fe when the number of objects of interest within e is less than k and the fluctuation of fe deviates from e slightly. We can also see that the smaller k is, the earlier fe reaches to e, for which the increasing of e will retrieve more objects.

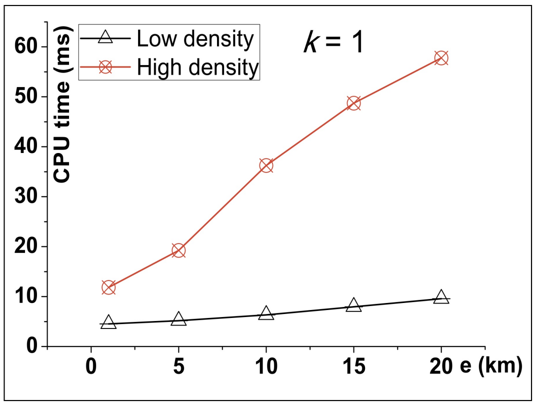


(a) k=5

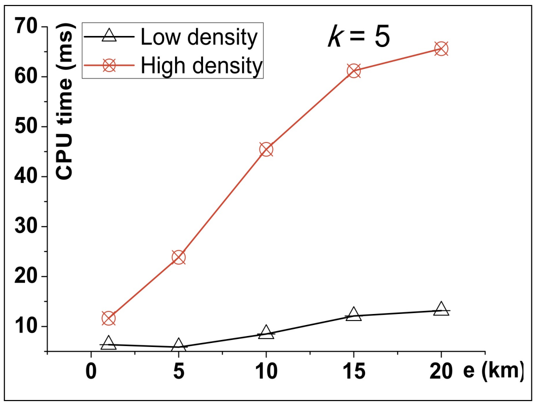


(b) k=10

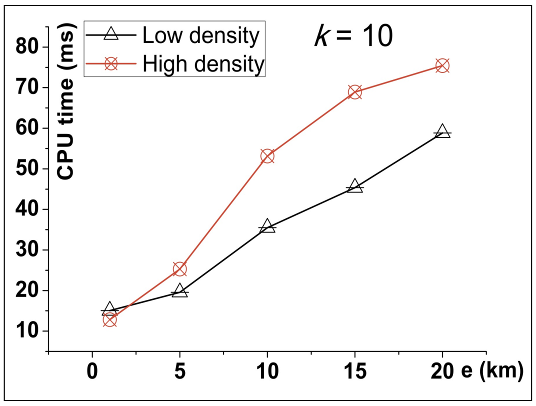
Figure 3.19: fe vs. e of kCR



(a) t=0.1h



(b) t=0.5h



(c) t=1h

Figure 3.20: CPU time of kCR

3.7 Summary

In this chapter, we proposes three NVD-based range search approaches to process the range search queries with several additional parameters (traveling time t, pre-defined region $\Re$ and the quantity of objects k) in the road networks.

The first approach, $TimeConstrainedRange$ (TCR), solves the problems that require the objects of interest within both network distance and traveling time range by utilizing the double weighted networks.

The second approach, *Region Constrained Ranges* (RCR), retrieves the objects of interest not only within the searching range, but also locating in some pre-defined region. For this purpose, we construct a Weighted-Hierarchical NVD that deals with the region involving the objects of interest as an big object in the first level NVD, while the objects of interest within these big object forms the second level NVD.

The last approach, $kNNConstrainedRange$ (kCR), discusses a kind of queries expecting to find k objects within the searching range. Then we study how to make a balance between searching range e and k to get the optimized result. These approaches can also combined to solve the extremely complex range search queries. Our experiments show that all proposed approaches can process the corresponding queries efficiently.

Chapter 4

Range Query Processing on Moving Objects

In this chapter, we present algorithms to monitor moving range queries and moving objects of interest in spatial networks. Our research reported in this chapter also appeared in [XZT+11a] and [XTSS10].

4.1 Overview

Mobile databases have benefited through the rapid advancement of global positioning systems (GPS) and geographic information system (GIS). One of the most common queries in mobile, as well as in spatial, databases is range search that finds all objects of interest within the given region or radius. It can be defined formally as: given a query point q, a user specified range e and a set of objects of interest $\wp$, find all objects of interest from $\wp$ within range e from q.

Range search queries in road networks have been extensively discussed in the last decade. The most two well-known methods in this area are Range Euclidean Restriction (RER) and Range Network Expansion (RNE) [PZMT03], which calculate the shortest network connection between two objects locating in the road networks, called network distance, by implementing Dijkstra's algorithm [Dij59] or other distance retrieving approach [SS09a]. All objects of interest returned to users are judged according to their network distances to the query point. The value of network distance does not only concern with the relative position of objects, but also with the pattern of the road networks. Therefore RER and RNE [PZMT03] provide more accurate results than Euclidean range search approaches [Gut84, BER85]. Whereas the disadvantages of these approaches [PZMT03, SSA08] are also obvious, since RER and RNE are based on expansion methods, lots of false hits are retrieved during the expansion, which makes them become time-consuming methods. Moreover if the density of the discrete objects is very low, the performance of expansion-based approaches will be decreased dramatically.

On the other hand, continuous range search is received significant research attention in the past few years [CHC04b], In our previous work, we proposed an approach named *Continuous Range Search* (CRS) [XZTS08] to address this kind of queries on both Euclidean distance and network distance. But *CRS* needs to divide the traveling path into some sub-segments according to the intersections of the road networks like most existing continuous methods [TPS02, KS05, SE06] that are not suitable for the complex networks, which is normal in reality, the performance of *CRS* will decrease dramatically.

The performance of monitoring moving objects for range search queries is another tough problem in spatial databases, especially, in road networks. As Monitoring moving objects should achieve equilibrium be- tween the frequency of updating and the accuracy of position of moving objects. If updates are very often, which has high cost, the error in location of the moving objects is kept very small. On the contrary, if we want to minimize the updating cost, then the error becomes larger.

In this chapter, we propose two approaches to monitoring and predicting the result of range queries in road networks.

One is called *Voronoi Continuous Range* (*VCR*) which are based on some natural properties of Network Voronoi Diagram (NVD) [OBSC00a], dedicated to range search query processing for both static and moving queries. *VCR* does not need to divide the moving path into some segments any more. Consequently the performance and the applicability of the continuous range search have been improved considerably. Since the details of connections in networks can be ignored during the moving of the query point, *VCR* can solve the undefined-trajectory continuous query properly as well.

The other is called *RangeMonitoring*, which discusses how the updating cost of the moving objects can be minimized during the range query processing.

The remainder of this chapter is structured as follows: the preliminary of our approaches, NVD and Voronoi-based range search are introduced in Section 4.2. Section 4.3 and Section 4.4 illustrate our proposed *VCR* and *RangeMonitoring* respectively. Section 4.5 evaluates our proposed techniques by showing the experimental and comparison results with some existing works. Section 4.6 summarizes this chapter.

4.2 Preliminary

4.2.1 Network Voronoi Diagram

Our proposed algorithms, *VCR* use Voronoi diagram as their underlying framework. Voronoi diagram has lots of geometry properties that can make immense improvement in the performance of range search queries processing. A Voronoi diagram is a decomposition of a plane space according to the position of a set of discrete points (site). Each site generates a Voronoi Polygon (VP), which involves all points closer to its site than to any other. A VP is formed by a set of Voronoi edges that are some subset of locus of points equidistant from two adjacent sites. The intersections of these edges for a site is called Voronoi vertex. The definition of Voronoi diagram is:

Definition 4.2.1. *Given a set of discrete objects $\wp = (o_1, ..., o_n)$ $(n > 1)$ in road networks, $VP(o_i) = \{\forall o \mid d_n(p, o_i) \leq d_n(p, o_j)\}$ $(i, j \in I_n$ and $i \neq j)$. The Voronoi*

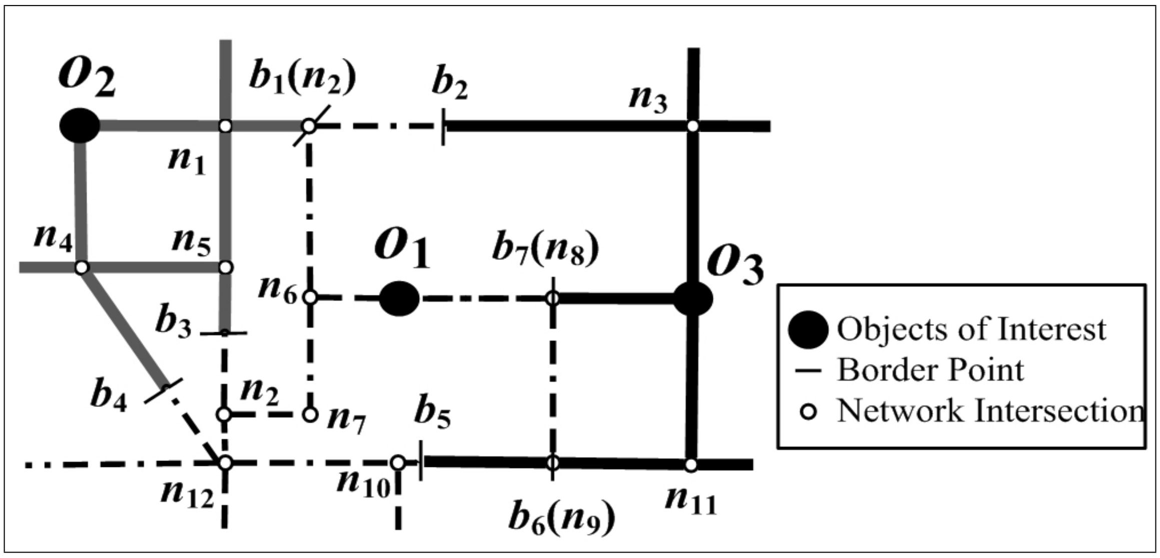


Figure 4.1: Network Voronoi Diagram

Diagram for $\wp$ is:

$$VD(\wp) = \bigcup_{i=1}^{n} VP(o_i)$$

This definition indicates that for any point inside the $VP(o_i)$, its distance to o_i must be smaller than to others generators, then $VP(o_i)$ is called Voronoi polygon associated with o_i.

An NVD is a special kind of Voronoi diagram constructed on spatial networks [OBSC00a]. The decomposition is based on the connection of the discrete objects rather than Euclidean distance. In the NVD, the Voronoi polygon changes to a set of road segments termed Network Voronoi polygon (NVP) and the edges of the polygons also shrink to some midpoints, termed border points, of the road network connection between two objects of interest.

Fig.4.1 shows an example of NVD. Besides the objects of interest (o), an NVD also includes some road network intersections (n) and border points (b). According to the properties of Voronoi diagram, from border points to a pair of adjacent objects is equidistant (e.g. $dis(b_7, o_1) = dis(b_7, o_3)$), then we just need to use Dijkstra's algorithm within one Voronoi polygon to get the distance from a generator to its borders. The distance between objects can be calculated by selecting the minimum distance to their shared borders and doubling this value (e.g. $MIN(dis(o_3, o_1)) = 2 * dis(b_7, o_3) = 2 * dis(b_7, o_1)$).

As all distances from border points to generators/other border points can be precalculated using Dijkstra's algorithm and stored in databases, then when a range query is issued, all needed distances can be retrieved from databases rather than calculated them online.

4.2.2 A Sub-function—VRS

Voronoi Range Search, VRS, is a Voronoi-based range search approach [XZT+09b] for static range queries on static point objects in road networks. *VRS* involves $contain(q)$ to retrieve the NVP containing the query point first and check whether its generator is in the range e or not. If it is in the range, then we bulk load a set of adjacent NVPs into $Q_{pre}()$ and compare the network distance from furthest generator o to query point with e. If $dis(o, q) < e$, then we repeat expansion by loading the next round of NVPs. Otherwise we discard o and select the next furthest object in Q_{pre}.

Because typical range search approaches are based on network expansion method, it can only retrieve a small number of road segments from the database at each step. While VRS utilizes the properties of NVD and changes the expansion method from road segments expansion to NVPs expansion, which improves the processing time greatly. It outperforms all other existing range search approaches. Even though VRS requires some pre-computed values (e.g. border to border, generator to border) to be retrieved from the database, VRS can retrieve the required value only once to improve the processing time.

Since all of our proposed methods in this paper are based on NVD and deal with range search queries in road networks, an NVD-based range search method is preferred as our foundation. VRS is the only NVD-based range search method that provides a better solution than other network range search methods by reducing the expansion area and declining the rate of false hits based on NVD, so VRS is utilized as the foundation for all our two proposed methods.

Algorithm 6: VRS(q, e)

```
    Input: query point:$q$, searching range:$e$
    Output: $Q_{pre}()$
1   Contain(q) finds the generator $o_q$;
2   $Q_{pre}() \leftarrow Q_{pre}() \cup o_q$;
3   if $e < dis_{net}(q, o_q)$ then
4       return o with true property in $Q_{pre}()$;
5   else
6       NVD_Expansion();
7   end
8   return o with true property in $Q_{pre}()$;
```

When the algorithm of VRS terminates, the structure of $Q_{pre}()$ must be as follow:

$$Q_{pre}() = (\ldots, (o_{out}, \ldots, False), (o_{in}, \ldots, True), \ldots)$$

The object with false value indicates the object out of the range (e.g. o_{out}), while the one with true value is the expected object in the range(e.g. o_{in}). The algorithm of VRS is shown below:

The NVD_Expansion() function includes step 2 and step 3, to get all objects of interest in the searching range e.

4.3 Moving Range Queries

Range query should not be confined to static range search, but also need to be feasible for continuous range queries. Our previous work, *Continuous Range Search* (CRS) as the only existed algorithm cannot avoid employing network segment division used by most continuous k nearest neighbors query processing methods (e.g. IE, UBA and eDAR), which performance depends on the number of intersections in the networks. Since the actual pattern of the road networks is normally extremely complex (numerous intersections), road segmentation creates a big problem in performance.

In this section, we propose a novel approach, *Voronoi Continuous Range* (VCR), a search method based on VRS, which does not require road segmentation.

4.3.1 Problem Definition

Firstly, we need to define continuous range search query.

Predefined trajectory continuous range search query is defined as retrieving all objects of interest on any point of a given trajectory in the networks. It is similar with continuous kNN query. The predefined path is a necessary element for this kind of query. Our previous work illustrates how CRS can solve the traditional continuous range search query properly.

S-D continuous range search query is defined as retrieving all objects of interest on any point during the moving of the query point from the start point (S) to the destination (D) in the networks. In this case, the moving trajectory of the query point is not predefined, which requires the method to have a better flexibility in the road network environment.

VCR is designed for both of the aforementioned queries.

4.3.2 Operational Principle of VCR

For Q_{pre} generated in VRS does not only involve the valid objects in the expected searching range, but also includes a few outside objects close to the searching range, the domain of the expected searching range can be located by o_{out}, o_{in}. o_{out} is the nearest object to the query point outside of the searching range, while o_{in} is the furthest one to the query point in the expected searching range.

Property 4.3.1. *Given the Q_{pre} return by* VRS*, then o of* $\min(|e - dis_{net}(o_{out}, q)|, |e - dis_{net}(o_{in}, q)|)$ *is the priority whose inside/outside property can be changed, when the query point is moving.*

In the continuous environment, when the query point is moving, it will cause a series of variations on the pattern of expected searching range respected to the moving distance of the query point during the movement of the query point. Some objects might move out, others could move in. To drop down the frequency of updating, we need to know where the result can be changed and that is the position the detection should be executed. According to Property 4.3.1, if the inside/outside property of o_{out}/o_{in} is unchanged, no object of interest could move in/out of the expected searching range, which provides us a method to monitor the changes during the movement of the query point.

Definition 4.3.1. Detection point *(dp) is a point used to predict the position where the range search result need to be updated on the trajectory of the moving query in the networks. It can be expressed by a network distance to the current position of query point,* $\min(|e - dis_{net}(o_{out}, q)|, |e - dis_{net}(o_{in}, q)|)$*,* o_{out} *has minimum* dis_{net} *to* q *with false value,* o_{in} *has maximum* dis_{net} *to* q *with true value.*

At every *dp*, VRS will be called to detect whether there is an object of interest that can be added/removed from Q_{pre}.

Because the velocity, the moving direction of the query point and the pattern of the road networks are ignored, the changes at the detection point cannot be guaranteed, and the number of *dp* may be slightly larger than split point used by other continuous approaches. But VCR focus on the moving distance of the query point that can be calculated and monitored easily, meaning that the continuous query processing is put in a more simple environment by removing most of the influencing factors, which provides VCR the capability to process both predefined trajectory and S-D continuous range search queries.

Definition 4.3.2. *A* **Critical point** *(cp) is the object of interest, which satisfies* $dis_{net}(q, o) = e$*,* $o \in \wp$*.*

Fig.4.2 shows that o_1 and o_2 are critical points. If there is a critical point in the Q_{pre}, then $\min(|e - dis_{net}(o_{out}, q)|, |e - dis_{net}(o_{in}, q)|) = 0$. So detection point cannot be found at this moment. To solve this problem, a new variable, *Precision Factor* (pf) need to be defined.

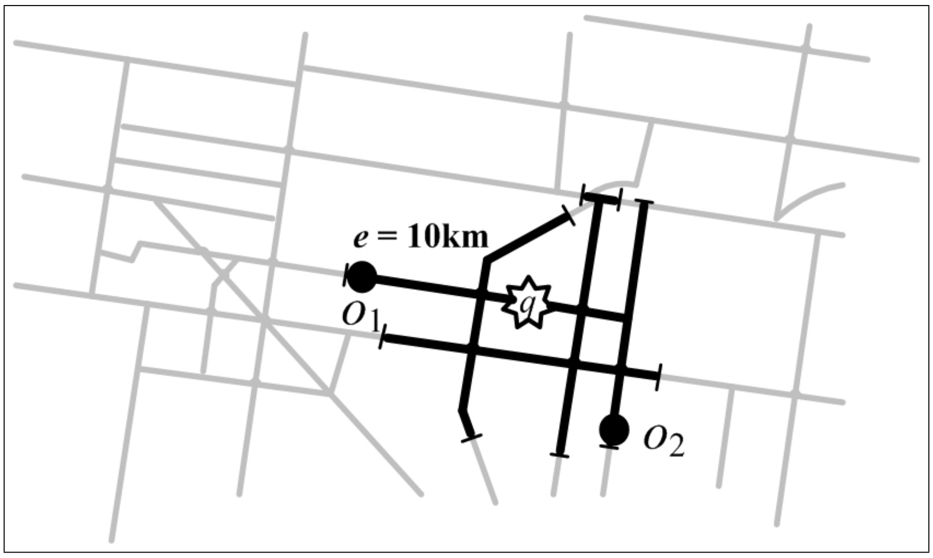


Figure 4.2: Critical points

Definition 4.3.3. Precision factor *(pf) is a minimum metric unit, which is determined by the precision of the network distance stored in the database.*

For example, if the network distance stored in database is 12.47km, then the pf = 0.01km. If $\exists cp \in Q_{pre}$, then dp is not calculated by $\min(|e - dis_{net}(o_{out}, q)|, |e - dis_{net}(o_{in}, q)|)$ any more. In this case, $dp = pf$, after updating Q_{pre} at the dp, cp will be cleared. To summarize, pf is used to detect the moving trend of a cp.

Using pf seems like a time consuming method, but actually, when we work over the processing which involve pf, we find its performance quite acceptable. Assuming $\min(|e - dis_{net}(o_{out}, q)|, |e - dis_{net}(o_{in}, q)|) =$ a and $\max(|e - dis_{net}(o_{out}, q)|, |e - dis_{net}(o_{in}, q)|) =$ A. In the worst case, the times of calculation to find the next object is $n = 1 + \log_2(A - a)/2pf$, and during these processing, the only data can be slightly changed is the network distance and the inside/outside property of the critical point. In other words, there is no other object of interest will move in/out to the range, except this critical point. Moreover, the critical point is rare in the real cases.

4.3.3 VCR Algorithm

The pseudo code of VCR is shown below and the main step of our proposed Voronoi-based continuous range will be illustrated as follow (see Algorithm 7):

Algorithm 7: VCR(S, D, e)

```
Input: start point:$S$, destination point:$D$, (pre-defined path:$SD$), searching
       range:$e$
Output: a set of $Q_{pre}$
VRS($S$, $e$);
$Total_{dis} \leftarrow dis_{net}(S, D)$;
repeat
    Find $o_{out}$ and $o_{in}$ from $Q_{pre}$;
    /* $o_{out}$ is the nearest object to $q$ out $e$; $o_{in}$ is the furthest
       object to $q$ in $e$                                              */
    $dp \leftarrow \min(|e - dis_{net}(o_{out}, q)|, |e - dis_{net}(o_{in}, q)|)$;
    if $dp \neq 0$ /*no object is a critical point*/ then
        VRS ($dp$, $e$);
        Update $Total_{dis} \leftarrow disnet(dp, D)$;
    else
        $dp = dp + pf$;
        Update the information of critical point at $dp$;
        Update $Total_{dis} \leftarrow disnet(dp, D)$;
    end
until $Total_{dis} = 0$;
```

Step 1 For a given start point S and end point D or a path SD, we define the value of $Total_{dis} = dis_{net}(S, D)$, which will be used to evaluate the position of query point to the destination.

Step 2 Find an object of interest, o_{out}, with the maximum network distance in the range, while find one, o_{in}, with the minimum network distance out of the range from Q_{pre} will be used to calculate the detection point where the result of the range search can be changed. If there is no O in the range or out of the range, then the corresponding network distance recorded as $dis_{net} = 0$.

Step 3 Find a detection point using $dp = \min(|e - dis_{net}(o_{out}, q)|, |e - dis_{net}(o_{in}, q)|)$.

Step 4 At the detection point, test any object of interest is a critical point to determine whether to employ pf for the next detection point or not. Update the corresponding information at the detection point.

Table 4.1: Q_{pre} at start point

False objects in Q_{pre}		True objects in Q_{pre}	
o_{out}	Properties	o_{in}	Properties
o_{11}	(15.23, False)	$\boldsymbol{o_{12}}$	**(11.64, True)**
o_9	(14.54, False)	o_5	(9.63, True)
o_6	(14.20, False)	o_2	(8.37, True)
o_{14}	(13.83, False)	o_8	(8.19, True)
o_{13}	(13.53, False)	o_7	(7.85, True)
o_{15}	(13.20, False)	o_1	(5.06, True)
o_4	(12.94, False)		
$\boldsymbol{o_3}$	**(12.82, False)**		

Step 5 Repeat steps 3 and 4 until the query point reach to the end point, $Total_{dis} = 0$.

Example 4.3.1. *The initial search range is 12km and* $Total_{dis} = dis_{net}(S, D) = 18.8$*, then at point* S*, the data in the* Q_{pre}*() is shown in Table 2. Learning through observation and comparison, we get the first detection point* $dp_1 = \min(|12-11.64|, |12-12.82|) = \min(0.36, 0.82) = 0.36$*. It means when the query point* q *moving 360 meters from the start point, we will do a range search using VRS to update the information of* Q_{pre}*.*

At the detection point dp_1, the $dis_{net}()$ of o_{12} and $Total_{dis}$ will be updated, if o_{12} is a critical point, $dis_{net}(o_{12}, q) = e = 12$km and $Total_{dis} \neq 0$ (not arrive at destination point), then pf which equals to 0.01 will intervene to test o_{12} will move into the range deeply or move out from the expected searching range in the next moment. So $dp_2 = pf = 0.01$, where the network distance and inside/outside property of o_{12} will be updated. On the other hand, if the $dis_{net}(o_{12}, q) \neq e$ at dp_1, then we will go through Q_{pre} to find the next detection point by estimating the new value of $\min(|e - dis_{net}(o_{out}, q)|, |e - dis_{net}(o_{in}, q)|)$.

If all information in Q_{pre} has been updated, we repeat the above procedure, until the query reaches the destination, $Total_{dis} = 0$ when the VCR will terminate.

Comparing to processing the pre-defined path which is just in a single NVP, processing the path through multiple NVPs will rise a massive updating. So if the pre-defined path is within a NVP, then the performance of VCR will be improved remarkably.

4.4 Range Query on Moving Objects

In this section we proposed a technology for static range search queries over moving objects in road networks.

Monitoring moving objects should achieve equilibrium between the frequency of updating and the accuracy of position of moving objects. If updates are very often, which has high cost, the error in location of the moving objects is kept very small. On the contrary, if we want to minimize the updating cost, then the error becomes larger.

To optimize the information loading, the road networks and the moving objects should be stored in two different layers. The static layer used to store the information of the road networks, including ID, end points and the length for each segment. Since the information of this layer is very rarely changed. While the variable layer stores the information for all moving objects, as their position, the moving speed and the moving direction change frequently in most cases. Then we discuss how the updating cost of the moving objects can be minimized.

On the other hand, to insure the correctness and the continuity of the range search result, we also illustrate how the updating cost of the range search result can be optimized. These two problems are discussed respectively in the remain of this section.

4.4.1 Moving Object Update

It is difficult to use VCR proposed in section 4.3, as unlike the query point, the movement of objects of interest is unpredictable. Therefore, we need to design a new technology to monitor the movement of these spatial objects.

Table 4.2: Data structure of a moving object

Column name	**Example**	**Comment**
ID	m_1	the ID of a moving object
MOTION	$d_{future-m_1}$	the distance of a moving object to its position at t_s between time interval t_st_e, $d_{future-m_1}$=$s_{m_1}(t-t_s)$
SPEED	s_{m_1}	the moving speed of a moving object
DIRECTION	n_1n_2	the segment which the object moves on, n_1n_2 indicate that the object moves from n_1 to n_2
START TIME	t_s	time interval
END TIME	t_e	

To minimize the updating cost, we store the motion as a function of time for each moving object instead of its position. Since the updating frequency of the motion is much lower than the position. Table 4.2 shows the detail of a moving object stored in database. In each time interval, the motion, the moving speed and the moving direction are invariable. In another word, when any change occur among the motion, the moving speed and the moving direction, this time interval is terminated and start a new one. With motion of the moving object, the future position $d_{future-m_1}$ after t_s can be calculated as:

$$d_{future-m} = s_m(t - t_s)$$

where t is the time to check the position of the moving object m. In Fig. 4.3, the segment n_1n_2 can be represented as a linear function:

$$y = ax + b,\ x \in (x_{n_1}, x_{n_2})$$

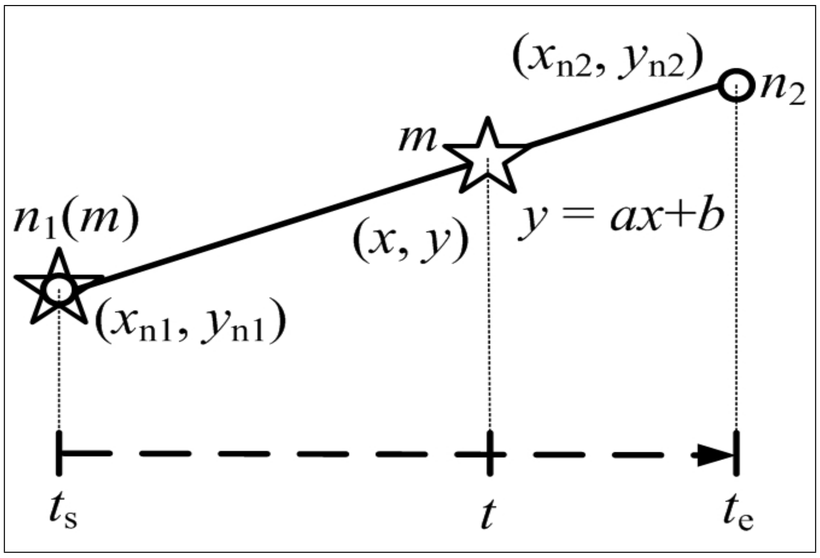


Figure 4.3: The future position of moving object m calculation

Since the road network is static and the positions of road network intersections n_1, n_2 are known. Then we have

$$\left.\begin{array}{l} y_{n_1} = a * x_{n_1} + b \\ y_{n_2} = a * x_{n_2} + b \end{array}\right\} \Rightarrow \left\{\begin{array}{l} a = \dfrac{y_{n_1} - y_{n_2}}{x_{n_1} - x_{n_2}} \\ b = \dfrac{x_{n_1}y_{n_2} - x_{n_2}y_{n_1}}{x_{n_1} - x_{n_2}} \end{array}\right.$$

The future position of object m can also be represented according to Euclidean distance as:

$$d_{future-m} = \sqrt{(x - x_{n_1})^2 + (y - y_{n_1})^2}$$

Then we have:

$$\left.\begin{array}{l} \sqrt{(x - x_{n_1})^2 + (y - y_{n_1})^2} = s_m(t - t_s) \\ y = \dfrac{y_{n_1} - y_{n_2}}{x_{n_1} - x_{n_2}} * x + \dfrac{x_{n_1}y_{n_2} - x_{n_2}y_{n_1}}{x_{n_1} - x_{n_2}} \\ x \in (x_{n_1}, x_{n_2}) \end{array}\right\} \Rightarrow \left\{\begin{array}{l} x = ... \\ y = ... \end{array}\right.$$

Then at any moment t within time interval t_st_e, the position of the moving object m can be predicted using its motion function instead of keep updating its position, by which, the updating cost improved dramatically (assuming the segments are straight lines). The motion function only need to be updated when a new time interval initialized, which is much less frequently than updating the position.

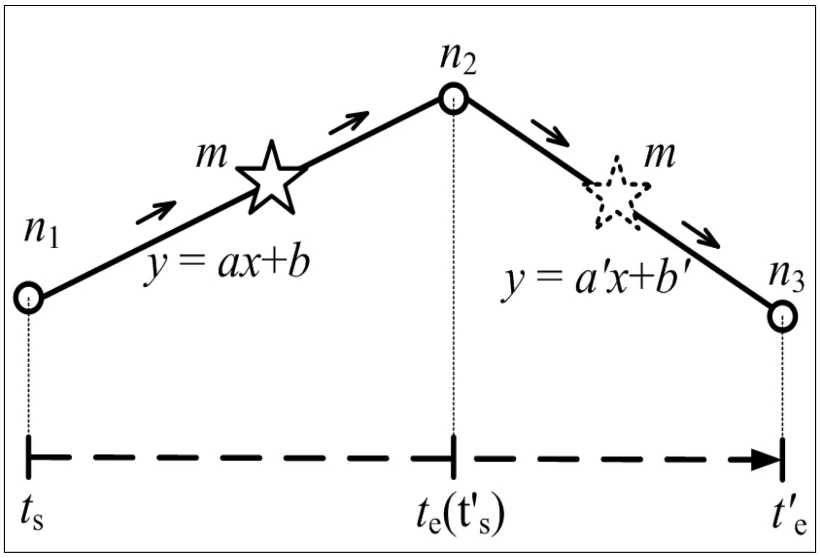


Figure 4.4: Updating the moving direction of m

Table 4.3: Updating the moving direction of m

ID	MOTION	SPEED	DIRECTION	ST	ET
m	$d_{future-m}$	s_m	n_1n_2	t_s	t_e
m	$d_{future-m}$	s_m	n_2n_3	t'_s	t'_e

We use the segment which the object moves on to represent the moving direction of this moving object. So n_1n_2 indicate that the object moves from n_1 to n_2 and $n_1n_2 \neq n_2n_1$. Table 4.3 shows the moving direction update of object m in Fig. 4.4

4.4.2 Range Search Query Update

In this section, we introduce the update of range search queries in road networks.

Grant Priority: Processing range search queries over moving objects should concern with the moving speed as well as the relative position with the searching range. Intuitively, high speed moving objects need to be processed earlier than low speed moving objects, but we also need to concern with the relative position with the searching range. So we define t_m as:

$$t_m = \frac{|(dis(m, q) - e)|}{s_m}$$

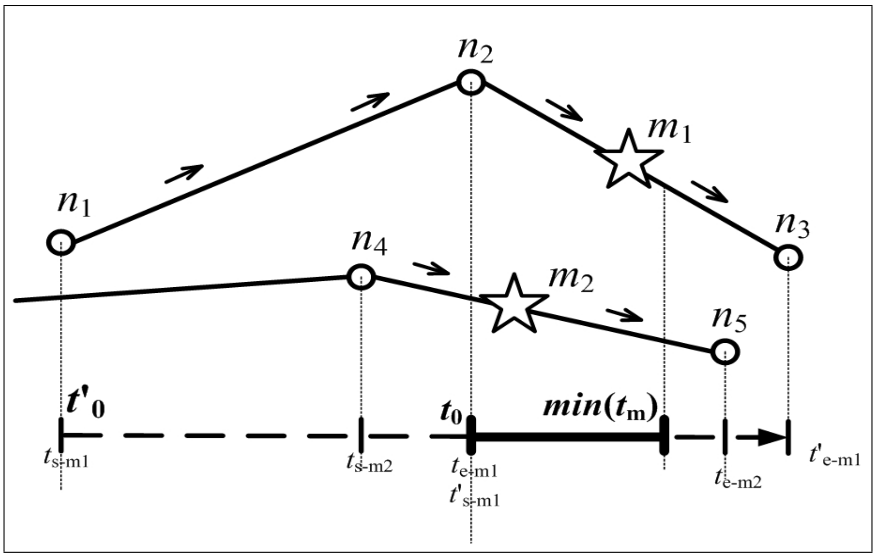


Figure 4.5: t_0 and t_m

Where $dis(m, q)$, which can be calculated easily by Dijkstra's algorithm or Voronoi diagram, is the network distance from moving object m to q at t_0 (traveling time is similar). Then the moving object m with $\min(t_m)$ need to be granted higher priority, since its possibility of effecting the searching result is the highest. See Fig. 4.5 as an example. Assuming the information of all moving objects are collected at t_0 and $t_0\min(t_m)$ is the only time interval between t_0 and $\min(t_m)$ for all moving objects, otherwise t_0 will be replaced by the last start time. Whereby we can predict the position of all moving objects according to their motion functions. As Fig. 4.5 illustrates, the positions of both m_1 and m_2 are predictable between t_0 and $\min(t_m)$. To simplify, we assume the moving speed of any moving object is invariable for each segment. Either m_1 or m_2 changing its motion in between $t_0\min(t_m)$ will issue a new start time t_s, then t_s will become the new $\min(t_m)$, whereby the motions of all objects are stable in the time interval $t_0\min(t_m)$.

Position Detection: For m with $\min(t_m)$, we detect its relative position with searching range according to $dis(m$, q) - e.

$$If\ dis(m,q) - e < 0 \Rightarrow m \in e\ at\ t_0$$
$$If\ dis(m,q) - e > 0 \Rightarrow m \notin e\ at\ t_0$$

This status will cooperate with moving trend detection to check whether we need to update range search result at $min(t_m)$.

Moving Trend Detection: When a moving object is traveling on a segment, it moves either away from q (denote as $\uparrow q$)or towards q (denote as $\downarrow q$). When the relative position of m with range is retrieved at t_0, if the moving trend of m can be detected within $t_0\min(t_m)$, then we can predict whether the movement of object m will effect the range search result or not at $\min(t_m)$.

The moving trend for a moving object can be detected by estimating the distance from the two ends of the segment the object moving on to the query point. Assuming the moving direction of m is n_1n_2 (no intersection on n_1n_2), the shortest distances $dis(n_1,\ q)$ and $dis(n_2,\ q)$ can be calculated easily. Then we estimate the moving trend of m as:

$$(dis(n_1,q) > dis(n_2,q)) \cap \begin{cases} (|dis(n_1,q) - dis(n_2,q)| = dis(n_1,n_2)) \Rightarrow m \downarrow q & ① \\ (|dis(n_1,q) - dis(n_2,q)| \neq dis(n_1,n_2)) \Rightarrow m \uparrow\downarrow q & ② \end{cases}$$

$$(dis(n_1,q) < dis(n_2,q)) \cap \begin{cases} (|dis(n_1,q) - dis(n_2,q)| = dis(n_1,n_2)) \Rightarrow m \uparrow q & ③ \\ (|dis(n_1,q) - dis(n_2,q)| \neq dis(n_1,n_2)) \Rightarrow m \uparrow\downarrow q & ④ \end{cases}$$

In condition ① and ③, the moving trends are unidirectional, while the moving trends in ② and ④ are bidirectional. So we also want to know the transition point on the segment n_1n_2 for condition ② and ④. We denote such a border point as bp that satisfies:

$$dis(n_1,q) + dis(n_1,bp) = dis(n_2,q) + dis(n_2,bp)$$

Suppose $dis(n_1,\ bp)$=z, $dis(n_2,\ bp)$=z', refers to Fig. 4.6 then we have:

$$\left.\begin{array}{l} z + z' = dis(n_1,n_2) \\ dis(n_1,q) + z = dis(n_2,q) + z' \end{array}\right\} \Rightarrow \begin{cases} z = \frac{dis(n_1,n_2) + dis(n_2,q) - dis(n_1,q)}{2} \\ z' = \frac{dis(n_1,n_2) - dis(n_2,q) + dis(n_1,q)}{2} \end{cases}$$

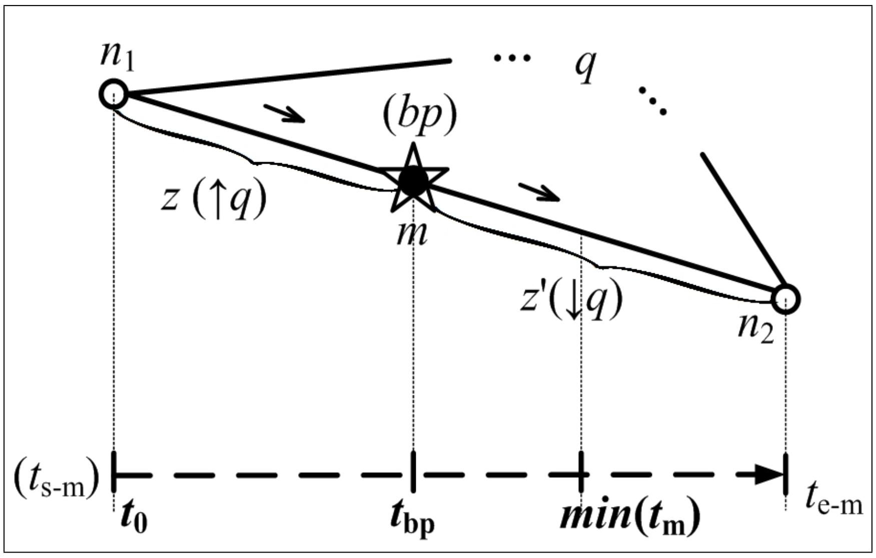


Figure 4.6: Moving trend detection of m

So if t_{bp} does not follow in between time interval $t_0\min(t_m)$, the moving trends of m are unidirectional for all ①②③④. Otherwise the moving trend will be divided into two sections in condition ②④ at t_{bp}, where need to update $\min(t_m)$ to t_{bp} and compare $dis(bp,\ q)$ with e to estimate whether we need to update the range search result.

