Advances on K Nearest Neighbour Search in Spatial Databases

by

Geng Zhao

Thesis

Submitted by Geng Zhao

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

March, 2013

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.

© Copyright

by

Geng Zhao

2013

To My Supportive Family, Friends and Academics

Contents

List of Tables

List of Figures

Advances on K Nearest Neighbour Search in Spatial Databases

Geng Zhao
[redacted]
Monash University, 2013

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

Abstract

A spatial database is a database that stores data and makes queries which are related to objects in space, including points, lines and polygons. The spatial database is designed to process the spatial data type which cannot be processed by typical databases. k Nearest Neighbor search is a type of query to classify objects based on closest distances in the feature space, as a result, a large portion of k nearest neighbor search queries are based on road network. Consequently, the most popular method that has been fully discussed is called Network Expansion. By processing more and more complex queries, network expansion methods show a significant drawback which is poor performance because the underlay road network is crossed and connected. Do expansion for each intersection points in road network is unrealistic. The other problem in k Nearest Neighbor (kNN) query processing is that most of the existing k nearest neighbor searches are concentrated on points. In other words, the discrete points are the input and output of k Nearest Neighbor search query. While in reality, points, lines as well as polygons are three types of spatial elements. How to utilize Lines/Route in spatial query becomes another question for our researchers. Motivated by these two points, the following paragraph summarizes the contribution of this thesis.

The first main contribution of this thesis is called *Voronoi Based k Nearest Neighbor search query*. Compared to the Network Expansion method, *the Voronoi*

based method aggregates the road segments using a Voronoi Diagram. Instead of expanding from each intersection in road network, the Voronoi Diagram divides the map into polygons by treating the points of interest as generators. In this part, we have proposed 2 algorithms which use the Voronoi Diagram to improve the performance for Multiple types of k Nearest Neighbor Search query and Continuous k Nearest Neighbor Search query respectively. Our experiments have shown the proposed algorithms can improve the performance significantly either from a cost and a processing time point of view compared to the traditional Network Expansion method.

The second main contribution of this thesis is called *Route and Path related kNN Queries*. The aim of this part in the thesis is to bring *lines/routes* into the input and(or) output of k Nearest Neighbor search. As a result, users can use lines to find points, use points to find routes, or even use route to find route. Correspondingly, there are three novel approaches discussed in this part, namely, Path Based kNN Search Queries, Path Branch Point based kNN Search Queries and Time Constraint Route Search over Multiple Locations. By proposing these three approaches, the k Nearest Neighbor Search has been enriched and can satisfy various types of user queries.

To sum up, the thesis is concentrated on two main category of query processing: i) *Voronoi based k Nearest Neighbor Search* which is aiming at improving the Network Expansion Method which is the most popular technique used by existing k Nearest Neighbor Search approaches. ii) *Route and Path related kNN Query* which brings route and path into the input or/and output of kNN queries and which fills up the blanket area in kNN query that only concrete points are utilized in spatial queries.

Advances on K Nearest Neighbour Search 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.

Geng Zhao
March 18, 2013

Acknowledgments

Doctoral thesis is a long and tough journey and, I would like to thank everyone who helped me to achieve the accomplishment of my doctoral degree. I learned to be self-initiated, self-organizable and optimistic.

First of all, I would like to express my gratitude to my supervisor, Associate Professor David Taniar. Without his insightful supervision, I could not have such a smooth path in this research. Without his patient guidance, I could not enjoy my study. Without his inspiriting encouragement, I a be so optimistic even facing the unprecedented difficulties and pressure. He is extremely considerate to me, especially during my maternity period. Moreover, what is a great honor that my baby is named by David.

I am also very thankful to my joint-supervisor Professor Bala Srinivasan. He taught me the overview of the research scope as well as urging me from time to time. He helped me out when I was suffering with minor technical details and outlined the big picture of my research.

I want to take this golden opportunity to thank my parents who support me since I came to Australia not only financially but also mentally. They gave me the environment in which I could do research without any other worries and supported all my decisions. The support from them is the most selfless friendship in the world.

Lastly, but the most importantly, I am thankful to my husband, Kefeng. Being my accompanier, and he has supported me along the way from my bachelor study until now. We went through hardness and shared joyful moments. Research time became enjoyable with his accompaniment. I want to take this opportunity to thank him and hope we can support each other in future.

Geng Zhao

Monash University

March 2013

Publications resulting from this thesis: Below is a list of publications resulting from this thesis. I am very grateful to all of the people who collaborated with me for these publications. Their comments and suggestions were always very helping and insightful.

1. Zhao, G., Xuan, K., Rahayu, W., Taniar, D., Safar, M., Gavrilova, M. and Srinivasan, B. Voronoi-based continuous k nearest neighbor search in mobile navigation. **IEEE Transactions on Industrial Electronics (TIE)**, 56(10): 2247-2257. 2010. (**Tier B**)

2. Zhao, G., Xuan, K. and Taniar, D. Path *k*NN query processing in Digital Ecosystems, **IEEE Transactions on Industrial Electronics (TIE)**. 2011. DOI (identifier) 10.1109/TIE.2011.2167113. (**Tier B**)

3. Zhao, G., Xuan, K., Taniar, D., Safar, M. and Srinivasan, B. Time Constraint Route Search over Multiple Locations. **in The Knowledge Engineering Review (KER)**. 2010. (Tier B)

4. Zhao, G., Xuan, K., Taniar, D. and Srinivasan, B. LookAhead Continuous *k*NN Mobile Query Processing. **International Journal of Computer Systems Science and Engineering (IJCSSE)**. 25(3). 2010.

5. Zhao, G., Xuan, K., Taniar, D., Safar, M., Gavrilova, M. and Srinivasan, B. Multiple object types *k*NN search using network Voronoi diagram. In Proceedings of **International Conference for Computational Science and Its Applications (ICCSA)**, pages 819-834, Yongin, Korea, 2009.

6. Zhao, G., Taniar, D., Safar, M., Rahayu, W. and Srinivasan, B. Path Branch Points in Mobile Navigation, Proceedings of **The 8th International Conference on Advances in Mobile Computing and Multimedia (MoMM'10)**, pages 329-336, Paris, France, 2010

7. 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**)

8. 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 329-336, Paris, France, November 2010.

9. 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***)

10. 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**)

Chapter 1

Introduction

1.1 Overview

Due to heavy traffic load and complex road connections, more and more users need an application to help them navigate crowded roads, guide them to the best route and even give answers to users' queries. With the development of personal computing devices and wireless networks, mobile devices using inexpensive wireless networks provide unlimited convenience to mobile users [WST05a]. A spatial database is a database that stores data and makes queries which are related to objects in space, including points, lines, polygons and paths. The spatial database is designed to process the spatial data type which can not be processed by typical databases. The application of spatial databases include Geographic Information Systems (GIS), Computer Aided Design (CAD), Very-Large-Scale Integration (VLSI) designs, Multimedia Information System (MMIS) and medicine and biological research. Spatial related query processing has played a more and more important role in our daily life with the decreasing cost of wireless network access, upgrading mobile device's processing ability and widening internet coverage.

With the developing processing efficiency and the growing complexity of road connection, nowadays users are sending out diverse spatial queries based on objects locations. A number of queries can be listed here as samples, e.g. navigating from

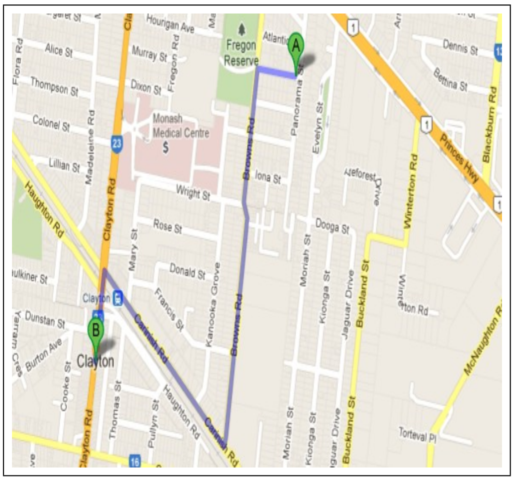


Figure 1.1: Example of road navigation

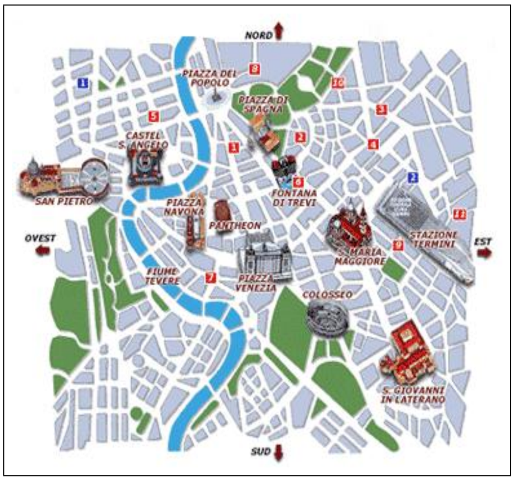


Figure 1.2: Example of route search

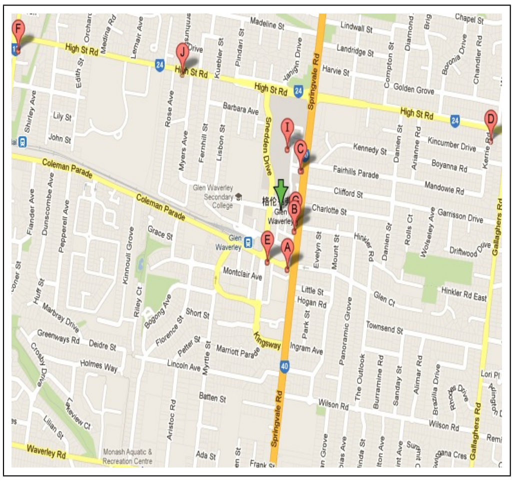


Figure 1.3: Example of object finding

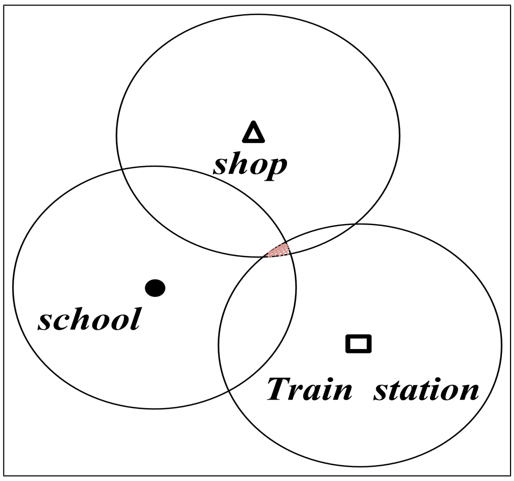


Figure 1.4: Example of range matching

A to B, finding the nearest neighbor from current locations, searching all objects within given range and so on. We summarize them using different points of view.

From the query point of view, these objects of interest can be a snapshot or continuously moving objects.

In a snapshot query, the results of the query are to be computed only once. In contrast to the snapshot queries, a continuous query requires the results to be continuously updated as the underlying data is updated.

Example 1.1.1. ***Snapshot query****: a user may want to find the nearest 5 petrol stations from Monash University. He may issue a snapshot k Nearest Neighbor query with a query location set as Monash University.*

Example 1.1.2. ***Continuous query***: *For instance, a person driving a car may want to find the nearest 5 petrol stations of his current location. Since the car is continuously moving, the results are required to be updated continuously. He may issue a continuous k Nearest Neighbor query with the query location set as the location of car..*

Example 1.1.3. ***Continuous moving query***: *While in the above example only the query is moving, in many applications, all of the query objects and data objects may be continuously moving. For instance, a taxi driver might want to continuously monitor his nearest walking customers of his location. In this example, the query and the data objects all are continuously moving.*

From the data accuracy point of view, points of interest can be certain or uncertain data.

Usually, it is assumed that the objects (e.g., locations) have accurate value and the spatial queries to use these locations. However, uncertain data is inherent most of the time such as sensor databases, moving object databases, market analysis, and quantitative economic research. The reason of the inaccuracy might be the limitation of measuring equipment, delayed data updates, incompleteness or data anonymization to preserve privacy. In such applications, the spatial queries are issued on the uncertain data and probabilistic results are returned.

Example 1.1.4. ***Certain Data***:*Find the nearest restaurant of point q* ($X = 116°23'17''$, $Y = 39°54'27''$)*. The query point (q) has accurate location.*

Example 1.1.5. ***Uncertain Data***:*Find the nearest restaurant of point q* ($116 < X < 117$, $38 < Y < 39$) *. The query point (q) has uncertain value.*

From the result's type point of view, we can summarize the spatial query into the following categories: route search, object finding and range matching.

Example 1.1.6. ***Route Search:*** *Find the best road that satisfied users' diverse conditions. For example, the route navigation we use everyday: find the shortest* *road starting from point A to destination point B (Fig.1.1 as an example). There might be some other road navigation variants: finding the route with shortest travel time or less traffic lights and so on. In addition, there are some other route search queries. Fig.1.2 gives another example of route search query. In this example, 16 famous tourist places have been chosen by the traveler. The query is to* ***find the route*** *that can visit 4 tourist places with the shortest distance.*

Example 1.1.7. ***Objects Finding:*** *Find the objects that fulfill users' requirements (k nearest neighbor or within the range and so on). Fig.1.3 shows an example of objects finding: show all pharmacies in my local suburb.*

Example 1.1.8. ***Range Matching:*** *Find the range or area that meets users' specification. Fig.1.4 shows an example of range matching: find the area that are close (within 2km walking distance) to shops, schools and train stations.*

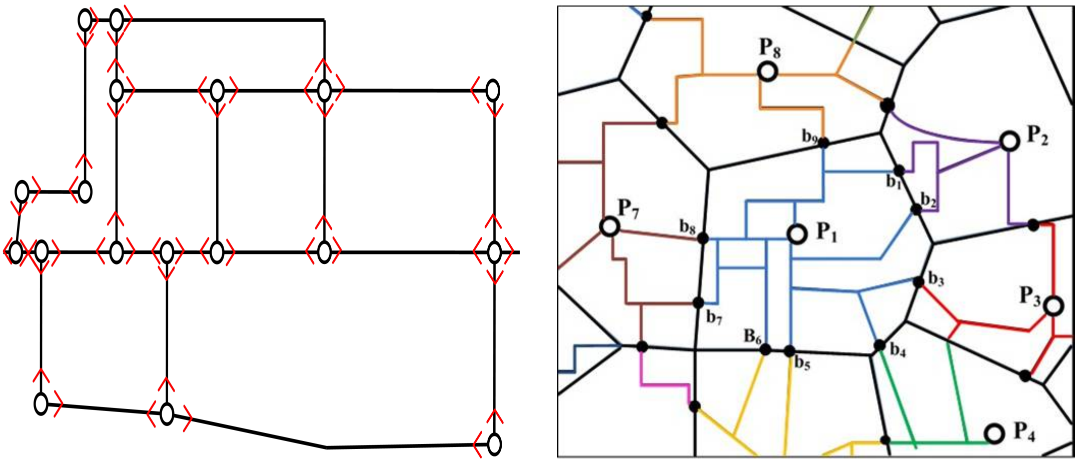
The following sections emphasis the major problems of the existing methods and the contributions in this thesis in detail.

1.2 Major Issues

Although k Nearest Neighbor query search has been fully investigated over the recent decades, there are still some significant problems or, in other words, there are still some gaps/blank zones which have not been explored. Moreover, even the existing methods which can solve the problems perform poorly under some circumstance. After summarizing various spatial queries, we draw several conclusions:

1.2.1 Problem 1: Poor performance of Network Expansion

All of the approaches are constructed based on the underlying road connection between objects. Within the road map, roads are connected and joined by thousands of intersection nodes which break the roads into small segments. The total distance of the road is calculated by summing up the component segments distances. As a



(a) Network Expansion (b) Voronoi Diagram

Figure 1.5: Network Expansion vs. the Voronoi Diagram

result, network expansion is the technique which has been widely used in existing methods. Network expansion is processed as follows: when encountering any intersection node, the traverse expansion takes every possible directions. In other words, if we suppose every node has four possible directions to go, then the expansion would be 4^n, where n is the number of expansion nodes. From this calculation, we can infer that the performance cost will behave like a parabola with the increasing number of intersection nodes. The poor performance is inevitable because the complex road connection will result in large number of intersection nodes. Consequently, how to merge the intersection nodes or how to avoid the expansion becomes a new topic to the researchers who are engaged in spatial query processing. This is the first main chapter of the thesis.

Fig.1.5(a) demonstrates the expansion directions for each intersection node. The expansion should be invoked for every possible moving direction when the query point is located at this intersection node. Fig.1.5 shows the illustration of a Network Voronoi Diagram which merges lots of segments into polygons.