By position and moving trend detection, we can decide whether the range search result need to be updated at $\min(t_m)$. After get the moving object m with minimum t_m at t_0, we estimate the relative position of m with searching range and detect the moving trend for m, which can be concluded as:

$$\left.\begin{array}{l}(m \notin e) \cap (m \downarrow q)\\ (m \in e) \cap (m \uparrow q)\end{array}\right\} \Rightarrow m\ can\ effect\ the\ result$$

$$\left.\begin{array}{l}(m \notin e) \cap (m \uparrow q)\\ (m \in e) \cap (m \downarrow q)\end{array}\right\} \Rightarrow m\ cannot\ effect\ the\ result$$

Algorithm 8 shows the pseudo code of range search over moving object.

Algorithm 8: *RangeMonitoring*(q, e)

```
Input: query point:$q$, searching range:$e$
Output: $Q_e$
1  $VRS(q, e)$ at $t_0$ ;
2  for moving objects $m(s)$ do
3      $t_m = \frac{|(dis(m,q) - e)|}{s_m}$;
4  end
5  Get $\min(t_m)$;
6  Find the minimum time interval $t_0\min(t_m)$;
   /* $t_0\min(t_m$ is the only interval in between */
7  Detect the relative position of $m$ with $e$;
8  Detect the moving trend of $m$;
9  while $m$ with $\min(t_m)$ does not effect the range searching result do
10     Move to next $m'_{t_{min}}$;
11 end
12 $VRS(q, e)$ at $t_{min}$
```

4.5 Performance Evaluation

We carried out several experiments to evaluate the performance of VCR. The data sets provided by *Whereis*® (http://www.whereis.com.au) [WHE], represents networks of thousands of links and nodes of the road system in Melbourne, Australia. We compare our proposed VCR with the previous work CRS. We performed 20 sets of tests for each experiment to calculated the tested data on average, and the range size was varied from 1km to 100km. All evaluated parameters are explained in Table 4.5.1.

We generated random road network segments with a set of objects to evaluate performance of RangeMonitoring in terms of the CPU time and the memory size. The data of all the experiments shown below are collected by averaging the results for 1000 random queries on each experiment to reduce the inaccuracy.

4.5.1 Experimental Result of VCR

The most frequency concept in VCR is *dp* that tries to detect the changes during the movement of query point. But sometimes the changes do not happen at *dp*, so the accuracy of *dp* is an important parameter to estimate the performance of VCR.

Table 4.4: Performance evaluation parameters

Parameter	**Description**
Segments	A subdivision of predefined path
sp	A split point in CRS, the result has to be changed at split point
dp	A detection point invoking VRS in VCR, the result can be changed at detection point

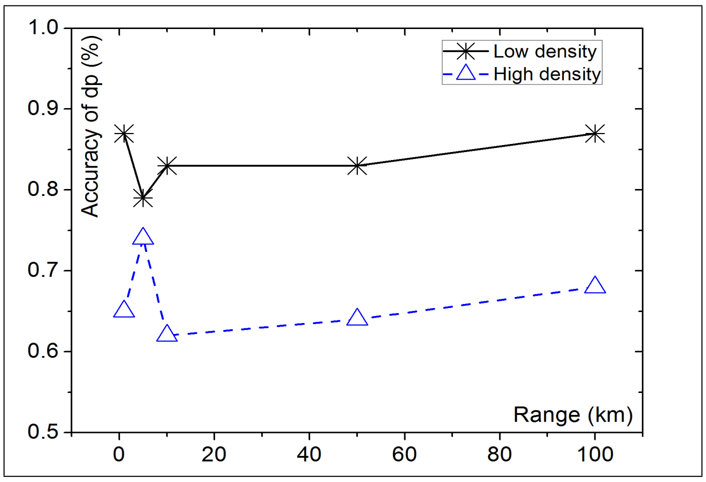


Figure 4.7: Accuracy of *dp*

Fig.4.7 shows the rate of the accuracy of *dp* in both high-density and low-density environments. Intuitively, rate of the accuracy of *dp* always remains at a good level when the searching range e increases. And the high-density objects will have a lower percentage than the low-density objects. So the density is the main factor that will affect the accuracy of *dp*.

Because of a very limited existing work in continuous range using NVD, we only compare VCR and CRS, by giving some experiment results in different scenarios, including, change the density of the object and alter the searching range to different location and resize it. These experiments focus on: number of comparison of dis_{net} with e, number of false hits and the proportion of the expansion area.

Table 4.5.1 shows the comparison results between VCR and CRS. It is obvious that VCR does not divide the path into segments while CRS has to implement the path segmentation to avoid changes when moving on the segment. If there is an intersection in the middle of the segment, we cannot guarantee the changes of

dis_{net} are unidirectional. No segmentation gives VCR a better performance and applicability.

On the other hand, the number of *dp* in VCR is larger than the split node in CRS most of the time. Unlike the split node, detection points do not ensure that there would be a change in that point. If the accuracy of split node is 100%, then *dp* keeps the accuracy in a lower level. But the number of *dp* is very similar with the corresponding value of split node, for VCR does not focus on the details of the network.

According to the result in Table 4.5.1, VCR is not suitable for a wide range and high density environment and when the value of range *e* approximate to the path length, CRS has a strong competitiveness.

Table 4.5: VCR vs. CRS

VCR vs. CRS												
	Low Density environment 10 objects/10sq.km											
	e = 1km				e = 10km				e = 100km			
	VCR	CRS	VCR	CRS	VCR	CRS	VCR	CRS	VCR	CRS	VCR	CRS
Segments	1	532	1	4028	1	532	1	4028	1	532	1	4028
sp(**CRS**)	–	33	–	390	–	465	–	4379	–	4308	–	38745
dp(**VCR**)	67	–	382	–	826	–	3659	–	7948	–	78976	–
	High Density environment 100 objects/10sq.km											
	e = 1km				e = 10km				e = 100km			
	VCR	CRS	VCR	CRS	VCR	CRS	VCR	CRS	VCR	CRS	VCR	CRS
Segments	1	532	1	4028	1	532	1	4028	1	532	1	4028
sp(**CRS**)	–	325	–	3885	–	4530	–	448764	–	46971	–	400653
dp(**VCR**)	638	–	4007	–	8487	–	3779	–	78754	–	764399	–

4.5.2 Experimental Result of RangeMonitoring

Since *Range Monitoring* manages the moving objects based on theirs priority to effect the range search result and only check the relative position and moving trend for the moving objects with highest priority, whereby it outperforms most existing range search queries approaches over moving objects. Both CPU time and memory size of *Range Monitoring* have direct proportion with the density of moving objects in road networks (refer to Fig. 4.8 and Fig. 4.9).

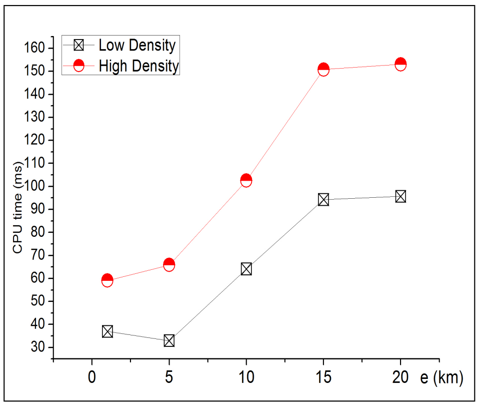


Figure 4.8: CPU time of range monitoring

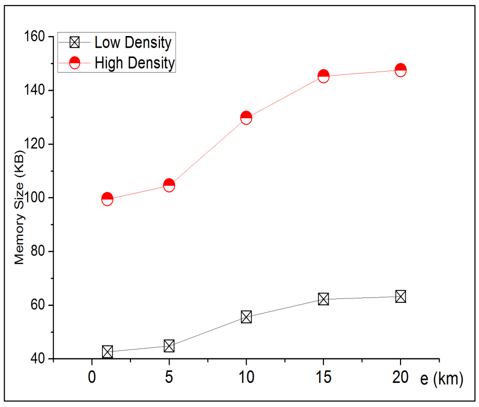


Figure 4.9: Memory size of range monitoring

4.6 Summary

In order to offer a better solution, we use Network Voronoi Diagram as the basis for our method, *Voronoi Continuous Range* (VCR), to process moving range queries. We utilize some new properties of NVD to get a result set Q_{pre} which as the preliminary data result for VCR. Differing from other continuous algorithm, VCR only focuses on the moving distance of the query point instead of the details of networks. When the result of Q_{pre} could be changed, we will implement range search at detection point to update the results. Our experiments show that VCR outperforms its competitors in most scenarios.

Moreover, we also proposed a technology to monitoring moving objects in road networks for range search queries, called *Range Monitoring*. Its manages the moving objects based on their priority to effect the range search result and only check the relative position and moving trend for the moving objects with highest priority, whereby it outperforms most existing range search queries approaches over moving objects.

Part II

Processing Region-Expected Queries

Chapter 5

On Finding k Nearest Neighbor-Regions

In this chapter, we propose a region-expected query to find a nearest region for a small cluster of objects in Cartesian space, and present two algorithms it. Our algorithms based on different geometric theories. We study their advantages and disadvantages according to the rigorous theoretical analysis and extensively experiments.

5.1 Overview

As described in Chapter 2, either a traditional range query or a kNN query can retrieve a set of point objects, also known as objects of interest (OOI), within a specific distance from the query point (Euclidean or network) or find several OOIs closer to the query point than any other OOIs. Many works consider range search and kNN queries in the context of some dynamic circumstances, such as continuous queries [TPS02, XZT$^+$11a] and moving OOIs [TP03]; or they change the threshold and add some constraints, such as reverse kNN [TPL04], aggregate kNN [PSTM04], range kNN [HL06], constrained kNN [FSAA] and constrained range [XZT$^+$11b]; or they solve similar problems in a different space, such as indoor space [YS10], road networks [PZMT03], weighted region [LGYL11] and land surface [DZS$^+$06].

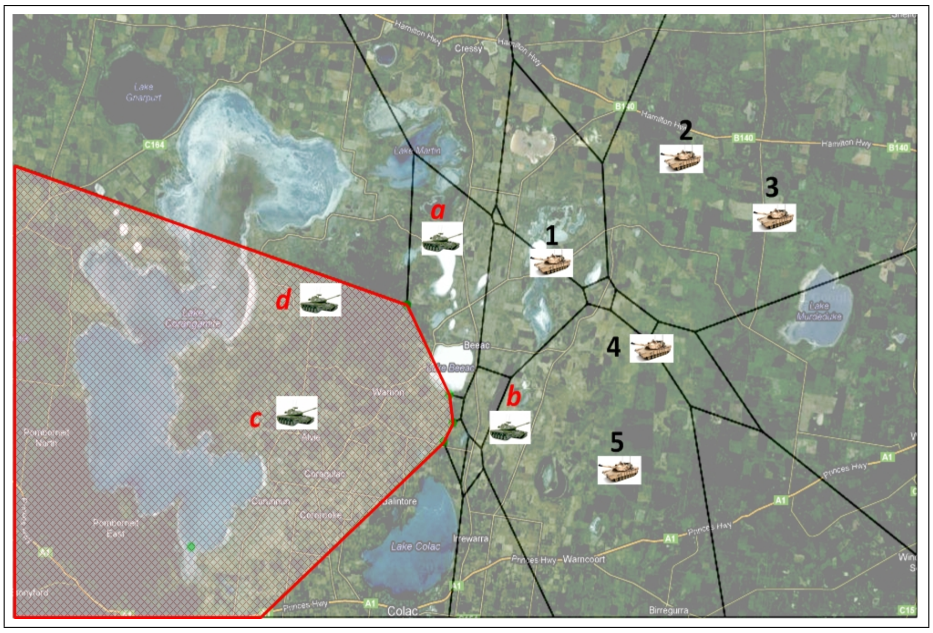


Figure 5.1: An example of kNN region query

However, they are still limited to retrieving point-objects, and the input is an independent object (a query point or a range centered at the query point).

In this chapter, we propose a novel spatial query, named k nearest neighbor (kNN) region, to retrieve a region including all points that consider the specific k objects in the entire set as the k nearest neighbors. The kNN region is very important in many practical applications:

To specify k branches of a company, we can obtain an area dominated by these k branches rather than the competitors. So it means the company is likely to retain the customers in this area.

A group wants to find an area closer to all the group members than their competitors.

One side wants to find an area to build a military base (a refuge) considering all friendly armies as nearest neighbors in the battle field, without exposing the exact location.

Fig. 5.1 illustrates the last case above as an example of kNN region query, where the shadowed area is the 4NN region of a, b, c, d. We can see that if the military base or a refuge was built in this 4NN region, it can give best support for objects

$a \sim d$ meanwhile it is far away from objects $1 \sim 5$, therefor it can be protected most sufficiently by a d. Moreover, the exact location of the military base or a refuge is not exposed, thus avoiding enemy's long range strike.

kNN range is a very challenging problem because of two main issues. One is that the information of the kNN region is not pre-stored in the spatial database, unlike point objects, which means the region cannot be retrieved from the database directly, but is identified according to the location and the distribution of the known objects. All the existing works and indexing approaches to spatial queries are incapable of processing kNN range queries. The other issue concerns the storage of the query result. The form of a kNN range is a convex polygon. So it is important to ensure the correctness and consistency of the vertices of the polygon in order. Intuitively, if the order of the vertices of a polygon is incorrect, then the formed polygon returned to the user is also incorrect.

This chapter concentrates on kNN region query processing for static objects in Cartesian space, and illustrates two algorithms, kth-order Voronoi diagram based algorithm, $\mathcal{VD}_k$-kR and Delaunay triangulation based algorithm, $\mathcal{DT}$-kR. $\mathcal{VD}_k$-kR which is designed to retrieve all the existing candidates for kNN regions of any k objects in the entire set, and then identify the kNN region of the specific k objects. $\mathcal{VD}_k$-kR performs well when the server processing a bunch of kNN region queries on the same data sets with specific k. $\mathcal{DT}$-kR first checks the relative position of the specific k objects in the entire set, which can greatly improve the performance of processing kNN region queries in most of the instances where the kNN region does not exist. When a number of objects in the entire set is massive, $\mathcal{DT}$-kR requires smaller storage and I/O access than $\mathcal{VD}_k$-kR to identify the kNN region by using several geometric properties of the Delaunay triangulation and convex hull.

The remainder of this chapter is organized as follows: Section 5.2 illustrates several geometric concepts used in our proposed algorithms. Section 5.3 illustrates several geometric concepts used in our proposed algorithms. Sections 5.4 and 5.5

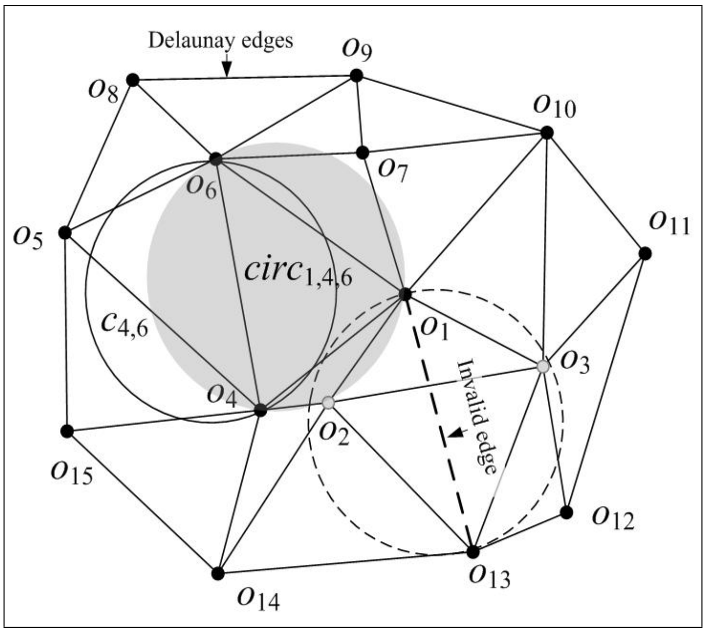


Figure 5.2: An example of Delaunay triangulation in $\mathbb{R}^2$

describe the details of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR respectively. Section 5.6 evaluates the performance of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR. Section 5.7 summarizes this chapter.

5.2 Preliminary

Our proposed algorithms are based on the important properties of a series of relative computational geometry concepts, namely, *Delaunay Triangulation*, *Convex Hull*, *Voronoi Diagram* and *High-Order Voronoi Diagram.* So in this section, we present the definitions and properties of these concepts.

5.2.1 Delaunay Triangulation and Convex Hull

In two-dimensional space $\mathbb{R}^2$, *Delaunay Triangulation* $\mathcal{DT}$ is a set of shortest connections named Delaunay edges on a set of discrete point objects U=$\{o_1, o_2,..., o_{n-1}, o_n\}$, which satisfies for each edge, o_io_j, we can find a circle containing the two ends of this edge o_i and o_j on its boundary, but not containing any other object of U. This is also called the *empty circle*[1] property of Delaunay triangulation. The Delaunay triangulation of a set U is unique.

[1]The empty circle is not unique

Definition 5.2.1. *Given a finite set of point objects* U=$\{o_1, o_2,..., o_{n-1}, o_n\}$ *in* $\mathbb{R}^2$*, the* Delaunay Triangulation *of* U *is a triangualtion* $\mathcal{DT}(U)$ *such that for each edge* o_io_j*,* $\{\exists\ c_{ij}|o_i,o_j \in c_{ij}\}$*, and (*$\forall o_m|o_m \notin c_{ij}$*,* $o_m \in U$ *and* $m \neq i \neq j\}$*).*

Fig.5.2 shows an example of $\mathcal{DT}$ in a two-dimensional space with an invalid edge. Since the empty circle for edge o_1o_{13} does not exist, o_1o_{13} cannot be a Delaunay edge. The empty circle can easily be found for any Delaunay edge, such as $c_{4,6}$ for o_4o_6. If two objects are the two ends of a Delaunay edge, these two objects are called neighbor objects of each other.

Another important property of Delaunay triangulation is the *empty circumcircle property*[1] whereby for any triangle in $\mathcal{DT}(U)$, its circumcircle does not contain any other object of U, such as $circ_{1,4,6}$ in Fig.5.2. The empty circumcircle is also called Delaunay circle.

Property 5.2.1. *Given a set of point objects* U=$\{o_1, o_2,..., o_{n-1}, o_n\}$ *in* $\mathbb{R}^2$*, a triangulation* $\mathcal{T}(U)$ *is a* $\mathcal{DT}(U)$ *iff the circumcircle of any triangle of* $\mathcal{T}(U)$ *does not contain a object of* U *in its interior.*

Convex hull, also called *convex envelope* is another important concept in computational geometry. It finds the smallest polygon containing the given set of point objects, U, by connecting some objects in U. The convex hull also has the smallest area and perimeter of all polygons containing the set U.

Definition 5.2.2. *Given a finite set of point objects* U=$\{o_1, o_2,..., o_{n-1}, o_n\}$ *in* $\mathbb{R}^2$*, the* convex hull *is the smallest Polygon* $Conv(U)$*, the vertices* VT *of which has,* $VT \subseteq U$*, containing* U*.*

One of the important properties of convex hull is that, for any point p in a two-dimensional space, we can always find a vertex in $VT(U)$ which is further from p than are any other points in U.

Property 5.2.2. *Given a set of objects* U*, then* $\forall p \in \mathbb{R}_2$*,* $\{\exists o_i|d(o_i,\ p) > d(o_j,\ p)\}$ *where* $o_i \in VT(U)$*,* $o_j \in U\text{-}VT(U)$

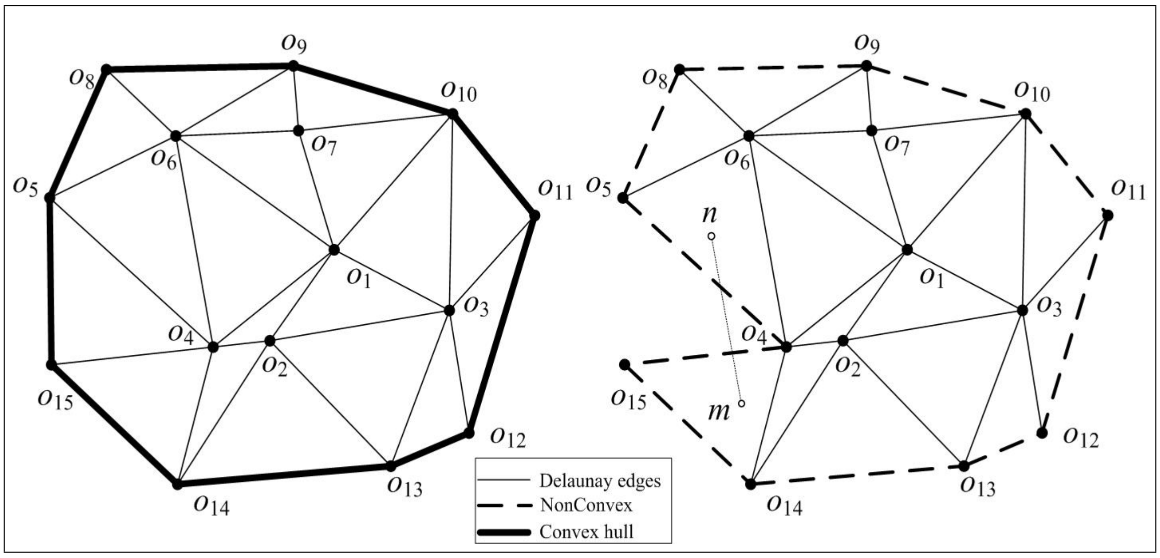


Figure 5.3: The comparison of the convex hull and the non-convex of Delaunay triangulation in Fig.5.2

Fig. 5.3 compares the convex hull with a non-convex generated by points in Fig. 5.2 and it shows another important property of a convex hull: if a polygon is a convex hull, then for any two points in this polygon, their connection is also contained by this polygon. In other words, all of the triangles of $\mathcal{DT}(U)$ are dominated by $Conv(U)$.

Property 5.2.3. *If a polygon $\mathcal{P}$ is a convex hull, then for any two points, $\forall n$, $\forall m \in \mathcal{P}$, the whole line segment also has $nm \in \mathcal{P}$.*

5.2.2 Voronoi Diagram and High-Order Voronoi Diagram

Voronoi Diagram is a special decomposition of a metric space according to the relative distance of a given set U of points [OBSC00b]. It is constructed by a set of *Voronoi polygons* where each polygon is associated with a given object o_i, denoted as $\mathcal{V}(o_i)$. Each edge of a Voronoi polygon is a segment of a perpendicular bisector of a pair of objects (o_i, o_j) and the half space including all points closer to o_i than o_j is denoted as $h(o_i, o_j)$. So $\mathcal{V}(o_i)$ can also be seen as the intersection of all of the half spaces closer to o_i than to any other object.

$$\mathcal{V}(o_i) = \bigcap_{i \neq j} h(o_i, o_j), (o_i, o_j \in U)$$

[1]The empty circumcircle is unique

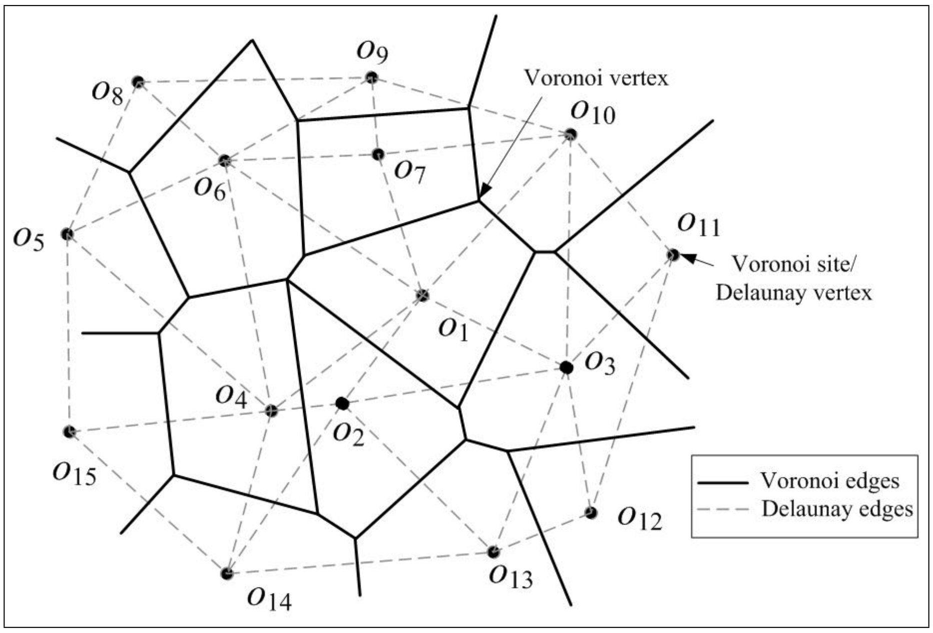


Figure 5.4: The comparison of Voronoi diagram and Delaunay triangulation in Fig.5.2

The definition of a Voronoi diagram can be stated as:

Definition 5.2.3. *Given a set of discrete objects* $U = (o_1, o_2, \ldots, o_{n-1}, o_n)$ *in a two-dimensional Cartesian space,* $\mathbb{R}^2$*, the* Voronoi Diagram *of* U *is:*

$$\mathcal{VD}(U) = \bigcup_{i=1}^{n} \mathcal{V}(o_i), (o_i \in U)$$

The Voronoi diagram of U, $\mathcal{VD}(U)$ is the dual structure of $\mathcal{DT}(U)$. Fig. 5.4 illustrates the relationship between the Voronoi diagram and the Delaunay Triangulation of all the objects shown in Fig. 5.2. Vertices of the Voronoi diagram are called Voronoi points and the edges are called Voronoi edges. An object, o, is called a Voronoi site or Delaunay vertex. If two objects have a shared edge, then these two objects are called neighbor objects of each other. Here is an important property of neighbor objects.

Property 5.2.4. *The nearest object of* o_i *has to be among its neighbor objects.*

A *High-Order Voronoi Diagram* is a variation of the ordinary Voronoi diagram. It also subdivides the metric space into several polygons. But the polygons of the high-order Voronoi diagram associate with a set of objects $S \subseteq U$ rather than a single

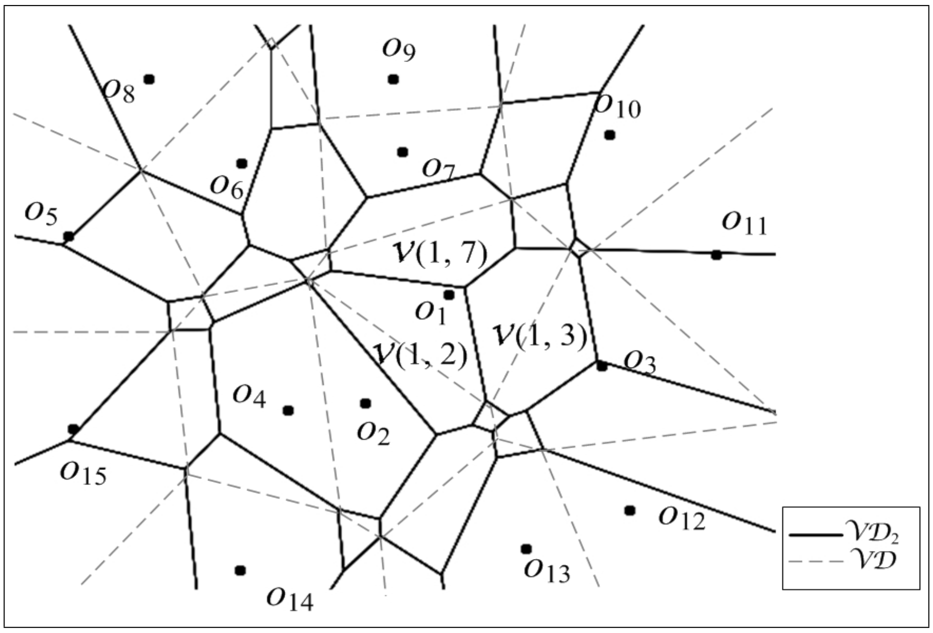


Figure 5.5: The comparison between the 2-order VD and the Voronoi diagram displayed in Fig.5.4

object in the ordinary Voronoi diagram, denoted as $\mathcal{V}(S)$. The "order" means the cardinality of S, denoted as $|S|$. If $|S|$=k, then this high-order Voronoi diagram is also called kth-order Voronoi diagram. When k=1, the kth-order Voronoi diagram degenerates into an ordinary Voronoi diagram. Since the points in $\mathcal{V}(S)$ are closer to all the objects in S than to any object in U - S, then the polygon of a high-order Voronoi diagram can be represented as:

$$\mathcal{V}(S) = \bigcap_{o_i \in S, o_j \in U-S} h(o_i, o_j),\ (S \subseteq U)$$

where U is the entire set of objects. Hence a high-order Voronoi diagram can be defined as:

Definition 5.2.4. *Given a set of discrete objects* U = (o_1, o_2,..., o_{n-1}, o_n) *in a two-dimensional Cartesian space,* $\mathbb{R}^2$, $\mathcal{P}(U)$ *is the power set including all the subsets of* U, $\mathcal{P}_k(U) \subset \mathcal{P}(U)$ *and* $\mathcal{P}_k(U)$ *contains all the subsets whose cardinality equals* k, $\mathcal{P}_k(U)$=(S_1, S_2, ..., $S_{c_n^k}$), *where* $|S_i|$=k *and* $|\mathcal{P}_k(U)|$= C_n^k, *the* kth-Order Voronoi Diagram *of* U *is:*

$$\mathcal{VD}_k(U) = \bigcup_{i=1}^{c_n^k} \mathcal{V}(S_i)$$

In this definition, we can see that the number of the subsets, each of which includes k objects, is C_n^k for a set U having n objects. The kth-order Voronoi diagram is a union set of all the Voronoi polygons of these subsets, S_i.

Property 5.2.5. *Any two adjacent Voronoi polygons $\mathcal{V}(S_i)$ and $\mathcal{V}(S_j)$, have the same k-1 NN.*

The details and the proof of this property are discussed in [Lee82]. The property means that if one goes from a Voronoi polygon to any adjacent polygon, only the kth nearest neighbor is changed.

Fig. 5.5 compares the $\mathcal{VD}$ of o_1 to o_{15} with their $\mathcal{VD}_2$. In $\mathcal{VD}_2$, each Voronoi polygon is dominated by two objects, and it is constructed by a set of perpendicular bisectors associated with these two objects. e.g. $\mathcal{V}(1,\ 7)$[1], in which all points are closer to o_1 and o_7 than to any other object.

5.3 Problem and Query Definition

For simplicity, in this chapter, we discuss only the kNN region query in a two-dimensional Cartesian space, $\mathbb{R}^2$, and use Euclidean distance as the metric. For any two points p and q, their Euclidean distance can be expressed as: $d(p,q)= \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$, where $(p_x,\ p_y)$, $(q_x,\ q_y)$ are Cartesian coordinates of p and q. But our proposed algorithms are not limited to this two-dimensional space; they can be simply modified to suit any higher metric space.

In $\mathbb{R}^2$, a kNN region of a set of objects of interest (OOIs), S and $|S|=k$, is the largest area containing all points considering the objects in S as its k nearest neighbors. The kNN region can be also seen as a Voronoi polygon $\mathcal{V}(S)$ in a $\mathcal{VD}_k$. So a kNN region query can be defined as:

Definition 5.3.1. *Given a set of OOIs, $U = (o_1,\ o_2, \ldots,\ o_{n-1},\ o_n)$ in $\mathbb{R}^2$, and specify a subset $S \subseteq U$, $|S|=k$, a* kNN region query *of S is to find the largest area which*

[1]The Voronoi polygon should be denoted as $\mathcal{V}(\{o_1,\ o_7\})$, but for simplicity, we omit symbol {} and represent the objects by using their indices.

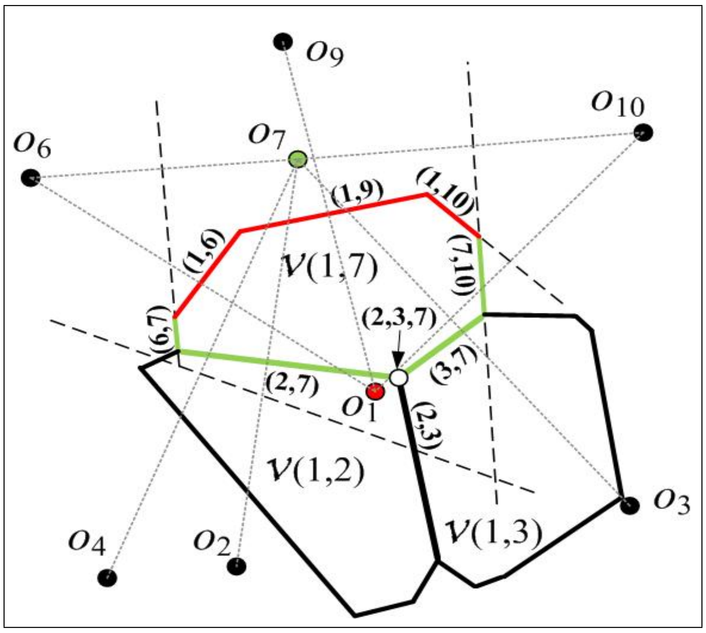


Figure 5.6: An example of some 2-NN regions

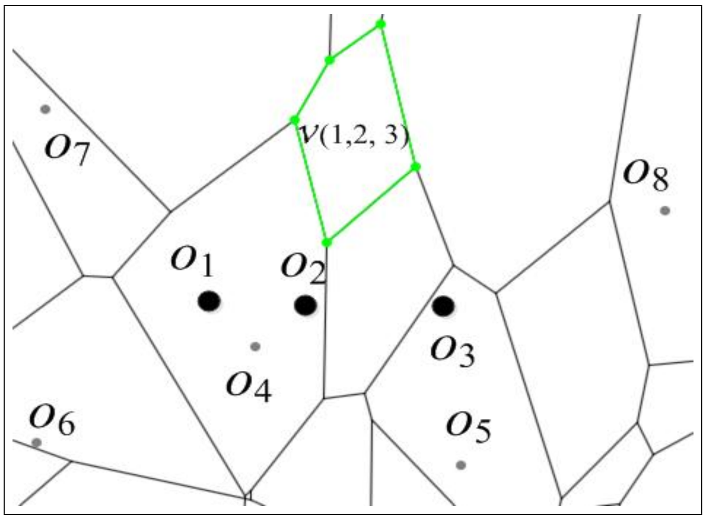
contains all points closer to OOIs in S than to any other OOI in U-S.

$$\mathcal{V}(S) = \{\forall p | d(p, o_i) < d(p, o_j)\}, (o_i \in S,\ o_j \in U - S)$$

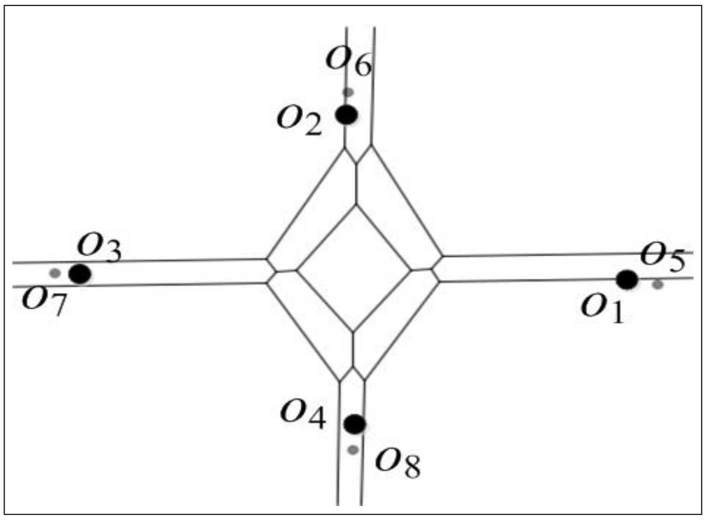
According to this definition, to find the kNN region, $\mathcal{V}(S)$, we need to find the closer space for each object in S, and then the intersection of these spaces is the kNN region of S. Fig. 5.6 shows some 2-NN regions and demonstrates how $\mathcal{V}(1, 7)$ is constructed. We can see that each edge of $\mathcal{V}(1, 7)$ is associated with a pair of indices (i,j), which means that the edge is a portion of the perpendicular bisectors of o_i and o_j, and all the edges of $\mathcal{V}(1, 7)$ are associated with o_1 or o_7. Each vertex is an intersection point of three edges ij, il and jl, and can be denoted by (i,j,l), such as (2,3,7).

Obviously, sometimes the kNN region of S, $\mathcal{V}(S)$, may not exist. In other words, there is no such a point that considers the OOIs in S as its nearest neighbors, such as, S=\{o_1, o_9\}. If $\mathcal{V}(S)$ exists, then it is unique. So the answer to the kNN region query is unique or null depending on the distribution of OOIs of S in U.

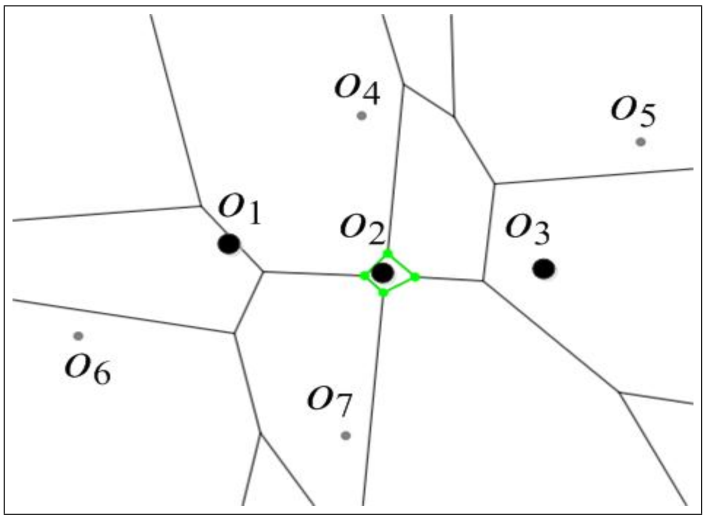
[1]Includes all of the neighbors of OOIs in S, but excludes all the objects in S



(a) A kNN region for three nonadjacent OOIs, o_1, o_2, o_3



(b) 4 adjacent OOIs, o_1, o_2, o_3 o_4, do not have a kNN region



(c) A 3NN region for o_1, o_2, o_3, where o_1, o_3 does not have 2NN region

Figure 5.7: The counter-examples for the misunderstanding of kNN region

Table 5.1: Frequently Used Symbols

Notation	**Description**
o_i	An object of interest in $\mathbb{R}^2$
U	The entire set of OOIs
S	A specified subset of U
$\|S\|$	The cardinality of S
$P_k(o_i)$	A polygon whose edges associated with o_i
$NB(S)$	The neighbor set of S [1]
$\mathcal{V}(o_i)$	The Voronoi polygon of object o_i
$\mathcal{V}(S)$	The kNN region of S
$\mathcal{VD}(U)$	The Voronoi diagram of U
$\mathcal{VD}_k(U)$	The kth-order Voronoi diagram of U
(i, j)	The Voronoi edge associated with o_i and o_j
$\mathcal{DT}(U)$	The Delaunay Triangulation of U
$Conv(S)$	The convex hull of S
$VT(S)$	The vertices of the convex hull of S
$d(o_i, o_j)$	The Euclidean distance between o_i and o_j

According to this definition, we can see that kNN region query focuses on processing a small cluster of OOIs locally, normally these OOIs adjacent to each other, but not necessarily. Several misunderstandings need to be clarified here, referring to Fig. 5.7

If a set of OOIs has a kNN region, all the OOIs in this set have to be adjacent to each other. Fig. 5.7(a) shows a counter-example

If all the OOIs in a set adjacent to each other, then the kNN region exist for sure. Fig. 5.7(b) shows a counter-example

If two OOIs does not have a 2NN region, then any set including these two OOIs does not have the kNN region. Fig. 5.7(c) shows a counter-example

We will discus these in the following sections. Table 5.1 lists the high frequency notations used in this chapter.

5.4 $\mathcal{VD}_k$-based kNN Region: $\mathcal{VD}_k$-kR

n this section, we propose an algorithm for kNN region queries based on kth-Order Voronoi diagram, and analyze its advantages and disadvantages.

Since the kNN region of S can be seen as the $\mathcal{V}(S)$ in a $\mathcal{VD}_k$, the kNN queries can be solved by using the kth-Order Voronoi diagram of the given set U. But, unlike the regular Voronoi diagram (k=1) where each polygon contains an objects, in a kth-order Voronoi diagram (k>1), the polygons may not contain these objects. For example, $\mathcal{V}(1, 7)$ does not contain o_1 and o_7 in Fig. 5.6. Although all the kNN regions of U are included in $\mathcal{VD}_k(U)$, we cannot identify the kNN region for a specific subset by using traditional indexing methods, such as R-Tree. In our $\mathcal{VD}_k$-kR, we propose two methods to identify the kNN region of S by using the generated $\mathcal{VD}_k$.

In the first method, for each of the points, o_i in S, we will identify all the Voronoi polygons considering o_i as one of its kNN by using the kth-order Voronoi diagram. According to Property 5.2.5, all the Voronoi polygons considering o_i as one of the kNN will be adjacent to each other. These polygons will be bounded by all the Voronoi edges associated with o_i. The polygon formed by these edges is signed as $P_k(o_i)$, where all the points consider o_i as one of the kNN. As Fig. 5.8 illustrates, $P_2(o_1)$ includes six Voronoi polygons that consider o_1 as one of the 2NN. The blue lines are the boundary of $P_2(o_1)$. We need to note that there can be more than one edge associated with a pair of objects, such as (1, 13) and (1, 11).

Then, the overlap of all such polygons P_k of S is the kNN region. Fig. 5.9 shows the 4NN region of o_1, o_2, o_3, o_{11}, which is found by using $\mathcal{VD}_k$ and $P_k(o_i)$ where k=4. For the given set of OOIs U=$\{o_1, o_2, ..., o_{15}\}$, generate a $\mathcal{VD}_4(U)$. Then for the specified 4 OOIs, namely, o_1, o_2, o_3, o_{11}, find $P_4(o_1)$, $P_4(o_2)$, $P_4(o_3)$ and $P_4(o_{11})$, and the overlap is the 4NN region of o_1, o_2, o_3, o_{11}, which is represented by the shaded polygon in Fig. 5.9. If any pair of P_ks does not have an overlap, then the null value can be returned immediately, which is an important property of the kNN region.

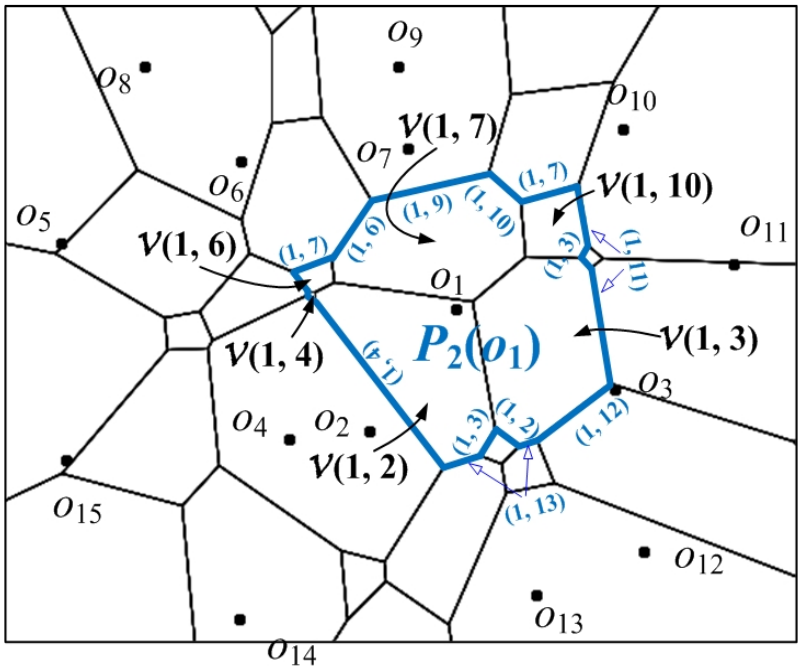


Figure 5.8: An example of P_k: $P_2(o_1)$ in $\mathcal{VD}_2$

Property 5.4.1. *Given a set of OOIs, $U = (o_1, o_2, \ldots, o_{n-1}, o_n)$, and specify a subset $S \subseteq U$, $|S|=k$, for any pair of objects o_i and $o_j \in S$, if $P_k(o_i) \cap P_k(o_j) = \emptyset$, then the kNN region of S does not exist.*

The main disadvantage of this polygon intersection method is the extremely high computation cost, which is caused by sorting the vertices of the non-convex polygon, $P_k(o_i)$, in a proper order [MK89]. Because if the order of vertices of $P_k(o_i)$ is incorrect, we may obtain a wrong result from the overlaps.

Hence, our second method estimates all the vertices of each $P_k(o_i)$ for all the objects in S. Since $\mathcal{VD}_k(U)$ includes all existing kNN regions of any k OOIs in U, so if the kNN region for a given set, S, exists, then $\mathcal{V}(S) \subset \mathcal{VD}_k(U)$. In other words, all the vertices of kNN region of S exist in the vertices of $\mathcal{VD}_k(U)$. According to the definition of the kth-order Voronoi diagram, if a point p considers the k OOIs of S as its kNN, then $p \in \mathcal{V}(S)$. Then we have the following lemma:

Lemma 5.4.1. *Given a set of objects $U = \{o_1, o_2, \ldots, o_n\}$, and a subset $S \subseteq U$, $|S|=k$, if a vertex v of $P_k(S)$s considers the k objects of S as kNN, then v is a vertex of the kNN region of S, $\mathcal{V}(S)$.*

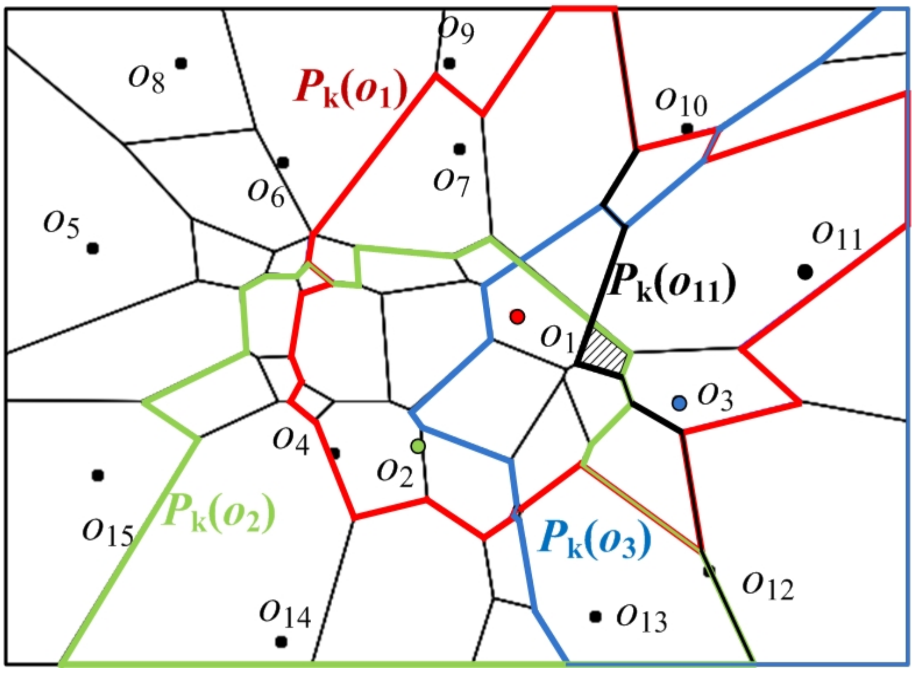


Figure 5.9: A 4NN region found by overlap of $P_k(S)$s

Proof. $\because$ v of $P_k(S)$s is a vertex of $\mathcal{VD}_k(U)$, according to the definition of high-order Voronoi diagram. $\therefore$ v is a vertex of a kNN region and consider k objects of U as its kNN and $\because$ v considers the k objects of S as kNN $\therefore$ v is the vertex of the kNN region of S. □

Algorithm 9: $\mathcal{VD}_k$-kR(U, S)

```
    Input: $U$, $S$={$o_1$, $o_2$, ..., $o_k$} $\subseteq U$
    Output: $k$NN Region
1   Generate a $\mathcal{VD}_k(U)$;
2   Find all the vertices $Vs$, associated with $P_k(S)$ in $\mathcal{VD}_k(U)$;
3   for ($i$=1; $i \leq Vs.size$; $i$++) do
4       if $S$ is not the $k$NN of $v_i$ then
5           delete $v_i$ from $Vs$
6       end
7   end
8   $kR \Leftarrow Vs$ ;
9   return $k$NN Region;
```

On the grounds of the above lemma, we keep all the vertices of $P_k(S)$s in a list and check whether there is an unspecified object closer than the furthest specific objects for each vertex, which means the specified k OOIs are not the kNN for this

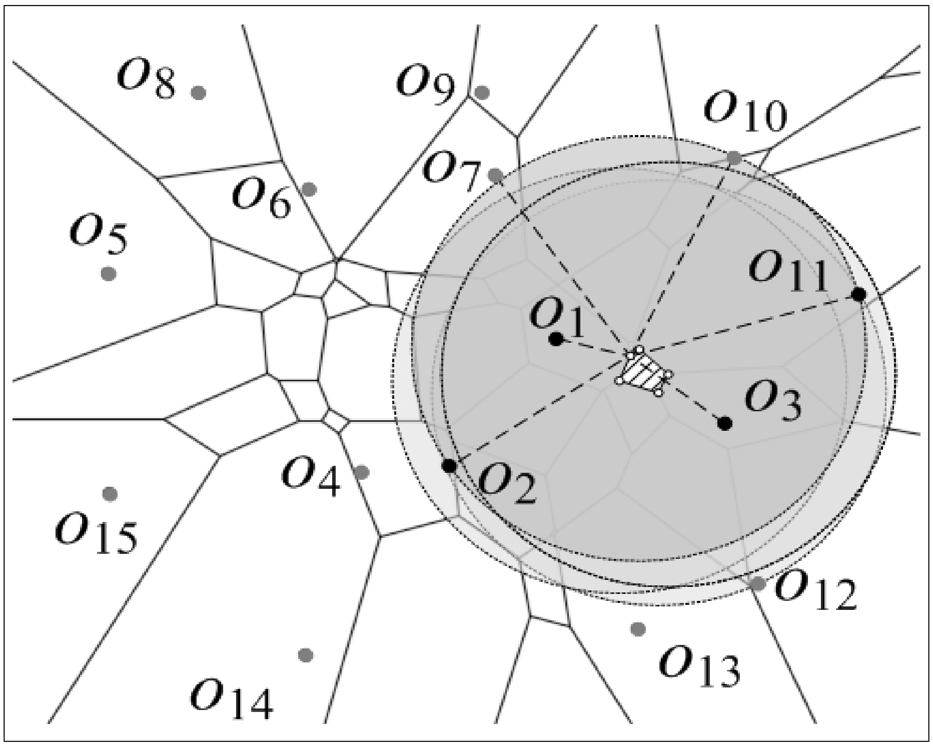


Figure 5.10: A 4NN region found by checking the vertices of $P_k(S)$s

vertex. In such a case, we delete this vertex from the list. After estimating all the vertices of $P_k(S)$s, the remaining vertices in the list will be all the vertices of the kNN region of S. Fig. 5.10 shows an example of finding a 4NN region by using our second method. In Fig. 5.10, the centers of all the circles are the vertices of $\mathcal{V}(S)$ and each circle contains o_1, o_2, o_3, o_{11} and does not include any other objects closer than these four objects to the center, which means these vertices considering o_1, o_2, o_3, o_{11} as their 4NN. So the polygon formed by these vertices is the 4NN region of objects o_1, o_2, o_3, o_{11}.

The $\mathcal{VD}_k$-kR is a pellucid algorithm to process kNN region queries; it can rapidly process a series of kNN queries with the same k on the same set U. But its disadvantages are also obvious. First, it takes very long computational time and requires very large storage spaces to generate $\mathcal{VD}_k(U)$[1] that not only depends on $|S|$, but also associates with $|U|$ which is usually extremely large. Second one $\mathcal{VD}_k$-kR can only process the kNN region for a particular U and $|S|$, which means if either the

[1]The time complexity and storage of generating $\mathcal{VD}_k(U)$ are $O(k^2 n \log n)$ and $O(k^2(n\text{-}k))$, $|U|=n$

given set, U or the number of nearest neighbors, k is changed, the $\mathcal{VD}_k$ has to be reconstructed. So we propose a novel algorithm $\mathcal{DT}$-kR to improve the performance.

5.5 $\mathcal{DT}$-based kNN Region: $\mathcal{DT}$-kR

$\mathcal{DT}$-kR processes the kNN region query based on a Delaunay triangulation and the convex hull generated on the specified k objects. It can return the null value immediately if the kNN region does not exist in most instances and optimize the computational time of finding the kNN region, and it can process any kNN region query even if U or k change.

5.5.1 Screening

Since the kNN region for the given set U may not exist, one of the most important improvements of $\mathcal{DT}$-kR compared to $\mathcal{VD}_k$-kR is the much faster reporting of ineligible cases. The screening step of $\mathcal{DT}$-kR can recognize most of the ineligible specified set S. We define two significant concepts, *cluster* and *independent object* used in this step of $\mathcal{DT}$-kR algorithm. Fig. 5.11 shows a cluster S'=$\{o_1, o_2, o_3, o_{11}, o_{13}\}$ and an independent object o_9 in the set S=$\{o_1, o_2, o_3, o_9, o_{11}, o_{13}\}$. The set S is not a cluster, as o_9 cannot connect other objects in S only through the objects in S, such as, the connection of o_9 and o_1 is via $o_7 \notin S$.

Definition 5.5.1. *Given a set of objects U=$\{o_1, o_2, ..., o_n\}$, a subset $S \subseteq U$, and the $\mathcal{DT}(U)$, if $\forall o_i, o_j \in S$ are connected to each other by edges of $\mathcal{DT}(U)$, but do not pass any object not in S, $(i \neq j)$, then the subset S is called a* cluster *in U, denoted by CL.*

Definition 5.5.2. *Given a set of objects U=$\{o_1, o_2, ..., o_n\}$, a subset $S \subseteq U$, and the $\mathcal{DT}(U)$, if $\exists o_i \in S$ does not link with $\forall o_j \in S$ in $\mathcal{DT}(U)$, $(i \neq j)$, then the object o_i is called an* independent object *in S, denoted by σ.*

Obviously, an independent object is a special cluster whose cardinality $|CL|$=1.

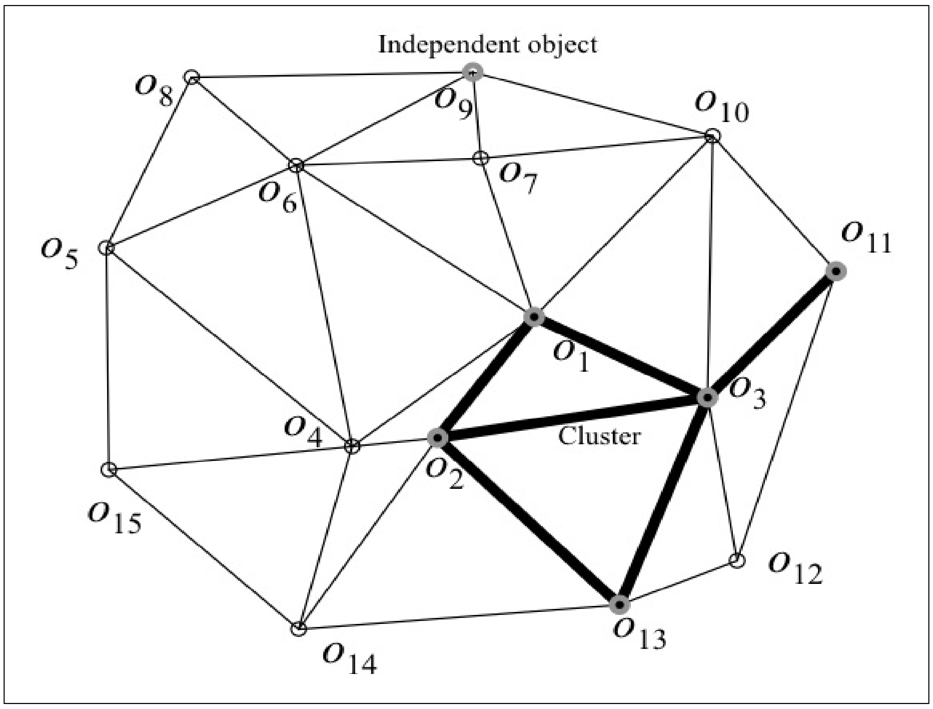


Figure 5.11: An example of a cluster and an independent object

Lemma 5.5.1. *Given a set of object U={o_1, o_2, ..., o_n}, and a subset S⊆U, |S|=k, if the number of CL > 1 in S, then the kNN region of S does not exist.*

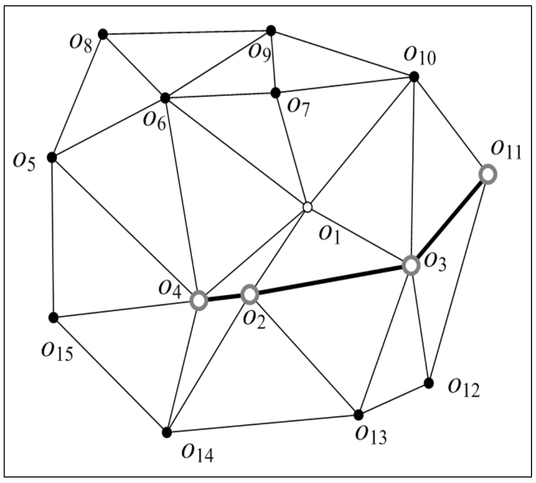
Proof. Given a subset S={o_1, o_2, ..., o_k}⊆U, and a set of clusters $\mathcal{CL}$ in S, CL_1∈$\mathcal{CL}$ is the largest cluster, and |CL_1|=k'<k, then we have:

$$\{\forall o_i | o_i \in S - CL_1,\ o_i \notin NB(CL_1)\}$$

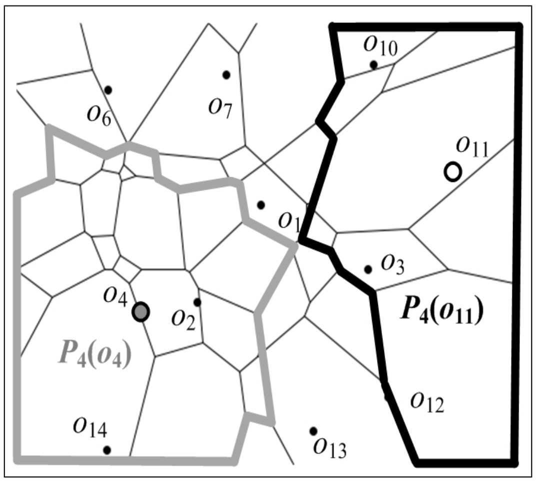
Assuming that a k'NN region exists $\mathcal{V}(CL_1)$, so according to the property of the Voronoi diagram, the (k'+1)NN of any point in $\mathcal{V}(CL_1)$ has to be in $NB(CL_1)$. But {∀ o_i|o_i∈ S-CL_1, o_i∉$NB(CL_1)$}, then the (k'+1)NN region for any subset of S does not exist. According to Definition 5.3.1 , then the kNN region of S does not exist. □

Corollary 5.5.1. *Given a set of objects U={o_1, o_2, ..., o_n}, and a subset S⊆U, if there exists an independent object, σ∈S, then the kNN region of S does not exist.*

Proof. The proof is similar to that of the above lemma, so it is omitted here. □



(a) Cluster of {o_2, o_3, o_4, o_{11}} in $\mathcal{DT}$



(b) $P_4(o_4)$ and $P_4(o_{11})$

Figure 5.12: An example of a false case

Lemma 5.5.1 and corollary 5.5.1 can recognize most of the ineligible cases, but there are still some that can by-pass these two screens, such as the example given in Fig.5.12. Even though the objects o_2, o_3, o_4, o_{11} form a cluster, their 4NN region does not exist. So lemma 5.5.1 and corollary 5.5.1 are necessary conditions for the existence of the kNN region, but are not sufficient conditions. Then the passed false cases will be found in the next step.

5.5.2 Optimizing

Unlike the $\mathcal{VD}_k$-kR algorithm which needs to generate a k-order Voronoi diagram for the entire set, $\mathcal{DT}$-kR finds only a few perpendicular bisectors to identify the kNN region in the optimizing step by using the following two lemmas:

Lemma 5.5.2. *Given a set of objects U={o_1, o_2, ..., o_n}, and a subset $S \subseteq U$, then $\mathcal{V}(S)= \mathcal{V}(VT(S))$, where $\mathcal{V}(S) \subseteq \mathcal{VD}(U)$ and $\mathcal{V}(VT(S)) \subseteq \mathcal{VD}(U\text{-}S+VT(S))$*

Proof. $\forall p \in \mathcal{V}(S)$ have $d(p, o_i)<d(p, o_j)$, $o_i \in S$, $o_j \in U\text{-}S$. $\because$ $VT(S) \subseteq S$ $\therefore$ $\forall$ $p \in \mathcal{V}(S)$ also have $d(p, o_{vt})<d(p, o_j)$, $o_{vt} \in VT(S)$, $o_j \in U\text{-}S$, which means $p \in \mathcal{V}(VT(S))$. So we get:

$$\mathcal{V}(S) \subseteq \mathcal{V}(VT(S)) \tag{5.1}$$

$\forall p \in \mathcal{V}(VT(S))$ have $d(p,\ o_{vt}) < d(p,\ o_j)$, $o_{vt} \in VT(S)$, $o_j \in U\text{-}S$. $\because$ the property of convex hull, $\forall p \in \mathbb{R}_2$, $\{\exists o_{vt} | d(p, o_i) < d(p, o_{vt})\}$, where $o_i \in S\text{-}VT(S)$ and $o_{vt} \in VT(S)$. $\therefore$ $\forall\ p \in \mathcal{V}(VT(S))$ also have $d(p,\ o_i) < d(p,\ o_{vt}) < d(p,\ o_j)$, $o_i \in (S\text{-}VT(S)) \subseteq S$, $o_j \in U\text{-}S$, which means $p \in \mathcal{V}(S)$. So we get:

$$\mathcal{V}(VT(S)) \subseteq \mathcal{V}(S) \tag{5.2}$$

According to equations 5.1 and 5.2, $\mathcal{V}(S) = \mathcal{V}(VT(\mathrm{S}))$. □

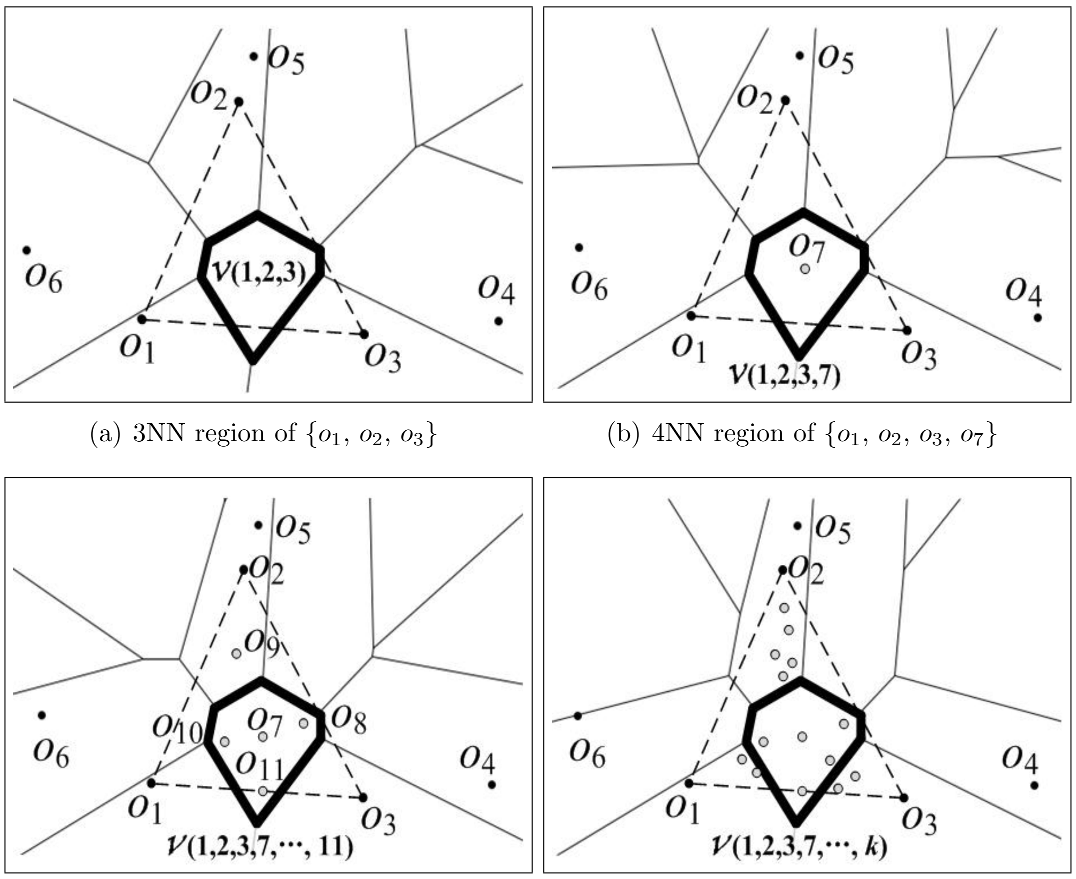
Lemma 5.5.2 indicates that the construction of the kNN region of a set S is associates only with the vertices of the convex hull of S, $VT(S)$. As shown in Fig. 5.13, objects o_1, o_2, o_3 form a convex hull in Fig. 5.13(a), in which the bold polygon is the 3NN region of o_1, o_2, o_3, if we add k' objects to this convex hull, then the (k+k')NN region will not change, such as Fig. 5.13(b) (k'=1), Fig. 5.13(c) (k'=5) and Fig. 5.13(d) (k'=k-3). So $\mathcal{V}$(1,2,3)= $\mathcal{V}$(1,2,3,7)=$\mathcal{V}$(1,2,3,7,...,11). With this optimizing method, since the cardinality of specified set S is very large (k is large), only the objects on the vertex of the convex hull will be used to generate the kNN region, which saves a lot of computation time and storage space.

Lemma 5.5.3. *Given a set of objects $U=\{o_1,\ o_2,\ ...,\ o_n\}$, and a subset $S \subseteq U$, then any edge of $\mathcal{V}(S)$ has to be a portion of $\perp(o_{vt},\ o_{nb(vt)})$, where $o_{vt} \in VT(S)$ and $o_{nb(vt)} \in NB(VT(S))$.*

Proof. Assuming an edge of $\mathcal{V}(S)$ is not associated with $o_{nb(vt)}$, and according to lemma 5.5.2, it has to be associated with o_{vt}, then this edge can be represented by $(o_i,\ o_{vt})$, in which a $o_i \notin NB(VT(S))$.

In respect to the property of the high-order Voronoi diagram, when moving out from $\mathcal{V}(S)$ through edge $(o_i,\ o_{vt})$, the kth NN changes to o_i, and according to the property of Voronoi diagram, the kth NN can change only to one of the neighbor objects of S, so we can get $o_i \in NB(VT(S))$ which conflicts with $o_i \notin NB(VT(S))$. □

Lemma 5.5.3 indicates that we need to find only the perpendicular bisector of the objects on the vertex of convex hull and their neighbors not in the given set.



(a) 3NN region of $\{o_1, o_2, o_3\}$

(b) 4NN region of $\{o_1, o_2, o_3, o_7\}$

(c) 8NN region of $\{o_1, o_2, o_3, o_7, ..., o_{11}\}$

(d) kNN region of $\{o_1, o_2, o_3, o_7, ..., o_k\}$

Figure 5.13: The vertices of a convex hull dominate the kNN region

The sizes of $VT(S)$ and $NB(VT(S))$ are very small no matter how large the given set is.

The algorithm of $\mathcal{DT}$-kR is illustrated in Algorithm 10. At the beginning, a Delaunay triangulation is generated for the entire set, U and if the specified k objects in S form more than one cluster or have an independent object, then $\mathcal{DT}$-kR terminates and returns a null value, which means the kNN region of S does not exist.

If S can pass the screening step, then generate the convex hull of S and find all the neighbor objects of the vertices of this convex hull, get two sets $VT(S)$ and $NB(VT(S))$. For all the objects in $VT(S)$ and in $NB(VT(S))$-S, find the perpendicular bisector for each pair, then the overlap of these areas will be the kNN region of S. If there is a pair of areas not having any overlap, then the kNN region of S does not exist.

5.5.3 Algorithm

Algorithm 10: $\mathcal{DT}$-kR(U, S)

```
    Input: $U$, $S$=\{$o_1$, $o_2$, ..., $o_k$\} $\subseteq U$
    Output: $k$NN Region
 1  Generate a $\mathcal{DT}(U)$;
 2  if (($the\ cluster\ of\ S>1$) or ($\exists\sigma\in S$)) then
 3      return $kR=\emptyset$
 4  else
 5      Generate a $Conv(S)$;
 6      Find the $NB(VT(S))$ in $\mathcal{DT}(U)$;
 7      $kR=h(o_x, o_y)$;
        /* $o_x\in Conv(S)$, $o_y\in NB(Conv(S))$, initialize $kR$ */
 8      for ($i=1$; $i\leq|VT(S)|$; $i$++) do
 9          for ($j=1$; $j\leq|NB(VT(S))|$; $j$++) do
10              if ($o_j\in NB(VT(S))$) and ($o_j\notin S$) then
11                  if ($h(o_i, o_j) \cap kR \neq \emptyset$) then
12                      $kR=h(o_i, o_j) \cap kR$;
13                  else
14                      return $\emptyset$
15                  end
16              end
17          end
18      end
19  end
20  return $kNN\ Region$;
```

5.6 Performance Evaluation

In this section, we evaluate the performance of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR. We first estimate their performance theoretically and analyze the cost of each step of these two algorithms. Then we evaluate the performance of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR based on the extensive simulation results for 60 different data sets. Table 5.2 illustrates all the variables appearing in the performance evaluation.

5.6.1 Theoretical Analysis

In this section, we analyze the performance of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR to process single kNN region query and multiple kNN region queries theoretically, where for multiple

Table 5.2: Performance evaluation parameters

Notation	**Description**
n	Number of OOIs in U(the value of the cardinality of U)
k	Number of OOIs in S(the value of the cardinality of S)
q	Number of queries
e	Number of edges
v	Number of vertices
v_s	Number of vertices of high order Voronoi diagram associate with S
v_c	Number of vertices of the convex hull(the value of the cardinality of $VT(S)$)
v_p	Number of vertices of generated polygon
m	Number of neighbor objects of the vertices of convex hull (the cardinality of $NB(VT(S))$)

Table 5.3: Single query time complexity comparison between $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR based on big O

	Time complexity							
	n=100		n=500		n=1000		n=5000	
k	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$
5	27080	**495**	147137	**1788**	290800	**3429**	1541371	**18924**
10	98020	**699**	603748	**2024**	1272400	**4174**	6711485	**19669**
15	207160	**1235**	1376930	**2930**	2899400	**4321**	15283341	**19816**
20	351040	**1446**	2404390	**2996**	5072200	**4606**	26758940	**20101**

queries, we evaluate the performance of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR to process a q kNN region queries for the same set of U and a specific k in total. The results show that if the number of kNN range queries is not extremely large, $\mathcal{DT}$-kR outperforms $\mathcal{VD}_k$-kR in all instances.