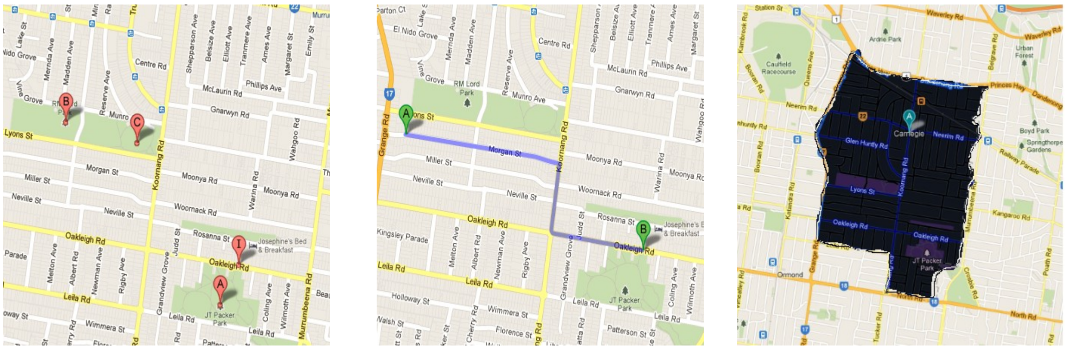
1.2.2 Problem 2: Discrete Points are the input and output of Spatial Queries

The second main chapter of the thesis is to bring line/route into spatial queries. In spatial databases, there are three types of spatial data elements: i) Point ii)Line and route iii)Region and polygon.

Points: A spatial point is a primitive notion upon which other concepts may be defined. In general, points are zero-dimensional; i.e., they do not have volume, area, length or any other higher-dimensional analogue. In branches of mathematics dealing with set theory, an element is sometimes referred to as a point. In spatial query processing, points of interest belong to this type. They are distributed discretely over the map. The discrete location on the surface of the planet, represented by an x and y coordinate pairs. Each point on the map is created by latitude and longitude coordinates, and is stored as an individual record in the shape file, see Fig.1.6(a).

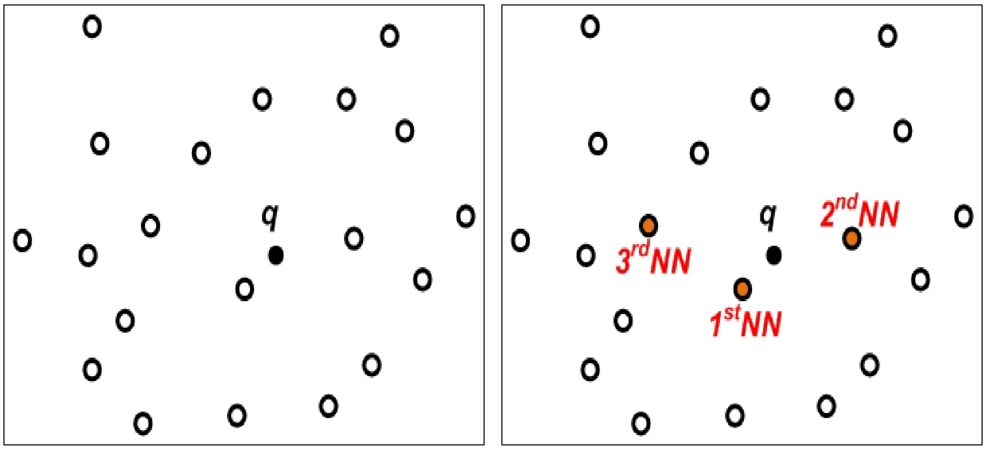
Lines: As an extension of a point, an elongated mark, is the connection between two points, Lines are the paths through networks, which may have line feature, such as a street, highway, river or pipe. Lines are formed by connecting two data points. The computer reads this line as straight, and renders the line as a vector connecting two x-y coordinates (X = longitude, Y = latitude). The more points used to create the line, the greater the detail. FPA requires that the line and polygon features include topology. For lines, this means that the system stores one end of the line as the starting point and the other as the end point, giving the line direction, see Fig.1.6(b).

Polygons: An area fully encompassed by a series of connected lines. Because lines have direction, the system can determine the area that falls within the lines comprising the polygon. Polygons are often of an irregular shape. Each polygon contains one type of data (e.g., vegetation, streets, and dispatch locations would be different polygons). All of the data points that form the perimeter of the polygon must connect to form an unbroken line. When preparing files, the polygons are verified as closed area, see Fig.1.6(c).



(a) Point (b) Line (c) Region

Figure 1.6: Spatial Element Types



(a) traditional kNN input (b) traditional kNN output

Figure 1.7: Traditional kNN input and output are discrete points

Nearly all spatial queries are objects related which means both the input(fig.1.7(a)) and output (fig.1.7(b)) of the queries are discrete points, for example, traditional k nearest neighbor search tries to find the nearest neighbors (points) of query points. But lines have not been fully introduced into the query processing. Motivated by this point, the second main chapter of my thesis is called *Route and Path related kNN Queries*.

While in reality, path/route is another important element in spatial space, the user might want to input a path to find a set of points, or input a set of points to create an optimal path, or even input a query path and output a result path at the same time.

1.3 Contributions

Based on the problems and possible extensions of existing works listed in section 1.2, we list our contributions generally in this section. Altogether there are 2 main categories of contribution which includes 5 topics.

1.3.1 Contribution 1: Using the Voronoi Diagram to enhance the performance

As stated in section 1.2, network expansion is the widely used methodology to process the spatial queries based on underlying road network. The performance has become a significant drawback because each intersection node will perform traverse expansion to different directions. As a result, our first contribution of the thesis is using a Voronoi Diagram to merge the road segment although it requires pre-calculation. In this chapter, we have proposed 2 approaches which use the Network Voronoi Diagram as the methodology.

- **Query Optimization on Continuous kNN Query Search**

  In Section 3.2, we propose an alternative approach for Continuous k Nearest Neighbor query processing, which is based on a Network Voronoi Diagram (we call our proposed method VCkNN, for Voronoi CkNN).

  This approach avoids the weakness of existing work [GR03, GR99] and improves the performance by utilizing a Voronoi diagram. VCkNN ignores intersections on the query path; instead, it uses Voronoi polygons to subdivide the path. In section 3.2, the Voronoi diagram, which originates in computational geometry and has been used successfully in other areas, such as industrial electronic area [VS08], and will demonstrate its effectiveness in a mobile environment.

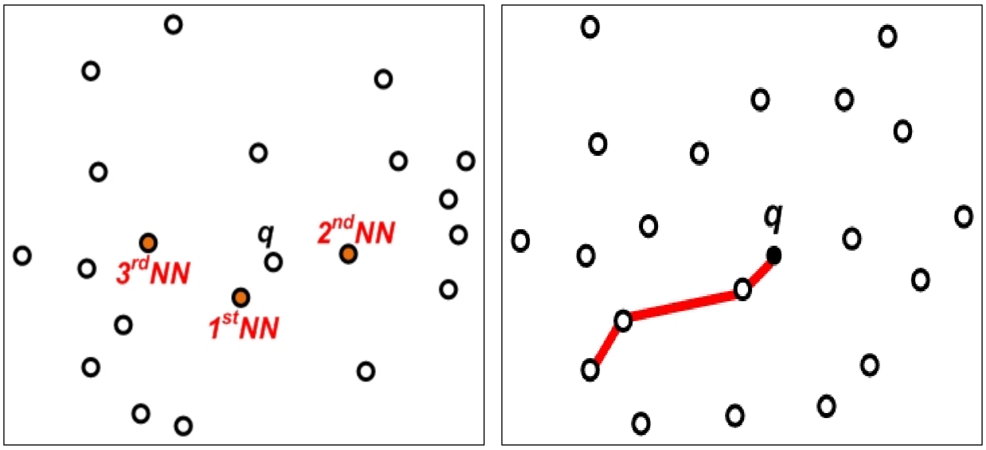
  Our proposed VCkNN approach is based on the attributes of the Voronoi diagram itself and using a piecewise continuous function to express the distance change of each border point. At the same time, we use Dijkstra's algorithm to expand the road network within the Voronoi polygon.

VCkNN, DAR [SE06] and IE [KS05] are all approaches for CkNN queries. But VCkNN is different from DAR and IE in most aspects. Therefore before introducing our VCkNN algorithm, we would like to highlight the main differences between VCkNN and DAR and IE: a) Path division mechanism, b)kNN processing, c) Sequence finding of split nodes, d)Processing split nodes. These are discussed in detail in section 3.2.

- **Query Extension on Multiple Objects Types**

  Current approaches of k nearest neighbor search focus on one object type, which narrows down the mobile query scope. For example, find the nearest 3 hospitals from my current location. In some cases, users may want to get kNN of different object types (multiple object types), as well as to obtain the shortest routes. Motivated by these, section 3.3 proposes new approaches on three different queries involving multiple object types using a network Voronoi Diagram. In these queries, more than one object type is considered and the query result is highly related with the object types. Every object belongs to one of the category and there is no overlap between categories. That is the basic property of a *multiple-object-type query.*

  Section 3.3 focuses on three different types of kNN mobile queries, including: a) query to find nearest neighbor for multiple types of interest point (or 1NN for each object type), b) query to give the shortest path to cover multiple-object-types in a pre-defined sequence, and c) query to find an optimum path for multiple object types that gives the shortest path that covers the required interest objects in a random sequence.



(a) kNN Demonstration (b) Path kNN Demonstration

Figure 1.8: Comparison of kNN and path kNN result

1.3.2 Contribution 2: Bringing route into kNN spatial queries

As stated in section 1.2, most of the existing queries put discrete points as input and output. Consequently, Chapter.4) concentrated on bringing route/path into the input or output or both of the queries.

- **Path based kNN Search Query**

  A possible query that a user may invoke is as follows: A market researcher wants to do a survey on restaurants and the sample size should be 10. The question is to find the shortest path for the user to visit all the 10 restaurants one by one. Range search cannot be used as there is no fix range. kNN search cannot be used either because after we visit the first interest point, the user may not want to return to query point and go to the second one. In this case, the user wants to continue to go to the second location from the first, and so on. This is a typical path based k nearest neighbor query (pkNN) and we propose a corresponding method in section 4.2 to process this type of query efficiently.

  Fig.1.8(b) shows the aim result of the Path based k nearest neighbor queries. Unlike fig.1.8(a) which considers all objects as discrete points, Path bases k nearest neighbor search is to find the shortest path which goes through k

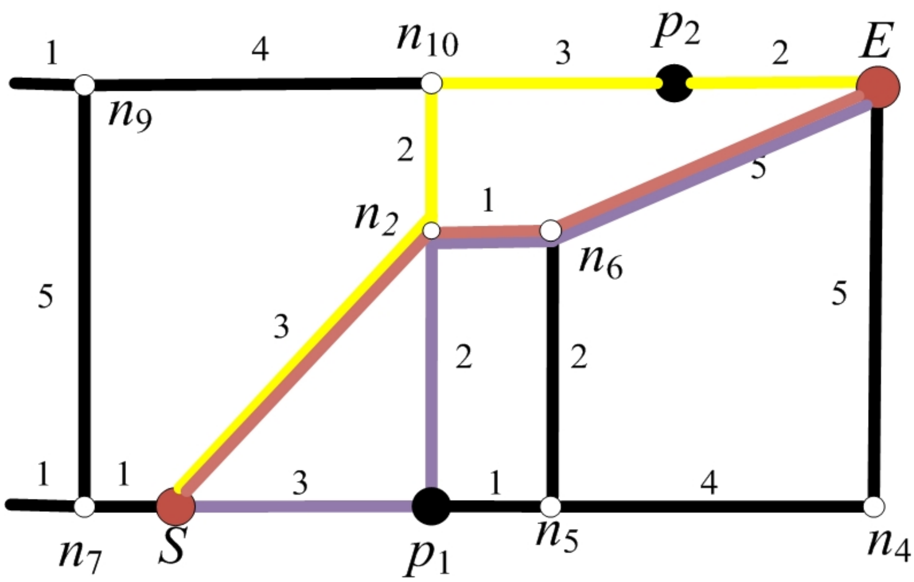


Figure 1.9: An example of Path Branch Point based kNN Search Queries

objects. In general, the overall distance of the path becomes the selection criteria.

- **Path Branch Point based kNN Search Queries**

  In section.4.3, we bring a novel query which is called path branch point (PBP). PBP can be defined as follows: given a set of candidate interest objects and a pre-defined path which starts at S and end at E, find a path which starts at S, via an interest point P and ends at E. This path should overlap with the pre-defined path as much as possible with acceptable distance increment. This is a novel query which is motivated by users' common requirements because most users have ad hoc paths in their daily travel and they can tolerate a longer driving distance to some extent if they can drive on a familiar path when they want to visit a certain type of object on the way. In this proposed approach, an Adjust Score is calculated for each path which is determined by overlapping distance and increased distance cost. The following example explains the query.

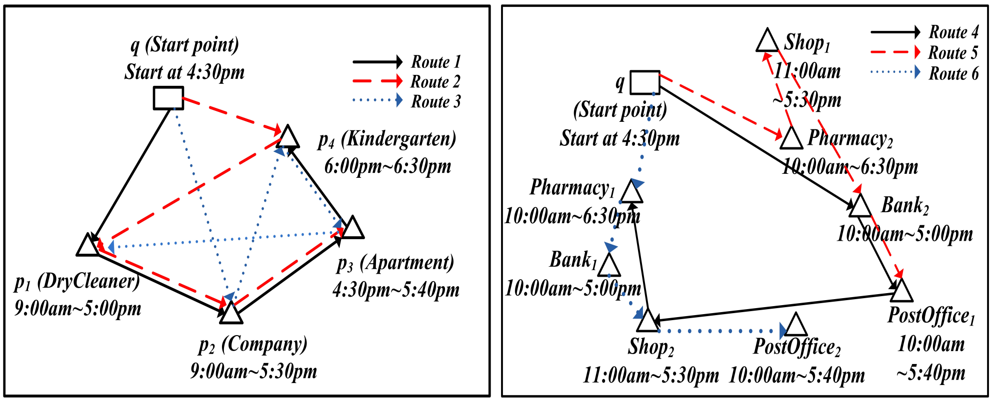
  Fig.1.9 shows an example of Path Branch Point based kNN Search Query. The path from S through red line to E is the query path. User query is to find an alternative path which starts from S and ends at E as well as on the way, an object (p) should be visited. So yellow (via object p_2) and purple lines (via object p_1) are two candidate results. We will calculate the *Adjust Score* for each candidate path and decide which path is optimal.

- **Time Constraint Route Search over Multiple Locations**

  Conventional route search queries aim at finding the shortest distance which is certainly useful, but often impractical, due to these following reasons: (i) each location or place, which are normally a spatial business entity (e.g. bank, dry cleaner, supermarket) has the opening hours - this implies that when this place is visited, it must be during their business hours; and (ii) the traveling time from one location to another needs to be considered, as in many cases, traveling time is more useful than the distance alone. Hence, in order to make route planning over visiting locations, we must take these two constraints into account. Section 4.4 defined these constraints as Time Constraints. Therefore, we proposed an approach called route search over multiple locations which takes time constraints into consideration (see fig.1.10).

  It is therefore imperative to assume that the route or path that arrives on the location outside the operation hours is considered as an invalid path. This problem exists in daily life, whereby we sometime have to choose a longer path to go back and forth to places just to meet the business hours of one location before its closing time. Hence, we need to draw time constraints into our proposed methods.

  In section 4.4, we focus on two problems of route search over multiple heterogenous locations: one for fixed locations, and the other for flexible locations. Fixed locations refer to predetermined locations by the user, such as Citibank on a specific location, Pharmore pharmacy on a specific location, etc. In this case, not only a specific business entity is specified, such as Citibank and not any bank, or Pharmore pharmacy and not any pharmacy, but also



(a) Fix Location Route Search (b) Flexible Location Route Search

Figure 1.10: Motivation of time constraint route search over multiple locations

the specific location, such as Citibank on 180 High Street, or Pharmore pharmacy on 25 Cure Road, etc. Hence, a Route Search over Fixed Locations (our proposed algorithm is then called $RFix$) finds the most efficient route to visit the user-defined fixed locations in a non-predefined order (see fig.1.10(a)).

Flexible locations, on other hand, refer to predetermined location types that are not the exact location itself. For example, if user wants to visit a pharmacy, which can be the pharmacy anywhere; or to visit Citibank, but can be in any branch. So, a route search over flexible locations for example is to find the most efficient route to visit Citibank, a pharmacy, etc, in a non-predefined order. Our proposed algorithm for Route Search over Flexible Locations is abbreviated as $RFlex$, (see fig.1.10(b)).

Both $RFix$ and $RFlex$ use the travel time network to estimate the travel time between any two locations, as well as using the time constraints imposed by not only the operating hours of each location, but also the traveling time itself.

1.4 Thesis Organization

This thesis is organized as follows (Refer to Fig. 1.11).

- Chapter 1 gives a brief introduction of spatial database, problems and contributions.