Single query processing: $\mathcal{VD}_k$-k**R**: $\mathcal{VD}_k$-kR algorithm includes two main steps, namely, construction of kth-order Voronoi diagram and identifying kNN region. In the first step, a kth-order Voronoi diagram for the set U needs to be constructed. The best well-known algorithm for constructing a kth-order Voronoi diagram is proposed in [Lee82], whose time complexity and storage are $O(k^2 n \log n)$

and $O(k^2(n\text{-}k))$ respectively, which are the lower boundary of the first step. The second step is to identify a kNN region for the specified k objects in S. In our proposed $\mathcal{VD}_k$-kR, we need to find all the vertices generated by the objects in S and estimate whether S includes all the kNN for each of vertices. So the cost of this step depends on the efficiency of kNN algorithm and the cardinality of the entire set U. Here, we use the distance of the furthest objects in S to the estimated vertex as the threshold. If any other object in U-S has a smaller distance, then we dispose of this vertex. The time complexity and storage of the the second step are $O(v_s n)$ and $O(v_s+n)$. The performance of second step can be improved slightly if you adopt a better kNN algorithm, but since the main cost of $\mathcal{VD}_k$-kR is in the first step, it will not improve the overall performance significantly. The total time complexity and storage of $\mathcal{VD}_k$-kR are: $O((k^2 \log n+v_s)n)$ and $O(k^2(n\text{-}k)+v_s)$.

$\mathcal{DT}$-kR: $\mathcal{DT}$-kR algorithm also has two steps, screening and optimizing. In the screening step, a Delaunay triangulation needs to be implemented on the given set U whereby the connectivity of the specified k objects of S can be checked. Compared with the construction of $\mathcal{DT}(U)$, the cost of checking connectivity can be ignored. So the main cost of the screening step depends on the cost of constructing a Delaunay triangulation for the entire set. The most efficient algorithm is the sweep line algorithm, also known as Fortune's algorithm [For87], the time complexity and the storage of which are $O(n\log n)$ and $O(n)$. In the optimizing step, all the vertices of the convex hull for the specified k objects need to be found as well as their neighbors to identify the kNN region of S. The minimum cost of generating the convex hull is $O(k \log k)$ [Gra72]. The time and space cost of identifying the kNN region are $O(mv_cv_p\log v_p)$ and $O(m+v_c+v_p)$, because the number of perpendicular bisectors depends on m and v_c, the cost of $v_p\log v_p$ guarantees the vertices of the overlap of two half space are in a proper order (clockwise or anticlockwise). The total time complexity and storage of $\mathcal{DT}$-kR are: $\mathcal{DT}$-k**R**$(U,\ S)$**:** $\mathbf{O}(n\text{log}n + k\text{log}k + mv_cv_p\text{log}v_p)$ and $\mathcal{DT}$-kR$(U,\ S)$: $\mathrm{O}(n+k+m+v_c+v_p)$.

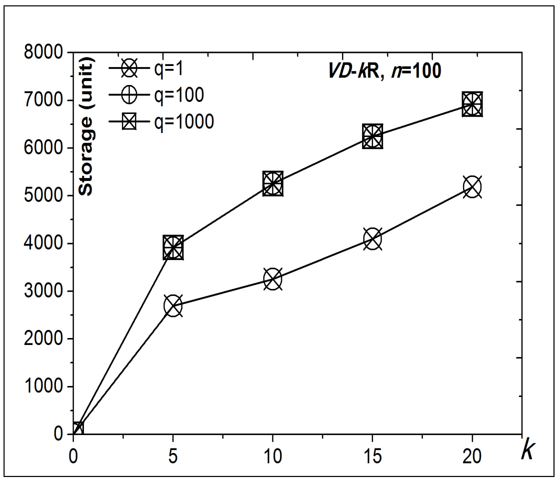
Table 5.4: Single query storage cost comparison between $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR based on O

	Storage							
	n=100		n=500		n=1000		n=5000	
k	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$
5	2695.8	**127.2**	13101.8	**530**	26090.8	**1029**	130091	**5029**
10	3255.2	**130.4**	13812.6	**532.2**	26847.4	**1037**	130847	**5037**
15	4096.6	**136.6**	15021.6	**540.6**	28099.4	**1039**	132099	**5039**
20	5185.4	**138.2**	16604.2	**541.6**	29747.2	**1041**	133747	**5041**

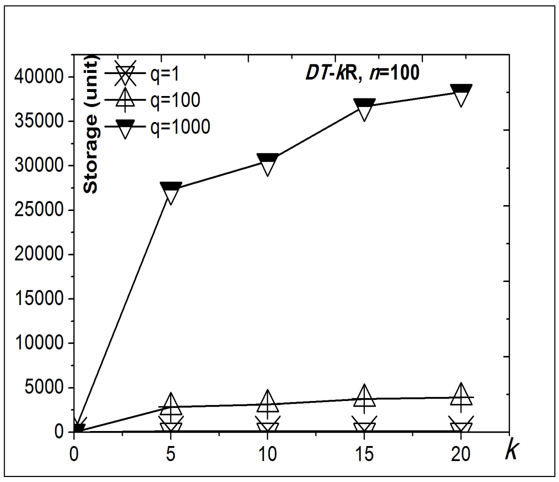
Table 5.3 and 5.4 compare the time complexity and storage of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR on several data sets with different density. Since m, v_c, v_p are always small, the performance of $\mathcal{DT}$-kR outperforms $\mathcal{VD}_k$-kR greatly in terms of cost for both time and space cost, and $\mathcal{DT}$-kR is very stable in any circumstance, even if n and k increase significantly. In contrast, the performance of $\mathcal{VD}_k$-kR declines dramatically when neither n nor k increases, because of the cost of constructing the kth-order Voronoi diagram.

Multiple queries processing: $\mathcal{VD}_k$-k**R**: Because the main cost of $\mathcal{VD}_k$-kR is the construction of kth-order Voronoi diagram for the entire set U and the cardinality of S, k, so it can process multiple queries for the same U and k more efficiently. Assuming there are q kNN region queries that need to be processed on the given U, then only one kth-order Voronoi diagram needs to be constructed for the given set in a specified order, then we just need to estimate only the kNN of the associate Voronoi vertices for each query. The total time complexity and storage of $\mathcal{VD}_k$-kR to process q kNN region queries are $O((k^2 \log n + v_s q)n)$ and $O(k^2(n\text{-}k) + \min(v_s q, v_{VD_k}))$

$\mathcal{DT}$-k**R**: $\mathcal{DT}$-kR also need to generate only one Delaunay triangulation for the entire set U, but still need to check the connectivity, generate convex hull and identify the kNN region for each query, so the performance of $\mathcal{DT}$-kR will not improve significantly compared with processing some single queries on different data sets. But because only a few objects in U are used in the $\mathcal{DT}$-kR algorithm, the time complexity is still much faster than $\mathcal{VD}_k$-kR. On the other hand, if the number



(a) $\mathcal{VD}_k$-kR



(b) $\mathcal{DT}$-kR

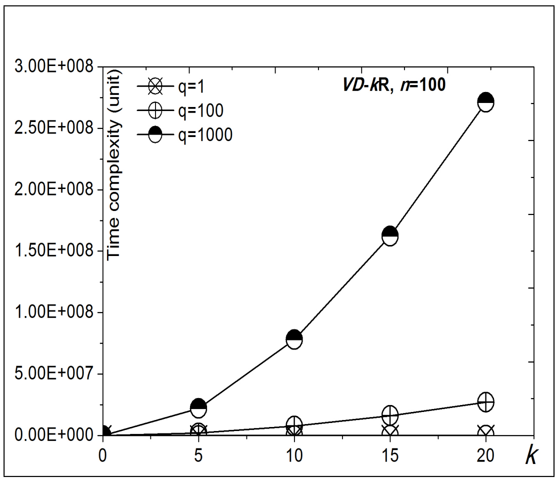
Figure 5.14: Storage comparison of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR for multiple queries

of queries is extremely large, then the generated convex hull and identified kNN regions will require a huge amount of storage, while $\mathcal{VD}_k$-kR can store only all the vertices of $\mathcal{VD}_k(U)$, which can be considerably smaller than $v_s q$.

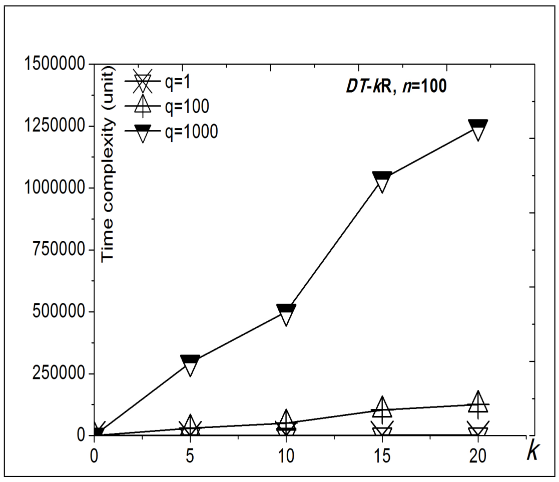
Fig. 5.15 and Fig. 5.14 show the effect of q and k on the performance of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR. The time complexity of both algorithms are proportional to q and k on the same U. When the q increases, $\mathcal{DT}$-kR always requires more storage space, while $\mathcal{VD}_k(U)$ requires only v_{VD_k}, if $v_{VD_k} < v_s q$.

The optimization of $\mathcal{VD}_k$-kR In the light of above theoretical analysis, the inefficiency of $\mathcal{VD}_k$-kR algorithm is due to the high cost of generating the kth-order Voronoi diagram on the entire set. The optimization of $\mathcal{VD}_k$-kR is reducing the size of OOIs to generate the kth-order Voronoi diagram locally.

First of all we need to generate $\mathcal{DT}(U)$ as the underlying framework. For a specific subset S of U, we can find all the neighbors of S based on the generated $\mathcal{DT}(U)$. Then the construction of the kth-order Voronoi diagram is implemented on $S \cup NB(S)$, $\mathcal{VD}_k(S \cup NB(S))$. The remaining steps will be the same with the regular $\mathcal{VD}_k$-kR algorithm, which is checking the all the vertices associated with $P_k(S)$ in $\mathcal{VD}_k(S \cup NB(S))$. Since it is a simple optimization of $\mathcal{VD}_k$-kR, we omit the algorithm and its empirical evaluation here.



(a) $\mathcal{VD}_k$-kR



(b) $\mathcal{DT}$-kR

Figure 5.15: Time complexity comparison of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR for multiple queries

The time complexity of generating $\mathcal{DT}(U)$ as the underlying framework is $O(n\log n)$ which outperforms $O(k^2 n\log n)$ of $\mathcal{VD}_k(U)$ remarkably. But we still need to construct a $\mathcal{VD}_k$ locally, the time complexity of this construction is $O(k^2(k{+}m)\log(k{+}m))$ relying on the number of k and $|NB(S)|$, where $m{=}k{+}|NB(S)|$. Consequently, the time complexity of optimized $\mathcal{VD}_k$-kR is:

$$O(n\log n + k^2(k+m)\log(k+m)) + v_s(k+m))$$

Comparing with the time complexity of the regular $\mathcal{VD}_k$-kR algorithm, we can see that the performance of the $\mathcal{VD}_k$-kR can be improved significantly if and only if $k{+}m{\ll}n$, otherwise the performance will be similar or even worse. Similarly, the storage cost will decreased seriously due to the same reason, so we do not discuss it here redundantly.

5.6.2 Empirical Results

We implemented our proposed $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR algorithms using Java in multiple platforms, including Win32, Win64 and Linux. The experimental results were collected under the Win32 platform with Intel(R) Core2 Duo T9600 2.8GHz CPU and 2GB Memory. We use two sorts of data sets, random data and clustered data. Fig.5.16 shows the patter of the 5000 OOIs distributing randomly and clustered.

Table 5.5: CPU time and I/O access comparison between $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR in term of k/n

$\frac{k}{n}$	CPU Time (ms)		I/O Access (MB)	
10%	20.83	0.852	6.99	0.174
30%	67.22	0.864	22.21	0.153
50%	193.54	1.319	60	0.233
70%	102.57	1.921	34.06	0.419
90%	17.9	0.687	5.72	0.133

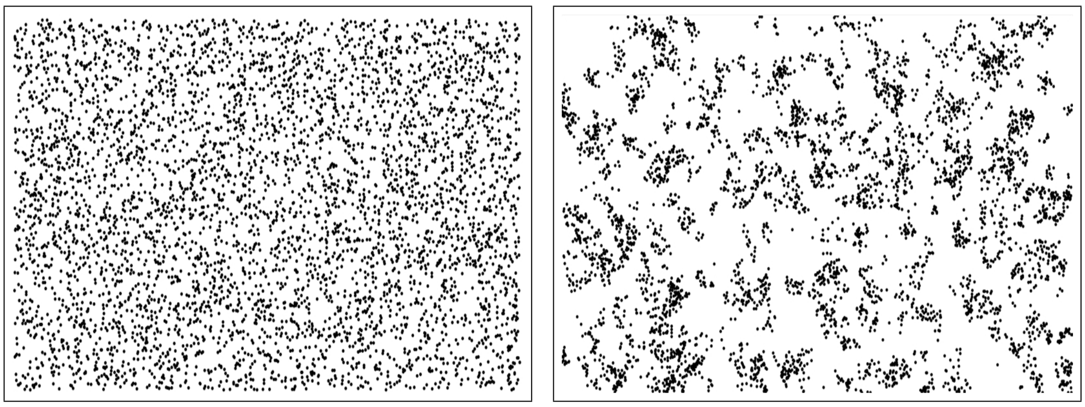

(a) 5000 Objects of Interest on the Random Pattern (b) 5000 Objects of Interest on the Clustered Pattern

Figure 5.16: The patter of random and clustered data with 5000 objects

The random data patter, such as Fig.5.16(a), is used to estimate the performance of our proposed algorithms on uniform distributed data while clustered pattern, such as Fig. 5.16(b), is more representative. We use low (100), medium (500), high (1000) and extremely high (5000) density data for both random patter and clustered pattern.

Query Processing Time: In this section, we measure the CPU time for $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR algorithms.

We first evaluate both algorithms in several low density circumstances (100 objects in a unit space). Table 5.6 and Table 5.7 show the trend of the CPU time with respect to k for random and clustered patterns respectively: the CPU times of both $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR algorithms are proportional to the number of k ($k/n \leq 50\%$). The processing time of $\mathcal{DT}$-kR are influenced much less than $\mathcal{VD}_k$-kR by k because v_c and v_p will not change significantly when k is increasing. $\mathcal{DT}$-kR greatly

Table 5.6: CPU time comparison between $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR in term of k on random pattern

	CPU Time (ms)							
	n=100		n=500		n=1000		n=5000	
k	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$
5	81.031	**0.829**	1424.54	**1.641**	6440.79	**3.423**	N/A	17
10	251.032	**1.11**	3747.8	**1.89**	13725.01	**3.639**	N/A	15
15	654.375	**1.296**	6548.75	**2**	25175.31	**3.297**	N/A	18
20	977.328	**1.329**	11094.69	**2.64**	35945.78	**4**	N/A	16

outperforms $\mathcal{VD}_k$-kR. Even when k=20, the CPU time of $\mathcal{DT}$-kR is about 1.4ms while $\mathcal{VD}_k$-kR is about 700ms. As Table 5.6 and Table 5.7 show, the CPU time of $\mathcal{DT}$-kR is 500 times faster than $\mathcal{VD}_k$-kR. This result validates our theoretical analysis of the time complexity of both algorithms.

We also conducted similar experiments in some medium, high and extremely density circumstances, refer to Table 5.6 and Table 5.7. When k=20 the CPU times of $\mathcal{DT}$-kR are about 2.7ms, 4.0ms and 15ms, 27ms respectively while these results are about 11100ms, 36000ms and 10000ms, 95009ms for $\mathcal{VD}_k$-kR. When the entire set is too large,$\mathcal{VD}_k$-kR does not work due to the high computational cost of $\mathcal{VD}_k(U)$, in that case we can use the optimized $\mathcal{VD}_k$-kR. The increasing of k and density of OOIs has fewer effects on $\mathcal{DT}$-kR than $\mathcal{VD}_k$-kR. Because the main cost of $\mathcal{VD}_k$-kR is the construction of the kth-order Voronoi diagram which is mainly affected by the cardinality of S (k), in contrast, the main cost of $\mathcal{DT}$-kR, the optimizing step, will not be affected significantly when the cardinality of S (k) is increasing. We also found that both algorithms performer worse on clustered data, if there is a high-dense cluster.

Next, we fix the cardinality of U and evaluate the performance of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR on different k/n from 10% to 90%. As Table 5.5 shows, the peak value of $\mathcal{VD}_k$-kR appears when k/n=50%, then its CPU time is inversely proportional to k/n. That is because when $k/n\geq$50%, the number of edges and vertices of the kth-order Voronoi diagram is inversely proportional to k/n for a specific n (The number

Table 5.7: CPU time comparison between $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR in term of k on clustered pattern

	CPU Time (ms)							
	n=100		n=500		n=1000		n=5000	
k	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$
5	110	**0.58**	1448	**2.39**	19878	**15**	N/A	47
10	235	**13**	3248	**3.21**	14399	**19**	N/A	18
15	497	**16**	5742	**32**	27427	**22**	N/A	31
20	901	**18**	9059	**15**	95009	**27**	N/A	32

Table 5.8: I/O access comparison between $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR in term of k on random pattern

	I/O Access (MB)							
	n=100		n=500		n=1000		n=5000	
k	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$
5	107.47404	**0.405**	2191.62	**9.96**	8512.82	**20.13**	N/A	387.84
10	347.716	**0.697**	4889.44	**12.25**	18351.36	**22.57**	N/A	482.01
15	886.189	**0.871**	8340.58	**12.8**	29232.83	**19.13**	N/A	628.65
20	1856.584	**0.678**	13135.88	**17.37**	41958.53	**24.76**	N/A	743.51

of Voronoi polygons which is equal C_n^k will decrease when k increases and greater than $n/2$). The maximum value of $\mathcal{DT}$-kR appears when k/n is about 70% since v_c and v_p are largest at this point.

I/O:

In this section, we evaluate the performance for $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR algorithms with respect to the I/O cost.

We also evaluate the I/O cost of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR on low, medium, high and extremely high density circumstances. Table 5.8 and Table 5.9 show the results of such four cases. As Table 5.8 and Table 5.9 show, the I/O cost of $\mathcal{VD}_k$-kR is much higher than $\mathcal{DT}$-kR, since the number v_s is very large in most of the instances and for each vertex, we need to check its kNN and compare it with S. The increasing of k and density of OOIs has less effect on the I/O cost of $\mathcal{DT}$-kR, since the construction cost of Delaunay triangulation is much less than the construction of the high-order

Table 5.9: I/O access comparison between $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR in term of k on clustered pattern

	I/O Access (MB)							
	n=100		n=500		n=1000		n=5000	
k	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$	$\mathcal{VD}_k$	$\mathcal{DT}$
5	120.89	**35.3**	2272.19	**111.82**	8588.45	**358.85**	N/A	338.14
10	363.31	**101.89**	4961.83	**243.62**	18951.3	**492.89**	N/A	823.04
15	871.39	**282.5**	8708.25	**324.12**	29596.7	**817.38**	N/A	1088.14
20	1568.3	**1065.09**	14157.19	**410.96**	43164.82	**1008.66**	N/A	900.55

Voronoi diagram. Because v_c and v_p remain stable in any density circumstance, the variation of the I/O of $\mathcal{DT}$-kR is primarily due to the construction of Delaunay triangulation on more OOIs, which is still better than $\mathcal{VD}_k$-kR.

Then we fix the cardinality of U and evaluate the I/O cost of $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR on different k/n from 10% to 90%. As Table 5.5 shows, the changes of I/O is very similar to the CPU with respect to k/n. $\mathcal{DT}$-kR also outperforms $\mathcal{VD}_k$-kR remarkably. The reason is also similar, so we omit the repeated explanation here.

5.7 Summary

In this chaper, we propose a novel spatial query, named, kNN region query, which is very important for theoretical studies and practical applications. We also illustrate two algorithms, $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR to process kNN region queries. $\mathcal{VD}_k$-kR is a high-order Voronoi diagram based approach, which is efficient to process a pile of kNN region queries on the same data set U with same order k. In such cases, only one kth-order Voronoi diagram needs to be generated and then it can process any kNN region query with the same order. But the processing time of $\mathcal{VD}_k$-kR is not efficient enough due to the cost of generating of the kth-order Voronoi diagram. To process a query with different k or on another data set, the kth-order Voronoi diagram has to be regenerated, which reduces the applicability of $\mathcal{VD}_k$-kR.

Then we proposed another algorithm, $\mathcal{DT}$-kR based on several computational geometric concepts, whereby the kNN region queries can be processed very fast in any circumstances.

Chapter 6

Locating an Optimum Region on Finite Point Objects

In this chapter, we study another region-expected query problem in Euclidean space called, optimum region. We propose a novel algorithms which indicating the relative position of objects of interest by utilizing polar coordinate system to answer this type of query.

6.1 Overview

Range search is a historic and significant queries not only in spatial databases, but also in many other applications, such as, computational geometry, geographical information systems (GIS), and computer-aided design (CAD). In chapter 5, we discuses finding a region for small set of objects (k); While sometimes the region might be expected on the entire set globally. Optimum region queries is such a type. Given a finite set of point objects and a positive value r (e.g, a distance), optimum region includes all the points which can covering the maximum number of objects in the finite set as the center of a circle with radius r.

All the existing spatial queries solve local spatial problem only, such as, range search, k nearest neighbors (kNN) and closest pairs, and these queries can only retrieve point objects. There is no technology for polygon objects indexing in spatial databases. Another interesting problem of optimum region is to find a region according to the distribution os a set of objects. Such queries have not been studied in spatial databases so far. Whereas the optimum region query is very useful and significant in many applications. Some examples are listed as follow:

Build a hospital in a community and let most resident reach to the hospital within 20 mins or 15km.

A company plans to establish a new Wi-Fi base station, the coverage range of which is 20 meters, to cover most Wi-Fi devices in the company.

A bombardment aircraft looks for a best area to drop a bomb for maximum destruction.

We give the formal definition of the optimum region in section 6.3

Unlike the typical range search query or other spatial queries, optimum region needs to construct a (set of) region(s) not stored in the database rather than retrieving point objects, the locations of which are stored as a pair of coordinates in the databases. Therefore all the existing works and indexing approaches on spatial queries are incapable of processing optimum region queries. The representation of the structure of the optimum region is also tough, as the boundary of the optimum region is not a (set of) simple polygon(s) but a (set of) polygon(s) with circular edges. Those two issues make the optimum region query to a challenging problem.

We concentrates on the optimum region query processing on a finite set of point-objects 2D Cartesian space, and proposed a novel algorithm, *Circle Partition and Arcs Superposition* (*CPAS*) based upon the polar coordinates system. *CPAS* first plots a circle with a specific radius centered at each object in the given set, and then divides each circle into a set of arcs which covered by other circles. For each circle we find an (a set of) overlap(s) of most arcs on this circle. Finally, the arcs covered by most circles form the boundary of the optimum region including all the points, any of which can be a center of a circle to cover maximum number of objects in the given object set.

The remainder of this chapter is organized as follows: Section 6.2 illustrates several geometric concepts used in our *CPAS* algorithms. Section 6.3 defines the problem formally. Section 6.4 illustrates the algorithm of *CPAS* in detail. Section 6.5 evaluates the performance by using both synthetic and real data. Section 6.6 summarizes this chapter.

6.2 Preliminary

In our algorithm, we use the polar coordinates to represent the relative position of a point on a circle with the center in a plane. In this section, we introduce the polar coordinates system briefly.

The polar coordinate system is a two-dimensional coordinate system where the position or the coordinates of a point is determined by a distance from a fixed point at the center of the coordinate space and an angle from a fixed direction. The fixed point (similar with the origin in a Cartesian coordinate system) is called the pole, and the reference axis, ray emanating from the pole in the fixed direction is the polar axis. In a common polar coordinate system, the polar axis points off toward the right (corresponding to the x-axis in Cartesian coordinates).

The polar coordinates of a point p are denoted as $p(r,\ \theta)$, where r, called *radial coordinate* or *radius*, is the distance of the point from the pole; θ, named as *angular coordinate* or *polar angle*, is the angle measured counterclockwise from the polar axis to a ray from the pole through p. Fig. 6.1 illustrates a polar coordinate system with the polar coordinates of a point $p(r,\ \theta)$. As shown in Fig. 6.1, an angular coordinate is normally expressed in either degree or radians.

The polar coordinate system is extremely useful when the relative position of any two points in a 2D space can be expressed by the distance and the angle. For many curves, the polar equations are the simplest way to describe themselves, whereas their Cartesian expressions are much more intricate; for some of them, such as, rhodonea curve, the polar equation is the only expression.

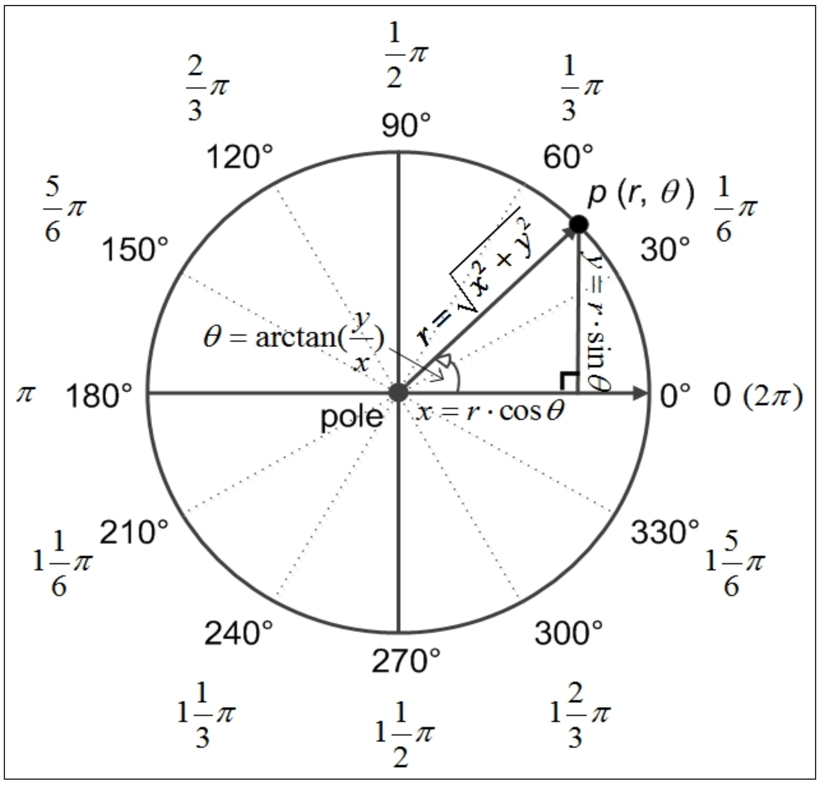


Figure 6.1: A polar coordinate system, an example of polar coordinates of p and the converting between its Cartesian coordinates and polar coordinates.

6.2.1 Multiple Representation and Uniqueness of Polar Coordinates

Because of the circular nature of the polar coordinate system, each point does not have a unique polar representation. Adding any number (either positive or negative) of a round, 360° or 2π, to the angular coordinates of a point does not change the corresponding position. In addition, because the radial coordinate r is a directed distance, r can be expressed by a negative number, which indicates the opposite direction to the positive radius. In general, a point $p(r,\,\theta)$ can be represented as an infinite number of polar coordinates:

$$(r,\theta)=(r,\theta\pm 2n\pi)$$

or

$$(r,\theta)=(-r,\theta\pm(2n+1)\pi)$$

where n is an integer. While for a particular pair of polar coordinates, there is one and only one point corresponding to this coordinates in a polar coordinate system.

To make the polar coordinates consistent and uniqueness, we limit r to non-negative numbers ($r \geq 0$) and θ to the interval [0, 360) in this chapter.

6.2.2 Converting between Cartesian and Polar Coordinates

Cartesian coordinates have a aptitude to describe the position of a point in the entire space. The Cartesian coordinates is also one of the most extensively used coordinates in spatial database; While polar coordinates focus on describing the relative position of a point to the pole precisely. So sometimes these two coordinate systems need to be converted. Fig. 6.1 illustrates the conversion of the coordinates of p between these two systems.

Cartesian coordinates to polar coordinates: For a given point $p(x, y)$ in a Cartesian coordinates system, the corresponding polar coordinates r and θ are:

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \arctan(\frac{y}{x})$$

Polar coordinates to Cartesian coordinates: For a given point $p(r, \theta)$ in a polar coordinates system, the corresponding cartesian coordinates x and y are:

$$x = r \cdot \cos\theta$$

$$y = r \cdot \sin\theta$$

Due to the simplicity and the circular geometric property of the polar coordinate system, it is widely employed in mathematics, physics, engineering, navigation, robotics, and other sciences. In this chapter, we also use polar coordinates of a point to represent the position of intersections on a circle.

6.3 Problem and Query Definition

The objective of *optimum region search* is to find the region where the center of a circle with a fixed radius can be located to cover most objects of interest (OOIs) in the given set. The set contained by such circles is called *covered set.*

Definition 6.3.1. *Given a finite set of OOIs,* U=$\{o_1, o_2, ..., o_{n-1}, o_n\}$ *in* $\mathbb{R}^2$*, and a specific radius* r*, then* $\forall p_i \in \mathbb{R}^2$*,* $\exists! \odot(p_i, r)$*. If* $\odot(p_i, r)$ *can cover a subset of* U*,* O_i *then this set is called the* ***covered set*** *of* p_i*, denoted as:* O_i=$\{o|d(o, p_i) \leq r\} \subseteq U$*.*

According to this definition, for two different points $p_i \neq p_j$, they may have the same covered set O_i=O_j

In this chapter, we discuss the optimum range query only in a two-dimension Euclidean space, where the metric is called Euclidean distance. For any two points, o and p, their Euclidean distance is calculated as:

$$d(o, p) = \sqrt{(o_x - p_x)^2, (o_y - p_y)^2}$$

where the subscripts x, y represent the Cartesian coordinates of an object. The formal description of the optimum region query is as following:

Definition 6.3.2. *Given a finite set of OOIs,* U=$\{o_1, o_2, ..., o_{n-1}, o_n\}$ *in* $\mathbb{R}^2$*, and a specific radius* r*, an* optimum range query *is to find the area(s) including all the centers of circles with radius* r *that can cover the maximum number of OOIs in* U*, denote as:* R_U=$\{p_i|d(o, p_i) \leq r\}$*, where* $o \in O_i \subseteq U$ *and* $|O_i| > \forall |O_j|$*, if* $p_j \notin R_U$*.*

We need to clarify that for a specific set, U, there exists exactly one result of the the optimum range query, which may contain several disjoint regions. Table 6.1 illustrates all the notations used in this chapter.

Table 6.1: Frequently used symbols

Notation	**Description**
R	A region in a Cartesian space, $\mathbb{R}^2$
o_i	An object of interest in $\mathbb{R}^2$
U	The entire set of OOIs
(x, y)	The Cartesian coordinates of a point
(r, θ)	The polar coordinates of a point
$\odot(p, r)$	A circle centered at p with r as radius
C_i	An infinite set including all the centers of circles covering o_i
O_i	A set of OOIs covered by $\odot(p_i, r)$
$\widehat{\rho\rho'}$=$[\theta, \theta']$	An arc from point ρ to ρ' in a counterclockwise with interval $[\theta, \theta']$, where θ and θ' represent their angular coordinates
$d(o, p)$	The Euclidean distance between any two points

6.4 Optimum Range Query Processing

In this section, we propose an algorithm, named *Circle Partition and Arcs Superposition* (*CPAS*) to process the optimum range queries on a set of OOIs in a 2-D Cartesian space.

Theorem 6.4.1. *Given any two points, o, $p \in \mathbb{R}$ and a specific value of r, if $p \in \odot(o, r)$, then $o \in \odot(p, r)$*

Proof. $p \in \odot(o, r) \Rightarrow d(o, p) \leq r \Rightarrow o \in \odot(p, r)$ □

Corollary 6.4.1. *Given a set of points, $U=\{o_1, o_2, ..., o_n\} \in \mathbb{R}$, an independent point p and a fix value of r, then $O=\{o_i | p \in \odot(o_i, r)\} \subseteq U \Leftrightarrow O \subset \odot(p, r)$*

Proof. $O=\{o_i | p \in \odot(o_i, r)\} \Leftrightarrow \forall o_i \in O$ have $d(o_i, p) \leq r \Leftrightarrow \bigcup_{i=1}^{n} o_i \subset \odot(p, r) \Leftrightarrow O \subset \odot(p, r)$ □

In general, to find the center of a circle with a specific radius to cover the maximum number of objects in U is that, first, for each object, o_i, the location of the centers, C_i, of all the circles which can cover o_i need to be identified. According to Theorem 6.4.1, the location of these centers are also covered by $\odot(o_i, r)$, then we

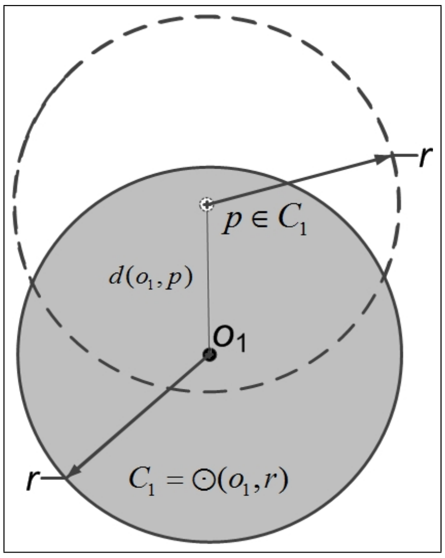


Figure 6.2: The location of all the centers can cover o_i, $C_i = \odot(o_i,\ r)$

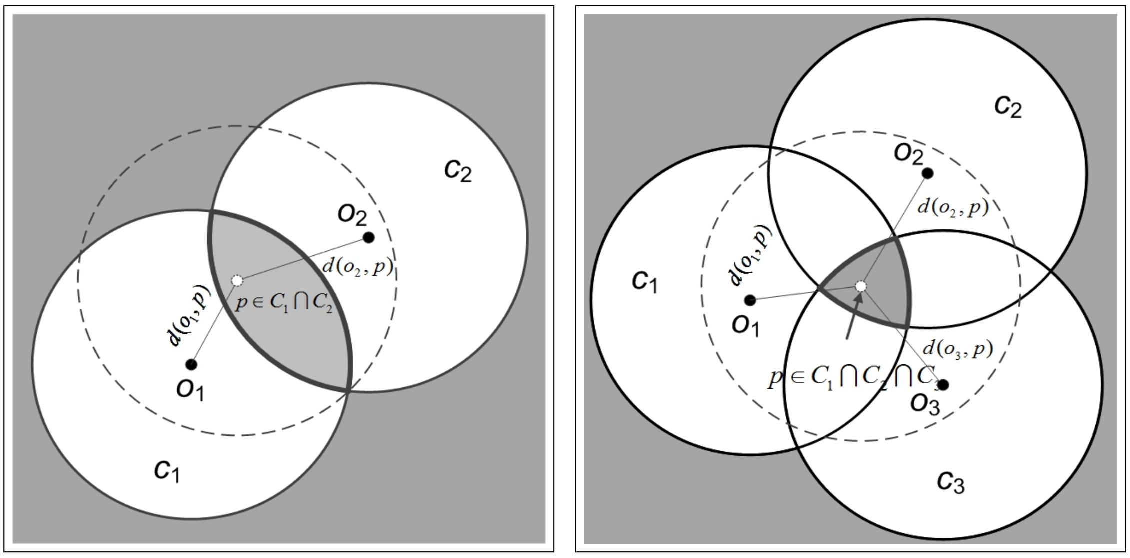
have $C_i = \odot(o_i,\ r)$ referring to Fig. 6.2. Based on the corollary to Theorem 6.4.1, as shown in Fig. 6.3(a) and Fig. 6.3(b) if a circle, $\odot(p,\ r)$, can cover a set of objects O=$\{o_1,\ o_2,\ ...,\ o_n\}$, then $p \in \bigcap_{i=1}^{n} C_i$. So we summarize that the area intersected by most C_i is the optimum region, which is bounded by a set of arcs of C_i.

Identifying the optimum region on a finite set includes two main operations, namely, *circle partition* and *arcs superposition*. For each circle generated by an object, we need to divide it into a set of arcs, one of which is covered by the circle of another object. All the arcs on a circle may overlap with each another. So for each circle, we want to find the arc(s) contained by the maximum number of arcs on a circle and store it (them) as a (set of) new arc(s). We also count the number of the coverage. Then each circle will keep a (set of) new arc(s) intersected by the arcs covered by other circles. Finally, all the arcs own the maximum counted number form the boundary of the optimum region.

6.4.1 Circle Partition

In this step, for each circle, $\odot(o_i,\ r)$ of an object $o_i \in U$, a set of arcs covered by other objects need to be identified.

If any pair of circles, $\odot(o_i,\ r)$, $\odot(o_j,\ r)$ intersect, $d(o_i,\ o_j) < 2r$, $o_i(x_i,\ y_i)$, $o_j(x_j,\ y_j) \in U$, where $(x,\ y)$ are the coordinates of an object, then $\odot(o_i,\ r)$, $\odot(o_j,\ r)$ will



(a) $\{p|p{\in}C_1{\cap}C_2\}$ (b) $\{p|p{\in}C_1{\cap}C_2{\cap}C_3\}$

Figure 6.3: The area includes all the centers of circles covering a set of O, where $|O|{\in}\{2,\ 3\}$

have two intersections (excluding that these two circles are tangent, and $o_i{\neq}o_j$, where there is one and only one intersection that is the middle point of line o_i and o_j. We ignore this case, science it will not generate any optimum region.), denoted as: $\rho(x_\rho,\ y_\rho)$ and $\rho'(x_{\rho'},\ y_{\rho'})$, which can be calculated according the known coordinates of o_i, o_j and the specific r.