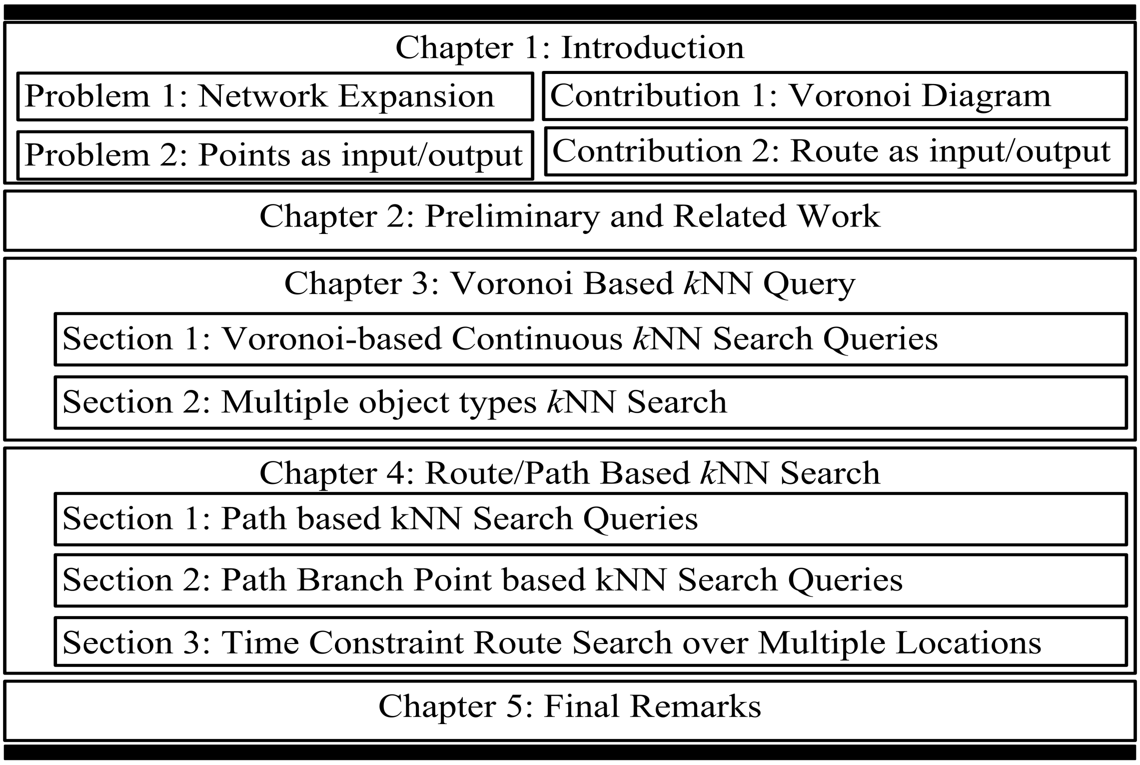


Figure 1.11: Thesis structure

- Chapter 2 provides a survey of the related works.
- Chapter 3 is the first main part of the thesis, which includes 2 approaches of Voronoi based k Nearest neighbor query. More specific descriptions are:
  - Section 3.2 optimizes the existing continuous k nearest neighbor search query by using a Network Voronoi Diagram. It improves the efficiency of the CkNN methods.
  - Section 3.3 presents a k nearest neighbor search query with multiple types of objects and uses a Voronoi Diagram to find k Nearest Neighbor which has been proven to outperform existing methods.
- Chapter 4 is the second main chapter, which includes 3 approaches of path/route based k Nearest neighbor search. More specific descriptions are:
  - Section 4.2 proposes a query that is called path based k nearest neighbor search. It aims at providing a path that visits k objects and the length of the path is the shortest.

  - Section 4.3 explains a query which is called path branch point route search. By given the query path and an object type, path branch point route search retrieves the optimal path that balances the overlap ratio of query path and the length of result path.
  - Section 4.4 describes a novel route research which adds time constraint into the search. In addition, a user may define the objects visiting sequence as sequential or random.

- Chapter 5 concludes our research, describes some of the open problems and provides several possible directions for future work.

Chapter 2

Preliminary and Related Work

2.1 Introduction

In this chapter, we consider the literature review on the work related to spatial query processing which requires the database system to access the objects with spatial features in the database. We briefly describe the queries which process the queries more accurately and efficiently. To sum up, in most cases, the distances between the objects are the merits in the results of these queries. The distance is calculated using Euclidean distance or network distance relying on the underlying road networks. The special purpose spatial index and query specific properties are used in order to reduce the system cost. As a result, the spatial objects are treated as points in the space throughout this thesis.

This chapter is constructed as follows:

Firstly, section 2.2 is the literature review with the motivation that originated this thesis. After introducing the basic concepts of spatial query processing elements, the following 5 subsections survey the related work on kNN queries summarized by Fig.2.4.

- Section 2.2.1, we review the basic concepts of spatial queries elements and the features of the Voronoi Diagram and the Network Voronoi Diagram.

- Section 2.2.2 describes the typical k nearest neighbor approaches including IER, INE and VN^3. IER, INE are using Network Expansion as the technique and VN^3 is using the Voronoi Diagram as its metrology. These two methods are the most popular methodology in spatial query processing named as Network Expansion and Voronoi Diagram. There is a brief demonstration of these methods illustrated and the differences are analyzes as well. The conclusion is drawn that under most of the case, the Voronoi Diagram outperforms Network Expansion. That is the motivation of the first main chapter of my thesis.

- Section 2.2.3 focuses on the continuous k nearest neighbor queries including DAR/eDAR and IE. Both DAR/eDAR and IE are using Network Expansion whereas we proposed another approach using Voronoi Diagram to merge the road segments into polygon. The performance Evaluation has proven that our new method can significantly improve the efficiency.

- Section 2.2.4 discusses the existing route search queries although it is still new to spatial queries. The second main chapter of the thesis is to enrich the route search queries.

- Section 2.2.5 introduces other k nearest neighbor queries including Reverse Nearest Neighbor Queries and Mutual k Nearest Neighbor Queries search.

Secondly, section 2.3 points out the outstanding problems after reviewing the existing works and formally defines the problems.

Finally, before proposing new approaches, section 2.4 concludes this chapter by using fig.2.13 to compare the contribution with the existing works in order to highlight the principle points of this thesis.

2.2 Related Work

2.2.1 Preliminaries

Spatial Queries Elements

Definition 2.2.1. ***(Road networks)*** *(R) is a weighted graph $G=\{V, E\}$, where V is a set of vertices $\{v_1, v_2,...v_n\}$, and E is a set of edges $\{e_1, e_2,...e_m\}$ and $\forall e_i \in E$, weight(e_i)$\in R^+$.*

In Def. 2.2.1, the underlying road network is constructed by choosing the layer of the map. In the road network diagram, the objects are called vertices while the connections between vertices are called edges, in other word, road segments.

Definition 2.2.2. ***(A vertex)*** *$v_i \in \{n_1, n_2,...n_j\} \cup \{p_1, p_2, ... p_k\}$, where n is an intersection node, and p is an interest point.*

Definition 2.2.3. ***(Vertex Scope)*** *Let $N=\{n_1, n_2,...n_j\}$ be a set of intersection nodes, and $P=\{p_1, p_2, ... p_k\}$ be a set of intersection points, then* ***a vertex*** *$v_i \in \{N \cup P\}$.*

Def. 2.2.2 and Def. 2.2.3 defines the scopes of vertices, which includes intersection nodes and interest objects. N represents the set of intersection nodes while P represents the set of interest objects.

Definition 2.2.4. ***(Weight)*** *$\forall e_i=(v_i, v_j) \in E$,* ***weight(**$e_i$**)*** *$= d_{net}(v_i, v_j)$, where d_{net} is the network distance between v_j and v_k.*

The weight of the edges is determined by the measure of the query (def.2.2.4. If the optimal result is based on the travel time, then the weight of the edges is the cost of travel time, while in this chapter, the shortest network distance determines the weight because the query result is defined as the shortest path.

Fig.2.1 is an example of road networks, in which road network intersections n_1-n_{10} (white points), and interest points p_1-p_3 (black points) are vertices and the solid lines connecting these vertices are edges. The number on each edge represents the

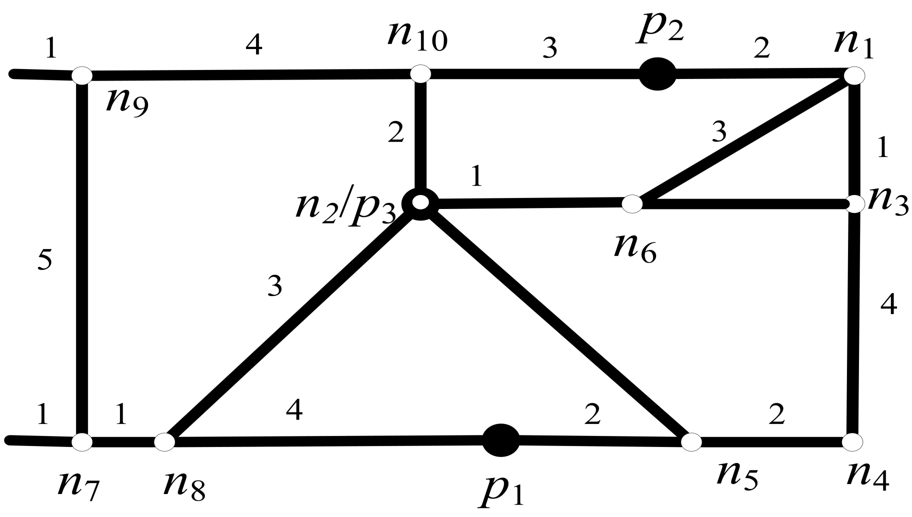


Figure 2.1: An example of road networks

shortest distance, in other words, the weight of the edge. Most of the spatial query is based on Euclidean distance. In reality, the distance between objects should not be determined by the length of the direct line linked objects. The network distance between objects suits the spatial query the most.

Voronoi Diagram based on Euclidean Distance

Voronoi Diagram is a special kind of decomposition of a metric space determined by distances to a specified discrete set of objects in the space [OBSC00]. Given a set of points S, the corresponding Voronoi diagram will be generated. Each point s has its own a Voronoi cell V(s), which consists of all points closer to s than to any other points. The border points between polygons are the collection of the points with equation of distance to shared generators. Fig.2.2 gives an example of Voronoi Diagram based on Euclidean distance. Pi represents the interest points and the lines are the shared border edge between polygons.

There are some basic properties associated with Voronoi Diagram, which have been well presented by Okabe, et al [Saf05]. We will list some of the relevant properties below:

- Property 1: The Voronoi diagram of a point set P, $V(P)$, is unique.

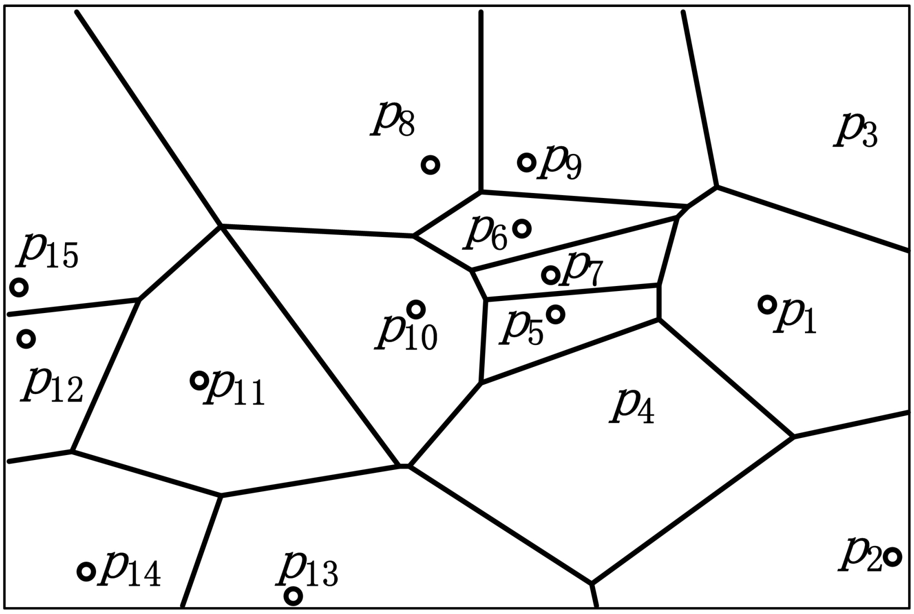


Figure 2.2: The Voronoi Diagram

- Property 2: The nearest generator point of p_i (e.g. p_j) is among the generator points whose Voronoi polygons share similar Voronoi edges with $V(p_i)$.

- Property 3: Let n and ne be the number of generator points and Voronoi edges, respectively, then $ne \leq 3n$-6.

- Property 4: From property 3, and the fact that every Voronoi edge is shared by exactly two Voronoi polygons, we notice that the average number of Voronoi edges per Voronoi polygon is at most 6, i.e., $2(3n\text{-}6)/n$ = 6-12/$n \leq 6$. This means that on average, each generator has 6 adjacent generators.

Using Voronoi Diagram to find nearest neighbor will let the algorithm perform more efficiently as all distance between borders and generators can be pre-calculated and stored. VN^3 and PINE utilize Voronoi diagram efficiently to find kNN. While currently there is no CkNN approach using Voronoi diagram to ignore the real network connection within the polygon, this point becomes our motivation of this chapter, Voronoi-based CkNN.

Figure 2.3: Network Voronoi Diagram

Network Voronoi Diagram

Voronoi diagram mentioned previously is the Voronoi diagram based on Euclidean distance. In the real world, when we want to search nearest neighbor or to generate the appropriate moving path, we use network distance, and not Euclidean distance. Network Voronoi Diagram is the Voronoi diagram, which uses network distance to generate the diagram, instead of Euclidean distance [XZTS08, Saf05]. In a typical Voronoi diagram, the shared borderline is the mid perpendicular of the links connected with two corresponding generators. However, in Network Voronoi Diagram, the borderline consists of discrete points, which are the middle points of network roads connected with two corresponding generators. A polygon in a network is the set of nodes and edges, which are closer to one generator than to any other. This is the principal difference between Voronoi Diagram and Network Voronoi Diagram. Network Voronoi Diagram will be used in our proposed method. The most basic property is the generators with shared border points have equal network distance to the same border point they shared. In fig.2.3, the different colors represent different polygons and the border points of the Network Voronoi Diagram are the discrete points on the roads.

Category	Query	POI	Remarks	Existing Works	Techniques
Typical *k*NN	Static	Static	*k* near neighbor of *q*	*Incremental Euclidean Restriction (IER)*	*Network Expansion*
				Incremental Network Expansion(INE)	
				Voronoi-based k nearest neighbor search (VN3)	*Voronoi Diagram*
Continuous *k*NN	Moving	Static	Find *k* near neighbor of a ***moving** q*	*DAR/eDAR*	*Segment Split / Network Expansion*
				Intersection Examination (IE)	
Route Search	Route	Static	Find *Route* for a query *q*	*Efficient Orienteering Route Search over Uncertain Data*	*Network Expansion/ Pruning&Fileting*
				Incremental Route Search Query	
*k*NN Variants	Static	Static	Find POIs consider *q* as *k*NN	*Reverse Nearest Neighbor Queries*	*R tree*
	Static	Static	Find mutual *k*NN of dataset	*Mutual k Nearest Neighbor Search*	*R tree, Pruning...*

Figure 2.4: Related Work Summary Chart

In the following 4 sections, we are going to categorize existing works which are highly related to this thesis. The following chart, fig.2.4, shows the general categorized criterion as well as the techniques that are used in these existing works.

2.2.2 Typical kNN Queries

The existing methods for static kNN cover Incremental Euclidean Restriction (IER), Incremental Network Expansion (INE) [PZMT03] and Voronoi Based Network Nearest Neighbor (VN^3).

Incremental Euclidean Restriction (IER)

Incremental Euclidean Restriction (IER) was proposed in 2004 [PZMT03]. Firstly, IER uses the entity R-tree to retrieve the k^{th} node's Euclidean distance. Secondly, IER restricts the interest point by this distance and calculates every point's network distance to a query point that is within the range of k^{th} node's Euclidean distance and then sorts them in ascending order of network distance to query point. Then set the k^{th} node's network distance as d_{max}. Finally, for the following interest points in Euclidean distance sequence continue to calculate their network distance to query point. If it is smaller than d_{max}, insert in into network distance queue and update

d_{max}. These operations will terminate until next node's Euclidean distance is larger than d_{max}.

IER algorithm performs well when there are not many false interest points which means its Euclidean distance falls into the restrict zone while its network distance is far from the query point. Too many false points will reduce the performance sharply and if the density of the interest points is high, the performance of IER leaves much to be desired.

For example, there are 10 interest points and their Euclidean distances to query point are $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. The query is 3NN.

Firstly, the 3^{rd} node according to its Euclidean distance will be p_3.

Secondly, calculate $\{p_1, p_2, p_3\}$ network distance to query point. Suppose $\{DN(p_1), DN(p_2), DN(p_3)\}= \{3, 7, 12\}$. $d_{max}= 12$.

Then as next node is p_4, calculate its network distance $DN(p_4) = 4$. Then insert p_4 into the queue and d_{max}=7. Continue to do p_5, $DN(p_5) = 5$. Then $d_{max}= 5$. Queue will become $\{p_1, p_4, p_5\}$.

Finally, next node will be $p_{6,}$, because its Euclidean distance (6) is larger than $d_{max}(5)$. This algorithm terminates.