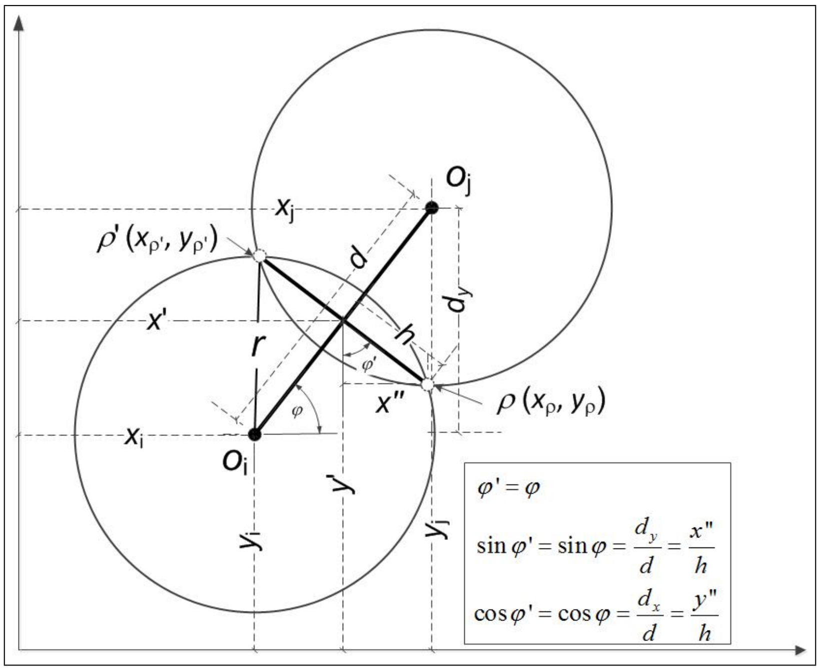
Lemma 6.4.1. *Given any two objects $o_i(x_i,\ y_i)$, $o_j(x_j,\ y_j)$ and a specific radius r, then the coordinates of the intersections, ρ and ρ', of $\odot(o_i,\ r)$, $\odot(o_j,\ r)$ can be expressed as:*

$$x_\rho = \frac{x_i + x_j}{2} \pm \frac{(y_i - y_j)\sqrt{r^2 - \frac{(x_j-x_i)^2+(y_j-y_i)^2}{4}}}{\sqrt{((x_j - x_i)^2 + (y_j - y_i)^2}}$$

$$y_\rho = \frac{y_i + y_j}{2} \pm \frac{(x_j - x_i)\sqrt{r^2 - \frac{(x_j-x_i)^2+(y_j-y_i)^2}{4}}}{\sqrt{((x_j - x_i)^2 + (y_j - y_i)^2}}$$

Proof. The proof is based on mathematical and geometrical calculation,the details refer to 6.4. We omit it here. □

If the Cartesian coordinates, $\rho(x_\rho,\ y_\rho)$ of an intersection on $\odot(o_i,\ r)$ is calculated, then it needs to be converted to the polar coordinates considering objects $o_i(x_i,\ y_i)$



o_j
x_j
$\rho'(x_{\rho'}, y_{\rho'})$
d
d_y
x'
h
r
φ'
x''
$\rho(x_\rho, y_\rho)$
φ
x_i
o_i
y'
y_i
y_j
$\varphi' = \varphi$
$\sin\varphi' = \sin\varphi = \frac{d_y}{d} = \frac{x''}{h}$
$\cos\varphi' = \cos\varphi = \frac{d_x}{d} = \frac{y''}{h}$

Figure 6.4: The coordinates calculation of ρ and ρ'

as the pole, in order to record the relative position of the arc on its circle. In other words, the two intersections and the relevant object represent one and only one unique arc on the circle centered at this object. As explained in section 3.2, then the angular coordinate of ρ are:

$$\theta = \arctan(\frac{y_\rho - y_i}{x_\rho - x_i})$$

Since the radial coordinate is given as r in an optimum range query, then the polar coordinates of ρ is denote as: (r, θ). Assuming the polar coordinates of two intersections between $\odot(o_i, r)$ and $\odot(o_j, r)$ are $\rho_j(r, \theta)$ and $\rho'_j(r, \theta')$ considering o_i as the pole, then the arc on $\odot(o_i, r)$ covered by $\odot(o_j, r)$ can be expressed as an interval $[\theta, \theta']$ or $[\theta', \theta]$, which depends on the value of θ and θ'. We can see that the polar coordinates of the intersections are different on two circles, as illustrated in Fig. 6.5. The arc of o_1 covered by o_2 is $[15.9°, 88.5°]$, while the corresponding arc on o_2 is $[196.2°, 268.8°]$, since the intersections consider different objects, o_1 and o_2 as the poles. Moreover, we have the following theorem.

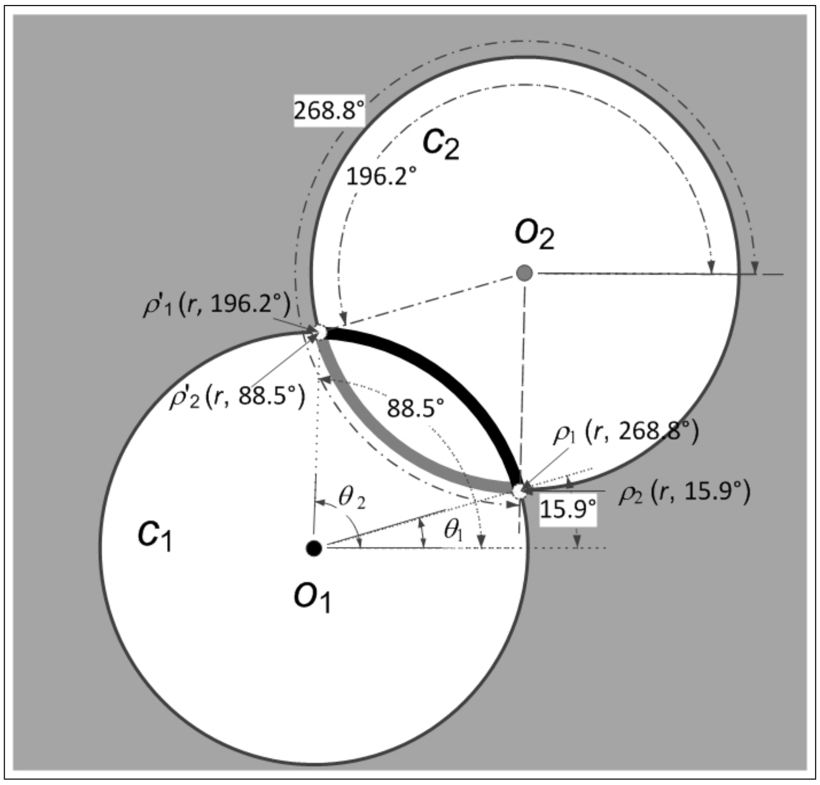


Figure 6.5: The polar coordinates of ρ and ρ' on $\odot(o_1, r)$ and $\odot(o_2, r)$

Theorem 6.4.2. *Given any two points o_i, o_j and specific radius r, $\odot(o_i, r)$ and $\odot(o_j, r)$ have two intersection points ρ and ρ', then the central angle $\angle\rho o_i\rho' = \angle\rho o_j\rho'$*

Proof. It based on trigonometric function, we omit the proof here. □

Corollary 6.4.2. *Given any two points o_i, o_j and specific radius r, $\odot(o_i, r)$ and $\odot(o_j, r)$ intersect at points ρ and ρ', then the central angle $\angle\rho o_i\rho' = \angle\rho o_j\rho' < 180°$*

Proof. Based on the sum of the inter angle of a quadrilateral equals 360° and Theorem 6.4.2, we have:

$$\left.\begin{array}{l}\angle\rho o_i\rho' + \angle\rho o_j\rho' + \angle o_i\rho' o_j + \angle o_i\rho o_j = 360^\circ \\ \angle\rho o_i\rho' = \angle\rho o_j\rho'\end{array}\right\}\Rightarrow$$

$$2\angle\rho o_i\rho' = 2\angle\rho o_j\rho' = 360^\circ - (\angle o_i\rho' o_j + \angle o_i\rho o_j) \Rightarrow$$

$$\angle\rho o_i\rho' = \angle\rho o_j\rho' = 180^\circ - \tfrac{1}{2}(\angle o_i\rho' o_j + \angle o_i\rho o_j) < 180^\circ$$

□

In our algorithm, the angular range of a circle is represented as [0°, 360°), if an angular coordinate of the intersection point is not in [0°, 360°), we need to adjust it by adding 360°.

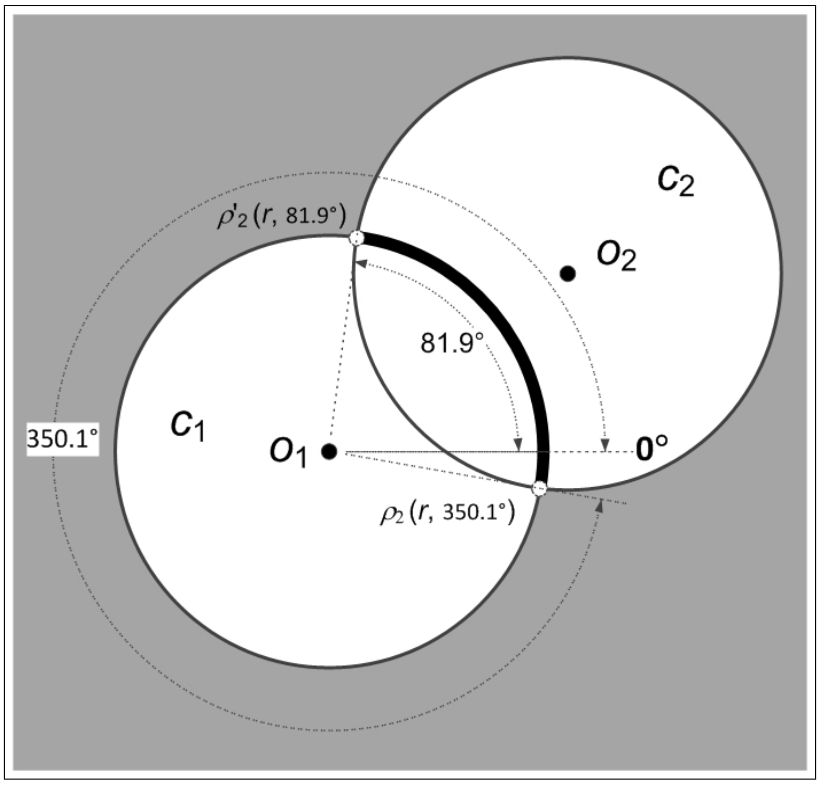


Figure 6.6: The interval crosses over 0°

However, the arc crosses over 0° will arise a problem. See Fig. 6.6 as an example, the angular coordinates of ρ_2 and ρ'_2 are 350.1° and 81.9° on $\odot(o_1,\ r)$, based upon the common definition of an interval, the range is [81.9°, 350.1°] which is not the factual result according to Fig. 6.6; While if we use [350.1°, 81.9°], it is inconsistent with the mathematic definition of an interval, as start point is greater than the end point. To solve this problem, we define this kind of arcs as a *cross-over arc*.

Definition 6.4.1. *Given any two points o_i, o_j and specific radius r, $\odot(o_i,\ r)$ and $\odot(o_j,\ r)$ intersect at points $\rho(r,\ \theta)$ and $\rho'(r,\ \theta')$, a **cross-over arc** is the arc having $|\theta\text{-}\theta'| > 180°$, whose interval called **cross-over interval** can be expressed as:*

$$[0°, \min(\theta, \theta')] \cap [\max(\theta, \theta'), 360°)$$

Although we need to check all the pairs of the objects in U, most of the pairs will be pruned if the $d(o_i,\ o_j) \geq 2r$. With all the theories illustrated in this step, each circle centered at an object of interest will be participated into a set of arcs, the interval of which is represented in a uniform format, $[\theta,\ \theta']$ where $\theta > \theta'$, from $\theta >$

to θ' in the counterclockwise. All the arcs on a circle might have some overlapped intervals with each other, which produce an (a set of) interval(s) covered by most arcs on a circle. In the next phase, we will extract this sort of intervals on each circle.

Algorithm 11: Circle Partition. CP(U, r)

```
Input: $U$={$o_1$, $o_2$, ..., $o_n$} and $r$
Output: The set of arc sets $\mathbb{A}$={$A_1$, $A_2$, ..., $A_i$}
for ($i$=0; $i \leq |U|$; $i$++) do
    for ($j$=0; $j \leq |U|$; $j$++) do
        if ($i \neq j$) and $dis(o_i, o_j) < 2r$ then
            $\rho(x, y) \leftarrow \odot(o_i, r) \cap \odot(o_j, r)$;
            $\rho'(x', y') \leftarrow \odot(o_i, r) \cap \odot(o_j, r)$;
            $\rho(r, \theta) \leftarrow \rho(x, y)$;
            $\rho'(r, \theta') \leftarrow \rho'(x', y')$;
            if $|\theta - \theta'| > 180°$ then
                Add $[0°, \min(\theta, \theta')]$ into $A_i$;
                Add $[\max(\theta, \theta'), 360°)$ into $A_i$;
            else
                Add $[\min(\theta, \theta'), \max(\theta, \theta')]$ into $A_i$;
            end
        else
            Continue;
        end
        return $A_i$;
    end
end
```

6.4.2 Arcs Superposition

In this step, for each circle, the intervals covered by most other circles need to be identified according to the overlaps of the arcs generated in the previous step. We start the discussion from any two arcs, and then we expand it to multiple arcs.

Assuming $\widehat{\rho_i\rho_i'}$ and $\widehat{\rho_j\rho_j'}$ are two arcs covered by $\odot(o_i, r)$ and $\odot(o_j, r)$ on a circle, then the arc covered by both circles is contained by both $\widehat{\rho_i\rho_i'}$ and $\widehat{\rho_j\rho_j'}$. See Fig. 6.7 as an example. We suppose that $[\theta_i, \theta_i']$ and $[\theta_j, \theta_j']$ are the intervals of $\widehat{\rho_i\rho_i'}$ and $\widehat{\rho_j\rho_j'}$ (the number of intervals could be more, if either $\widehat{\rho_i\rho_i'}$ or $\widehat{\rho_j\rho_j'}$ is a cross-over arc), then the interval of the arc covered by both circles is $[\theta_i, \theta_i'] \cap [\theta_j, \theta_j']$. Fig. 6.8

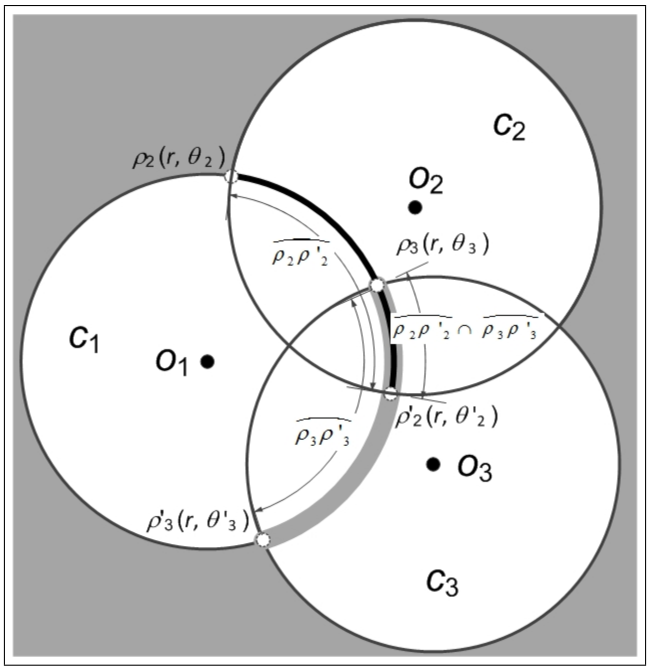
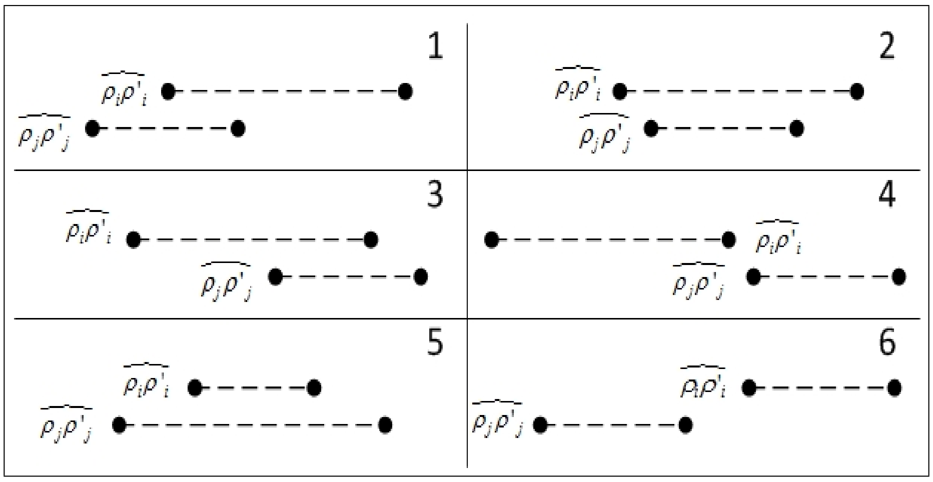


Figure 6.7: The superposition of two arcs

shows all the possible results of the superposition of any two arcs, in which the arcs are abstracted as straight lines. The new interval will be recorded according to the values of θ_i, θ'_i, θ_j and θ'_j, excluding case 4 and 6 in Fig. 6.8. For case 4 and 6, we will keep arcs $\widehat{\rho_i\rho'_i}$ and $\widehat{\rho_j\rho'_j}$ as the arcs covered by most circles, since they covered by the same number of circles, and no other arc is covered by more circles than them.

Now we explain the arcs superposition in a more universal scenario. For each circle $\odot(o_i, r)$, A is the set including all the arcs generated by $\odot(o_i, r)$ with others



1
$\widehat{\rho_i\rho'_i}$
$\widehat{\rho_j\rho'_j}$
2
$\widehat{\rho_i\rho'_i}$
$\widehat{\rho_j\rho'_j}$
3
$\widehat{\rho_i\rho'_i}$
$\widehat{\rho_j\rho'_j}$
4
$\widehat{\rho_i\rho'_i}$
$\widehat{\rho_j\rho'_j}$
5
$\widehat{\rho_i\rho'_i}$
$\widehat{\rho_j\rho'_j}$
6
$\widehat{\rho_i\rho'_i}$
$\widehat{\rho_j\rho'_j}$

Figure 6.8: All the cases of two arcs superposition

circles, A=$\{\widehat{\rho_1\rho_1'}$, $\widehat{\rho_2\rho_2'}$, ..., $\widehat{\rho_n\rho_n'}\}$. Then we want to find the arc which is covered by the maximum circles intersecting with $\odot(o_i, r)$ and we define this arc as:

Definition 6.4.2. *S is a subset of A, $S \subseteq A$, $|S|=m$ and $\forall\widehat{\rho_l\rho_l'} \in S$ satisfy $\bigcap_m^{l=1} \widehat{\rho_l\rho_l'} \neq \emptyset$, and $\nexists S' \subset A$, has $|S'|=k > m$ and $\forall\widehat{\rho_l\rho_l'} \in S'$ satisfying $\bigcap_k^{l=1} \widehat{\rho_l\rho_l'} \neq \emptyset$, then the arc $\widehat{max} = \bigcap_m^{l=1} \widehat{\rho_l\rho_l'}$ is called the* ***maximum arc****, the interval of is called* ***maximum interval***

Intuitively, the maximum arc is included by most arcs on a circle. To find the interval of $\widehat{max}$, we propose an algorithm named arcs superposition, $AS()$. First of all, a set of intervals in A, is accepted from the previous step. All the arcs in A are assigned a counter indicating that how many times it is covered. The initiate value is 1, since each arc is covered by the circle containing itself. Then A is traversed to calculate all the intervals generated by each pair of arcs, and store them into a new list I_{new}. $AS()$ is invoked recursively on the new list until any two arcs do not have a superposition, The counter increases by 1 in each recursion. Fig. 6.9 illustrates an example of multiple arcs superposition for $\widehat{\rho_1\rho_1'}$, in reality, we also need to check $\widehat{\rho_2\rho_2'}$ $\widehat{\rho_5\rho_5'}$ similarly. The pseudo code of Arc Superposition algorithm is shown in Algorithm 12.

6.4.3 Optimum Range Identification

Based on the algorithms introduced above, identifying the boundary of the optimum range is quite straightforward. After the step of circle partition, each circle relevant with an object of interest has a set of arcs, whereby an (a set of) arc(s) covered by most other circles, maximum arc(s), on each circle can be found in the step of arcs superposition. So finally, we only need to retrieve all the maximum arcs having the greatest count number.

Lemma 6.4.2. *The boundary of the optimum region is a (set of) closed polygon(s) with circular edges. Only all the maximum arcs having the largest count number form the boundary.*

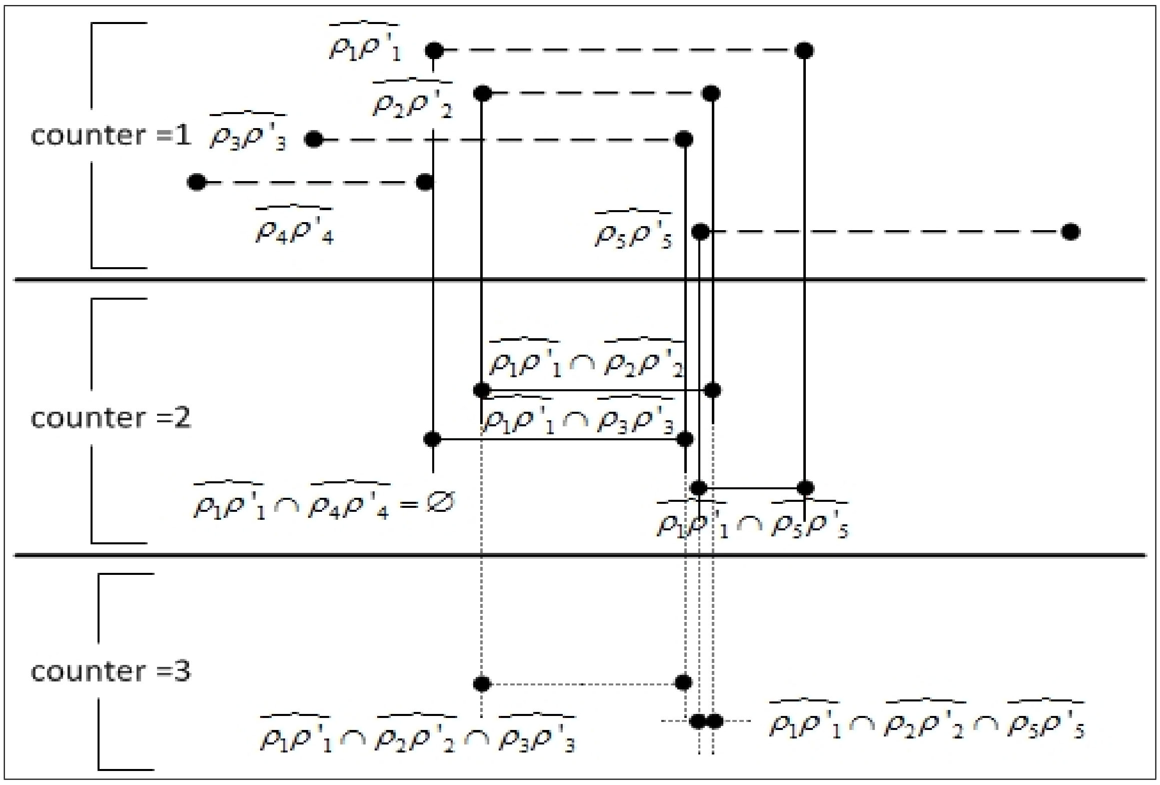

counter =1
$\widehat{\rho_1\rho'_1}$
$\widehat{\rho_2\rho'_2}$
$\widehat{\rho_3\rho'_3}$
$\widehat{\rho_4\rho'_4}$
$\widehat{\rho_5\rho'_5}$
counter =2
$\widehat{\rho_1\rho'_1} \cap \widehat{\rho_2\rho'_2}$
$\widehat{\rho_1\rho'_1} \cap \widehat{\rho_3\rho'_3}$
$\widehat{\rho_1\rho'_1} \cap \widehat{\rho_4\rho'_4} = \varnothing$
$\widehat{\rho_1\rho'_1} \cap \widehat{\rho_5\rho'_5}$
counter =3
$\widehat{\rho_1\rho'_1} \cap \widehat{\rho_2\rho'_2} \cap \widehat{\rho_3\rho'_3}$
$\widehat{\rho_1\rho'_1} \cap \widehat{\rho_2\rho'_2} \cap \widehat{\rho_5\rho'_5}$

Figure 6.9: Multiple arcs superposition for $\widehat{\rho_1\rho'_1}$

Algorithm 12: Arcs Superposition, AS(A)

```
Input: $A$
Output: $\widehat{max}_A$
1  List<Arc> $I_{new}$ = new LinkedList<Arc>();
2  for ($i$=0; $i \leq |A|$; $i$++) do
3      for ($j$=$i$+1; $j \leq |A|$; $j$++) do
4          if (superposition($\widehat{\rho_i\rho'_i}$, $\widehat{\rho_j\rho'_j}$)) then
5              Add superposition($\widehat{\rho_i\rho'_i}$, $\widehat{\rho_j\rho'_j}$) into $I_{new}$;
6          else
7              Continue;
8          end
9      end
10 end
11 if $I_{new} \neq \emptyset$ then
12     $A$=$I_{new}$;
13     $A.counter$+=1;
14     AS($A$);
15 else
16     return $\widehat{max}_A$=$A$;
17 end
```

Proof. If an arc $\widehat{\rho_j\rho_j'}$ on $\odot(o_i,\ r)$ is covered by $\odot(o_j,\ r)$, then there has to be an arc $\widehat{\rho_i\rho_i'}$ on $\odot(o_j,\ r)$ covered by $\odot(o_i,\ r)$ because of the mutual coverage property of circles. These two arcs form a closed polygon since they share the same two intersections. Similarly, if an arc $\widehat{\rho_O\rho_O'}$ on $\odot(o_i,\ r)$ is covered by a set of circles associate with O=$\{o_1,\ o_2,\ ...,\ o_n\}$ and $o_i \in O$, then there must exist at least one $o_j \in O$ that has an arc $\widehat{\rho_O\rho_O'}$ also covered by the circles generated by O, and these two arcs formed a closed polygon. $\square$

Fig. 6.10 shows a result of optimum region which includes two areas bounded by the thick lines. A circle centered in these two areas with r as radius can cover o_1, o_2, o_3, o_5 or o_2, o_3, o_4, o_5. We can see that even $\odot(o_3,\ r)$ covers both area, but the count number of its max arcs is less than the greatest count number, so any arcs of $\odot(o_3,\ r)$ will not be involved in forming the boundary. Which means that not all of the objects can be covered by the optimum region are involved in forming the optimum region.

The optimum region is not the simple overlap of all the circles, which may be an empty set normally; While the optimum region is always existing by chosen the best location among a set of objects. As shown in Fig. 6.10, although o_1 to o_5 do not have any shared area, the optimum region still can be found.

Algorithm 13: CPAS(U, r)

```
Input: $U$ and $r$
Output: Optimum region
List<Arc[ ] > $A$ = new LinkedList<Arc[ ] >();
$A$ = CP($U$, $r$);
for ($o_i$:$U$) do
    $\widehat{max}_{A[i]}$=AS($A$[i]);
end
return optimum region ← $\widehat{max}_{A[i]}$ has max($\widehat{max}_{A[i]}$.counter);
```

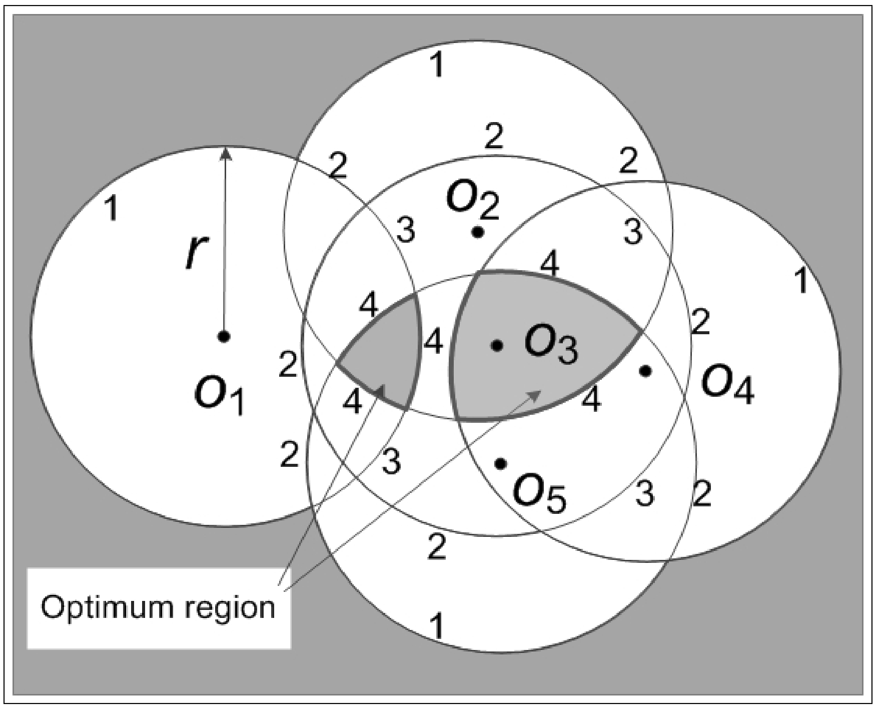


Figure 6.10: An example of optimum region including two areas

6.5 Performance Evaluation

In this section, we evaluate the performance of the two main steps, circle partition and arcs superposition, of *CPAS* in term of CPU time, I/O access and number of objects covered by the optimum region. First we describe the parameters of the experiments, followed by the results and the discussion.

6.5.1 Experiment Setup and Datasets

All the experiments were conducted on Intel(R) Core(TM)2 Duo CPU T9600@2.8GHz with 4 GBytes memories. All the algorithms were implemented in Java and executed on Windows 7 64-bits. We use two sorts of data sets, random data and clustered data. Table 6.2 shows the parameters in our experiments. Fig.6.11 shows the patter of the 5000 OOIs distributing randomly and clustered. The random data patter, such as Fig.6.11(a), is used to estimate the performance of CPAS on uniform distributed data while clustered pattern, such as Fig. 6.11(b), is more representative.

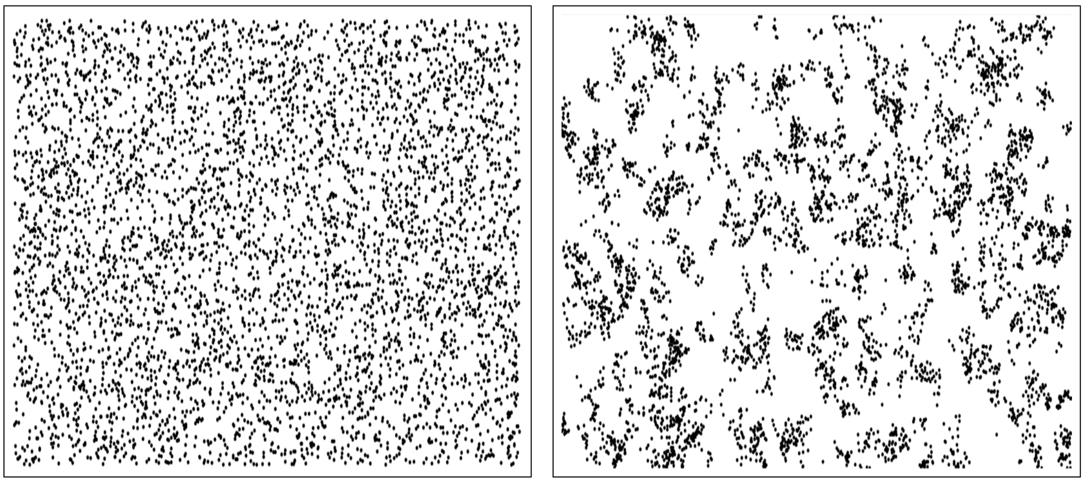

(a) 5000 Objects of Interest on the Random Pattern (b) 5000 Objects of Interest on the Clustered Pattern

Figure 6.11: The patter of random and clustered data with 5000 objects

Table 6.2: System parameters for experiments

Parameter	**Range**
Number of objects of interest ($\times$ 100)	1, 5, 10, 20, 50
The coverage capacity of the query	5, 10, 15, 20
The threshold distance (Clustered data)	30
The map size	1280 $\times$ 760

6.5.2 Experimental Results

For each experiment, the performance is conducted on three metrics, namely, CPU time, I/O access and the cardinality of the OOI set which can be covered by the query with the specific radius. We also estimate the proportion of the two steps, circle partition and arcs superposition, of our algorithm on these three metiers and we can observe that they are effected by different factors. We perform our algorithm 100 times on each data set and take the average as the result.

CPU Time:

Table 6.3 and Table 6.4 compare the response time of processing an optimum region query on uniformed and clustered data sets. By observing the CPU time of circle partition (CP) and arcs superposition (AS), the variation of the radius r does not effect the response time of CP seriously, as the extension of r may only involve a few more OOIs to partite the circle generated by an OOI and due to the

Table 6.3: CPU Time comparison of CPAS on random data

♯	5000		2000		1000		500		100	
r	CP	AS	CP	AS	CP	AS	CP	AS	CP	AS
5	34.904	0.776	5.426	0.273	0.133	1.266	0.311	0.037	0.0133	0.00057
10	37.072	0.658	5.674	0.077	1.399	0.051	0.33	0.018	0.0135	0.00041
15	33.945	17.388	5.579	0.374	1.34	0.153	0.331	0.022	0.01356	0.00044
20	34.508	247.27	5.58	1.313	1.351	0.053	0.334	0.024	0.0136	0.00035

Table 6.4: CPU Time comparison of CPAS on clustered data

♯	5000		2000		1000		500		100	
r	CP	AS	CP	AS	CP	AS	CP	AS	CP	AS
5	32.35	5.432	6	0.026	1.31	0.158	0.355	0.089	0.01389	0.00011
10	36.55	49.811	6.58	0.284	1.34	0.389	0.34	0.478	0.01396	0.00604
15	35.2	1553.156	5.36	24.07	1.4	3.335	0.33	17.376	0.01429	0.06071
20	33.2	23161.048	5.58	227.502	1.41	30.467	0.34	194.829	0.01906	0.23394

low computation cost of CP, the sightly increasing involving OOIs will not effect the performance of CP significantly. Intuitively, the clustered patter should have lower response time than random patter. Whereas, based on the experiment result, the effecting is very limited, due to the high performance of CP. So the response time of CP mainly relies on cardinality of the entire set of OOIs. While the CPU time of arcs superposition (AS) is proportional to the specified radius, cardinality of set U and the distribution of OOIs. The variation of any of these three parameters will effect the performance of AS considerably, and AS performs much better on the uniformed data than the clustered data.

I/O Access

Table 6.5 and Table 6.6 show the I/O access comparison of CPAS on random and clustered data for a set of optimum region queries. These results on CP and AS are very similar with the CPU time discussed above. AS is the main cost of CPAS algorithm on I/O access as well and it works better on the uniformed data than clustered data.