Incremental Network Expansion(INE)

Incremental Network Expansion (INE) is an approach of k nearest neighbor query that was proposed in the same paper of IER. The INE algorithm is based on Dijkstra's algorithm. The basic idea of INE is network expansion.

INE firstly locates the query point to find which segment includes the query point. It records the start and end nodes of this segment with their distance to query point, puts them in RS and sorts them in ascending sequence. Then checks whether there is any interest point on this segment and add these points into the result list. Expand the top node in RS, add its adjacent nodes into RS and loop these operations until kNN has been found. This distance should be record as d_{max}.

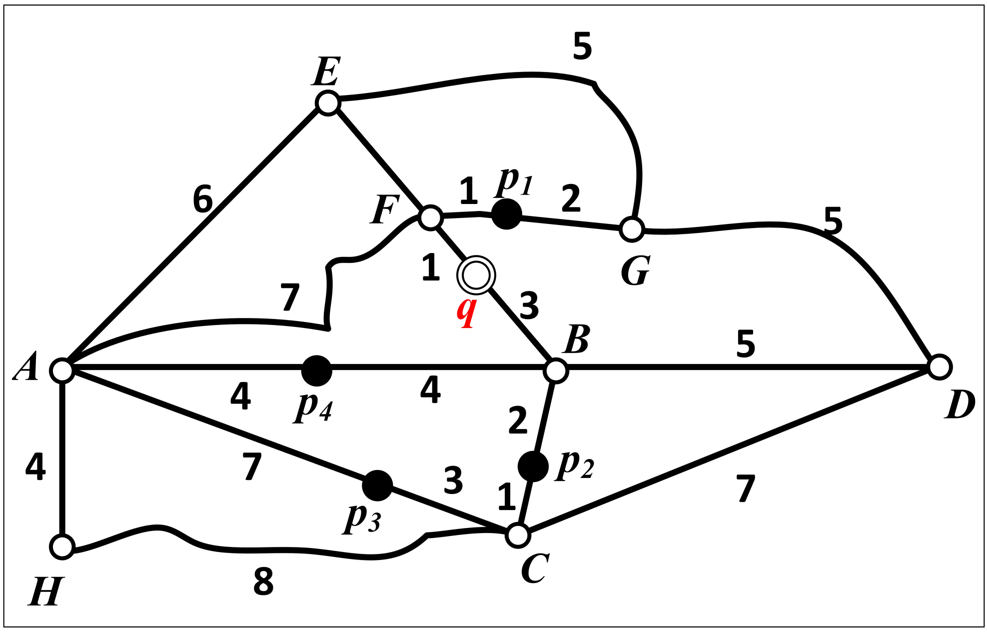


Figure 2.5: An example of INE query

Continue to expand the other nodes in RS, once the kNN has been found, update d_{max} if the new distance is smaller than origin d_{max}. Terminate the algorithm if all other nodes in RS have a larger distance to q than d_{max}.

Take Fig.2.5 as an example. The query is defined as finding 2NN from query point q.

Firstly, locate q to find the segment which covers q, the result is FB. Then check whether any object exists in this segment.

Secondly, add F and B into S set. $S = \{(F,1), (B,3)\}$. Sort the nodes in S by their distance to R in ascending order. Then add F in the top of S, expand F and check whether any object is on FA or FG or FE. We find p_1 as first NN. Then add all F's adjacent node into S set and sort again. $S = \{(B,3), (G,4), (E,4), (A,8)\}$.

Thirdly, expand B and check whether any object is on BA. Then we find p_4 as the second NN, and $d_{max} = 7$. Check whether any object is on BC. If p_2 is nearer than p_4, update $d_{max} = 5$ and $S = \{(G,4), (E,4), (C,6), (A,8), (D,8), (A,11)\}$.

Finally after expanding G and E, the distance is over d_{max}, INE terminates.

The main advantage of INE is that its architecture can be used in other query solutions, although compared to PINE its performance is not the best [Saf05].

Voronoi-based k nearest neighbor search (VN^3)

Voronoi-based k nearest neighbor search (VN^3) was proposed [KS04] in 2004. VN^3 is based on the properties of the Network Voronoi diagrams and also localized pre-computation of the network distances for a very small percentage of neighboring nodes in the network. In general, it keeps the result in ascending order, adopts a filter and makes refinement steps to generate and filter candidate results, it also uses localized pre-computed network distances to save response time.

To talk in detail, the first nearest neighbor query point can be told directly by intuition via the Voronoi diagram. The polygon that contains the query point will be the first nearest neighbor. Subsequently, $1^{st}NN$'s adjacency information can be utilized to provide a candidate set for other nearest neighbors of q. Finally, the actual network distances from q to the generators in the candidate set can be pre-computed and this step will refine the set. The filter/refinement process in VN^3 is iterative: at each step, firstly, a new set of candidates is generated from the NVPs of the generators that are already selected as the nearest neighbors of q, then the pre-computed distances are used to select only the next nearest neighbor of q. Hence, the filter/refinement step must be invoked k times to find the first k nearest neighbors of q ([KS04]).

The following figure.2.6 shows the example of VN^3. Suppose the query is 3NN.

First of all, we can tell 1^{st}NN of query point is p_1. Then candidate set CS updates as $CS = \{p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$.

Secondly, calculate the distance from candidate interest points in CS to query point and select the 2^{nd}NN. Suppose it is p_6. Then update CS by pop out 2^{nd}NN and add all its adjacent interest points into CS.

$CS = \{p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_{18}, p_{17}, p_{16}, p_{15}\}$.

Subsequently, repeat the previous step and suppose p_5 is the 3^{rd}NN. As $k = 3$, which means all interest points have been found, the algorithm terminates.

In summary, VN^3 performs well if we are only concerned with a static kNN query.

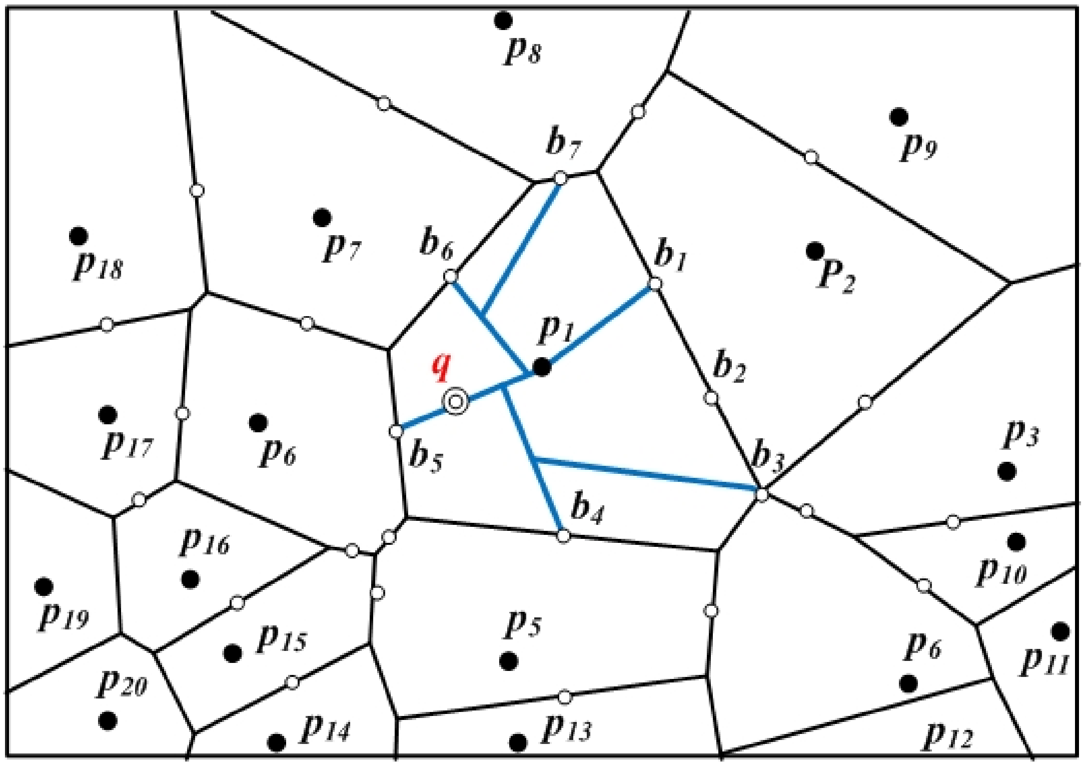


Figure 2.6: An example of VN^3 query

2.2.3 Continuous kNN Queries

If the query point is moving, it is infeasible to apply kNN at every point of the line, because it will generate a large number of queries and a large overhead. So the objective of a moving or continuous query is to efficiently find the location of the split node(s) on the path, in other words, where kNN changes. There are two important existing works on continuous k nearest neighbor (CkNN) based on network distance. The first one is DAR and eDAR based on PINE, proposed by Safar and Ebrahimi [Saf06]. Another CkNN work is Intersection Examination (IE) based on VN^3 proposed by Kolahdouzan and Shahabi [KS05]. Hence, the following section will discuss these two works and analyze their strengths and weaknesses.

DAR/eDAR

DAR/eDAR was proposed by Safar and Ebrahimi [Saf06]. These are based on PINE, which uses road networks as the underlying map. These two algorithms start by dividing the query path into segments, each of which is separated by a network

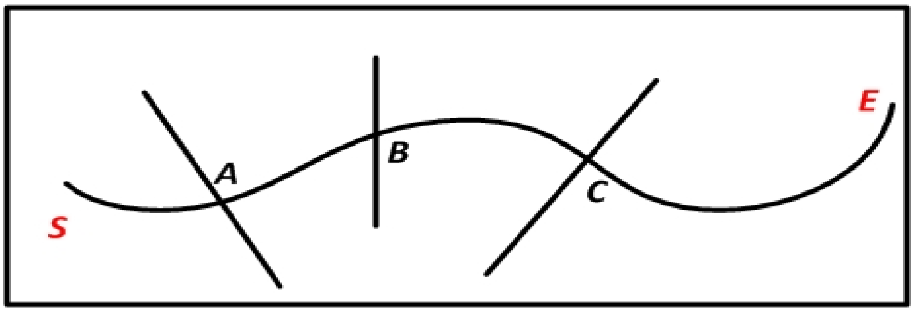


Figure 2.7: An example of DAR: step one

intersection node. Then they find kNN tables for two adjacent nodes, compare the two tables, and swap the position to make these two the same. Every swap would incur a split node, and when the two tables are exactly same, all split nodes have been found. Then split nodes' position and kNN tables are the result of the query. In order to illustrate this clearly, the following shows an example of the process.

Firstly, we divide the query path into segments using the intersect nodes on the path as shown in fig.2.7. In this example, the query starts from S and ends in E. The path from S to E has a number of intersections, and the path separated by an intersection is a segment. In this example, the path from S to A is one segment, and from A to B is another, and so on.

Secondly, for every segment (e.g. like AD in fig.2.8), we find the kNN of the two ending points (A and D), from which we generate two kNN lists for both ending points (see fig.2.9, assume the query is 2NN). Then we aggregate these lists to form one complete list (see fig.2.9).

Then for every adjacent interest points, calculate λ according to the following formula (note that I is the distance column in the ready queue RQ for a particular interest point, and $Dist$ is a distance function).

$$\lambda_{i,i+1} = \frac{Dist(A, D) + Dist(D, I'_i) - Dist(A, I'_i)}{2}$$

Then apply the same operation between the last interest point and every point in RQ. The smallest λ will be the moving direction of query point. Swap the list to find another split until the two lists are the same.

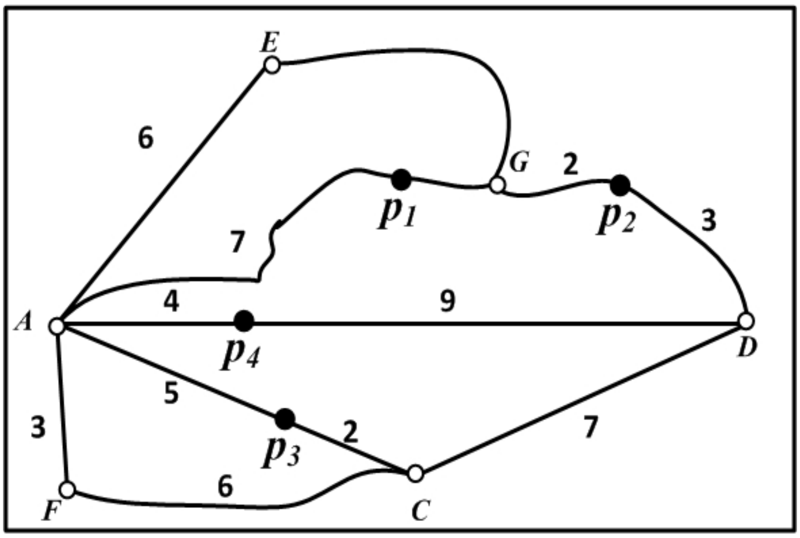


Figure 2.8: An example of DAR: Step 2

A's list will be

Interest point	Distance
P_4	4
P_3	5

D's list will be

Interest point	Distance
P_2	3
P_1	7

Then create a complete list for A and D

A	Interest point	Distance
PQ	P_4	4
	P_3	5
RQ	P_3	9
	P_3	9

D	Interest point	Distance
PQ	P_2	3
	P_1	7
RQ	P_1	7
	P_2	11

Figure 2.9: An example of DAR: Step 3

It is an undeniable fact that DAR and eDAR perform well for a CkNN query, except that they divide the query path into segments. This will let the performance go worse as the number of intersections increases. Also a large number of overheads will be incurred even if there is no split node in some segments. Nevertheless, we need to do kNN for every segment although we find no split node. In view of the above mentioned reasons, an approach should be proposed which does not take intersections into account.

Intersection Examination (IE)

The second approach of CkNN is Intersection Examination (IE) which is based on VN^3. In general, similar to eDAR, IE separates the query path into segments. IE then tries to find the split nodes by defining the trend for each interest point in the current kNN result list and sorts them in an ascending order. When there is any change in the position of the interest point, it becomes a split node.

To be specific, if the query is to find continuous 1NN, it can simply find all nodes that intersect with the border of the Voronoi diagram. The IE algorithm divides the query path into smaller segments using the intersection nodes on the path. From every segment, IE uses VN^3 to find kNN for the two terminating nodes.The kNN results of every segment should be within the combination congregation of the kNN result of the two terminating nodes. We can obtain the trend of every interest point at the start point's kNN results, and then find the point where two adjacent nodes have the same distance to the query point, that is the split node.

Similar to DAR and eDAR, IE indeed is an alternative approach to a CkNN query, except that it also needs to divide the query path into segments. Using IE, the trend of interest points can be monitored either moving closer or away from the current position of the query. Our approach of Voronoi CkNN will provide a more comprehensible way to let the user read kNN results for any node on the query path.

2.2.4 Route Search query

Route search has many important applications in various fields such as commerce, transportation, tourism, security and health-care services. In such applications, a route search should be efficient, intuitive and expressive, allowing a user to specify complex search queries and receive an immediate answer. However, current route-search applications on the Web are limited to a point-to-point search. When computing a route, different goals and constraints can be defined, such as minimizing the traveling length, limiting the route to be over roads of a certain type, etc.

In this section, we are going to review some popular route search query although the area has not been widely discussed followed by couple of algorithms discussed in detail because they are close to the approaches I proposed in the thesis. More specifically, we will present the related work on efficient orienteering-route search over uncertain spatial data sets and Incremental Route Search Query.

Efficient Orienteering-Route Search over Uncertain Spatial Data sets

Paper [DKD08] considers route search over uncertain data sets. Spatial data might be instinctively uncertain due to various reasons such as its acquisition process, imprecise modeling and manipulation. An uncertain data set can contain correct and incorrect objects. The uncertainty of the data represents a confidence value indicating its probability to be correct. When it is a real-world entity, the object is considered as correct, it is considered as incorrect otherwise. A user may be able to test the correctness of an object by visiting the entity at the location of that object. In this paper, the author defined a problem called a generalization of the Orienteering Problem (OP). OP considers a route search where the aim is finding a route that starts at a given location and traverses through as many correct objects as possible without exceeding a given distance. Finding a solution to OP is a problem that cannot be computed efficiently because OP is a generalization of TSP (Traveling Salesman Problem); hence, it is an NP-hard (nondeterministic polynomial-time hard) problem that is unlikely to have a polynomial-time algorithm. This paper presents heuristics to OP that are efficient and scalable.

The Greedy Algorithm: The first algorithm proposed in paper [DKD08] is called the greedy algorithm. The greedy algorithm constructs a route iteratively by making the most profitable increase in each step. Suppose P_i is the path constructed in step i and let o_i is the last object of P_i. P_i will be considered as the starting point s in step 0. For each step i, the algorithm checks the set N of objects that are in D and are not already in P_i. In each step, the object o' is retrieved from N if $distance(o_i, o')/confidence(o') \leqslant distance(o_i, o'')/confidence(o'')$ for any object

o'' in N. If $length(P_i)$ $+distance(o_i, o')\leqslant L_{max}$,it adds the $edge(o_i, o')$ to P_i and continues to step i+1. Else, it returns P_i. The performance evaluation illustrated that the greedy algorithm is simple and relatively efficient as it does not require any preprocessing and its time complexity is $O(|D|^2)$ where $|D|$ is the size of D. The greedy algorithm is effective when the objects of D are uniformly distributed, i.e., the data set is uniform in all directions and their confidence values have a small variance, i.e., when all the confidence values are approximately equal. In other word, the greedy algorithm for any direction performs as well as in any other direction, and the produced route will have an expected prize value close to the optimal. However, when the data set is not uniform, the greedy algorithm may not provide good results.

The Double-Greedy Algorithm: The Double-Greedy Algorithm (DG) in paper [DKD08] is an improvement of the Greedy Algorithm. The Double-Greedy Algorithm (DG) intuitively examines pairs of edges for deciding which node to add. Formally, in step i, the algorithm extends P_i by adding the object o' such that there exists o'' for which $confidence(o')/distance(o_i, o') + confidence(o'')/distance(o', o'')$ $\geqslant confidence(o*)/distance(o_i, o*) + confidence(o**)/distance(o', o**)$ for any $o*$ and $o**$ that are in D and are not in P_i (Note that also o' and o'' are in D and are not in P_i). Algorithm DG has time complexity $O(|D|^3)$. In order to increase efficiency, DG checks a pair of edges only when the following condition holds: α * $distance(o_{i-1}, o_i) \geqslant distance(o_i, o')$, where $\alpha\geqslant 1$ is a fixed factor. Intuitively this condition is satisfied when the next edge we consider to add to the route is much longer than its preceding one. The factor α is to detect when the route leaves a cluster and we want to direct the route to a new cluster.

The Adjacency-Aware Greedy Algorithm: Motivated by the entities's distribution, for example, hotels are usually located near the coast or near tourist sites; restaurants are located in the city center, clusters should be taken into account. Given a data set that contains clusters of objects, a good heuristic for constructing an OP route is to give precedence to objects that are in a cluster over objects that are not in a cluster. This is defined as The Adjacency-Aware Greedy Algorithm (AAG).

AAG does modeling on the given data set as a directed weighted graph where the objects of the data set are the nodes and the weight of the edge between every pair of nodes is a combination of the distance between the objects and the confidence of the target node. Then, AAG computes for each node the probability of reaching this node in a random walk on the graph. Next, AAG replaces the confidence values on nodes by a combination of the confidence values and the random-walk probabilities. Finally, it applies the greedy algorithm using the new values. AAG outperforms the other algorithms of the Greedy for data sets that have clusters. The AAG improves the Greedy algorithm by giving a higher weight to objects that have many near neighbors, especially if the near neighbors have high confidence values.

The Adjacency-Aware Greedy Algorithm with Buffering: The Adjacency-Aware Greedy Algorithm with Buffering starts by a similar computation as in AAG, and for each edge in the route, AAGB builds a buffer. It applies a pre-processing step similar to AAG by the calculation of new weights. In addition, it finds the distance between every pair of objects in D, and it computes the mean of these distances, denoted this mean by $\overline{L}$. AAGB constructs the route greedily in the same way as AAG, but uses a buffer to add objects that are near the route constructed by AAG. The buffering is computed as follows. Suppose in step i the last object is o_i, AAGB increases the route by adding the object o_j. $d_{i,j} = distance(o_i, o_j)$ and $b_{i,j}$ is the width of the buffer. We compute the size of $b_{i,j}$ to guarantee that $\Delta L_{i,j}$ $= distance(o_i, o') + distance(o', o_i)$ - $d_{i,j} \leqslant \overline{L}$. That is, the added distance by going to some object o' in the buffer $\Delta L_{i,j}$ should not exceed the mean distance between objects in the dataset.

Incremental Route Search Query: The following paragraphs are going to summarize some existing works of route search query which have attracted increasing attention nowadays. Li et al. [LLT11] propose a new query called Trip Planning Query (TPQ) in spatial databases, in which each spatial object has a location and a category, and the objects are indexed by an R-tree. Each Trip Planning Query consists of three components: a start location s, an end location t, and a set of categories C, and it is to find the shortest route that starts at s, passes through at least one object from each category in C and ends at t. TPQ has been proven that it is a deduction from the Traveling Salesman problem (NP problem). Based on the triangle inequality property of metric space, two approximation algorithms including a greedy algorithm and an integer programming algorithm are proposed.

Compared with TPQ, keyword-aware optimal route query, denoted by KOR, which is to find an optimal route such that it covers a set of user-specified keywords, a specified budget constraint is satisfied and the objective score of the route is optimized. The problem of answering KOR queries is NP Problem. Paper [CCCX12] devises two approximation algorithms, i.e., OSScaling and BucketBound. Results of empirical studies show that all the proposed algorithms are capable of answering KOR queries efficiently, while the algorithms BucketBound and Greedy run faster. We also study the accuracy of approximation algorithms.

Sharifzadeh et al. [SKS08] propose a variant problem of TPQ, called optimal sequenced route query (OSR). A total order of OSR on the categories C is imposed and only the starting location s is specified. Two elegant exact algorithms LLORD and R-LORD are proposed to deal with query OSR. OSR are constructed under the same setting which is indexed by an R-tree. The metric space based pruning strategies are developed in the two exact algorithms.

Chen et al. [CKSZ08] define the multi-rule partial sequenced route (MRPSR) query, which is a unified query of TPQ and OSR. Three heuristic algorithms are proposed to answer MRPSR. KOR is different from OSR and MRPSR and their algorithms are not applicable to process KOR.

Kanza et al. [KSSD08] brings in a different route search query on the spatial database: the length of the route should be smaller than a specified threshold while the total text relevance of this route is maximized. Greedy algorithm is proposed without guaranteeing to find a feasible route. Their team develop several heuristic algorithms for answering a similar query in an interactive way [KLSS09]. The progress is like this: the user provides feedback on whether the object satisfies the query after visiting each object and the feedback is considered when computing the next object to be visited. Another work proposed by this team, [LKSS10], developed approximate algorithms to solve OSR with order constraints in an interactive way. Kanza et al. also study the problem of searching optimal sequenced route in probabilistic spatial database [KSS09].

Malviya et al. [MMB11] is aiming at answering continuous route planning queries over a road network, in other words, to find the shortest path in the presence of updates to the delay estimates.

Roy et al. [RDAYY11] consider the problem of interactive trip planning. The query helps the users' itineraries based on the users preferences and time budget.

Yao et al. [YTL11] propose the multi-approximate-keyword routing (MARK) query. A MARK query is specified by a starting and an ending location, and a set of (keyword, threshold) value pairs. It searches for the route with the shortest length such that it covers at least one matching object per keyword with the similarity larger than the corresponding threshold value.

2.2.5 Other k Nearest Neighbor queries

In this section, the popular variants of nearest neighbor queries are described. More specifically, we present the related work on reverse nearest neighbor queries in section 2.2.5. Then section 2.2.5 reviews the mutual k nearest neighbor queries.

Reverse Nearest Neighbor Queries

Reverse nearest neighbor queries search, as one of the most popular variant of nearest neighbor query, focuses on the inverse relation of k nearest neighbor search. The definition of reverse neighbor query (RNN) is to find all the objects that consider q as nearest neighbor, which is formally defined in Definition 2.2.5.

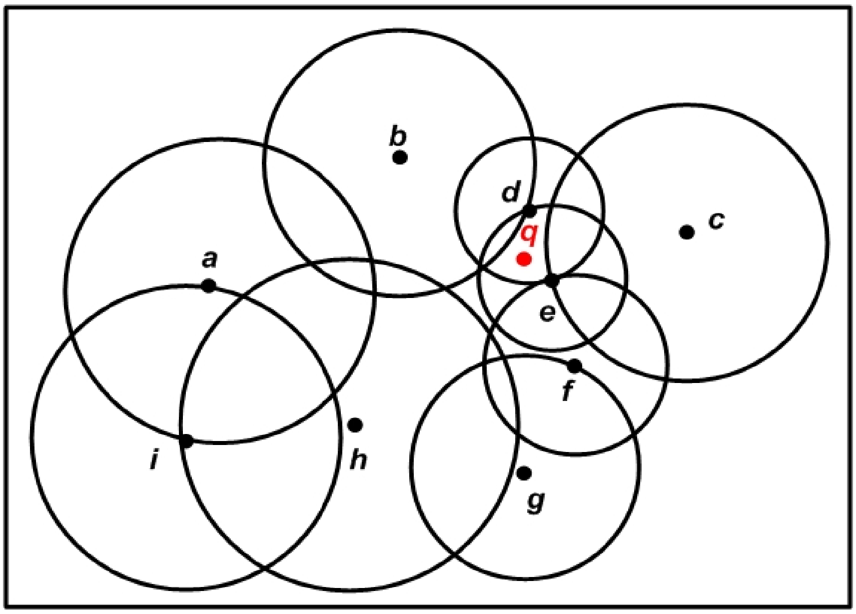


Figure 2.10: An example of reverse nearest neighbor query approach

Definition 2.2.5. *Reverse Nearest Neighbor: Given a set of objects P and a query object q, a reverse nearest neighbor query RNN* $= \{p \mid q = NN_p,\ p \in P\}$.

The query result set of RNN may contain 0 element or one or more elements. In [Ber93], the formal definition of reverse nearest neighbor query and some applications are proposed. For example, when a shopping mall chooses a site to open a new branch, we may use RNN to find the customers effected by this shopping mall. Moreover, RNN can also be used to choose the location which maximizes the number of potential customers. That is a bichromatic example. Take another monochromatic example, a RNN query may be issued to find petrol stations that are affected by opening a new petrol station at the new site. In summary, a bichromatic query (the first example) is to find the reverse nearest neighbors within two different types of objects. A monochramtic RNN query (the second example) is to find the reverse nearest neighbors where the data set contains only one type of object [Ber93].

There are a lots of existing approaches of reverse nearest neighbors proposed in the past few years [SRAE01, SAE00, MVZ02, LNY03, YL01]. We will briefly describe some most popular and general algorithms in the following paragraphs.

In paper [KM00], a RNN query firstly pre-calculates a circle of each object p that its nearest neighbor lies on the perimeter of the circle as shown in Fig.2.10.

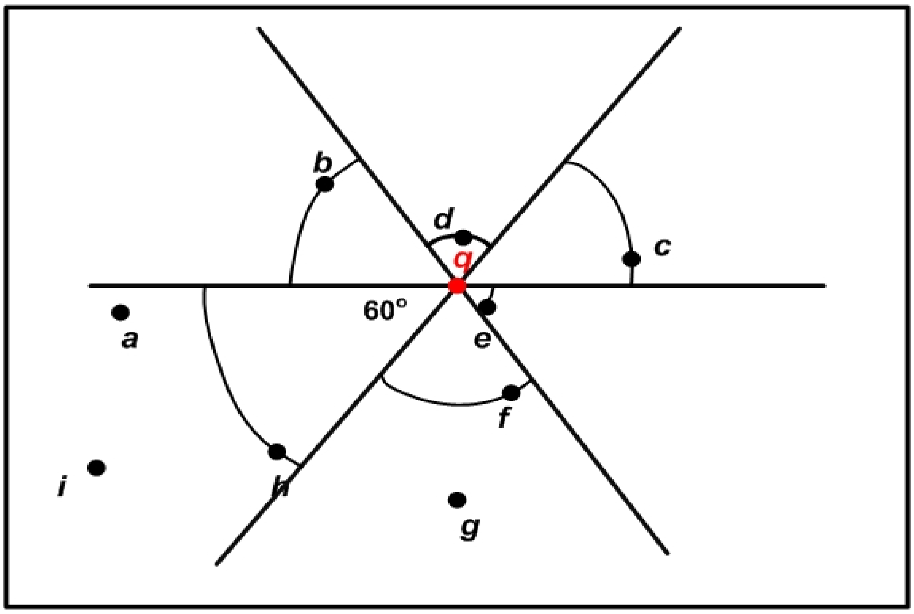


Figure 2.11: An example of reverse nearest neighbor query approach - SAA

Another technique that does not have any preprocessing involved was proposed by Stanoi et al. [SAE00], denoted as SAA. They partitioned the whole space centered at the query point q into six equal regions of 60 degrees each (b, c, d, e, f and h in Fig. 2.11). In each region, only the nearest neighbor to q can be the reverse k nearest neighbor result. So the other point in the same region can be pruned. Take fig.2.11 as an example, in the left-below region, assume h is the nearest neighbor of q, they observe that for a nearest neighbor object h of q in this region; either h is the RNN of q or there is no RNN in this region. But we can observe that h is closer to i than q. As a result, there is no RNN of q in this left below region and we do not need to consider other objects in this region. To sum up, SAA processes RNN queries in two steps: firstly, find the nearest neighbor for each of the six regions and then form as a candidate list. Secondly, for each point in the candidate list, generate its nearest neighbor query. If the result is q, it should be included into q's RNN result. Otherwise, discard this point. As a result, a, i and g are pruned. When $k \geq 1$, the RkNN queries can be solved in a similar way, i.e., in each region, the kth nearest neighbor of q defines the pruned area.

Another reserve k nearest neighbor search approach is proposed by Tao et al [TPL04], denoted as Half-Plane Pruning RNN. It brought the idea of perpendicular

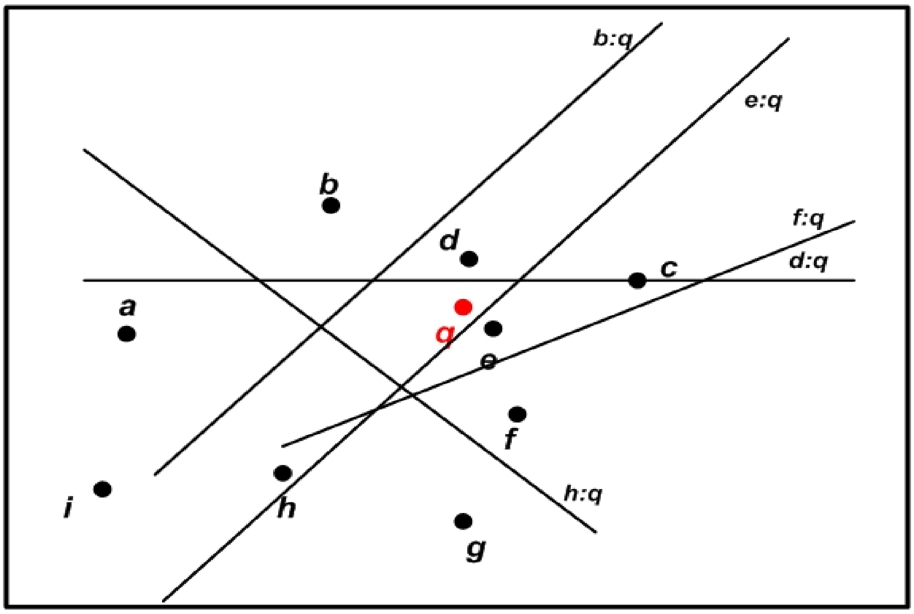


Figure 2.12: An example of reverse nearest neighbor query approach - Half-Plane Pruning RNN

bisector into the methods in order to reduce the search space. Firstly we link d with q, then find the midpoint of the the link dq. After that we form the perpendicular bisector line (line $d:q$) with the link dq. Line $d:q$ divides the space into two half planes PL_q and PL_d where PL_q contains q and PL_d contains d. In other words, there will not be any point considering q as nearest neighbor in plane PL_d because in this plane, points are closer to d other than q. Based on this property, we can use the line to prune the MBRs which completely fall into Plane PL_d. The same steps are invoked for the rest of the objects until the smallest region is found. Their approach can also be extended to answer RkNN queries that is to find all objects for which q is one of their k nearest neighbors (see fig.2.12).

Mutual k Nearest Neighbor Search

Definition 2.2.6. *Given a dataset P, a query point q and user defined k, mutual k Nearest Neighbor search is to find the set of object $S \subseteq P$, that $S = p \mid p \in NN_k(q)$ and $q \in NN_k(p)$, $\forall p \in S$.*

kNN search is asymmetric. However, MkNN retrieval is symmetric. For example, MkNN$(p_1) = \{p_2\}$ indicating that MkNN$(p_2) = \{p_1\}$.

The following items list the popular mutual k nearest neighbor approaches [GZCL09]. SP is very inefficient in terms of I/O overhead and CPU cost, especially for large values of k. To overcome this deficiency, the last 4 approaches are proposed to improve the performance of MkNN query processing via different optimization techniques.

- Simple processing algorithm (SP)

  Simple processing algorithm (SP) is proposed based on the definition of MkNN query by [GZCL09]. It firstly conducts a kNN search to retrieve the candidate set $CandidateSet = kNN_q$. Then it verified each candidate $p \in CandidateSet$. The verification of a candidate p can be conducted again via a kNN search to check whether $q \in kNN_p$. If the result is yes, it means that q is among the kNN of p and hence p is returned as an answer object. Otherwise, p is discarded, i.e., it is a false hit.