Table 6.5: I/O Access comparison of CPAS on random data

♯	5000		2000		1000		500		100	
r	CP	AS	CP	AS	CP	AS	CP	AS	CP	AS
5	805.16	2.65	128.49	0.18	32.57	0.02	12.031	0.003	0.492	0.002
10	807.52	157.27	128.76	3.19	33.11	0.22	8.43	0.03	0.492	0.001
15	809.7	5526.21	130.04	41.83	32.43	1.79	8.33	0.11	0.342	0.003
20	815.1	127928.37	130.39	461.95	32.78	10.75	8.28	0.47	0.335	0.01

Table 6.6: I/O Access comparison of CPAS on clustered data

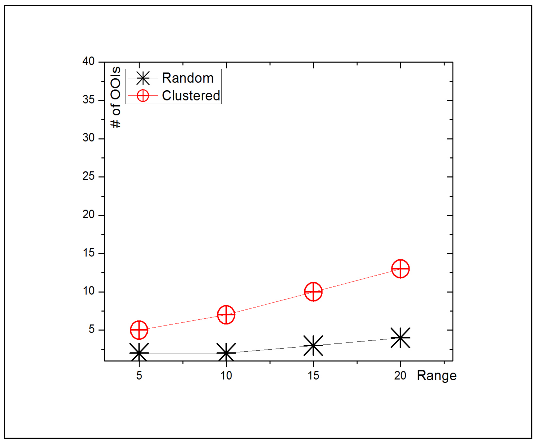
♯	5000		2000		1000		500		100	
r	CP	AS	CP	AS	CP	AS	CP	AS	CP	AS
5	802.81	175.13	130.5	3.71	32.48	0.84	8.29	0.97	0.35	0.08
10	807.82	27344.91	129.66	375.88	32.8	81.74	8.38	218.22	0.39	1.51
15	813.4	835375.88	131.33	13981.48	33.21	1842.01	8.75	9478.03	0.42	36.25
20	819.12	10250399.25	131.89	130305.1	33.56	17230.54	9.03	104160.4	0.43	126.61

Cardinality of the Result Set

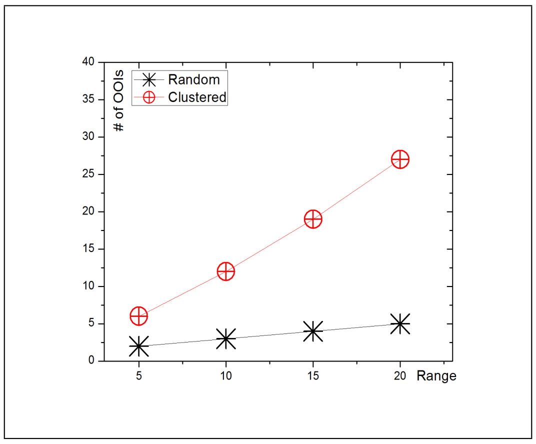
Fig. 6.12(a)- Fig. 6.12(e) show how many OOIs in the entire set U can be covered by the query point with the specified radius. Obliviously, clustered data will have a large result set than the uniformed data. For the uniformed data, the cardinality of the result set is totally relies on $|U|$, and it is in proportion to $|U|$; While for the clustered data, the cardinality of the result set is not only depends on $|U|$, but also relevant with the distribution of OOIs in U. In other words, it depends on the densest cluster.

6.6 Summary

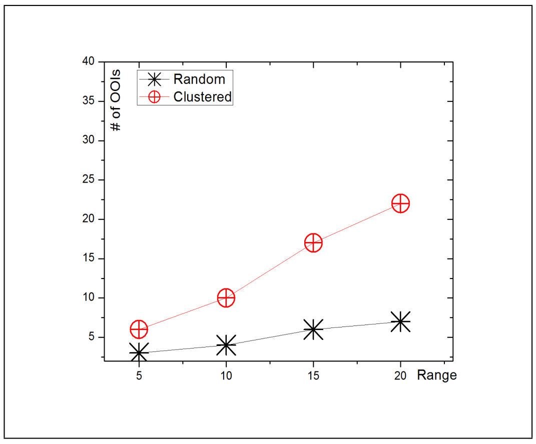
In this chapter, we propose a novel spatial query, named, optimum region query that locate an (a set of) area(s) containing all the centers of circles with a specific radius covering most objects in the given set. We propose an algorithm, *Circle Partition and Arcs Superposition* (*CPAS*) to process optimum region queries, whereby the optimum region queries can be processed efficiently in the most circumstances.



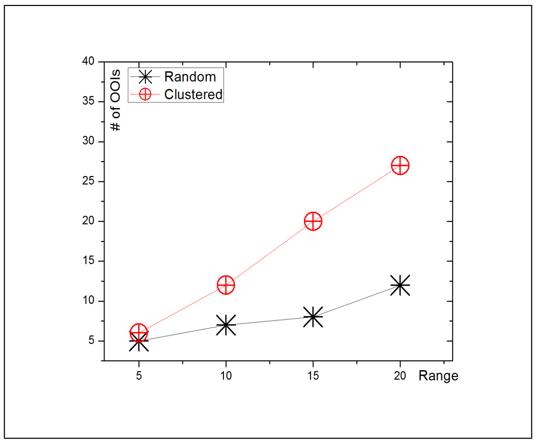
(a) $|U|$ =100



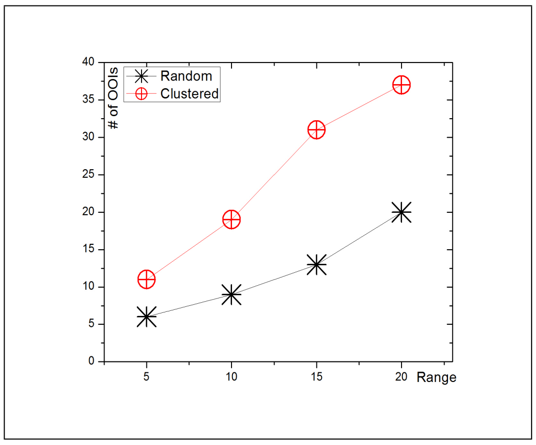
(b) $|U|$ =500



(c) $|U|$ =1000



(d) $|U|$ =2000



(e) $|U|$ =5000

Figure 6.12: Cardinality of OOIs set covered by the query point

Chapter 7

Final Remarks

7.1 Contributions

In this section, we emphasize the contributions of this thesis again. In the first part, we studied several novel point-expected range queries extensively to improve the computation cost and reduce the communication cost. In the second part, we are the first to propose region-expected queries as well as discuss the techniques for such sort of range queries in spatial databases to enrich the diversity of spatial queries. The contributions are briefly described as follow.

7.1.1 Point-Expected Range Queries

Processing range queries in constrained circumstances: in this thesis, we propose three novel constrained range search queries and an approach for each query based on network Voronoi diagram, which makes the range search query processing more flexible to satisfy various requirements in different circumstances. The performances of these these algorithms are analyzed theoretically and evaluated experimentally. The results show that our approaches can process constrained range search queries very efficiently.

Processing continuous range queries: we study the problem of continuously monitoring moving range queries on a set of data objects that do not change their locations. Our proposed technique is based on network Voronoi diagram, which is a spatial decomposition of a metric space. Our algorithm reduces the computation and communication cost because it does not require to recompute the results as long as the query does not move out from a Voronoi cell. We conduct extensive experimental analysis to study the effectiveness of our network Voronoi based approach. Moreover, the experimental results demonstrate that the proposed approach outperforms its competitors.

Processing range queries on moving objects: the spatial queries of moving objects monitoring is restricted by frequent updates, as a result, processing spatial queries over moving objects becomes a tough task, especially in road networks. Our technique stores the motion of the moving objects as a function of time instead of its position to achieve equilibrium between the updating and communication cost, and the accuracy of the location of moving objects. By experimental studies, we show that our proposed range monitoring algorithm can process moving objects monitoring range queries efficiently.

7.1.2 Region-Expected Queries

kNN Region Query Processing: none of existing spatial queries can find or retrieve regions closer to a set of specific objects than to any other objects, even though this is an important problem in spatial databases and practical applications. In this thesis, we propose a novel query, k Nearest Neighbor region searching, which retrieves a region where every point considers specified k objects as the k nearest neighbors. In addition, we propose two algorithms, $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR based on high-order (kth-order) Voronoi diagram and Delaunay triangulation respectively for it. We discuss the kNN region searching in a 2D Cartesian space, but it can extend to a higher dimensional space. An extensive theoretical and empirical study was conducted to compare and evaluate the performance of these two algorithms. The study showed that $\mathcal{VD}_k$-kR and $\mathcal{DT}$-kR outperforms the other ones in different scenarios with respect to k, the number of objects in the entire set U, and the number of queries to be processed.

Optimum Region Query Processing: another region-expected problem is that database users might be interested in find a region that can cover maximum objects among a set of objects with a specific radius r. We propose such a query in this thesis, named optimum region. In addition, we developed an algorithm, *Circle Partition and Arcs Superposition* (*CPAS*), to solve this problem. An extensive empirical study was conducted to evaluate the performance of *CPAS*. The results showed that *CPAS* can process the optimum region queries efficiently in most of the circumstances.

7.2 Conclusions

In this thesis, we present efficient techniques to answer various range/region-related spatial queries under different circumstances. Chapter 3 and 4 present our research on finding point objects within a specific region in constrained or dynamic circumstances. In Chapter 5 and Chapter 6, we illustrate our approaches to answer region-expected queries, including, kNN region queries and optimum region queries. Below are the details.

In chapter 3, we study the problems of constrained range searching in spatial networks. We design some variations of Voronoi diagram as the underlying frameworks of our proposed approaches, which reduce the computation cost remarkably. We propose three approaches, *TCR*, *RCR*, *kCR* to process these constrained queries. We conduct a set of experiments extensively on each approach and the results demonstrate that they can process the corresponding range queries efficiently.

In chapter 4, we introduce two technologies to efficiently process distance based moving range queries, and monitor moving objects in spatial networks respectively. Our proposed approach does not only significantly improve the computation time but also reduces the communication cost for client-server architectures, as we only update the data if the range search result might be effected by the moving objects or the query point itself. Moreover, we design a structure to store the movement of spatial objects as a function, which reduce the storage and communication cost significantly.

In chapter 5, we propose a region-expected query for a small set of specific objects among the entire set, named k nearest neighbors region. We illustrate two algorithms based on kth-order Voronoi diagram and Delaunay triangulation for this sort of queries. We conduct a rigorous theoretical analysis to study the preciseness and effectiveness of our algorithms. The theoretical results are verified by an extensive experimental study. The experiment results also demonstrate that the proposed approaches for Euclidean distance based kNN region queries performs well. We also show that the kth-order Voronoi diagram based algorithm and the Delaunay triangulation based algorithm have advantages in different circumstances.

In chapter 6, we introduce another region-expected query, optimum region query, which finds the area including all the centers of such circles that can covering maximum number of objects in the entire set. We present a novel algorithm based on polar coordinates of each point. We are also first to study the performance of optimum region query. The expected performance of the proposed algorithm is optimal when the entire set is a not large. Extensive experiments illustrate the efficiency of our proposed algorithm.

7.3 Open Problems

In chapter 3, we propose three constrained region queries and corresponding approaches. While for different constrained queries, we have to use the corresponding algorithm to process them. Ideally, a system should automatically determine which algorithm need to be retrieved, but with the swift blossom of LBS and spatial database, there has to be more and more new queries demanded by the clients and users. As of a result, the underlying application will be more and more complex and difficult to be developed and maintained. Therefore, a unified algorithm for constrained range queries is required, which has to be flexible to handle various of range queries in spatial databases.

In chapter 4, we design two algorithms for moving range query and moving objects. But our algorithm only works when either query or objects of interest are static. The moving range query on moving objects of interest is still an open problem.

We analyze the performance of VD_k-kR in chapter 5, and we claim that the generation of the underlying structure, kth-order VD is expensive. Thus, we optimize the naive VD_k-kR by constructing kth-order VD locally. But intuitively, the domain of local kth-order VD is not optimal, which is another open problem.

In chapter 5, we present an algorithm for optimum region queries. We analyze the performance of steps, circle partition and arcs superposition. We find that most of the computation cost is on arcs superposition. We feel that identify the arcs covered by maximum number of other circles can be optimized.

7.4 Future Work

In this section, we propose several possible directions for future work.

Query processing in P2P and Ad-hoc networks: In this thesis, all algorithms and queries are based for client-server architectures. Whereas, it is interesting to adjust our proposed algorithms for P2P based networks or Ad-hoc networks [Muh09]. As our algorithms outperform existing works in most of the circumstances, we conjecture that Voronoi based algorithms in P2P or Ad-hoc networks will outperform the existing techniques as well. We also would like to investigate whether it is possible to let each peer handle a portion of Voronoi diagrams, which may reduce the load and computation cost of the server seriously. Another potential problem is how to let moving objects or queries cooperate to manage the relative positions in the P2P or Ad-hoc networks.

Spatial indexes for non-point objects: All the existing spatial indexes are constructed for point object in Euclidean distance, where the point objects are the abstraction of real-entities the extent of which can be ignored; While for non-point spatial objects, such as, a river, or an area with a specific vegetation, no existing algorithm can create a spatial index on them. A naive solution is that simulate any polygon as a rectangle and then constructed a R-Tree [Gut84] index on it. But this approach may extend the boundary of non-point object greatly, such as a river will be simulated as a rectangle, which causes the search result very inaccurate. Although we proposed many algorithms for polygon objects, but they are still lack of support on the bottom level.

Spatial queries in a high dimensional space: Most of the existing works of spatial query concentrate on the processing in a 2D space. The spatial query processing in a high dimensional space, e.g., 3D space, land surface, inside space of a building, has only become a hit ever since the past few year [STX08,XSP09]. It is a interesting problem to investigate how the higher dimensional Voronoi diagram or other geometric theories would help to improve the performance of the spatial query processing.

Appendix A

Fundamental Spatial Operations

In this appendix, we introduce the fundamental spatial operations. Let o_1 and o_2 be two spatial data objects (points, lines or regions), then we have the following operations where the topological relationship operations are illustrated in Fig. A.1

CONTAINS(o_1, o_2): Checks if o_1 have the CONTAINS topological relationship with o_2, if yes return true.

COVERS (o_1, o_2): Checks if o_1 have the COVERS topological relationship with o_2, if yes return true.

INTERSECTS (o_1, o_2):Checks if o_1 have the INTERSECTS topological relationship with o_2, if yes return true.

TOUCH (o_1, o_2): Checks if o_1 have the TOUCH topological relationship with o_2, if yes return true.

EQUAL (o_1, o_2): Checks if o_1 have the EQUAL topological relationship with o_2, if yes return true.

DISJOINT (o_1, o_2): Checks if o_1 have the DISJOINT topological relationship with o_2, if yes return true.

LENGTH(o_1): Return the length of a spatial object o_1 (only for line object).

AREA(o_1): Return the area of a spatial object o_1 (only for polygon object).

DISTANCE(o_1, o_2): Return the distance between two spatial objects o_1, o_2. If o_1 and/or o_2 are not points then the distance function relies on the definition

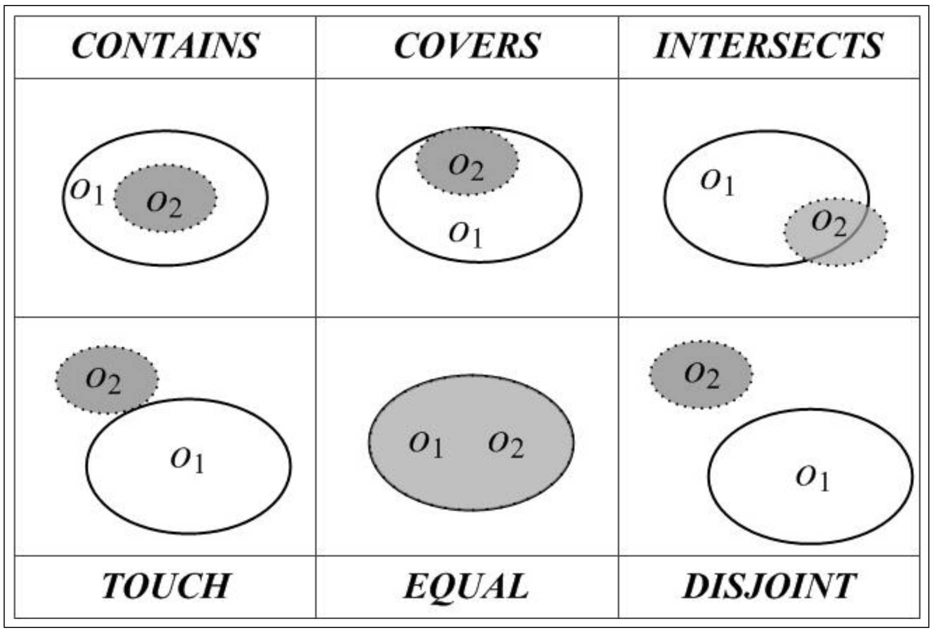


Figure A.1: Fundamental topological spatial relationship

of such distance. For example, a possible definition of the distance between two polygon objects is the minimum distance between them.

WITHIN-DISTANCE(o_1, o_2, dis): Checks if o_1 and o_2 are within a specific distance dis, if yes return true.

Appendix B

Simulation Source Codes

B.1 Simulation of Point-Expected Queries

Point.Java

```
package classes;
import java.util.ArrayList;
public class Point {
    boolean iostatus=true;
    ArrayList<Integer> adjacents=new ArrayList<Integer>();
    ArrayList<ArrayList> others=new ArrayList<ArrayList>();
    int id=0;
    double distToQ=0;
    boolean added=false;
    public Point(){}
    public Point
    (int id,  ArrayList<Integer> adjacents, ArrayList<ArrayList>others){
        this.id=id;
        this.adjacents=adjacents;
        this.others=others;
    }
    public void setIOStatus(boolean iostatus){this.iostatus=iostatus;}
    public void setDistToQ(double distToQ){this.distToQ=distToQ;}
    public ArrayList<Integer> getAdjacents(){return adjacents;}
    public ArrayList<ArrayList> getOthers() {return others;}
    public boolean getIOStatus(){return iostatus;}
    public int getID(){return id;}
    public double getDistToQ(){return distToQ;}
    public void setAdded(boolean added){this.added=added;}
    public boolean getAdded(){return added;}
    public void printPoint(){
        System.out.println("ID: "+id);
        System.out.println("Adjacents: "+adjacents);
        System.out.println("Others: "+others);
        System.out.println("IOStatus: "+iostatus);
        System.out.println("Distance To Q: "+distToQ);
    }
}
```

NVD. java

```
package classes;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
public class NVD {
    ArrayList<Point> generator=new ArrayList<Point>();
    ArrayList<ArrayList> ga=new ArrayList<ArrayList>();
    ArrayList<ArrayList> go=new ArrayList<ArrayList>();
    public NVD(int noGens, FileReader fr1, FileReader fr2)
    throws IOException{
        readAdjacents(fr1);
        readOthers(fr2);
        for(int i=0; i<noGens; i++){
            int id=i+1;
            generator.add(new Point(id,findAdjacents(id), findOthers(id)));
        }
    }
    public void readAdjacents(FileReader fr1) throws IOException{
        BufferedReader br=new BufferedReader(fr1);
        String s;
        while ((s=br.readLine())!=null){
          String f[] = s.split("\t");
          ArrayList<Integer> record=new ArrayList<Integer>();
          record.add(Integer.parseInt(f[0]));
          record.add(Integer.parseInt(f[1]));
          ga.add(record);
        }
        br.close();
    }
    public ArrayList<Integer> findAdjacents(int id){
        ArrayList<Integer> adjacents=new ArrayList<Integer>();
        for (int i=0; i<ga.size(); i++){
            int a=(Integer)ga.get(i).get(0);
            if(a==id) adjacents.add((Integer)ga.get(i).get(1));
        }
        return adjacents;
    }
    public void readOthers(FileReader fr2) throws IOException{
        BufferedReader br=new BufferedReader(fr2);
        String s;
        while ((s=br.readLine())!=null){
          String f[] = s.split("\t");
          ArrayList record=new ArrayList();
          record.add(Integer.parseInt(f[0]));
          record.add(Integer.parseInt(f[1]));
          record.add(Double.parseDouble(f[2]));
          go.add(record);
        }
        br.close();
    }
    public ArrayList<ArrayList> findOthers(int id){
        ArrayList<ArrayList> others=new ArrayList<ArrayList>();
        for (int i=0; i<go.size(); i++){
            int a=(Integer)go.get(i).get(0);
            if(a==id) {
                ArrayList record=new ArrayList();
                record.add((Integer)go.get(i).get(1));
                record.add((Double)go.get(i).get(2));
                others.add(record);
            }
        }
        return others;
    }
    public void printGens(){
        for (int i=0; i<generator.size(); i++){
            System.out.println("PID "+generator.get(i).getID());
            System.out.println("PID "+generator.get(i).getAdjacents());
            System.out.println("PID "+generator.get(i).getOthers());
        }
    }
    public ArrayList<Point> getGens(){
        return generator;
    }
}
```

TCR

```
package vrs;
import classes.NVD;
import classes.Point;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.TreeSet;
public class Main {
```

```java
    static NVD aNVD=null;
    public static void main(String[] args) throws FileNotFoundException, IOException {
        for(int looptimes=0; looptimes<=100; looptimes++)
        {
            FileReader fr1=new FileReader("D:/GIS Performance/TCR/data/20_Adjacent.
         txt");
            FileReader fr2=new FileReader("D:/GIS Performance/TCR/data/20
         _Distance_Time_M_Cogestion.txt");
            int noGens=20;
            aNVD=new NVD(noGens,fr1,fr2);
            Point q=generateQueryPoint(noGens);
            double e=20;
            double t=0.5;
            ArrayList<Point> distanceQpre=VRSDistance(q, e);
            ArrayList<Point> timeQpre=VRSTime(q, t);
            System.out.println(distanceQpre.size());
            System.out.println(timeQpre.size());
            ArrayList<Point> result=getIntersection(distanceQpre, timeQpre);
            System.out.println("**** FINAL RESULT ****");
            for(int i=0; i<result.size(); i++){
                result.get(i).printPoint();
            }
        }
    }
    public static ArrayList<Point> getIntersection(ArrayList<Point> a, ArrayList<Point> b)
         {
        ArrayList<Point> result=new ArrayList<Point>();
        for(int i=0; i<a.size(); i++){
            if(b.contains(a.get(i))) result.add(a.get(i));
        }
        return result;
    }
    public static ArrayList<Point> VRSDistance(Point q, double e){
        q.printPoint();
        System.out.println("***********");
        ArrayList<Point> Qpre= new ArrayList<Point>();
        Qpre.add(q);
        addToQpre(q, Qpre);
        sortQpre(q, Qpre, true);
        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getDistToQ()>e){Qpre.get(i).setIOStatus(false);}
        }
        Point inPoint=new Point();
        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getIOStatus()==true){
                inPoint=Qpre.get(i);
                break;
            }
        }
        int size=0;
        do{
            size=Qpre.size();
            for (int i=0; i<Qpre.size(); i++){
                if(Qpre.get(i).getIOStatus()==true && Qpre.get(i).getAdded()==false){
                    addToQpre(Qpre.get(i), Qpre);
                    Qpre.get(i).setAdded(true);
                }
            }
            sortQpre(q, Qpre, true);
            int inIndex=Qpre.indexOf(inPoint);
            for(int i=inIndex-1; i>-1; i--) {
                if(Qpre.get(i).getDistToQ()>e){
                    inPoint=Qpre.get(i+1);
                    break;
                }
            }
            inIndex=Qpre.indexOf(inPoint);
            for (int i=0; i<inIndex; i++){
                Qpre.get(i).setIOStatus(false);
            }
        }while(Qpre.size()!=size);
        ArrayList<Point>result=new ArrayList<Point>();
        for(int i=0; i<Qpre.size(); i++)
            if(Qpre.get(i).getIOStatus()==true){
            Qpre.get(i).printPoint();
            result.add(Qpre.get(i));
            }
        return result;
    }
    public static ArrayList<Point> VRSTime(Point q, double t){
        System.out.println("***********");
        ArrayList<Point> Qpre= new ArrayList<Point>();
        Qpre.add(q);
        addToQpre(q, Qpre);
        sortQpre(q, Qpre, false);
        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getTimeToQ()>t){Qpre.get(i).setIOStatus(false);}
        }
        Point inPoint=new Point();
        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getIOStatus()==true){
                inPoint=Qpre.get(i);
                break;
            }
        }
        int size=0;
        do{
            size=Qpre.size();
            for (int i=0; i<Qpre.size(); i++){
                if(Qpre.get(i).getIOStatus()==true && Qpre.get(i).getAdded()==false){
                    addToQpre(Qpre.get(i), Qpre);
                    Qpre.get(i).setAdded(true);
                }
            }
            sortQpre(q, Qpre, false);
```

```
                int inIndex=Qpre.indexOf(inPoint);
                for(int i=inIndex−1; i>−1; i−−) {
                    if(Qpre.get(i).getTimeToQ()>t){
                        inPoint=Qpre.get(i+1);
                        break;
                    }
                }
                inIndex=Qpre.indexOf(inPoint);
                for (int i=0; i<inIndex; i++){
                    Qpre.get(i).setIOStatus(false);
                }
            }while(Qpre.size()!=size);
            ArrayList<Point>result=new ArrayList<Point>();
            for(int i=0; i<Qpre.size(); i++)
                if(Qpre.get(i).getIOStatus()==true){
                Qpre.get(i).printPoint();
                result.add(Qpre.get(i));
                }
            return result;
        }
        public static void addToQpre(Point q, ArrayList<Point> Qpre){
            for(int i=0; i<q.getAdjacents().size(); i++) {
                int id1=q.getAdjacents().get(i);
                for(int j=0; j<aNVD.getGens().size(); j++){
                    int id2=aNVD.getGens().get(j).getID();
                    if(id1==id2)Qpre.add(aNVD.getGens().get(j));
                }
            }
        }
        public static Point generateQueryPoint(int noGens){
            Random r=new Random();
            int id=r.nextInt(noGens);
            return aNVD.getGens().get(id);
        }
        public static void sortQpre(Point q, ArrayList<Point> Qpre, boolean distance){
```

```
        Map aMap=new LinkedHashMap();
        for(int i=0; i<Qpre.size(); i++){
            aMap.put(Qpre.get(i),getDist(q,Qpre.get(i), distance));
        }
        List mapKeys = new ArrayList(aMap.keySet());
        List mapValues = new ArrayList(aMap.values());
        TreeSet sortedSet = new TreeSet(mapValues);
        Object[] sortedArray = sortedSet.toArray();
        Qpre.clear();

        for(int i=sortedArray.length; i>0;){
            double d=(Double)sortedArray[--i];
            Point a=(Point)mapKeys.get(mapValues.indexOf(d));
            if(distance)a.setDistToQ(d);
            else a.setTimeToQ(d);
            Qpre.add(a);
        }
    }
    public static double getDist(Point a, Point b, boolean distance){

        for(int i=0; i<aNVD.getGens().size(); i++){
            int id1=aNVD.getGens().get(i).getID();
            if(id1==a.getID()){
                for(int j=0; j<aNVD.getGens().get(i).getOthers().size(); j++)
                {
                    int id2=(Integer)aNVD.getGens().get(i).getOthers().get(j).get(0);
                    if(id2==b.getID()&&distance) return (Double)aNVD.getGens().get(i).
        getOthers().get(j).get(1);
                    if(id2==b.getID()&&!distance) return (Double)aNVD.getGens().get(i).
        getOthers().get(j).get(2);
                }
            }
        }
        return 0;
    }
```

```
}
```

RCR

```
package vrs;
import classes.NVD;
import classes.Point;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.TreeSet;
public class Main {
    static NVD aNVD=null;
    public static void main(String[] args) throws FileNotFoundException, IOException {
      for(int looptime=0; looptime<1000; looptime++){
        FileReader fr1=new FileReader("D:/GIS Performance/RCR/data/20-RCR_Adjacent.
         txt");
        FileReader fr2=new FileReader("D:/GIS Performance/RCR/data/20-RCR_4_Inside.
         txt");
        FileReader fr3=new FileReader("D:/GIS Performance/RCR/data/20-RCR_Distance.
         txt");
        int noGens=5;
        aNVD=new NVD(noGens,fr1,fr2,fr3);
        Point q=generateQueryPoint(noGens);
        double e=1;
        System.out.println("***** QUERY *****");
        q.printPoint();
        ArrayList<Integer> result=RCR(q, e);
        System.out.println("***** RESULT *****");
        Collections.sort(result);
        System.out.println( result );
    }
    }
    public static ArrayList<Integer> RCR(Point q, double e){
        ArrayList<Integer> PQ=new ArrayList<Integer>();
        ArrayList<Point> RQ=new ArrayList<Point>();
        ArrayList<Point> pRQ=new ArrayList<Point>();
        ArrayList<Point> Qpre=VRS(q, e);
        setDmax(Qpre, q);
        for(int i=0; i<Qpre.size(); i++){
            double dmax=Qpre.get(i).getDistToQmax();
            double dmin=Qpre.get(i).getDistToQ();
            if(dmax<e){
                if(testAP(dmax,dmin, e,Qpre.get(i)))
                RQ.add(Qpre.get(i));
            }
            else pRQ.add(Qpre.get(i));
        }
        for(int i=0; i<RQ.size(); i++){
            for(int j=0; j<RQ.get(i).getInners().size(); j++){
                if(!PQ.contains((Integer)RQ.get(i).getInners().get(j).get(1)))
                PQ.add((Integer)RQ.get(i).getInners().get(j).get(1));
            }
        }
        for(int i=0; i<pRQ.size(); i++){
            double newE=e-pRQ.get(i).getDistToQ();
            for(int j=0; j<pRQ.get(i).getInners().size(); j++){
                double d=(Double)pRQ.get(i).getInners().get(j).get(2);
                if(d<=newE&&!PQ.contains((Integer)pRQ.get(i).getInners().get(j).get(1)))PQ.
         add((Integer)pRQ.get(i).getInners().get(j).get(1));
            }
        }
        return PQ;
    }
    public static boolean testAP(double dmax, double dmin, double e, Point p){
        ArrayList<Integer> apList=new ArrayList<Integer>();
        for(int i=0; i<p.getInners().size(); i++)
        {
            if(!apList.contains((Integer)p.getInners().get(i).get(0)))
                apList.add((Integer)p.getInners().get(i).get(0));
        }
        ArrayList<Double> a=new ArrayList<Double>();
        ArrayList<Double> b=new ArrayList<Double>();
        for(int j=0; j<p.getInners().size(); j++){
            int x=(Integer)p.getInners().get(j).get(0);
            if(apList.get(0)==x) a.add((Double)p.getInners().get(j).get(2));
            if(apList.get(1)==x) b.add((Double)p.getInners().get(j).get(2));
        }
        Collections.sort(a);
        Collections.sort(b);
        if(dmin+a.get(a.size()-1)>e) return false;
        if(dmax+b.get(b.size()-1)>e) return false;
        return true;
    }
    public static void setDmax(ArrayList<Point> Qpre, Point q){
        for(int i=0; i<Qpre.size(); i++){
            double dmax=getDmax(Qpre.get(i), q);
            Qpre.get(i).setDistToQmax(dmax);
        }
    }
    public static double getDmax(Point a, Point b){
        for(int i=0; i<aNVD.getGens().size(); i++){
            int id1=aNVD.getGens().get(i).getID();
            if(id1==a.getID()){
                for(int j=0; j<aNVD.getGens().get(i).getOthers().size(); j++)
                {
                    int id2=(Integer)aNVD.getGens().get(i).getOthers().get(j).get(0);
                    if(id2==b.getID()) return (Double)aNVD.getGens().get(i).getOthers().get
        (j).get(2);
                }
            }
        }
        return 0;
    }
    public static ArrayList<Point> VRS(Point q, double e){
        ArrayList<Point> Qpre= new ArrayList<Point>();
        Qpre.add(q);
        addToQpre(q, Qpre);
        q.setAdded(true);
        sortQpre(q, Qpre);
        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getDistToQ()>e){Qpre.get(i).setIOStatus(false);}
        }
        Point inPoint=new Point();
        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getIOStatus()==true){
                inPoint=Qpre.get(i);
                break;
            }
        }
        int size=0;
        do{
            size=Qpre.size();
            for (int i=0; i<Qpre.size(); i++){
                if(Qpre.get(i).getIOStatus()==true && Qpre.get(i).getAdded()==false){
                    Qpre.get(i).setAdded(true);
                    addToQpre(Qpre.get(i), Qpre);
                }
            }
            sortQpre(q, Qpre);
            int inIndex=Qpre.indexOf(inPoint);
            for(int i=inIndex-1; i>-1; i--) {
                if(Qpre.get(i).getDistToQ()>e){
                    inPoint=Qpre.get(i+1);
                    break;
                }
                else
                {
                    inPoint=Qpre.get(i);
                }
            }
            inIndex=Qpre.indexOf(inPoint);
            for (int i=0; i<inIndex; i++){
                Qpre.get(i).setIOStatus(false);
            }
        }while(Qpre.size()!=size);
        ArrayList<Point> result=new ArrayList<Point>();
        for(int i=0; i<Qpre.size(); i++)
            if(Qpre.get(i).getIOStatus()==true)
                result.add(Qpre.get(i));
        return result;
    }
    public static void addToQpre(Point q, ArrayList<Point> Qpre){
        for(int i=0; i<q.getAdjacents().size(); i++) {
            int id1=q.getAdjacents().get(i);
            for(int j=0; j<aNVD.getGens().size(); j++){
                int id2=aNVD.getGens().get(j).getID();
                if(id1==id2&&!Qpre.contains(aNVD.getGens().get(j)))Qpre.add(aNVD.
        getGens().get(j));
            }
        }
    }
    public static Point generateQueryPoint(int noGens){
        Random r=new Random();
        int id=r.nextInt(noGens);
        //int id=1;
        return aNVD.getGens().get(id);
    }
    public static void sortQpre(Point q, ArrayList<Point> Qpre){
        Map aMap=new LinkedHashMap();
        for(int i=0; i<Qpre.size(); i++){
            aMap.put(Qpre.get(i),getDist(q,Qpre.get(i)));
        }
        List mapKeys = new ArrayList(aMap.keySet());
        List mapValues = new ArrayList(aMap.values());
        TreeSet sortedSet = new TreeSet(mapValues);
        Object[] sortedArray = sortedSet.toArray();
        Qpre.clear();
        for(int i=sortedArray.length; i>0;){
            double d=(Double)sortedArray[--i];
            Point a=(Point)mapKeys.get(mapValues.indexOf(d));
            a.setDistToQ(d);
            Qpre.add(a);
        }
    }
    public static double getDist(Point a, Point b){
        for(int i=0; i<aNVD.getGens().size(); i++){
            int id1=aNVD.getGens().get(i).getID();
            if(id1==a.getID()){
                for(int j=0; j<aNVD.getGens().get(i).getOthers().size(); j++)
                {
                    int id2=(Integer)aNVD.getGens().get(i).getOthers().get(j).get(0);
                    if(id2==b.getID()) return (Double)aNVD.getGens().get(i).getOthers().get
         (j).get(1);
                }
            }
        }
        return 0;
    }
}
```

kCR

```
package vrs;
import classes.NVD;
import classes.Point;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.TreeSet;
public class Main {
    static NVD aNVD=null;
    public static void main(String[] args) throws FileNotFoundException, IOException {
    double sumfe = 0;
    for(int looptimes=0; looptimes<1000; looptimes++)
    {
        FileReader fr1=new FileReader("D:/GIS Performance/kCR/data/20_Adjacent.txt");
        FileReader fr2=new FileReader("D:/GIS Performance/kCR/data/20_Distance.txt");
        int noGens=20;
        aNVD=new NVD(noGens,fr1,fr2);
        Point q=generateQueryPoint(noGens);
        System.out.println("***** QUERY POINT *****");
        q.printPoint();
        double e=5;
        int k=5;
        ArrayList<Point> Qpre=VRS(q, e);
        ArrayList<Point> Qek=new ArrayList<Point>();
        double fe=0.0;
        double dismax=Qpre.get(0).getDistToQ();
        int countQ=Qpre.size();
        if(k<=Qpre.size()){
            Qek=findKNN(Qpre, k);
            fe=e;
            System.out.println("***** IN RANGE *****");
            for (int i=0; i<Qek.size(); i++) Qek.get(i).printPoint(); }
        else{
            --k;
            ArrayList<Point> Qk=findKNN(q.getOthers(), k, q);
            fe=Qk.get(Qk.size()-1).getDistToQ();
            int l= k;
            boolean accepted=isAcceptable(Qk.get(l-1).getDistToQ(),
            dismax, e, l, k, countQ);
            while(!accepted && l>countQ){
                Qk=subtractPoint(Qk, Qk.get(l-1));
                l--;
                accepted=isAcceptable(Qk.get(l-1).getDistToQ(), dismax, e, l, k, countQ);
                fe=Qk.get(l-1).getDistToQ();
            }
            Qek=Qk;
            System.out.println("***** IN RANGE *****");
            for (int i=0; i<Qpre.size(); i++) Qpre.get(i).printPoint();
            System.out.println("***** OUT RANGE *****");
            for (int i=0; i<Qek.size(); i++) {
                if(Qek.get(i).getIOStatus()==false) Qek.get(i).printPoint();
            }
            sumfe = sumfe+fe;
            //System.out.println("fe: "+fe);
            //System.out.println("sumfe: "+sumfe);
        }
    }
    double avefe = sumfe/1000;
    System.out.println("***** Factual Range *****");
    System.out.println("avefe: "+avefe);
    }
    public static ArrayList<Point> subtractPoint(ArrayList<Point> Qk, Point p){
        Qk.remove(Qk.indexOf(p));
        return Qk;
    }
    public static boolean isAcceptable(double dPl, double dismax, double e, int l, int k,
        int countQ){
        double drel=(dPl-dismax)/e;
        double l1 = Double.parseDouble(Integer.toString(l));
        double k1 = Double.parseDouble(Integer.toString(k));
        double countQ1 =  Double.parseDouble(Integer.toString(countQ));
        double drkl=(l1-countQ1+1)/k1;
        if(drel<=drkl) return true;
        return false;
    }
    public static ArrayList<Point> VRS(Point q, double e){
        System.out.println("***** VRS *****");
        ArrayList<Point> Qpre= new ArrayList<Point>();
        Qpre.add(q);

        addToQpre(q, Qpre);
        sortQpre(q, Qpre);

        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getDistToQ()>e){Qpre.get(i).setIOStatus(false);}
        }
        Point inPoint=new Point();
        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getIOStatus()==true){
                inPoint=Qpre.get(i);
                break;
            }
        }
        int size=0;
        do{
            size=Qpre.size();
            for (int i=0; i<Qpre.size(); i++){
                if(Qpre.get(i).getIOStatus()==true && Qpre.get(i).getAdded()==false){
                    addToQpre(Qpre.get(i), Qpre);
                    Qpre.get(i).setAdded(true);
                }
            }
            sortQpre(q, Qpre);
            int inIndex=Qpre.indexOf(inPoint);
            for(int i=inIndex-1; i>-1; i--) {
                if(Qpre.get(i).getDistToQ()>e){
                    inPoint=Qpre.get(i+1);
                    break;
                }
                else inPoint=Qpre.get(i);
            }
            inIndex=Qpre.indexOf(inPoint);
            for (int i=0; i<inIndex; i++){
                Qpre.get(i).setIOStatus(false);
            }
        }while(Qpre.size()!=size);
        ArrayList<Point> result=new ArrayList<Point>();
        for(int i=0; i<Qpre.size(); i++)
            if(Qpre.get(i).getIOStatus()==true){
                Qpre.get(i).printPoint();
                result.add(Qpre.get(i));
            }
        return result;
    }
    public static void addToQpre(Point q, ArrayList<Point> Qpre){
        for(int i=0; i<q.getAdjacents().size(); i++) {
            int id1=q.getAdjacents().get(i);
            for(int j=0; j<aNVD.getGens().size(); j++){
                int id2=aNVD.getGens().get(j).getID();
                if(id1==id2)Qpre.add(aNVD.getGens().get(j));
            }
        }
    }
    public static Point generateQueryPoint(int noGens){
        Random r=new Random();
        int id=r.nextInt(noGens);
        return aNVD.getGens().get(id);
    }
    public static void sortQpre(Point q, ArrayList<Point> Qpre){
        Map aMap=new LinkedHashMap();
        for(int i=0; i<Qpre.size(); i++){
            aMap.put(Qpre.get(i),getDist(q,Qpre.get(i)));
        }
        List mapKeys = new ArrayList(aMap.keySet());
        List mapValues = new ArrayList(aMap.values());
        TreeSet sortedSet = new TreeSet(mapValues);
        Object[] sortedArray = sortedSet.toArray();
        Qpre.clear();
        for(int i=sortedArray.length; i>0;){
            double d=(Double)sortedArray[--i];
            Point a=(Point)mapKeys.get(mapValues.indexOf(d));
            a.setDistToQ(d);
            Qpre.add(a);
        }
    }
    public static double getDist(Point a, Point b){
        for(int i=0; i<aNVD.getGens().size(); i++){
            int id1=aNVD.getGens().get(i).getID();
            if(id1==a.getID()){
                for(int j=0; j<aNVD.getGens().get(i).getOthers().size(); j++)
                {
                    int id2=(Integer)aNVD.getGens().get(i).getOthers().get(j).get(0);
                    if(id2==b.getID()) return (Double)aNVD.getGens().get(i).getOthers().get
         (j).get(1);
                }
            }
        }
        return 0;
    }
    public static ArrayList<Point> findKNN(ArrayList<Point> VRSResult, int k ){
        if (k==VRSResult.size())return VRSResult;
        ArrayList<Point> result = new ArrayList<Point>();
        int lastIndex=VRSResult.size();
        for(int i=0; i<k; i++){
            result.add(VRSResult.get(--lastIndex));
        }
        return result;
    }
    public static ArrayList<Point> findKNN(ArrayList<ArrayList> others, int k, Point q){
        ArrayList<Point> result=new ArrayList<Point>();
        ArrayList<Integer> kID=new ArrayList<Integer>();
        Map aMap=new LinkedHashMap();
        for(int i=0; i<others.size(); i++){
            aMap.put((Integer)others.get(i).get(0),(Double)others.get(i).get(1));
        }
        List mapKeys = new ArrayList(aMap.keySet());
        List mapValues = new ArrayList(aMap.values());
        TreeSet sortedSet = new TreeSet(mapValues);
        Object[] sortedArray = sortedSet.toArray();
        for(int i=0; i<sortedArray.length; i++){
            kID.add((Integer)mapKeys.get(mapValues.indexOf(sortedArray[i])));
        }
        for(int i=0; i<k; i++){
            for(int j=0; j<aNVD.getGens().size(); j++){
                int id=(Integer)aNVD.getGens().get(j).getID();
                if(id==kID.get(i)){
                    aNVD.getGens().get(j).setDistToQ(getDist(aNVD.getGens().get(j),q));
                    result.add(aNVD.getGens().get(j));
                }
            }
        }
        return result;
    }
}
```