- Two-step Algorithm (TS)

  As every object included in the candidate set $CandidateSet$ need to be verified in the SP algorithm, the overhead and cost are extremely high. Since reverse k nearest neighbor search can verify the object p as well, so this step can be made to do the reserver k nearest neighbor of query point q. This method is defined as two-step algorithm in [GZCL09].

- Reuse Two-heap Algorithm (RTH)

  Reuse two-heap algorithm (RTH) is proposed together with SP and TS in [GZCL09] , which attempts to fully use locally available nodes in order to reduce the redundant node accesses. In addition, an early termination condition is developed to be applied in the verification process of the $CandidateSet$. It is possible that any p in $CandidateSet$ may be terminated earlier without finding all the kNN of p.

- NN search with Pruning (NNP)

  NN search with Pruning (NNP) introduces pruning heuristics at two places to improve the search performance. The first pruning is conducted in integrating with kNN search, handled by an NNP Finding algorithm; and the second pruning is introduced as a self-pruning process. The main target is to remove those candidates that will not have any possibility to be RkNN(q).

- RNN search with pruning (RNNP)
  When Size(kNN_q) $\geqslant$ Size(RkNN_q), there is a better way to do the reverse k nearest neighbor search of query point first, then verify each object in RkNN_q set. It is a reverse way of the traditional mutual k nearest neighbor, which does the kNN_q first and verifies it after that.

To sum up, this chapter reviews the existing works related to typical k nearest neighbor search, continues k nearest neighbor search as well as k nearest neighbor variants. Although the areas of interest have been filled up in the recent decades, there are still some significant problems or, in other words, there are still some gaps/blank zones which have not been explored. Moreover, even the existing methods which can solve the problems, but perform poorly under some circumstances.

2.3 Problem Definition

As mentioned above, the following list summarizes the general problems of existing methodologies which are the motivation of this thesis as well. Fig.2.13 compares the our proposed approaches with related works.

- **Poor performance of Network Expansion Methodology**
  All of the approaches are constructed based on the underlying road connection between objects. Within the road map, roads are connected and joined by thousands of intersection nodes which break the roads into small segments. The total distance of the road is calculated by summing up the component segment distances. As a result, network expansion is the technique which has been widely used in the existing methods. Network expansion is processed as follows: when encountering any intersection node, the traverse expansion is done in every possible direction. In other words, if we suppose every node has

four possible directions to go, then the expansion would be 4^n, n is the number of expansion nodes. From this calculation, we can infer that the performance cost will behave like a parabola with the increasing number of intersection nodes. This poor performance is inevitable because the complex road connection will result in large number of intersection nodes. Consequently, network expansion methodology has an instinctive drawback which will lead to poor performance of spatial queries. How to merge the intersection nodes or how to avoid the expansion becomes a new topic to researchers who are engaged in spatial query processing.

- **Discrete points as kNN input and output.**

  Nearly all spatial queries are objects related which means both the input and output of the queries are discrete points. While in reality, path/route is another important element in spatial space. The second outstanding problem is the unicity of the input/output types. Consider that the user might want to input a path to find a set of points, or input a set of points to create an optimal path, or input a query path and output a result path at the same time, the second contribution can be made is bringing the path into kNN input and output.

Consequently, the problem and contributions are summarized as follows:

Contribution 1 Network Voronoi Diagram is used to merge the road segment which highly proves the performance of k Nearest Neighbor Search compared to Network Expansion Methodology. Two Voronoi based k Nearest Neighbor search queries are proposed in chapter 3, namely Voronoi-based Continuous kNN Search Queries in section 3.2 and Multiple object types kNN Search in section 3.3.

Contribution 2 As stated above, most of the existing queries put discrete points as input and output. Consequently, Chapter 4 concentrated on bringing route/-path into the input or output or both of the queries. Three route search queries

Techniques	Category	Remarks	Existing/Our Proposed Works	Structure
Network Expansion	Typical *k*NN	*k* near neighbors of *q*	*INE & IER & VN*[3]	*Related Work*
	Continuous *k*NN	Find *kNN* of a ***moving*** *q*	*DAR/eDAR/Intersection Examination (IE)*	
Improve Performance ***Network Voronoi Diagram***	Continuous *k*NN	Find *kNN* of a ***moving*** *q*	*Voronoi-based Continuous kNN Search*	***Chapter 3*** ***Section 3.1***
	Typical *k*NN & Route Search	Find *k*NN for *q* in each type	*Multi-object-types NN (M_NN)*	***Section 3.2***
		Find path via multi-types in *pre-defined* sequence.	*Incremental Multi-object-types NN (iM_NN)*	
		Find path via multi-types in *random* sequence.	*Optimum Path Multi-object-types NN (PM_NN)*	

Input→Output	Category	Remarks	Existing/Our Proposed Works	Structure
POIs & q → Points	Typical *k*NN	*k* near neighbors of *q*	*INE & IER & VN*[3]	*Related Work*
Bring Route into in/output ***POIs & q → Route*** ***Route → Route***	Route Search queries	*Find path via the most objects*	*Efficient Orienteering-Route Search*	
		Find path via the all types	*Incremental Route Search Query*	
		Find shortest path via *kNN*	*Path based kNN Search*	***Chapter 4*** ***Section 4.1***
		Find the path via *POI* type with most overlap	*Path Branch Point based kNN Search*	***Section 4.2***
		Find the path via *POI* type with time constraint	*Time Constraint Route Search over Multiple Locations*	***Section 4.3***

Incremental Route Search Query

Figure 2.13: Related Work vs. Approaches proposed in this thesis

are proposed in this chapter which are all route related, namely Path based kNN Search Queries in section 4.2, Path Branch Point based kNN Search Queries in section 4.3 and Time Constraint Route Search over Multiple Locations in section 4.4.

2.4 Summary

In this chapter, firstly we introduce the divisions of spatial queries from a query point of view, a result point of view as well as the accuracy point of view. Then it outlines the structure of this related work chapter. Secondly, a comprehensive summary of the existing work is explained. Preliminarily it goes first followed by several example and goes through a process of static k Nearest Neighbor Search, continuous k Nearest Neighbor Search, route search queries and other spatial queries.

After reviewing these approaches, two outstanding problems are pointed out which also lead our two main part of contribution in my thesis. Finally, it summarizes this chapter in this conclusion section and illustrates the contribution to existing work in fig. 2.13.

Chapter 3

Voronoi Based k Nearest Neighbor Search[1]

3.1 Introduction

With the developing wireless devices and booming spatial query searching, nearly all of the approaches are constructed based on the underlying road connection between objects. Within the road map, roads are connected and joined by thousands of intersection nodes which break the roads into small segments. The total distance of the road is calculated by summing up the component segments distances. As a result, network expansion is the technique which has been widely used in the existing methods. Network expansion is processed as follows: when encountering any intersection node, the traverse expansion is made to every possible directions. In other words, if we suppose every node has four possible directions to go, then the expansion would be 4n, n is the number of expansion nodes. From this calculation, we can infer that the performance cost will behave like a parabola with the increasing number of intersection nodes. This poor performance is inevitable because the complex road connection will result in a large number of intersection nodes.

[1]Part of this chapter has been published in Zhao, G., Xuan, K., Rahayu, W., Taniar, D., Safar, M., Gavrilova, M., and Srinivasan, B. Voronoi-based continuous k nearest neighbor search in mobile navigation. IEEE Transactions on Industrial Electronics (TIE), 56(10):2247-2257. 2010.

Consequently, Network Voronoi Diagrams are chosen to merge the network segments which obviously improves the performance and reduces the cost. In this chapter, we propose two approaches, namely Voronoi based k Nearest neighbor search and Voronoi based multiple types k Nearest neighbor. In both queries, Voronoi Diagrams are used as the methodology which has been proven to be applicable and efficient. The following paragraphs introduce them in detail.

Voronoi based k Nearest neighbor search is an approach to deal with the Continuous kNN (abbreviated as CkNN) query. CkNN [SE06, KS05, TPS02a] also have attracted other researchers' interest. In order to find split nodes, all existing continuous kNN approaches divide the query path into segments, find kNN results for the two terminate nodes of each segment and then, for each segment, find the split nodes. One segment of the path starts from an intersection and ends at another intersection. For every segment, a kNN process is invoked to find split nodes for each segment. If there are too many intersections on the path, there will be many segments, and consequently, the processing performance will degrade. These are the obvious limitations of the current CkNN approaches. As a result, section 3.2 proposes an alternative approach for CkNN query processing, which is based on the Network Voronoi Diagram (we call our proposed method VCkNN, for Voronoi CkNN). This approach avoids these weakness mentioned above and improves the performance by utilizing a Voronoi diagram. VCkNN ignores intersections on the query path; instead, it uses Voronoi polygons to subdivide the path. In this chapter, the Voronoi diagram, which originated in the computational geometry [GR03, GR99] and has been used successfully in other areas, such as industrial electronic area [VS08], will be demonstrated in its effectiveness in a mobile environment.

Current approaches for kNN mainly use network expansion. Network expansion consumes a large amount of processing time because segments invoke functions iteratively. Consequently, a Voronoi diagram is adopted as the most suitable tool to solve kNN queries because it aggregates lots of segments into polygons. However, current approaches focus on one object type, which narrows down the mobile query scope. For example, to find the nearest 3 hospitals to a current location. In some cases, users may want to get kNN of different object types (multiple object types), as well as to obtain the shortest routes. Motivated by these, this chapter proposes new approaches on three different queries involving multiple object types using a network Voronoi Diagram. In these queries, more than one object type is considered and the query result is highly related with the object types. Every object belongs to one category and there is no overlap between categories. That is the basic property of *multiple-object-type query*. In section 3.3 focuses on three different types of kNN mobile queries, including: a) query to find nearest neighbor for multiple types of interest point (or 1NN for each object type), b) query to give the shortest path to cover multiple-object-types in a pre-defined sequence, and c) query to find an optimum path for multiple object types that gives the shortest path that covers the required interest objects in a random sequence.

From this point, two methods are proposed in the following sections followed after the performance evaluation.

3.2 Approach 1: Voronoi-based Continuous kNN Search

Continuous k nearest neighbor search is not a novel type of query in a mobile environment, as it has been well studied in the past. Continuous k nearest neighbor can be defined as given a moving query point, its pre-defined moving path and a set of candidate interest points, to find the point on the way where k nearest neighbor changes. This is a traditional query in mobile navigation. To get the exact point on the road in short response time is not easy. Almost all of the approaches try to find the *split nodes*, which are the locations where the kNN results are changed. The already existing works on CkNN have some limitations as follows.

- Both DAR/eDAR and IE need to divide the pre-defined query path into segments using the intersections on the road. It means that once there is an intersect road in the path, it becomes a new segment, and we need to check whether there are any split nodes on this segment.

- Using DAR/eDAR and IE, for every segment we should find kNN for the start and end nodes of the segment. It obviously reduces the efficiency of the performance when the number of intersections on the query path becomes large.

- DAR/eDAR uses PINE (based on a Voronoi diagram) to do the kNN for the start and end nodes of each segment. But when doing continuous kNN, DAR/eDAR discards the Voronoi diagram and adopts another method to detect split nodes. While in our proposed approach, we use Voronoi diagram all the way through both in the kNN and CkNN stages. Hence, the properties of the Voronoi diagram are used to enhance the CkNN process.

- Both DAR/eDAR and IE cannot predict where split nodes will appear. In our proposed Voronoi-CkNN (VCkNN), it is known even before we reach the point and also it gives us the visibility of which interest point is moving out or into the list and at which position the node will become a split node.

Our proposed VCkNN approach is based on the attributes of the Voronoi diagram itself and using a piecewise continuous function to express the distance change of each border point. At the same time, we use Dijkstra's algorithm to expand the road network within the Voronoi polygon.

Comparison (VCkNN vs. DAR vs. IE)

VCkNN, DAR and IE are all approaches for CkNN queries. But VCkNN is different from DAR and IE in most of aspects. Therefore before introducing our VCkNN algorithm, we would like to highlight the main differences between VCkNN and DAR and IE.

- Path division mechanism

  For the same network connection, DAR and IE divide the query path into

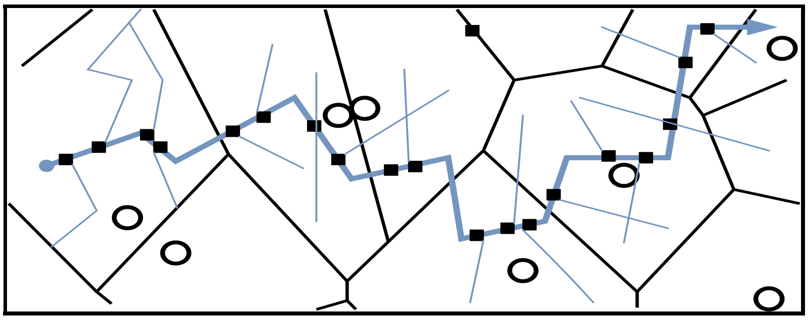


Figure 3.1: Segments using DAR and IE

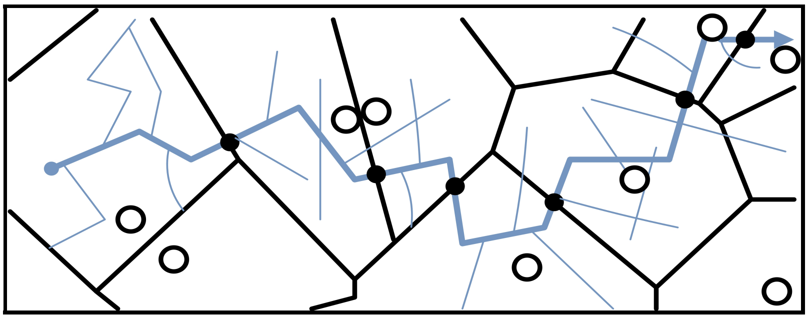


Figure 3.2: Segments using VCkNN

segments as shown in Fig.3.1, whereas VCkNN processes the path as in 3.2. Note that in Fig.3.1, for every intersection in the query path, it becomes a segment. In this example, the query path is divided into 18 segments, as there are as many intersections along the query path. In contrast, using the same query path, our approach has only 5 segments (see 3.2). The number of segments is determined by the number of Voronoi polygons. Even though there are many intersections in each Voronoi polygon, our method will process each Voronoi polygon as a unit, and hence, there is no need to check intersection by intersection.

- kNN processing

  For each segment, DAR and IE use either PINE or VN^3 to perform kNN processing for the two terminating nodes (e.g. start and end of the segment). In contrast, VCkNN does not need any algorithm to do kNN on any point on the path. VCkNN finds kNN level by level (from 1^{st}NN, then 2^{nd}NN, then 3^{rd}NN, and so on) for the entire query path. Hence, kNN results can easily be visualized using VCkNN.

- Sequence finding of split nodes

  DAR and IE use formulae to calculate the distance between two adjacent split nodes. Subsequently, we find split nodes one by one. This also means that we do not know the $(k+1)^{th}$ split node until we find k^{th} split node. In contrast, VCkNN locates split nodes using query point moving distance. For each interval, we identify the split nodes directly, which are the nearest distance between the query point and the intersected paths in the Voronoi polygon. Consequently, all split nodes are identified in one go.

- Processing split nodes

  DAR and IE compare the kNN results of the two terminate nodes of each segment to find all split nodes within this segment. On the other hand, VCkNN finds all split nodes top down from 1^{st}NN, and then 2^{nd}NN and so on. The following Tab.3.1 summarizes the differences between DAR, IE and VCkNN.

	VCkNN	DAR	IE
Query	**Continuous k nearest neighbor search**		
Basic idea	Monitor border points	Swap the position to calculate the split nodes	Monitor candidate POI and using trend to find split nodes
Segment	Ignore	Need to check segment by segment	
Voronoi polygon	Expansion polygon by polygon	Ignore	Ignore
Split node predicable	Yes	No	No
Visible	Yes	No	No
Do kNN	No	Yes	Yes

Table 3.1: VCkNN vs. DAR vs. IE

VCkNN Algorithm

The benefits offered by the proposed VCkNN processing are supported by the inherent propositions of a Network Voronoi Diagram, which are as follows:

Proposition 1. *The generator of the Voronoi polygon that includes the query point must be the nearest neighbor of the query point.*

Proof. It is self-evident because the polygon defines the area where any point in this area is closer to the polygon's generator than other generators (refer to Property 2 listed in section 2.2.1). □

The split nodes in Network Voronoi Diagram are determined by the following lemmas, which are the basis of our VCkNN algorithm. The first lemma is about the split nodes, whereas the second lemma is about kNN results.

Lemma 3.2.1. *In Voronoi CkNN, all border points that intersect with the query path and the generator edge are* ***split nodes****.*

Proof. It is obvious that when the query path reaches the generator edge, the 1stNN will change because the distance to the shared edge generators are the same (refer to Property 2 listed in section 2.2.1). □

Axiom 3.2.1. *If the query path overlaps with generator edge for a while, the first time when they intersect will be the split node and the last point where they no longer overlap will be the split node too.*

Lemma 3.2.2. *Suppose q's kNN* $= p_1, , p_k$*, the* $(k+1)^{th}$*NN of q should be within the neighbor of* $p_1, , p_k$*.*

Proof. According to the a property of the Voronoi diagram, Let $G = g_1,, g_k$ P be the set of the first k nearest generators of a location q inside $V(g_1)$, then g_k is among the adjacent generators of $G \backslash g_k$. □

Before the VCkNN algorithm is presented in Algorithm.1, we need to define moving interval (ML):

Definition 3.2.1. ***(Moving interval)*** *(ML) is the interval between two split nodes; in other words, ML is determined by two split nodes.*

The location of split node is marked by the query point moving out distance. For example, if ML is $0.7\tilde{3}.0$, whereby 0.7 and 3.0 are two adjacent split nodes in current split nodes list, then 0.7 refers to the split node that is located at the point

Algorithm 1 Algorithm VCkNN (q, k, moving path SE)

$1^{st}NN$ = contain (q)
Initial $CS = 1^{st}NN$'s neighbor generator.
$M = 1$
$Result = 1^{st}NN$ (moving interval of query point)
if $M>1$ **then**
    for each polygon where SE goes across **do**
        Expand q to each border point
        Draw the line for each border point AND get piecewise function for each border point
        Add border to generator distance to the line
    end for
end if
The lowest line will the $2^{nd}NN$. Intersect points will be split nodes. Set $M = 2$
$Result$+= $2^{nd}NN(MovingInterval_1)$,,$2^{nd}NN(MovingInterval_n)$
while $M<K$ **do**
    for each intervals which separate by split node **do**
        $CS = CS + M^{th}$ neighbor generator
        for each interest point in CS **do**
            draw a line for this interval
            The lowest line will be the $(M+1)^{th}NN$
            $Result$+=$(M+1)^{th}MovingInterval_1$,,$(M+1)^{th}MovingInterval_n$
            $M = M + 1$
            Intersect nodes are split nodes
        end for
    end for
end while
if $M = K$ **then**
    Terminate the algorithm
end if

where query point moves out in a distance of 0.7. The same is applied to 3.0 which is the split node location away from the current query point. The proposed VCkNN algorithm is given in Algorithm.1. Our VCkNN algorithm is explained as follows:

- Step 1: 1^{st} NN

  Use the contain(q) function to get the Voronoi polygon, which includes the query point. This polygon's generator will be the 1^{st} NN until it moves out from this polygon (according to preposition 1).

- Step 2: Split nodes

  The intersections between query paths and polygon borders are split nodes (refer to lemma 1).

- Step 3: Moving Interval (ML)

  Moving interval (ML) will have segments within the Voronoi polygons and the query path is divided into several MLs. For each ML, we do the following. From the beginning point of the interval, expand the road network to every border point of this polygon and record the distance. For each border point, monitor the change of the distance. Get the piecewise function for each border point according to query point's moving out distance, and then a set of candidate interest points (CS) is initialized that contains all adjacent neighbors of 1^{st} NN.

- Step 4: Candidate Interest Points (CS)

  For all interest points in CS, calculate its distance to the beginning of the interval and generate the corresponding lines and functions. Every time a line is generated, put it into a chart which x axis is the moving distance of the query point. The chart records all the changes of kNN. One thing should be mentioned here is that, if one interest point has more than one border point in the current polygon, keep the one which has the shortest distance.

- Step 5: 2^{nd} NN and more split nodes

  After finishing all interest points in CS, the lowest line (the one closest to x

axis) will be the 2^{nd} NN and the intersections of lines will be the split nodes. These split nodes divide the current interval into multiple small ones. Then add the 2^{nd} NN's adjacent interest points into CS.

- Step 6: $k > 2$

  If $k > 2$, then for every new interval, do the following: Remove the lowest lines from the chart in this interval. For all interest points in CS, calculate its distance to the beginning of interval and generate the corresponding line and functions. Every time we generate a line, put it into the chart. The lowest lines will be the next level of NN. New split nodes are the intersections on the lowest lines, and new intervals are generated by these split nodes. Update CS by adding new NN's neighbor into CS. If the NN level is still less than k, do this step again until all kNNs have been found.

- Step 7: Process termination conditions

  Finally, after all Voronoi polygons where a query path goes across have been checked, and all split nodes have been found, the algorithm terminates.

Walk through Example of VCkNN

This section describes a walk through of the VCkNN process. It not only explains the VCkNN step by step but also compares it with other works, including DAR and IE. We will list the piecewise function and draw the line in the chart to make it easy to understand. Fig.3.3 shows an example. The query is to find CkNN along the query path, shown as a thick black line, which starts from q and ends at p_{10}. The borders of $V(P_1)$ and the paths from P1 to the border points are also shown.

The first set of split nodes is the intersection nodes between Voronoi polygons and the moving path. In this case, $SplitNodes = b_2, b_9, b_{10}$. Refer to Lemma 3.2.1 on the split nodes. Split node b2 is the border point between Voronoi polygon $V(P_1)$ and $V(P_2)$, split node b9 is the border point between $V(P_2)$ and $V(P_3)$, and split node b_{10} is the border point between $V(P_3)$ and $V(P_{10})$.

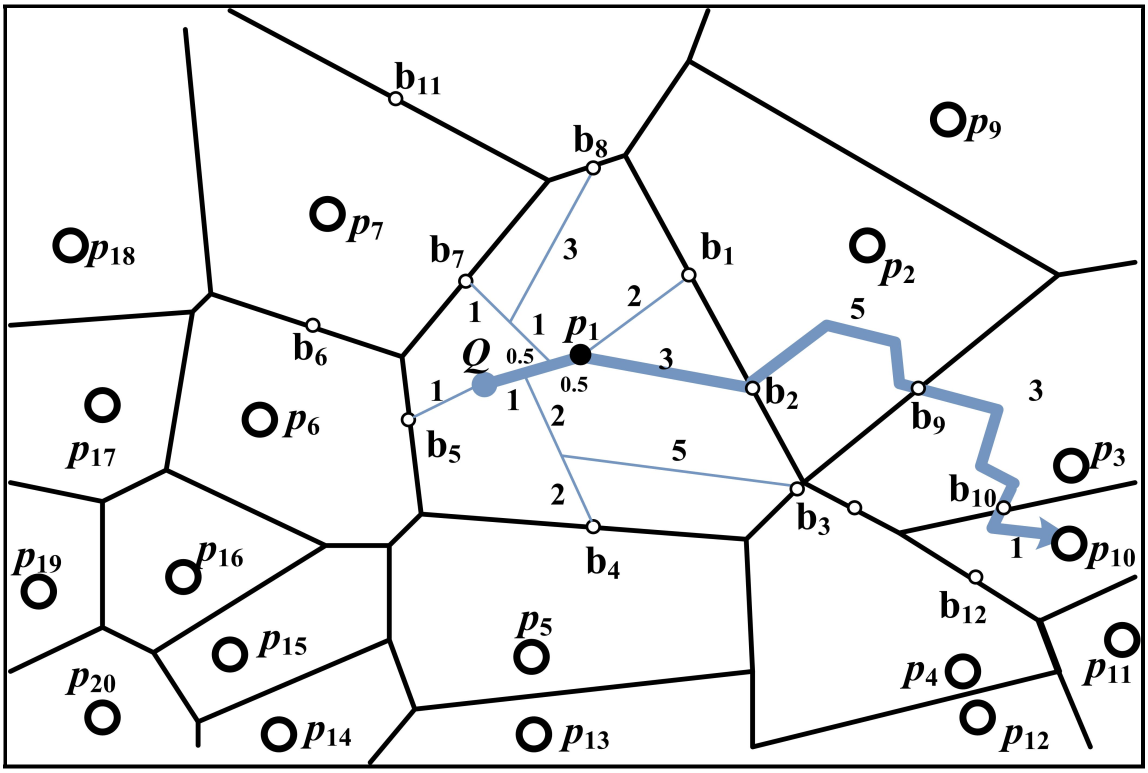


Figure 3.3: Example of VCkNN

Note the 1^{st} NN results are p_1 with a range of distance from 0.0 and 5.0, p_2 with a range of distance from 5.0 and 10.0, p_3 with a range of distance from 10.0 and 13.0, and P10 with a range of distance 13.0 and 14.0. In short, we can write the 1^{st} NN results something like this:

1^{st} NN = $p_1(0.0\ 5.0)$, $p_2(5.0\ 10.0)$, $p_3(10.0\ 13.0)$, $p_{10}(13.0\ 14.0)$

All ranges of distance are the distance from the starting query point. This means that when the query point q moves from 0.0 to 5.0, p_1 is 1^{st} NN, and when q moves from 5.0 to 10.0, p_2 will be the 1^{st} NN, and so on.

Then for $V(P_1)$, $V(P_2)$, $V(P_3)$ and $V(P_{10})$, do the following steps. Take $V(P_1)$ as an example.

Firstly we need to set some initial values according to the VCkNN algorithm (1: $M = 1$ as we have found the 1^{st} level of kNN, and $CS = p_2, p_3, p_4, p_5, p_6, p_7, p_8$, which are the adjacent nodes of $V(P_1)$.

Secondly, expand the query point q to every border point in this polygon. With the movements of q, draw a line for every border point and get the piecewise function for each border point. Table 3.2 shows the line along the movement of query point. The first column is the moving distance from the current location of query point q.

Tab.3.2 Movement of each border point in p_1.

q moving distance (km) NN	b_1	b_2	b_3	b_4	b_5	b_7	b_8
0.0	4.0	5.0	8.0	5.0	1.0	3.5	5.5
1.0	3.0	4.0	7.0	4.0	2.0	2.5	4.5
1.5	2.5	3.5	7.5	4.5	2.5	2.0	4.0
2.0	2.0	3.0	8.0	5.0	3.0	2.5	4.5

Table 3.2: Movement of each border point in p_1

The corresponding chart (see fig.3.4) and the piecewise function 3.2 are shown as follows. Note from fig.3.4, the line for border point b_2 goes down from 5 when the position of q is 0, to 0 when the position of q is around 5. The opposite is the line for border point b_5 where it goes up as q is moving from 0 to 5 (in this case the line for b_5 increases from 1 to 6). These two lines explain that when q moves, the distance from q to b_2 will be decreasing and b_2 is getting closer to q. The opposite is to b_5, where q is actually moving away from it.

The rest of the border points, such as b_1, b_3, b_4, b_7, and b_8, are all getting closer to q when q moves from 0 to some points before 2, but then they all increase after that. This indicates that initially the distance from q to these border points is decreasing (the border points are getting closer to q), but later it will be increased as q moves away from these border points.

In term of mathematical functions, these distance movements can be expressed in a piecewise function as shown in fig.3.4. Note that the functions for b_2 and b_5 are straight functions, whereas the rest have some conditions when to increase and when to decrease.

$$b_1 = \begin{cases} 4 - x & x \in [0, 2] \\ x & x \in (2, 5] \end{cases} \qquad b_3 = \begin{cases} 8 - x & x \in [0, 1] \\ x + 6 & x \in (1, 5] \end{cases} \qquad \begin{array}{l} b_5 = x + 1[0, 5] \\ \\ b_2 = 5 - x[0, 5] \end{array}$$

$$b_8 = \begin{cases} 5.5 - x & x \in [0, 1.5] \\ x + 2.5 & x \in (1.5, 5] \end{cases} \quad b_4 = \begin{cases} 5 - x & x \in [0, 1] \\ x + 3 & x \in (1, 5] \end{cases} \quad b_7 = \begin{cases} 3.5 - x & x \in [0, 1.5] \\ x + 0.5 & x \in (1.5, 5] \end{cases} \tag{3.1}$$

Thirdly, for each interest point, add its distance to the corresponding border into table 3.2 and do the chart again (as shown in fig.3.5). Suppose their distances to the borders are as follows:

$$\begin{cases} Distn(b_1, P_2) = 2.2 \\ Distn(b_2, P_2) = 3.2 \\ Distn(b_3, P_2) = 7.8, Distn(b_3, P_3) = 7.8, Distn(b_3, P_4) = 7.8 \\ Distn(b_4, P_5) = 4.8 \\ Distn(b_5, P_6) = 2.8 \\ Distn(b_7, P_7) = 2.3 \\ Distn(b_8, P_8) = 4.3 \end{cases} \tag{3.2}$$

Note that p_2 is adjacent to b_1, b_2, and b_3. Also note that the adjacent polygons of b_3 are p_2, p_3, and p_4. Others indicate that b_4 adjacent with p_5, b5 with p_6, b7 with p_7, and finally b8 with p_8. Fig.3.5 shows how each interest point, p_2 to p_8 are adjacent with the corresponding border points. For example, the top line in fig.3.5 indicates the distance from q to p_4, p_3 and p_2 (all through b_3). The line, as explained previously, shows that initially p_4, p_3 and p_2 are getting closer to q but are then getting farther.

In fig.3.6, we only focus on the bottom lines. The intersections between all bottom lines are new split nodes. The first intersection is between line p_6 and line p_7 at $q = 1.0$ (This is pointed by the first vertical dotted line). The second intersection is between line p_7 and line p_2 at $q = 1.7$ (pointed by the second vertical dotted line). The third intersection is between line p_2 (through border point b_2) and line p_2 (through border point b_1). And finally the last intersection for this interval is the lowest point of line p_2 (through border point b_2). Hence, we have four new split nodes for this interval.

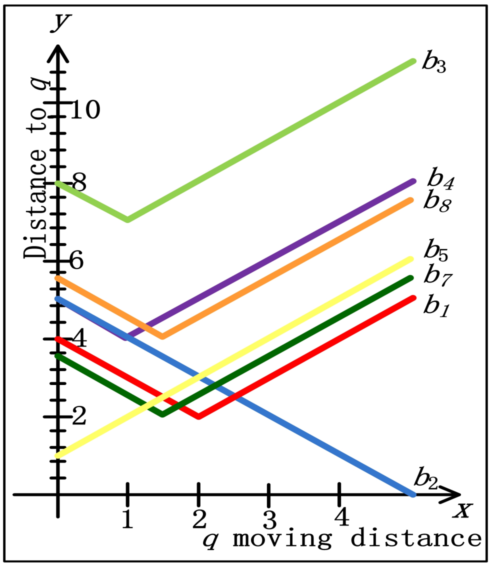


Figure 3.4: p_1 border

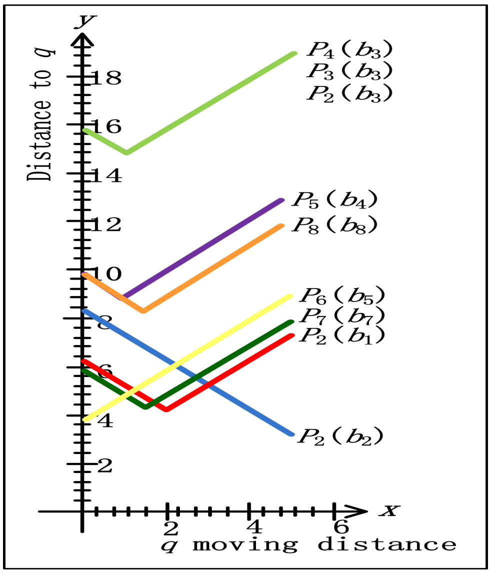


Figure 3.5: Each p

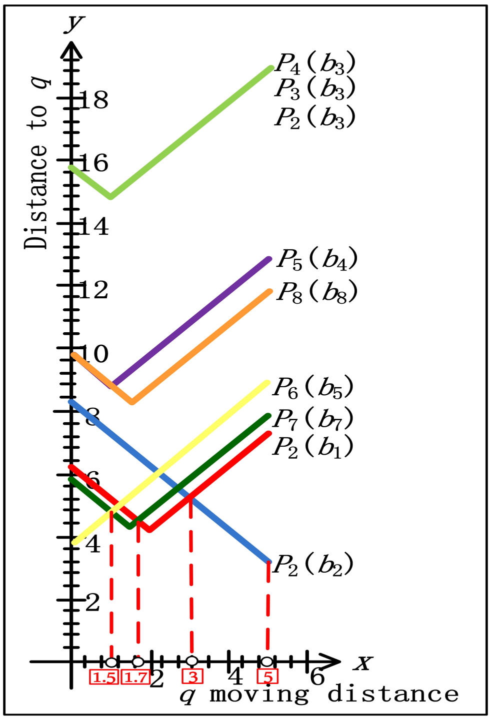


Figure 3.6: 2^{nd}NN

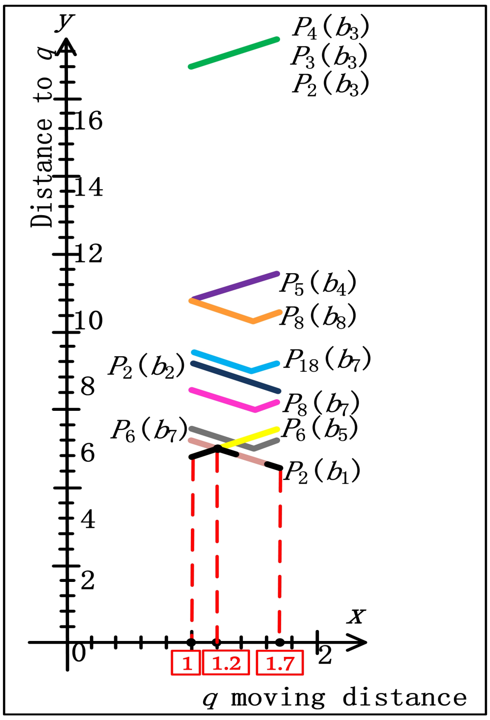


Figure 3.7: 3^{rd}NN

2^{nd}NN for this interval are: p_6, p_7, p_2 (through b_1), and p_2 again (but through b_2). In summary, 2^{nd}NN for the 0.0–5.0 interval are:

2^{nd}NN for 0.0–5.0 interval = $\{p_6(0.0\text{–}1.0),\ p_7(1.0\text{–}1.7),\ p_2(1.7\text{–}3.0),\ p_2(3.0\text{–}5.0)\}$

In other words:

- When q moves from 0.0 to 1.0, 2^{nd}NN = p_6
- When q moves from 1.0 to 1.7, 2^{nd}NN = p_7
- When q moves from 1.7 to 3.0, 2^{nd}NN = p_2 (through b_1)
- When q moves from 3.0 to 5.0, 2^{nd}NN = p_2 (through b_2)

Fourthly, after we get 2^{nd}NN, we update CS for every interval of the new split nodes, that is interval 0.0–1.0, interval 1.0–1.7, and interval 1.7–5.0. There is no need to split interval 1.7–5.0 into two intervals of 1.7–3.0 and 3.0–5.0, since the 2^{nd}NN for this interval is the same, that is p_2.