VCR and Monitoring

```
package vrs;
import classes.NVD;
import classes.Point;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.TreeSet;
public class Main {
    static NVD aNVD=null;
    public static void main(String[] args) throws FileNotFoundException, IOException {
        FileReader fr1=new FileReader("D:/GIS Performance/VRS/data/5_Adjacent.txt");
         //TO BE CHANGED
        FileReader fr2=new FileReader("D:/GIS Performance/VRS/data/5_Distance.txt");
         //TO BE CHANGED
        int noGens=5; // TO BE CHANGED
        aNVD=new NVD(noGens,fr1,fr2);
        Point q=generateQueryPoint(noGens);
        double e=15; //TO BE CHANGED
        VRS(q, e);
    }
    public static void VRS(Point q, double e){
        q.printPoint();
        System.out.println("***********");
        ArrayList<Point> Qpre= new ArrayList<Point>();
        Qpre.add(q);
        addToQpre(q, Qpre);
        sortQpre(q, Qpre);
        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getDistToQ()>e){Qpre.get(i).setIOStatus(false);}
        }
        Point inPoint=new Point();
        for (int i=0; i<Qpre.size(); i++){
            if(Qpre.get(i).getIOStatus()==true){
                inPoint=Qpre.get(i);
                break;
            }
        }
        int size=0;
        do{
            size=Qpre.size();
            for (int i=0; i<Qpre.size(); i++){
                if(Qpre.get(i).getIOStatus()==true && Qpre.get(i).getAdded()==false){
                    addToQpre(Qpre.get(i), Qpre);
                    Qpre.get(i).setAdded(true);
                }
            }
            sortQpre(q, Qpre);
            int inIndex=Qpre.indexOf(inPoint);
            for(int i=inIndex-1; i>-1; i--) {
                if(Qpre.get(i).getDistToQ()>e){
                    inPoint=Qpre.get(i+1);
                    break;
                }
                else inPoint=Qpre.get(i);
            }
            inIndex=Qpre.indexOf(inPoint);
            for (int i=0; i<inIndex; i++){
                Qpre.get(i).setIOStatus(false);
            }
        }while(Qpre.size()!=size);
        for(int i=0; i<Qpre.size(); i++)
            if(Qpre.get(i).getIOStatus()==true)
            Qpre.get(i).printPoint();
    }
    public static void addToQpre(Point q, ArrayList<Point> Qpre){
        for(int i=0; i<q.getAdjacents().size(); i++) {
            int id1=q.getAdjacents().get(i);
            for(int j=0; j<aNVD.getGens().size(); j++){
                int id2=aNVD.getGens().get(j).getID();
                if(id1==id2&&!Qpre.contains(aNVD.getGens().get(j)))Qpre.add(aNVD.
        getGens().get(j));
            }
        }
    }
    public static Point generateQueryPoint(int noGens){
        Random r=new Random();
        int id=r.nextInt(noGens);
        return aNVD.getGens().get(id);
    }
    public static void sortQpre(Point q, ArrayList<Point> Qpre){
        Map aMap=new LinkedHashMap();
        for(int i=0; i<Qpre.size(); i++){
            aMap.put(Qpre.get(i),getDist(q,Qpre.get(i)));
        }
        List mapKeys = new ArrayList(aMap.keySet());
        List mapValues = new ArrayList(aMap.values());
        TreeSet sortedSet = new TreeSet(mapValues);
        Object[] sortedArray = sortedSet.toArray();
        Qpre.clear();
        for(int i=sortedArray.length; i>0;){
            double d=(Double)sortedArray[--i];
            Point a=(Point)mapKeys.get(mapValues.indexOf(d));
            a.setDistToQ(d);
            Qpre.add(a);
        }
    }
    public static double getDist(Point a, Point b){
        for(int i=0; i<aNVD.getGens().size(); i++){
            int id1=aNVD.getGens().get(i).getID();
            if(id1==a.getID()){
                for(int j=0; j<aNVD.getGens().get(i).getOthers().size(); j++)
                {
                    int id2=(Integer)aNVD.getGens().get(i).getOthers().get(j).get(0);
                    if(id2==b.getID()) return (Double)aNVD.getGens().get(i).getOthers().get
        (j).get(1);
                }
            }
        }
        return 0;
    }
}
```

B.2 Simulation of Region-Expected Queries

$\mathcal{VD}_k$-k**R**

```
public double distance(Point2D site1, Point2D site2) {
    return Math.sqrt((Math.abs(site1.getX() - site2.getX()) * Math
        .abs(site1.getX() - site2.getX()))
        + (Math.abs(site1.getY() - site2.getY()) * Math.abs(site1
            .getY() - site2.getY())));
    }
    public List<VEdge> myEdges;
    public List<Point2D> myVertices;
    public List<VEdge> jasonResult = new LinkedList<VEdge>();
    public int numberOfKeptVertices = 0;
    public void JasonsAlgorithm(List<Point2D> selectedSites) {
    Set<VEdge> allEdges = this.edges;
    System.out.println("All edges: " + allEdges);
    PointSet allSites = this.sites;
    List<VEdge> myEdges = new LinkedList<VEdge>();
    List<Point2D> myVertices = new LinkedList<Point2D>();
    for (Point2D site : selectedSites)
        for (VEdge edge : allEdges)
        if (edge.critical1 == site | edge.critical2 == site)
            myEdges.add(edge);
    this.myEdges = myEdges;
    System.out.println("Found edges: " + myEdges);
    for (VEdge edge : myEdges) {
        VVertex[] edgeVertices = { edge.v1, edge.v2 };
        for (VVertex vertex : edgeVertices) {
        if (!myVertices.contains(vertex)) {
            myVertices.add(vertex);
            double maxDistance = 0;
            for (Point2D site : selectedSites ) {
            double theDistance = this.distance(vertex, site);
            if (theDistance > maxDistance)
                maxDistance = theDistance;
            }
            for (Point2D site : allSites )
            if (distance( site , vertex) < maxDistance - 0.00001
                & !selectedSites .contains( site ))
                myVertices.remove(vertex);
        }
        }
    }
    this.myVertices = myVertices;
    System.out.println("My vertices: " + myVertices);
    this.numberOfKeptVertices = myVertices.size();
    if (this.numberOfKeptVertices != 0) {
        List<VEdge> edgesKept = new ArrayList<VEdge>();
        for (Point2D vertex : myVertices)
        for (VEdge edge : allEdges) {
            ArrayList<VVertex> edgeVertices = new ArrayList<VVertex>();
            edgeVertices.add(edge.v1);
            edgeVertices.add(edge.v2);
            if (edgeVertices.contains(vertex)) {
            edgesKept.add(edge);
            edgeVertices.remove(vertex);
            VVertex theVertex = edgeVertices.get(0);
            if (theVertex. isAtInfinity ()) {
                double infinitX, infinitY ;
```

```
                    infinitX  = vertex.getX() + theVertex.getX() * 5000;
                    infinitY  = vertex.getY() + theVertex.getY() * 5000;
                    Point2D theInfinityPoint = new Point2D(infinitX,
                        infinitY);
                    double maxDistance = 0;
                    for (Point2D site : selectedSites) {
                    double theDistance = this.distance(
                        theInfinityPoint, site);
                    if (theDistance > maxDistance)
                        maxDistance = theDistance;
                    }
                    System.out
                        .println("Max distance is:" + maxDistance);
                    for (Point2D site : allSites) {
                    System.out.println("Distance from "
                        + theInfinityPoint + " to site " + site
                        + " is : "
                        + distance(site, theInfinityPoint));
                    if (distance(site, theInfinityPoint) < maxDistance - 0.00001
                        & !selectedSites.contains(site)) {
                        edgesKept.remove(edge);
                        break;
                    }
                    }
                } else if (!myVertices.contains(theVertex))
                    edgesKept.remove(edge);
                }
            }
            this.jasonResult = edgesKept;
        } else {
            System.out.println("Area does not exist.");
        }
        }
}
```

$\mathcal{DT}$-kR

```
package knnregion;
import intersect2Dalgorithm.Intersect2DAlgorithm;
import intersect2Dalgorithm.Segment;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import convexhullalgorithm.ConvexHull;
import data.Point;
import data.Site;
import delaunayalgorithm.Voronoi;
public final class Algorithm {
    public Algorithm(double canvasWidth, double canvasHeight, List<Site> sites,
        List<Site> selectedSites) {
    this.canvasHeight = canvasHeight;
    this.canvasWidth = canvasWidth;
    this.sites = sites;
    this.selectedSites = selectedSites;
    }
    /**
     * Step 1: Generate DT for the sites.
     */
    public void step1() {
    Voronoi v = new Voronoi();
    v.generateVoronoi(this.sites);
    this.connections = v.connections;
    this.numberOfEdges = v.numberOfEdges;
    // System.out.println("Step 1 result: " + this.connections);
    System.out.println("Step 1 done!");
    }
    /**
     * For step 2.
     */
    private void traverse(Site selected) {
    for (Site connected : this.connections.get(selected))
```

```
        if (!connected.visited & this.selectedSites.contains(connected)) {
        connected.visited = true;
        traverse(connected);
        }
    }
    /**
     * Step 2: Test whether selected sites are interconnected.
     */
    public boolean step2() {
    this.traverse(this.selectedSites.get(0));
    this.interconnected = true;
    for (Site s : this.selectedSites)
        if (!s.visited) {
        this.interconnected = false;
        break;
        }
    // System.out.println("Step 2 result: " + this.interconnected
    // + this.selectedSites);
    return this.interconnected;
    }
    /**
     * Step 3: Generate convex hull of selected sites.
     */
    public void step3() {
    ConvexHull convexHull = new ConvexHull(selectedSites);
    this.convexHull = convexHull.chainHull();
    System.out.println("Step 3 result: " + this.convexHull);
    }
    /**
     * Step 4: Find neighbors of the convex hull vertices.
     */
    public List<Site> step4() {
    this.neighbours = new LinkedList<Site>();
    for (Site s : convexHull)
        for (Site connected : this.connections.get(s))
        if (!this.selectedSites.contains(connected)
            & !this.neighbours.contains(connected))
            this.neighbours.add(connected);
    // System.out.println("Step 4 result: " + this.neighbours);
    return this.neighbours;
    }
    /**
     * Determines the location of a point in relative to a line segment. Returns
     * > 0 if point is on left, returns < 0 if point is on right, return = 0 if
     * point is on the segment.
     */
    private double p2s(Segment theSegment, Point point) {
    return (theSegment.P1.x − theSegment.P0.x)
        * (point.y − theSegment.P0.y) − (point.x − theSegment.P0.x)
        * (theSegment.P1.y − theSegment.P0.y);
    }
    /**
     * Step 5:   .........
     */
    public List<Point> step5() throws Exception {
    this.polygon = new LinkedList<Point>();
    this.bisectors = new LinkedList<Point>();
    // Initially, the polygon is a rectangle of drawable area.
    this.polygon.add(new Point(this.canvasWidth, 0));
    this.polygon.add(new Point(0, 0));
    this.polygon.add(new Point(0, this.canvasHeight));
    this.polygon.add(new Point(this.canvasWidth, this.canvasHeight));
    LinkedList<Point> toBeRemoved = new LinkedList<Point>();
    Segment toIntersect = new Segment(new Point(0, 0), new Point(0, 0));
    Segment theIntersection = new Segment(new Point(0, 0), new Point(0, 0));
    Point middle;
    double yIntersection, xIntersection, yMax, k, intersectionP2S, polygonP2S, convexP2S;
    boolean canIntersect;
    for (Site onHull : this.convexHull)
        for (Site neighbour : this.neighbours) {
```

```
        // Make a bisector line
        k = -1
            / ((onHull.location.y - neighbour.location.y) / (onHull.location.x - neighbour.
         location.x));
        middle = new Point(
            (onHull.location.x + neighbour.location.x) / 2,
            (onHull.location.y + neighbour.location.y) / 2);
        System.out.println("Between: " + onHull + " and " + neighbour);
        // Intersect the polygon with the line
        if (k < 0) {
            yIntersection = -k * middle.x + middle.y;
            xIntersection = -middle.y / k + middle.x;
            toIntersect.P0 = new Point(xIntersection, 0);
            toIntersect.P1 = new Point(0, yIntersection);
        } else if (k >= 0) {
            yIntersection = -k * middle.x + middle.y;
            yMax = k * (this.canvasWidth - middle.x) + middle.y;
            toIntersect.P0 = new Point(this.canvasWidth, yMax);
            toIntersect.P1 = new Point(0, yIntersection);
        }
        canIntersect = new Intersect2DAlgorithm().intersect(
            toIntersect, this.polygon, theIntersection);
        System.out.println("To Intersect: " + toIntersect);
        if (canIntersect
            && !(theIntersection.P0.x == Double.NaN
                | theIntersection.P1.x == Double.NaN
                | theIntersection.P0.y == Double.NaN | theIntersection.P1.y == Double.NaN)
         ) {
            toBeRemoved.clear();
            System.out.println("The intersection: " + theIntersection);
            // Which side of polygon points shall remain?
            intersectionP2S = this
                .p2s(theIntersection, onHull.location);
            for (Point p : this.polygon) {
            polygonP2S = this.p2s(theIntersection, p);
            // Those points which are not on same side of the convex
            // hull are to be removed
            if (Math.abs(polygonP2S − intersectionP2S) > Math
                .abs(polygonP2S + intersectionP2S))
                toBeRemoved.add(p);
            }
            for (Point p : toBeRemoved)
            this.polygon.remove(p);
            // Store new intersections of the line and the polygon
            this.polygon.add(theIntersection.P0);
            this.polygon.add(theIntersection.P1);
            this.bisectors.add(theIntersection.P0);
            this.bisectors.add(theIntersection.P1);
            System.out.println("POLYGON: " + this.polygon);
            // For informative purpose, collect some statistics data
            ++this.numberOfIntersects;
            if (this.polygon.size() > this.maxPolygonVerticesNumber)
            this.maxPolygonVerticesNumber = this.polygon.size();
        } else {
            // If the intersection (or toIntersect, they're on same
            // line) is on opposite side of the convex hull, terminate
            // the algorithm, KNN region does not exist in such case
            polygonP2S = this.p2s(toIntersect, this.polygon.get(0));
            convexP2S = this.p2s(toIntersect, onHull.location);
            if (Math.abs(polygonP2S − convexP2S) > Math.abs(polygonP2S
                + convexP2S)) {
            throw new Exception();
            }
        }
        }
    System.out.println("Step 5 result: " + this.polygon);
    return this.polygon;
    }
    private double canvasWidth, canvasHeight;
    public List<Site> sites, selectedSites; // KNN algorithm input
    public Map<Site, List<Site>> connections; // Step 1 output
    public boolean interconnected; // Step 2 output
    public List<Site> convexHull; // Step 3 output
    public List<Site> neighbours; // Step 4 output
    public List<Point> bisectors; // Step 5 output
    public List<Point> polygon; // Step 5 output
    // For informative purpose
    public int numberOfIntersects = 0, maxPolygonVerticesNumber = 0,
        numberOfEdges = 0;
    public static void main(String args[]) {
    System.out.println(4 / 2);
    }
}
```

Point.java

```
package algorithm;
public final class Point {
    public Point(final double x, final double y) {
        this.x = x;
        this.y = y;
    }
    public double distanceTo(final Point other) {
        return Math.hypot(this.getX() − other.getX(), this.getY() − other.getY());
    }
    @Override
    public boolean equals(final Object other) {
        final Point p = (Point) other;
        return this.getX() == p.getX() && this.getY() == p.getY();
    }
    public double getX() {
        synchronized (this) {
            ++Point.accessCounter;
        }
        return this.x;
    }
    public double getY() {
        synchronized (this) {
            ++Point.accessCounter;
        }
        return this.y;
    }
    @Override
    public int hashCode() {
        return super.hashCode();
    }
    public void setX(final double x) {
        this.x = x;
    }
    public void setY(final double y) {
        this.y = y;
    }
    @Override
    public String toString() {
        return "(" + (int) this.getX() + ", " + (int) this.getY() + ")";
    }
    private double x, y;
    public static int accessCounter = 0;
}
```

Segment.java

```
package algorithm;
import java.util.ArrayList;
import java.util.List;
public final class Segment {
    public Segment(final double begin, final double end) {
        this.begin = begin;
        this.end = end;
    }
    @Override
    public boolean equals(final Object other) {
        final Segment s = (Segment) other;
        return this.getBegin() == s.getBegin() && this.getEnd() == s.getEnd();
    }
    public double getBegin() {
        return this.begin;
    }
    public double getEnd() {
        return this.end;
    }
    @Override
    public int hashCode() {
        return super.hashCode();
    }
    public Segment intersect(final Segment another) {
        if (this.getBegin() < another.getBegin()) {
            if (this.getEnd() <= another.getEnd()
                    & this.getEnd() >= another.getBegin()) {
                return new Segment(another.getBegin(), this.getEnd());
            } else if (this.getEnd() > another.getEnd()) {
                return new Segment(another.getBegin(), another.getEnd());
            } else {
                return null;
            }
        } else if (this.getBegin() <= another.getEnd()
                & this.getBegin() >= another.getBegin()) {
            if (this.getEnd() <= another.getEnd()
                    & this.getEnd() >= another.getBegin()) {
                return new Segment(this.getBegin(), this.getEnd());
            } else if (this.getEnd() > another.getEnd()) {
                return new Segment(this.getBegin(), another.getEnd());
            } else {
                return null;
            }
        } else {
            return null;
        }
    }
    public void setBegin(final double begin) {
        this.begin = begin;
    }
    public void setEnd(final double end) {
        this.end = end;
    }
    @Override
    public String toString() {
        return "(" + (int) this.getBegin() + " to " + (int) this.getEnd() + ")";
    }
    private double begin, end;
    static class MostIntersected {
        public MostIntersected(final int count, final List<Segment> segments) {
            this.count = count;
            this.segments = segments;
        }
        @Override
        public String toString() {
            return "Intersected " + this.count + " times: " + this.segments;
        }
        int count;
        final List<Segment> segments;
    }
    public static MostIntersected mostIntersected(final List<Segment> segments,
            final int count) {
        final List<Segment> next = new ArrayList<Segment>();
        boolean intersected = false;
        for (final Segment s1 : segments) {
            for (final Segment s2 : segments) {
                if (s1 == s2) {
                    continue;
                }
                final Segment intersection = s1.intersect(s2);
                if (intersection != null & !next.contains(intersection)) {
                    intersected = true;
                    next.add(intersection);
                }
            }
        }
        if (intersected) {
            return Segment.mostIntersected(next, count + 1);
        }
        return new MostIntersected(count, segments);
    }
    public static int accessCounter = 0;
}
```

Circle.java

```
package algorithm;
import algorithm.Segment.MostIntersected;
public final class Circle {
    public Circle(final Point centre) {
        this.centre = centre;
    }
    public double distanceTo(final Circle other) {
        return this.centre.distanceTo(other.getCentre());
    }
    @Override
    public boolean equals(final Object other) {
        final Circle c = (Circle) other;
        return this.centre.equals(c.centre);
    }
    public Point getCentre() {
        return this.centre;
    }
    public MostIntersected getMi() {
        return this.mi;
    }
    @Override
    public int hashCode() {
        return super.hashCode();
    }
    public Point[] intersect (final Circle other) {
        final double dx = other.centre.getX() − this.centre.getX();
        final double dy = other.centre.getY() − this.centre.getY();
        final double d = Math.hypot(dx, dy);
        if (d > Circle.radius ∗ 2 || d < 0.00000001) {
            return null;
        }
        final double a = d / 2;
        final double x2 = this.centre.getX() + dx ∗ a / d;
        final double y2 = this.centre.getY() + dy ∗ a / d;
        final double h = Math.sqrt(Circle.radius ∗ Circle.radius − a ∗ a);
        final double rx = −dy ∗ (h / d);
        final double ry = dx ∗ (h / d);
        return new Point[] { new Point(x2 + rx, y2 + ry),
                new Point(x2 − rx, y2 − ry) };
    }
    public Segment[] intersection(final Circle other) {
        final Point[] intersections = other.intersect(this);
        if (intersections != null) {
            final Point i1 = intersections[0], i2 = intersections[1];
            double arcBegin = Math.toDegrees(Math.atan2(i1.getY()
                    − this.getCentre().getY(), i1.getX() − this.getCentre().getX()));
            double arcEnd = Math.toDegrees(Math.atan2(i2.getY()
                    − this.getCentre().getY(), i2.getX() − this.getCentre().getX()));
            if (arcBegin < 0) {
                arcBegin += 360;
            }
            if (arcEnd < 0) {
                arcEnd += 360;
            }
            double lesser, greater;
```

```
                if (arcBegin < arcEnd) {
                    lesser = arcBegin;
                    greater = arcEnd;
                } else {
                    greater = arcBegin;
                    lesser = arcEnd;
                }
                if (greater − lesser > 180) {
                    return new Segment[] { new Segment(0, lesser),
                            new Segment(greater, 360) };
                }
                return new Segment[] { new Segment(lesser, greater) };
            }
            return null;
        }
    public void setMi(final MostIntersected mi) {
        this.mi = mi;
    }
    @Override
    public String toString() {
        return this.centre.toString();
    }
    private final Point centre;
    private MostIntersected mi;
    public static double radius = 0.0;
}
```

Optimum Region

```
package algorithm;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import algorithm.Segment.MostIntersected;
```

```
public final class Algorithm {
    static class Result {
        public Result(final Map<Circle, List<Segment>> all,
                final Map<Circle, List<Segment>> result, final int max) {
            this.all = all;
            this.result = result;
            this.max = max;
        }
        public final Map<Circle, List<Segment>> all, result;
        public final int max;
    }
    @SuppressWarnings("boxing")
    public static Result solve(final List<Circle> circles) {
        final Map<Circle, List<Segment>> all = new HashMap<Circle, List<Segment>>();
        final Map<Integer, List<Circle>> numberSegmentsCircle = new HashMap<Integer,
         List<Circle>>();
        for (final Circle c1 : circles) {
            final List<Segment> segments = new ArrayList<Segment>(20);
            for (final Circle c2 : circles) {
                if (c2 == c1 || c2.distanceTo(c1) > Circle.radius * 2) {
                    continue;
                }
                for (final Segment s : c1.intersection(c2)) {
                    segments.add(s);
                }
            }
            final MostIntersected mi = Segment.mostIntersected(segments, 0);
            all.put(c1, mi.segments);
            if (!numberSegmentsCircle.containsKey(mi.count)) {
                numberSegmentsCircle.put(mi.count, new ArrayList<Circle>(20));
            }
            numberSegmentsCircle.get(mi.count).add(c1);
        }
        final Integer max = Collections.max(numberSegmentsCircle.keySet());
        final Map<Circle, List<Segment>> result = new HashMap<Circle, List<Segment
         >>();
        for (final Circle c : numberSegmentsCircle.get(max)) {
            result.put(c, all.get(c));
        }
        return new Result(all, result, max);
    }
    public static Result solve2(final List<Circle> circles,
            final List<Circle> specified) {
        // ------------------ Step 1 ----------------------
        final Map<Circle, List<Segment>> all = new HashMap<Circle, List<Segment>>();
        final Map<Circle, List<Segment>> result = new HashMap<Circle, List<Segment
         >>();
        for (final Circle c : circles) {
            final List<Segment> cSegments = new ArrayList<Segment>();
            if (specified.contains(c)) {
                cSegments.add(new Segment(0, 360));
            }
            for (final Circle s : specified) {
                final Segment[] segments = c.intersection(s);
                if (segments == null) {
                    continue;
                }
                for (final Segment segment : segments) {
                    cSegments.add(segment);
                }
            }
            c.setMi(Segment.mostIntersected(cSegments, 1));
            all.put(c, new ArrayList<Segment>());
            result.put(c, c.getMi().segments);
        }
        final Map<Circle, List<Segment>> step2 = new HashMap<Circle, List<Segment
         >>();
        int maxCounter = 0;
        for (final Circle r : result.keySet()) {
            if (r.getMi().count != specified.size()) {
                continue;
            }
            final List<Segment> tmp = new ArrayList<Segment>();
            for (final Circle c : circles) {
                if (r == c) {
                    continue;
                }
                if (!specified.contains(c)) {
                    final Segment[] segments = r.intersection(c);
                    if (segments != null) {
                        for (final Segment s : segments) {
                            for (final Segment sOnR : r.getMi().segments) {
                                final Segment sIntersection = s.intersect(sOnR);
                                if (sIntersection != null) {
                                    tmp.add(sIntersection);
                                }
                            }
                        }
                    }
                }
            }
            if (tmp.size() == 0) {
                tmp.addAll(r.getMi().segments);
            } else if (!specified.contains(r)) {
                tmp.add(new Segment(0, 360));
            }
            final MostIntersected rMi = Segment.mostIntersected(tmp,
                    r.getMi().count + 1);
            r.setMi(rMi);
            if (r.getMi().count > maxCounter) {
                maxCounter = r.getMi().count;
            }
        }
        for (final Circle r : result.keySet()) {
            if (r.getMi().count == maxCounter) {
                step2.put(r, r.getMi().segments);
            }
        }
        return new Result(all, step2, maxCounter);
    }
}
```

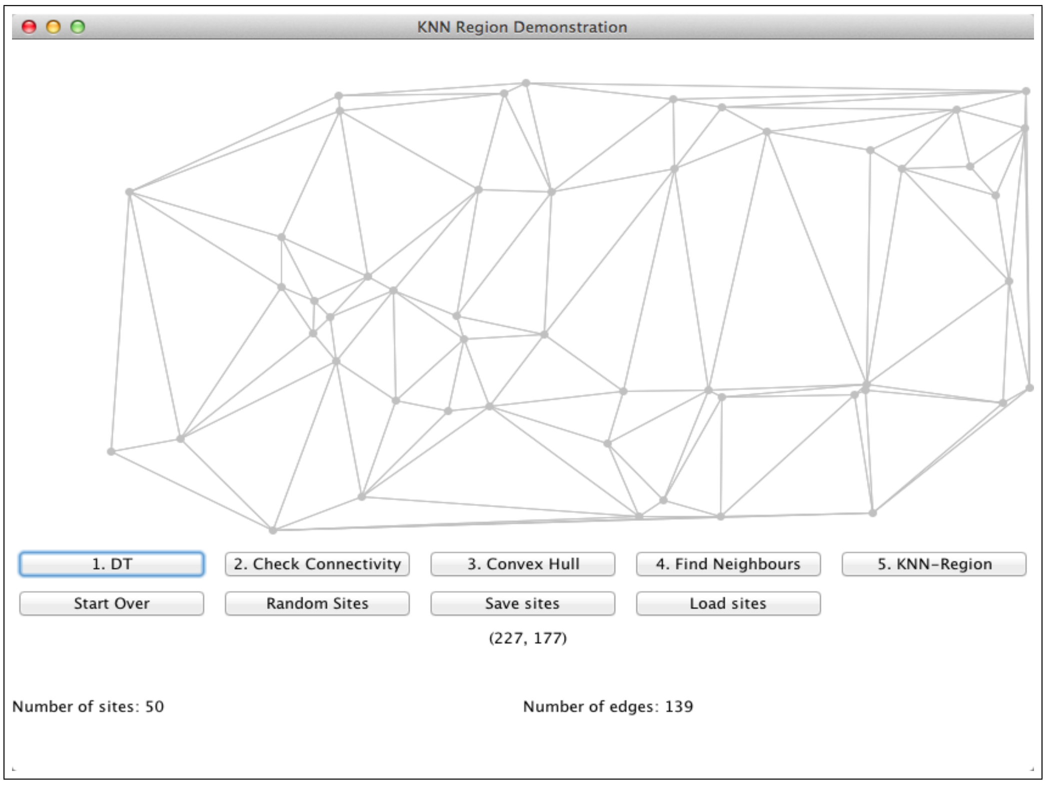
Appendix C

Visualization of Expected Regions

In this Appendix, the visualizations of the region-region expected algorithms are illustrated. Fig. C.1 to Fig. C.4 show the procedure of Delaunay triangulation based algorithm to identify the kNN region step by step.

Fig. C.5 demonstrates a kNN region retrieved by kth-order Voronoi diagram based algorithm which based on the algorithm of kth-order Voronoi diagram proposed by Prof. Andreas Pollak.

Fig. C.6 shows the visualization of optimum region finding by our proposed algorithm.

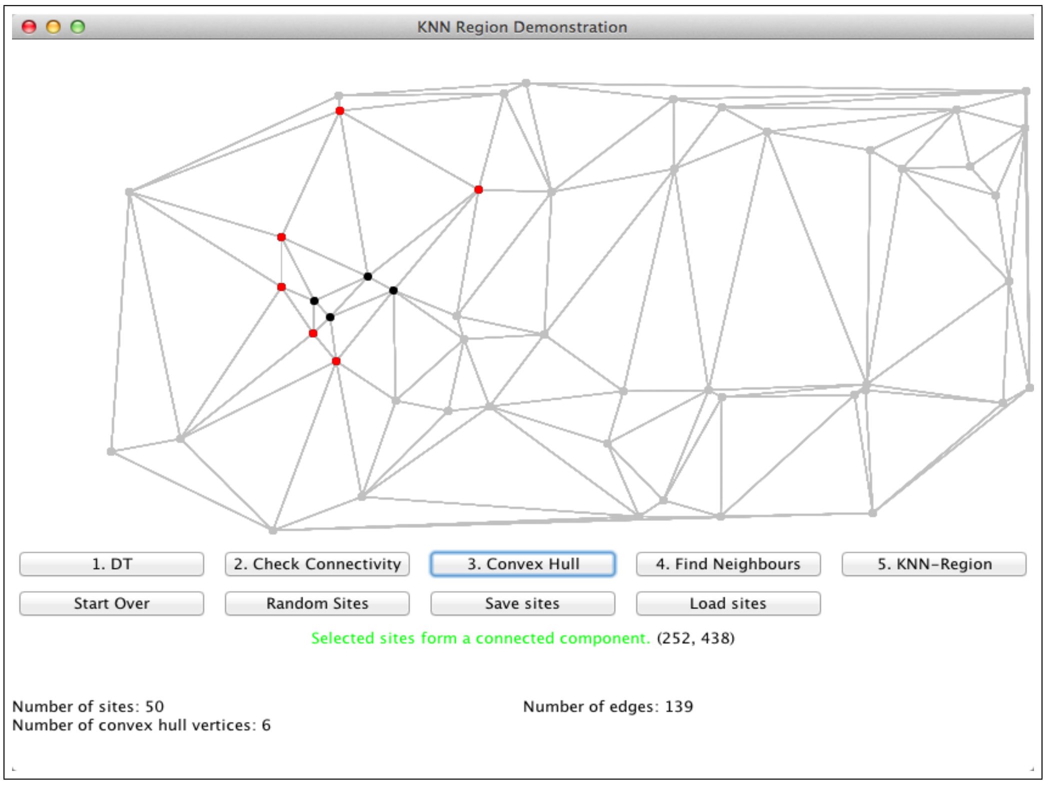


(227, 177)

Number of sites: 50

Number of edges: 139

Figure C.1: The visualization of Delaunay triangulation

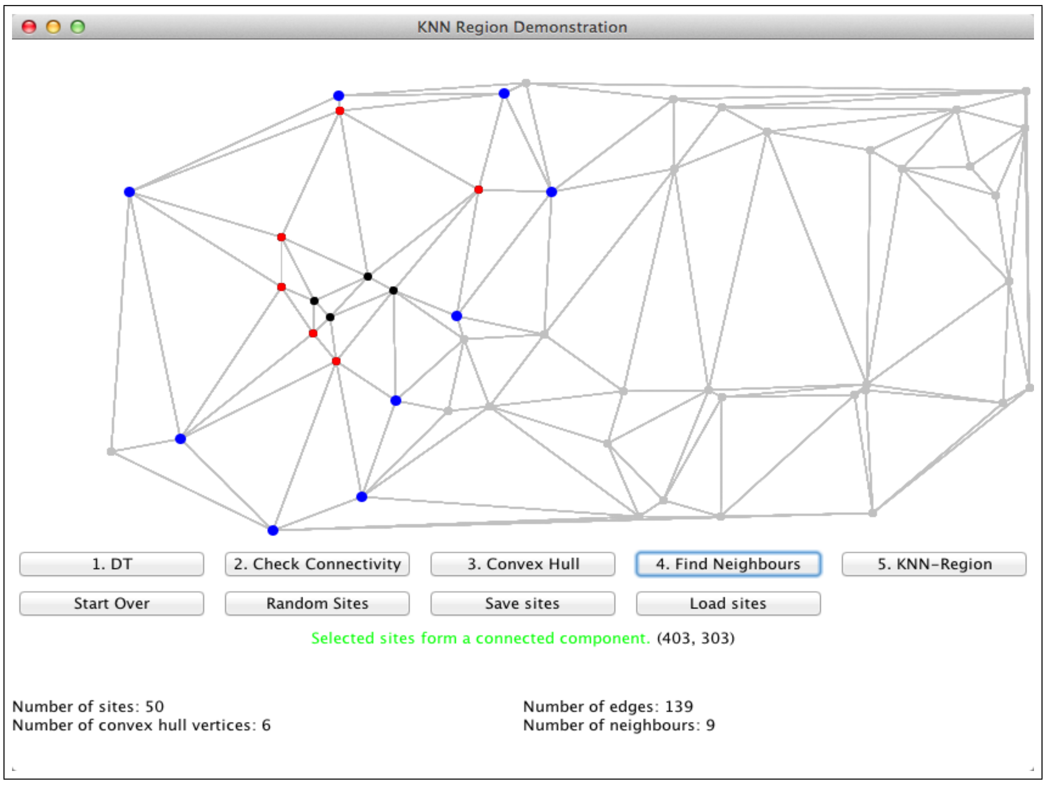


Selected sites form a connected component. (252, 438)

Number of sites: 50
Number of convex hull vertices: 6

Number of edges: 139

Figure C.2: The visualization of the convex hull of specific points

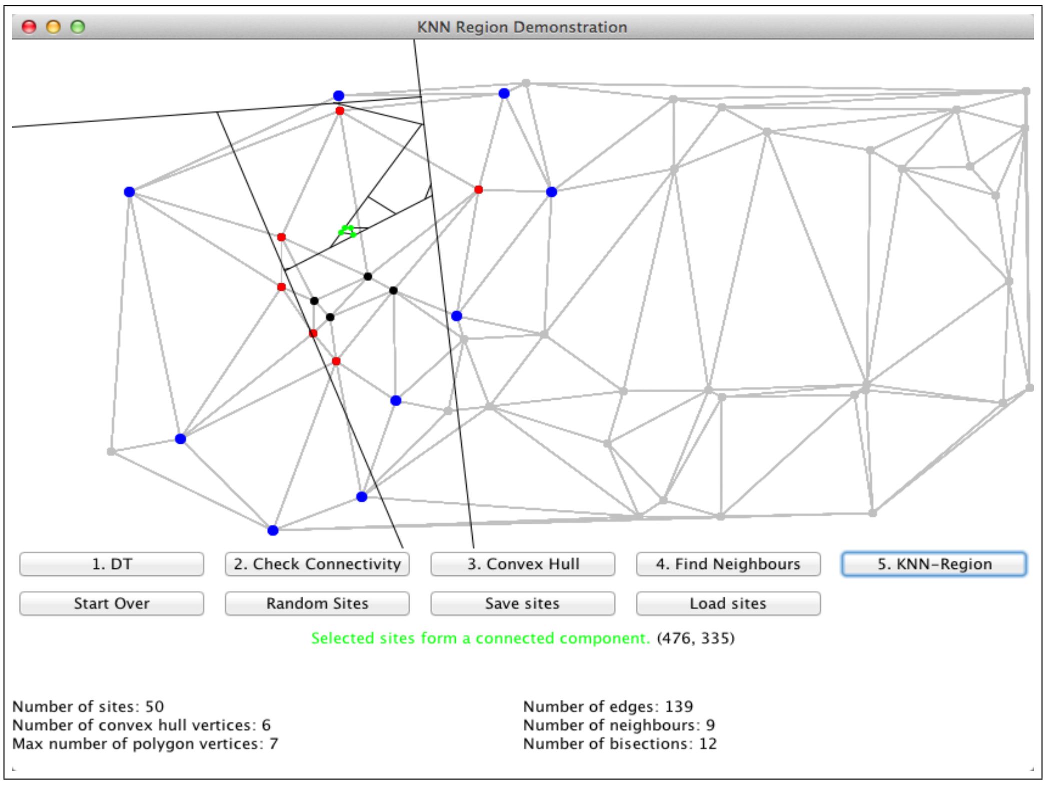


Selected sites form a connected component. (403, 303)

Number of sites: 50
Number of convex hull vertices: 6

Number of edges: 139
Number of neighbours: 9

Figure C.3: The visualization of neighbors of specific points



Selected sites form a connected component. (476, 335)

Number of sites: 50
Number of convex hull vertices: 6
Max number of polygon vertices: 7

Number of edges: 139
Number of neighbours: 9
Number of bisections: 12

Figure C.4: The visualization of kNN region finding by $\mathcal{DT}$-kR

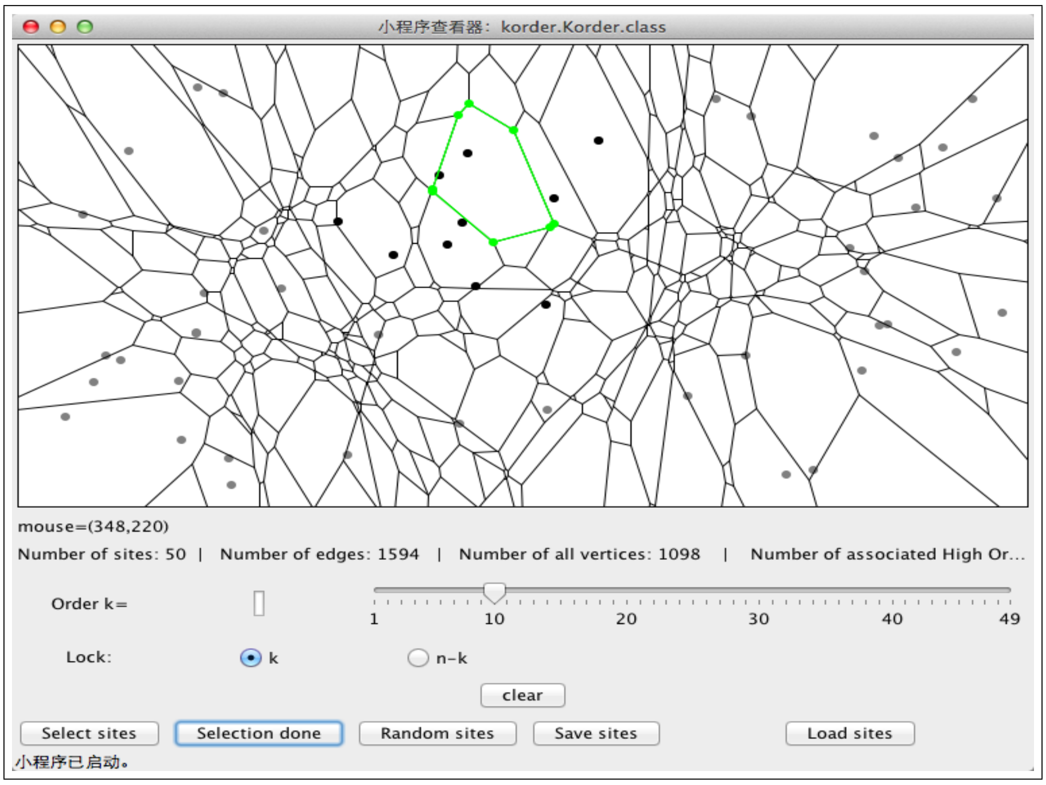

小程序查看器: korder.Korder.class
mouse=(348,220)
Number of sites: 50 | Number of edges: 1594 | Number of all vertices: 1098 | Number of associated High Or...
Order k=
1
10
20
30
40
49
Lock:
k
n-k
clear
Select sites
Selection done
Random sites
Save sites
Load sites
小程序已启动。

Figure C.5: The visualization of kNN region finding by $\mathcal{VD}_k$-kR

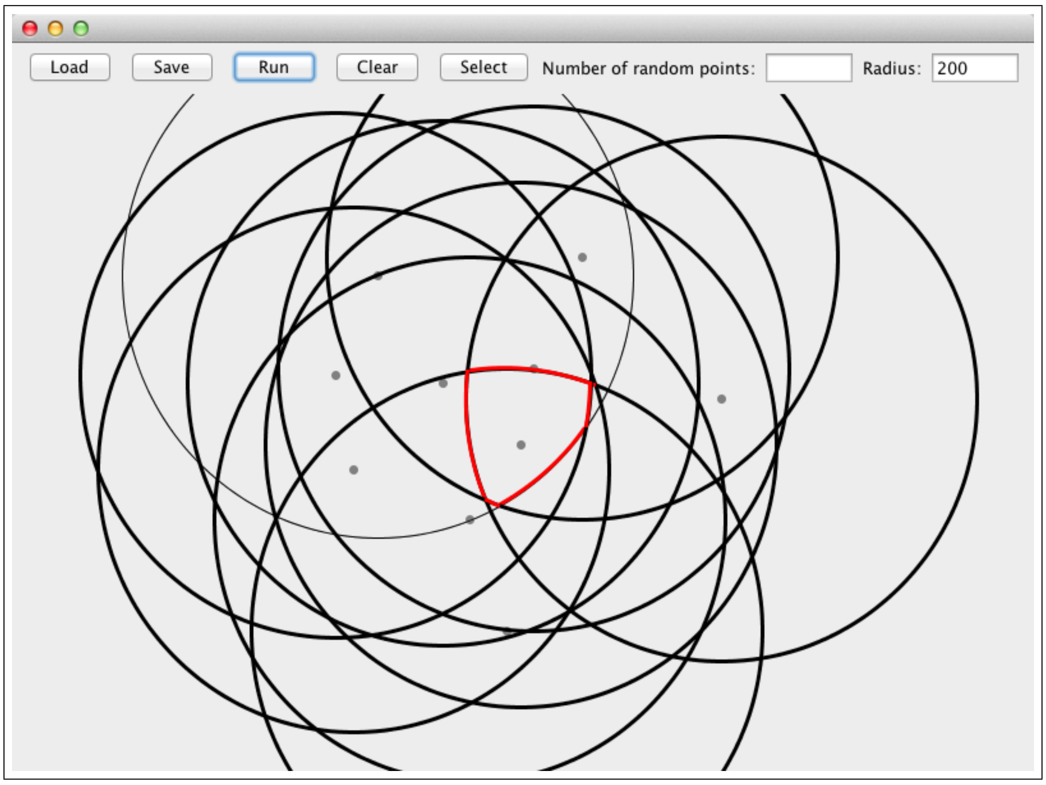


Figure C.6: The visualization of an optimum region

Vita

Xuan, K., Zhao, G., Taniar, D., Rahayu, W., Safar, M. and Srinivasan, B., Voronoi-based range and continuous range query processing in mobile databases. *J. Comput. Syst. Sci.*(*JCSS*), **77**(4):637–651, 2011. (**A***)

Xuan, K., Zhao, G., Taniar, D., Safar, M. and Srinivasan, B., Constrained range search query processing on road networks. *Concurrency and Computation: Practice and Experience* (*CONCURRENCY*), **23**(5):491–504, 2011. (**A**)

Xuan, K., Zhao, G., Taniar, D., Safar, M. and Srinivasan, B., Voronoi-based multi-level range search in mobile navigation. *Multimedia Tools Appl.*, **53**(2):459–479, 2011. (**B**)

Xuan, K., Taniar, D., Safar, M. and Srinivasan, B. (2010), Time constrained range search queries over moving objects in road networks. In *The 8th International Conference on Advances in Mobile Computing and Multimedia* (*MoMM*), pages 329–336, Paris, France, November 2010. (**B**)

Xuan, K., Taniar, D., Safar, M. and Srinivasan, B., and etc. (2009) Network Voronoi Diagram Based Range Search. *The 23rd International Conference on Advanced Information Networking and Applications* (*AINA*), pages 741–748, Bradford, United Kingdom, 2009 (**Best Paper**, **B**)

Xuan, K., Zhao, G., Taniar D., Srinivasan, B., Safar, M. and etc., Continuous range search based on network Voronoi diagram. *International Journal of Grid and Utility Computing* (*IJGUC*), **1**(4): 328–335, 2009.

Xuan, K., Taniar, D., Safar, M. and Srinivasan, B. (2008), Continuous Range Search Query Processing in Mobile Navigation. *14th International Conference on Parallel and Distributed Systems* (*ICPADS*), pages 361–368, Melbourne, Australia, December, 2008. (**B**)

Zhao, G., Xuan, K., Rahayu W., Taniar, D., Safar, and etc., Voronoi-based continuous k nearest neighbor search in mobile navigation. *IEEE Transactions on Industrial Electronics*, **58**(6):2247–2257, 2011. (**Current IF=5.16**)

Zhao, G., Xuan, K. and Taniar, D., Path knn query processing in mobile systems. IEEE Transactions on Industrial Electronics, 99, doi: 10.1109/ TIE.2011.2167113, 2011. (**Current IF=5.16**)

Zhao, G., Xuan, K. and Taniar, D., Rahayu, W. and Srinivasan, B. (2009), Intelligent transport navigation system using lookahead continuous kNN. In Proc. of ICIT, pages 1–6, Churchill, Victoria, Australia, February 2009.

Zhao, G., Xuan, K., Taniar, D. and Srinivasan, B., LookAhead continuous KNN mobile query processing. *Comput. Syst. Sci. Eng.*, **25**(3), 2010 (**B**)

Zhao, G., Xuan, K., Taniar, D., Safar, M., Gavrilova, M.L. and etc. (2009), Multiple Object Types KNN Search Using Network Voronoi Diagram. *International Conference of Computational Science and Its Applications* (*ICCSA*), pages 819–834, Seoul, Korea, 2009.

Zhao, G., Xuan, K., Taniar, D., and Srinivasan, B., Incremental k-Nearest-Neighbor Search on Road Networks. *Journal of Interconnection Networks*, **9**(4): 455–470, 2008

Permanent Address: Clayton School of Information Technology
Monash University
Australia

This thesis was typeset with LaTeX 2ε[1] by the author.

[1]LaTeX 2ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Glenn Maughan and modified by Dean Thompson and David Squire of Monash University.

Bibliography

[AB04] Peter F. Ash and Ethan D. Bolker. Generalized dirichlet tessellations. *Geometriae Dedicata*, 20(2):209–243, October 2004.

[ABS08] Markus Aleksy, Thomas Butter, and Martin Schader. Architecture for the development of context-sensitive mobile applications. *Mobile Information Systems*, 4(2):105–117, 2008.

[Bay97] Rudolf Bayer. The universal b-tree for multidimensional indexing: general concepts. In *Proc. of Worldwide Computing and Its Applications (WWCA)*, pages 198–209. Springer, March 1997.

[BdAG06] Thomas Behr, Victor Teixeira de Almeida, and Ralf Hartmut Guting. Representation of periodic moving objects in databases. In *Proc. of 14th ACM-GIS*, pages 43–50, Arlington, Virginia, November 2006. ACM.

[BER85] Dennis Albert Beckley, Martha Walton Evens, and V. K. Raman. Multikey retrieval from k-d trees and quad-trees. In *Proc. of ACM SIGMOD*, pages 291–301. ACM Press, May 1985.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, Atlantic City, NJ, 1990.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[BS67] L J Bass and S R Schubert. On finding the disc of minimum radius containing a given set of points. *Mathematics of Computation*, 12:712–714, 1967.

[CC] Hyung-Ju Cho and Chin-Wan Chung. An efficient and scalable approach to cnn queries in a road network. In *VLDB*, pages 865–876. ACM.

[CFP+05] Domenico Cantone, Alfredo Ferro, Alfredo Pulvirenti, Diego Reforgiato Recupero, and Dennis Shasha. Antipole tree indexing to support range search and k-nearest neighbor search in metric spaces. *IEEE Trans. Knowl. Data Eng.*, 17(4):535–550, 2005.

[CHC04a] Ying Cai, Kien A. Hua, and Guohong Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Proc. of 5th MDM*, pages 27–38, Berkeley, California, January 2004. IEEE Computer Society.

[CHC04b] Ying Cai, Kien A. Hua, and Guohong Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Mobile Data Management*, pages 27–38, Berkeley, CA, January 2004.

[CHC04c] Ying Cai, Kien A. Hua, and Guohong Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Mobile Data Management*, pages 27–38, Berkeley, CA, USA, January 2004. IEEE Computer Society.

[CL07] Edward P. F. Chan and Heechul Lim. Optimization and evaluation of shortest path queries. *VLDB J.*, 16(3):343–369, July 2007.

[CLZ+09] Muhammad Aamir Cheema, Xuemin Lin, Ying Zhang, Wei Wang, and Wenjie Zhang. Lazy updates: An efficient technique to continuously monitoring reverse knn. *PVLDB*, 2(1):1138–1149, 2009.

[CMG+06] Jidong Chen, Xiaofeng Meng, Yanyan Guo, Stephane Grumbach, and Hui Sun. Modeling and predicting future trajectories of moving objects in a constrained network. In *Proc. of 7th MDM*, page 156, Nara, Japan, May 2006. IEEE Computer Society.

[CMNN09] Chi-Yin Chow, Mohamed F. Mokbel, Joe Naps, and Suman Nath. Approximate evaluation of range nearest-neighbor queries with quality guarantee. In *Proc. of 11th SSTD*, pages 283–301, Aalborg, Denmark, July 2009. Springer.

[Com79] Douglas Comer. The ubiquitous b-tree. *Computing Surveys*, 11(2):123C137, 1979.

[CSS08] Shigang Chen, Meongchul Song, and Sartaj Sahni. Two techniques for fast computation of constrained shortest paths. *IEEE/ACM Trans. Netw.*, 16(1):105–115, February 2008.

[CSZY] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, and Jeffrey Xu Yu. Monitoring path nearest neighbor in road networks. In *SIGMOD Conference*, pages 591–602. ACM, June.

[dA] Victor Teixeira de Almeida. Towards optimal continuous nearest neighbor queries in spatial databases. In *GIS*, pages 227–234. ACM, November.

[dAG05] Victor Teixeira de Almeida and Ralf Hartmut Guting. Supporting uncertainty in moving objects in network databases. In *Proc. of 13th ACM-GIS*, pages 31–40, Bremen, Germany, November 2005. ACM.

[Dij59] Edsger W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1(22):269–271, 1959.

[DKS] Ugur Demiryurek, Farnoush Banaei Kashani, and Cyrus Shahabi. Efficient continuous nearest neighbor query in spatial networks using euclidean restriction. In *SSTD*, pages 25–43. Springer, July.

[DKS09] Ugur Demiryurek, Farnoush Banaei Kashani, and Cyrus Shahabi. Efficient continuous nearest neighbor query in spatial networks using euclidean restriction. In *SSTD*, pages 25–43, Aalborg, Denmark, July 2009. Springer.

[Dye86] M E Dyer. On a multidimensional search technique and its application to the euclidean one-centre problem. *SIAM Journal on Computing*, 15:725–738, 1986.

[DZS$^+$06] Ke Deng, Xiaofang Zhou, Heng Tao Shen, Kai Xu, and Xuemin Lin. Surface k-nn query processing. In *Proc. of 22nd ICDE*, page 78, Atlanta, GA, USA, April 2006. IEEE Computer Society.

[EH72] J Elzinga and D W Hearn. Geometrical solutions to some minimax location problems. *Transportation Science*, 6:379–394, 1972.

[FB74] Raphael Finkel and J.L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[For87] Steven Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[FSAA] Hakan Ferhatosmanoglu, Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Constrained nearest neighbor queries. In *SSTD*, pages 257–278. Springer, July.

[GG98] Volker Gaede and Oliver Günther. An introduction to spatial database systems. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[GGPS07] Stephen R. Gulliver, George Ghinea, M. Patel, and Tacha Serif. A context-aware tour guide: User implications. *Mobile Information Systems*, 3(2):71–88, 2007.

[GKTD05] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *VLDB J.*, 14(2):137–154, April 2005.

[Gra72] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1(4):132–133, 1972.

[GT04a] Jen Ye Goh and David Taniar. Mobile data mining by location dependencies. In *Proc. of 5th Intelligent Data Engineering and Automated Learning (IDEAL)*, pages 225–231, Wellington, New Zealand, September 2004. Springer.

[GT04b] John Goh and David Taniar. Mining frequency pattern from mobile users. In *Proc. of 8th Knowledge-Based Intelligent Information and Engineering Systems (KES)*, pages 795–801, Wellington, New Zealand, September 2004. Springer.

[GT05] John Goh and David Taniar. Mining parallel patterns from mobile users. *International Journal of Business Data Communication and Networking*, 1(1):50–76, 2005.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD*, pages 47–57. ACM Press, June 1984.

[Gut94] Ralf Hartmut Guting. Multidimensional access methods. *VLDB J.*, 3(4):357–399, 1994.

[GZ09] Yunjun Gao and Baihua Zheng. Continuous obstructed nearest neighbor queries in spatial databases. In *SIGMOD Conference*, pages 577–590, Providence, Rhode Island, USA, June 2009. ACM.

[GZC$^+$09a] Yunjun Gao, Baihua Zheng, Gencai Chen, Wang-Chien Lee, Ken C. K. Lee, and Qing Li. Visible reverse k-nearest neighbor queries. In *ICDE*, pages 1203–1206, Shanghai, China, April 2009. IEEE.

[GZC$^+$09b] Yunjun Gao, Baihua Zheng, Gencai Chen, Wang-Chien Lee, Ken C. K. Lee, and Qing Li. Visible reverse k-nearest neighbor query processing in spatial databases. *IEEE Trans. Knowl. Data Eng.*, 21(9):1314–1327, 2009.

[HCLZ09] Mahady Hasan, Muhammad Aamir Cheema, Xuemin Lin, and Ying Zhang. Efficient construction of safe regions for moving knn queries over dynamic datasets. In *SSTD*, pages 373–379, Aalborg, Denmark, July 2009. Springer.

[HGNM08] Nicola Honle, Matthias GroBmann, Daniela Nicklas, and Bernhard Mitschang. Preprocessing position data of mobile objects. In *Proc. of 9th MDM*, pages 41–48, Beijing, China, April 2008. IEEE.

[HL06] Haibo Hu and Dik Lun Lee. Range nearest-neighbor query. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 18(1):78–91, January 2006.

[HXL05] Haibo Hu, Jianliang Xu, and Dik Lun Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD Conference*, pages 479–490, Baltimore, Maryland, USA, June 2005. ACM.

[JLO07a] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Continuous clustering of moving objects. *Proceedings of the VLDB Endowment*, 19(9):1161–1174, September 2007.

[JLO07b] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Continuous clustering of moving objects. *IEEE Trans. Knowl. Data Eng.*, 19(9):1161–1174, September 2007.

[JT05] James Jayaputera and David Taniar. Data retrieval for location-dependent queries in a multi-cell wireless environment. *Mobile Information Systems*, 1(2):91–108, 2005.

[KKR08] Hans-Peter Kriegel, Peer Kroger, and Matthias Renz. Continuous proximity monitoring in road networks. In *Proc. of 16th ACM-GIS*, page 10, Irvine, California, November 2008. ACM.

[KS04] Mohammad R. Kolahdouzan and Cyrus Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *Proc. of 30th VLDB*, pages 840–851, Toronto, Canada, August 2004. Morgan Kaufmann Publishers Inc.

[KS05] Mohammad R. Kolahdouzan and Cyrus Shahabi. Alternative solutions for continuous k nearest neighbor queries in spatial network databases. *GeoInformatica*, 9(4):321–341, December 2005.

[KZWW05] Wei Shinn Ku, Roger Zimmermann, Haojun Wang, and Chi Ngai Wan. Adaptive nearest neighbor queries in travel time networks. In *Proc. of 13th ACM-GIS*, pages 210–219, Bremen, Germany, November 2005. ACM.

[LCLC09] Dongsheng Li, Jiannong Cao, Xicheng Lu, and Kaixian Chen. Efficient range query processing in peer-to-peer systems. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 21(1):78–91, January 2009.

[Lee82] Der-Tsai Lee. On k-nearest neighbor voronoi diagrams in the plane. *IEEE Trans. Computers*, 31(6):478–487, 1982.

[LGYL11] Chuanwen Li, Yu Gu, Ge Yu, and Fangfang Li. wneighbors: A method for finding k nearest neighbors in weighted regions. In *DASFAA*, pages 134–148, Hong Kong, China, April 2011. Springer.

[LJOS05] Dan Lin, Christian S. Jensen, Beng Chin Ooi, and Simonas Saltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *Proc. of 6th MDM*, pages 59–66, Ayia Napa, Cyprus, May 2005. ACM.

[LWF08] Wenting Liu, Zhijian Wang, and Jun Feng. Continuous clustering of moving objects in spatial networks. In *Proc. of 12th Knowledge-Based Intelligent Information and Engineering Systems (KES)*, Zagreb, Croatia, September 2008. Springer.

[LZZ06] Dan Lin, Rui Zhang, and Aoying Zhou. Indexing fast moving objects for knn queries based on nearest landmarks. *GeoInformatica*, 10(4):423–445, 2006.

[Meg83] N Megiddo. Linear-time algorithms for linear programming in r^3 and related problems. *SIAM Journal on Computing*, 12:759–776, 1983.

[Meg84] N Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of ACM*, 31:114–127, 1984.

[MK89] Avraham Margalit and Gary D. Knott. An algorithm for computing the uniou, intersection or difference of two polygons. *Comput & Graphics*, 13(2):167–183, 1989.

[Muh09] Rashid Bin Muhammad. Range assignment problem on the steiner tree based topology in ad hoc wireless networks. *Mobile Information Systems*, 5(1):53–64, 2009.

[MYPM06] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias, and Nikos Mamoulis. Continuous nearest neighbor monitoring in road networks. In *Proc. of 32th VLDB*, pages 43–54, Seoul, Korea, September 2006. ACM.

[NTZ07] Sarana Nutanong, Egemen Tanin, and Rui Zhang. Visible nearest neighbor queries. In *DASFAA*, pages 876–883, Bangkok, Thailand, April 2007. Springer.

[NTZ10] Sarana Nutanong, Egemen Tanin, and Rui Zhang. Incremental evaluation of visible nearest neighbor queries. *IEEE Trans. Knowl. Data Eng.*, 22(5):665–681, 2010.

[NZTK08] Sarana Nutanong, Rui Zhang, Egemen Tanin, and Lars Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *Proceedings of the VLDB Endowment*, 1(1):1095–1106, August 2008.

[OBSC00a] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley and Sons Ltd., West Sussex, England, second edition, 2000.

[OBSC00b] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley and Sons Ltd., West Sussex, England, second edition, 2000.

[PJ03] Dieter Pfoser and Christian S. Jensen. Indexing of network constrained moving objects. In *GIS*, pages 25–32, New Orleans, Louisiana, USA, November 2003. ACM.

[PMS07] Kostas Patroumpas, Theofanis Minogiannis, and Timos K. Sellis. Approximate order-k voronoi cells over positional streams. In *Proc. of 15th ACM-GIS*, page 36, Seattle, Washington, November 2007. ACM.

[PS07a] Kostas Patroumpas and Timos K. Sellis. Semantics of spatially-aware windows over streaming moving objects. In *Proc. of 8th MDM*, pages 52–59, Mannheim, Germany, April 2007. IEEE.

[PS07b] Kostas Patroumpas and Timos K. Sellis. Semantics of spatially-aware windows over streaming moving objects. In *MDM*, pages 52–59, Mannheim, Germany, May 2007. IEEE.

[PSTM04] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. Group nearest neighbor queries. In *Proc. of 20th ICDE*, pages 301–312, Boston, MA, USA, March 2004. IEEE Computer Society.

[PTMH05] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576, 2005.

[PZMT03] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *Proc. of 29th VLDB*, pages 802–813, Berlin, Germany, September 2003. Morgan Kaufmann Publishers Inc.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, San Jose, California, June 1995. ACM Press.

[Saf05] Maytham Safar. K nearest neighbor search in navigation systems. *Mobile Information Systems*, 1(3):207–224, 2005.

[SE06] Maytham Safar and Dariush Ebrahimi. edar algorithm for continuous knn queries based on pine. *International Journal of Information Technology and Web Engineering*, 1(4):1–21, 2006.

[SH75] M I Shamos and D Hoey. Closest-point problems. In *Proc. of 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, Los Angeles, 1975. IEEE Computer Society Press.

[Sha75] M I Shamos. Geometric complexity. In *Proc. of 7th Annual ACM Symposium on Theory of Computing*, pages 224–233, New York, 1975. ACM.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518. Morgan Kaufmann, September 1987.

[SS08] Mehdi Sharifzadeh and Cyrus Shahabi. Processing optimal sequenced route queries using voronoi diagrams. *GeoInformatica*, 12(4):411–433, December 2008.

[SS09a] Jagan Sankaranarayanan and Hanan Samet. Distance oracles for spatial networks. In *Proc. of 25th ICDE*, pages 652–663, Shanghai, China, April 2009. IEEE.

[SS09b] Jagan Sankaranarayanan and Hanan Samet. Distance oracles for spatial networks. In *ICDE*, pages 652–663, Shanghai, China, April 2009. IEEE.

[SSA08] Hanan Samet, Jagan Sankaranarayanan, and Houman Alborzi. Scalable network distance browsing in spatial databases. In *Proc. of ACM SIGMOD*, pages 43–54, Vancouver, BC, Canada, June 2008. ACM Press.

[STX08] Cyrus Shahabi, Lu An Tang, and Songhua Xing. Indexing land surface for efficient knn query. *PVLDB*, 1(1):1020–1031, 2008.

[Syl75] J J Sylvester. A question in the geometry of situation. *Quarterly Journal of Mathematics*, 1(79), 1875.

[TBPM05] Manolis Terrovitis, Spiridon Bakiras, Dimitris Papadias, and Kyriakos Mouratidis. Constrained shortest path computation. In *SSTD*, pages 181–199, Angra dos Reis, Brazil, August 2005. Springer.

[TG07] David Taniar and John Goh. On mining movement pattern from mobile users. *International Journal of Distributed Sensor Networks*, 3(1):69–86, 2007.

[TP03] Yufei Tao and Dimitris Papadias. Spatial queries in dynamic environments. *ACM Transactions on Database Systems (TODS)*, 28(2):101–139, June 2003.

[TPL04] Yufei Tao, Dimitris Papadias, and Xiang Lian. Reverse knn search in arbitrary dimensionality. In *Proc. of 30th VLDB*, pages 744–755, Toronto, Canada, August 2004. Morgan Kaufmann Publishers Inc.

[TPS02] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *Proc. of 28th VLDB*, pages 287–298, Hong Kong, China, August 2002. Morgan Kaufmann Publishers Inc.

[TR02] David Taniar and J. Wenny Rahayu. A taxonomy of indexing schemes for parallel database systems. *Distributed and Parallel Databases*, 12(1):73–106, July 2002.

[TR04] David Taniar and J. Wenny Rahayu. Global parallel index for multi-processors database systems. *Information Sciences*, 165(1-2):103–127, September 2004.

[TST$^+$11] David Taniar, Maytham Safar, Quoc Thai Tran, J. Wenny Rahayu, and Jong Hyuk Park. Spatial network rnn queries in gis. *Comput. J.*, 54(4):617–627, 2011.

[TTS09] Quoc Thai Tran, David Taniar, and Maytham Safar. Reverse k nearest neighbor and reverse farthest neighbor search on spatial networks. *T. Large-Scale Data- and Knowledge-Centered Systems*, 1:353–372, 2009.

[TWHC04] Goce Trajcevski, Ouri Wolfson, Klaus Hinrichs, and Sam Chamberlain. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.*, 29(3):463–507, March 2004.

[TXC07] Yufei Tao, Xiaokui Xiao, and Reynold Cheng. Range search on multidimensional uncertain data. *ACM Trans. on Database Systems (TODS)*, 32(3):15, August 2007.

[TYSK09] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD Conference*, pages 563–576, Providence, Rhode Island, USA, June 2009. ACM.

[WHE] Telstra Corporation, 2009, whereis, Melbourne, viewed 10 July, 2009, http://www.whereis.com.

[WRTS09] Agustinus Borgy Waluyo, J. Wenny Rahayu, David Taniar, and Bala Srinivasan. Mobile service oriented architectures for nn-queries. *Journal of Network and Computer Applications*, 32(2):434–447, March 2009.

[WST03] Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar. Optimal broadcast channel for data dissemination in mobile database environment. In *Proc. of 5th Advanced Parallel Programming Technologies (APPT)*, pages 655–664, Xiamen, China, September 2003. Springer.

[WST04] Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar. A taxonomy of broadcast indexing schemes for multi channel data dissemination in mobile database. In *Proc. of 18th Advanced Information Networking and Applications (AINA)*, pages 213–218, Fukuoka, Japan, March 2004. IEEE Computer Society.

[WST05] Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar. Research on location-dependent queries in mobile databases. *Comput. Syst. Sci. Eng.*, 20(2), March 2005.

[XSP09] Songhua Xing, Cyrus Shahabi, and Bei Pan. Continuous monitoring of nearest neighbors on land surface. *PVLDB*, 2(1):1114–1125, 2009.

[XTSS10] Kefeng Xuan, David Taniar, Maytham Safar, and Bala Srinivasan. Time constrained range search queries over moving objects in road networks. In *MoMM*, pages 329–336, Paris, France, November 2010.

[XZT$^+$09a] Kefeng Xuan, Geng Zhao, David Taniar, Bala Srinivasan, Maytham Safar, and Marina Gavrilova. Continuous range search based on network voronoi diagram. *International Journal of Grid and Utility Computing*, 1(4):328 – 335, 2009.

[XZT$^+$09b] Kefeng Xuan, Geng Zhao, David Taniar, Bala Srinivasan, Maytham Safar, and Marina Gavrilova. Network voronoi diagram based range search. In *Proc. of 23rd Advanced Information Networking and Applications (AINA)*, pages 741–748, Bradford, UK, May 2009. IEEE Computer Society.

[XZT$^+$11a] Kefeng Xuan, Geng Zhao, David Taniar, J. Wenny Rahayu, Maytham Safar, and Bala Srinivasan. Voronoi-based range and continuous range query processing in mobile databases. *J. Comput. Syst. Sci. (JCSS)*, 77(4):637 – 651, 2011.

[XZT$^+$11b] Kefeng Xuan, Geng Zhao, David Taniar, Maytham Safar, and Bala Srinivasan. Constrained range search query processing on road networks. *Concurrency and Computation: Practice and Experience (CONCURRENCY)*, 23(5):491 – 504, 2011.

[XZT$^+$11c] Kefeng Xuan, Geng Zhao, David Taniar, Maytham Safar, and Bala Srinivasan. Voronoi-based multi-level range search in mobile navigation. *Multimedia Tools Appl.*, 53(2):459–479, 2011.

[XZTS08] Kefeng Xuan, Geng Zhao, David Taniar, and Bala Srinivasan. Continuous range search query processing in mobile navigation. In *Proceedings of the 14th ICPADS 2008*, pages 361–368, Melbourne, Victoria, Australia, December 2008. IEEE.

[YLK09] Bin Yao, Feifei Li, and Piyush Kumar. Reverse furthest neighbors in spatial databases. In *ICDE*, pages 664–675, Shanghai, China, April 2009. IEEE.

[YMP05] Man Lung Yiu, Nikos Mamoulis, and Dimitris Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(6):820–833, June 2005.

[YS10] Wenjie Yuan and Markus Schneider. Supporting continuous range queries in indoor space. In *Mobile Data Management*, pages 209–214, Kanas City, Missouri, USA, May 2010. IEEE Computer Society.

[ZJDR10] Rui Zhang, H. V. Jagadish, Bing Tian Dai, and Kotagiri Ramamohanarao. Optimized algorithms for predictive range and knn queries on moving objects. *Inf. Syst.*, 35(8):911–932, 2010.

[ZXR$^+$08] Geng Zhao, Kefeng Xuan, Wenny Rahayu, David Taniar, Maytham Safar, Marina L. Gavrilova, and Bala Srinivasan. Incremental k-nearest-neighbor search on road networks. *Journal of Interconnection Networks(JOIN)*, 9(4):455–470, December 2008.

[ZXR$^+$11] Geng Zhao, Kefeng Xuan, Wenny Rahayu, David Taniar, Maytham Safar, Marina L. Gavrilova, and Bala Srinivasan. Voronoi-based continuous k nearest neighbor search in mobile navigation. *IEEE Transactions on Industrial Electronics*, 58(6):2247–2257, 2011.

[ZXT$^+$09] Geng Zhao, Kefeng Xuan, David Taniar, Wenny Rahayu, and Bala Srinivasan. Intelligent transport navigation system using lookahead continuous knn. In *Proc. of ICIT*, pages 1–6, Churchill, Victoria, Australia, February 2009. IEEE.

[ZXT11] Geng Zhao, Kefeng Xuan, and David Taniar. Path knn query processing in mobile systems. *IEEE Transactions on Industrial Electronics*, 99, 2011.

[ZZS$^+$05] Panfeng Zhou, Donghui Zhang, Betty Salzberg, Gene Cooperman, and George Kollios. Close pair queries in moving object databases. In *GIS*, pages 2–11, Bremen, Germany, November 2005. ACM.

Last Thing

“With most people, unbelief in one thing is founded upon blind belief in another.”

– Georg C. Lichtenberg, Scientist