- For interval 0.0–1.0: $CS = \{p_2, p_3, p_4, p_5, p_7, p_8, p_{15}, p_{16}, p_{17}, p_{18}\}$, and 2^{nd}NN = $\{p_6(0.0\text{–}1.0)\}$. This means that when q moves from 0.0 to 1.0, p_6 is 2^{nd}NN.
- For interval 1.0–1.7: $CS = \{p_2, p_3, p_4, p_5, p_6, p_8, p_{18}\}$, and 2^{nd}NN = $\{p_7(1.0\ 1.7)\}$. This means that when q moves from 1.0 to 1.7, p_7 is 2^{nd}NN.
- For interval 1.7–5.0: CS = $\{p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$, and 2^{nd}NN = $\{p_2(1.7\ 5.0)\}$. This means that when q moves from 1.7 to 5.0, p_2 is 2^{nd}NN.

Fifthly, if $k > 2$, for every interval listed above, we need to process further. Note that the process is done iteratively from a larger interval to a smaller interval, until the smallest interval cannot further be divided. To illustrate our example, we take the 1.0–1.7 interval. This process can be thought like using a magnifying glass on the 1.0–1.7 interval of the previous process (in fig.3.6), and the result is shown in fig.3.7. We need to update the line in fig.3.7 for all interest points in CS.

Fig.3.7 shows the 1.0–1.7 interval, where the lines are updated for all interest points in CS. CS for 1.0–1.7 interval is $CS = \{p_2, p_3, p_4, p_5, p_6, p_8, p_{18}\}$, and the 2^{nd}NN for this interval is p_7. The adjacent nodes of p_7 are p_6, p_{18}, and p_8.

Suppose the distances between b_7 and these adjacent nodes are:

$$\begin{cases} Distn(b_7, P_6) = 3 \\ Distn(b_7, P_{18}) = 5 \\ Distn(b_7, P_8) = 4 \end{cases} \tag{3.3}$$

Note that we only need to get the distance between border point b_7 and all adjacent polygons of the 2^{nd}NN which is p_7, because border point b_7 is the border between p_7 and p_1 (the Voronoi polygon of the query point).

After calculating the above three distances, which represent three lines on the chart, we draw the three lines on the chart again. The split nodes are found at the interactions of the bottom lines (refer to figure 13). As a result the 1.0–1.7 intervals is now divided into two smaller intervals: 1.0–1.2 and 1.2–1.7.

For interval 1.0–1.2:$CS = \{p_2, p_3, p_4, p_5, p_8, p_{15}, p_{16}, p_{17}, p_{18}\}$, and 3^{rd}NN $= \{p_6$ (1.0–1.2)$\}$. This means that when q moves from 1.0 to 1.2, p_6 is 3^{rd}NN.

And For interval 1.2–1.7: $CS = \{p_3,\ p_4,\ p_5,\ p_8,\ p_9,\ p_{18}\}$, and 3^{rd}NN $= \{p_2$ (1.2–1.7)$\}$. This means that when q moves from 1.2 to 1.7, p_2 is 3^{rd}NN.

In summary, 3^{rd}NN for the 1.0 – 1.7 interval are: 3^{rd}NN for 1.0 – 1.7 interval $=$ $\{p_6$ (1.0–1.2), p_2 (1.2–1.7) $\}$

We need to do the same thing for the other two intervals of the 2nd NN, which are 0.0 – 1.0, and 1.7 – 5.0. This is repeated until the desired k is achieved.

Finally, after the algorithm finishes, we can see clearly where the split nodes are and also every point on query path; in other words, we can tell the kNN results straightaway, without processing kNN on every single split node like DAR and IE.

If we just look at the 1.0 – 1.2 interval, for an example sake, if the query is 3^{rd}NN, then the 3^{rd}NN for this interval is p_1, p_7, and p_6. p_1 will remain 1stNN until distance 5.0 (p_1 actually starts becoming the 1^{st}NN from distance 0.0), and p_7 will remain 2^{nd}NN until distance 1.7. Finally, p_6 is only the 3^{rd}NN for this interval only (e.g. 1.0–1.2 interval).

3.3 Approach 2: Voronoi based Multiple kNN Search

In this section, we present our proposed algorithms for the three kinds of multiple-object-type kNN queries. The first two proposed query processing (M_NN and iM_NN) use two approaches, namely: using one NVD for each object type, and using one NVD for all object types, whereas the last proposed query processing for PM_NN uses one NVD for all object types model.

Before proposing approaches for kNN queries for multiple object types, we firstly introduce the taxonomy of these three queries with examples:

1. *Multiple-object-types Nearest Neighbor* (M_NN) query is to find nearest neighbors for multiple object types. It is common in mobile navigation. Around the query point, there are k different types of interest points. For each object type, to find the nearest neighbor among the same object type is the objective of the query.

   Example 3.3.1. *Suppose a group of colleagues wants to have dinner together, and around their company there are hundreds of restaurants. They prefer French, Italian and Chinese food. As a result, they want to know the nearest French, Italian and Chinese restaurant respectively first and then make the final decision.*

2. *Incremental Multiple-object-types Nearest Neighbors* (iM_NN) query is to find optimum/shortest path to pass multiple object types in the pre-defined sequence. This query can be used when the sequence of passed interest points is critical for the user.

   Example 3.3.2. *Suppose a person falls ill at home suddenly, the family wants to tell their driver the following path. Firstly, obviously they want to go to the nearest hospital because the sickness is acute. Secondly, they need to go to the nearest medical checkup clinic. After that, they will go to find the nearest*

GP office to check the checkup result and finally go to the nearest pharmacy according GP's prescription.

3. *Optimum Path Multiple-object-type Nearest Neighbors* (PM_NN) query is to find optimum/shortest path to pass multiple object types in random sequence. Although it seems similar with 2^{nd} query, it is a novel issue actually because the interest points can be random passed.

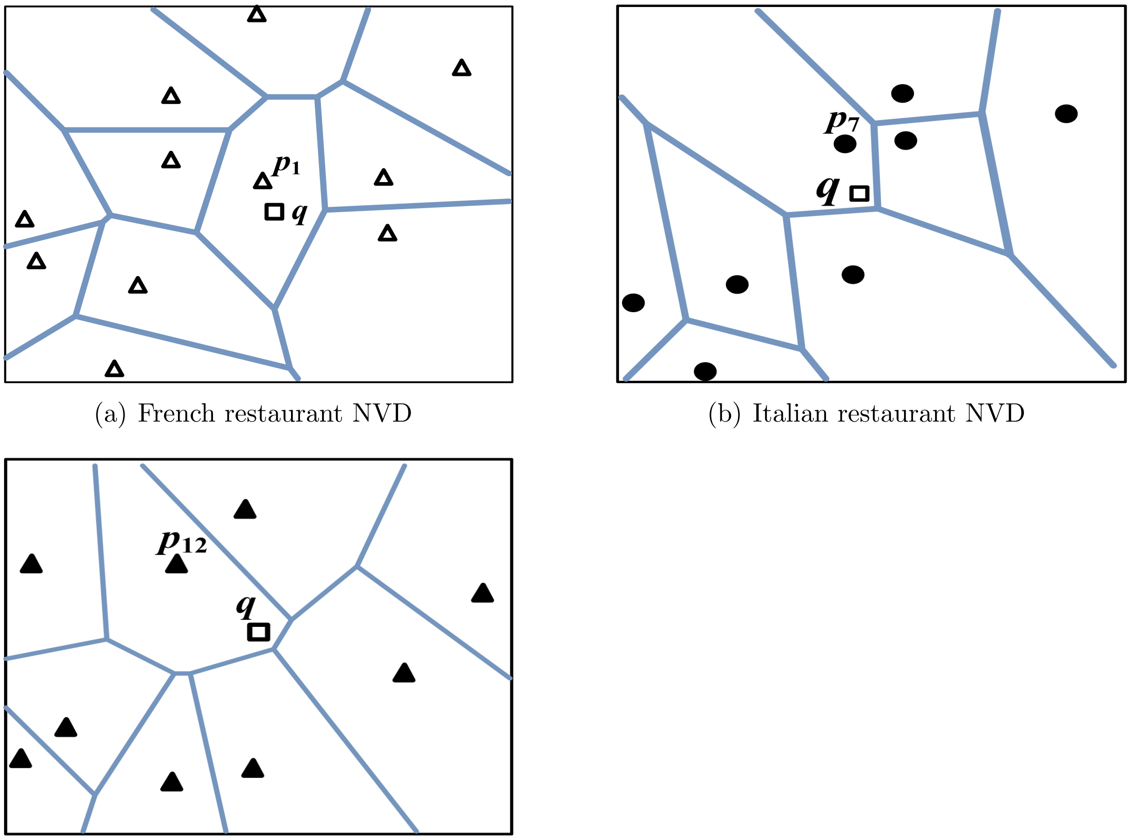
   Example 3.3.3. *Suppose a secretary has plan to do the following things: go to post office to post a letter, go to bank to deposit a cheque, go to shop to buy some print chapter and go to dry cleaner to deliver a piece of clothes. So she wants to get the best routine which not only covers all places but also makes her travelling path shortest.*

In summary, they are novel queries as there is no approach touching the query about kNN of multiple object types and they are reality-oriented and practical. Now let using the following 3 sections to proposed approaches to these quires in sequence.

Multiple-object-types Nearest Neighbor(M_NN)

In this section, we propose two ways to solve M_NN query: (*i*) For each object type, generate a NVD and find the nearest neighbor. (*ii*) Generate one NVD for all objects then filter them while searching the target result. **NVD for each object type:**

A straight approach is firstly generating NVD for each object type. For each type, find its nearest neighbor for query point using its NVD. The result comes out directly when all nearest neighbors of each type have been found. There is no reason to doubt its correctness. But concerning its efficiency, it becomes infeasible because if there are too many different kinds of object, loading different NVD*s* will consume most of the processing time. Consequently in this section, an alternative way is proposed for the query: one NVD for all objects.



(a) French restaurant NVD

(b) Italian restaurant NVD

(c) Chinese restaurant NVD

Figure 3.8: Example 3.3.1 - One NVD for each object type

Based on example 3.3.1, Fig.3.8(a), 3.8(b) and 3.8(c) represent NVDs of French, Italian and Chinese restaurants respectively. As a result, the nearest French (p_1), Italian (p_7) and Chinese (p_{12}) restaurant can be told directly.

One NVD for all object types: Generate just one NVD for all objects which not only includes the types that the user concerns but also includes the objects of other types. It will definitely improve the performance both in time and storage aspects. The algorithm performs as follows.

Firstly, generate NVD considering all objects as polygon generators. Then "contain" function is invoked to get the generator whose polygon covers query point. This generator is the first nearest neighbor of its type.

Secondly, do an expansion within this polygon and record the distance from the query point to all border points. Calculate the distances from the query point to

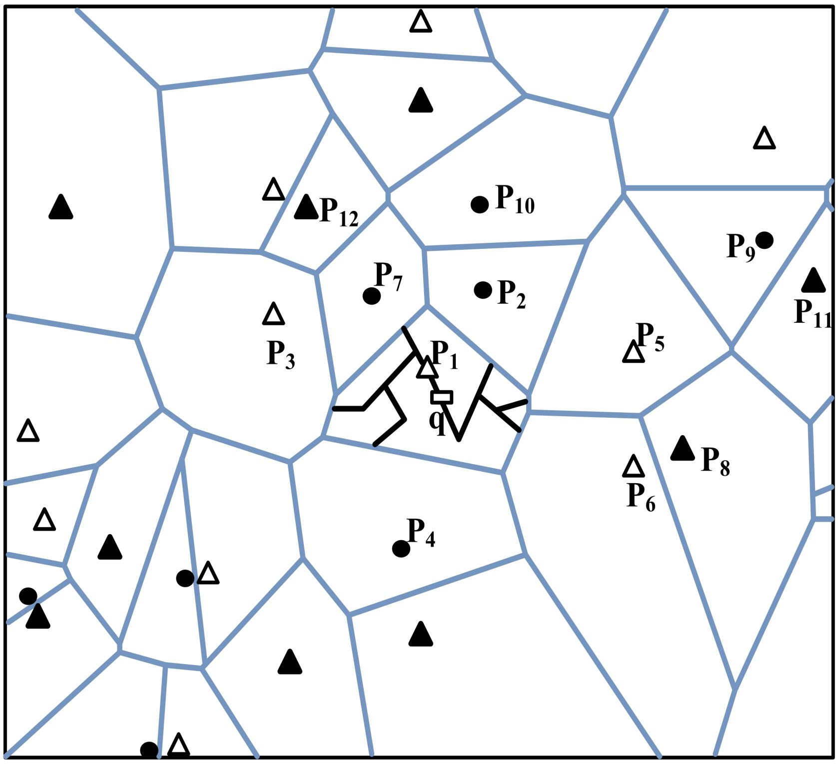


Figure 3.9: Example 3.3.1 - One NVD for all objects

all adjacent polygon generators. As all border points to generators' distances are pre-computed, this process can be finished transitorily.

Thirdly, the generators will be put in a queue sorting by their distance to the query point. From the shortest one, if by now a query result for its type have not found, it will be recorded as query result for this type; otherwise, just discard it. Then add its adjacent generators into the list and sort again. Do this step iteratively until all object types' nearest neighbors have been found.

Finally, we get a result list which is for each object type there is an interest point nearest to query point among others in this type.

The algorithm can be expressed in Algorithm 2.

The following example fully illustrates how the algorithm works. The scenario is based on example 3.3.1 as well. In this case, the query should retrieve 3 restaurants because the user only concerns 3 types of restaurants, French, Italian and Chinese. The processing steps are as follows:

- Generate NVD as in Fig. 3.9. White triangle, black dot and black triangle indicate French, Italian and Chinese restaurants respectively. Use *contain*() function to locate p_1 which is the 1^{st} NN of q.

- As Type(p_1) = French, initial RL = {(French, p_1), (Italian, $\emptyset$), (Chinese, $\emptyset$)} Initial NP = {p_2, p_3, p_4, p_5, p_6, p_7} by adding all p_1's adjacent into NP.

- Expand q within p_1's polygon and record all distance from q to border points. Calculate the distance from q to each p in NP and sort them in ascending order. Update NP, suppose NP = {(p_5, 5), (p_7, 7), (p_2, 9), (p_3, 11), (p_6, 16), (p_4, 18)}

- Pop out p_5. As type(p_5) = French & in RL, French already has value p_1, ignore p_5. Add p_5's adjacent neighbors into NP and update NP. Suppose the distance is: NP = {(p_7, 7), (p_2, 9), (p_3, 11), (p_8, 14), (p_6, 16), (p_4, 18), (p_9, 19), (p_{10}, 22), (p_{11}, 28)}

- Then Pop out p_7. As type(p_7) =Italian & in RL, Italian has null value, update RL as RL = {(French, p_1), (Italian, p_7), (Chinese, $\emptyset$)}. After that, add all p_7's adjacent neighbors into NP and update NP. Suppose the distance is: NP = {(p_2, 9), (p_{12}, 10), (p_3, 11), (p_8, 14), (p_6, 16), (p_4, 18), (p_9, 19), (p_{10}, 22),(p_{11}, 28)}

- Then pop out p_2 and ignore it as it is Italian restaurant. Then Pop out p_{12}. As type(p_{12}) = Chinese & in RL, Chinese has null value, update RL as RL = {(French, p_1), (Italian, p_7), (Chinese, p_{12})}. Algorithm terminates.

Incremental Multiple-object-types Nearest Neighbors (iM_NN)

The query of incremental nearest neighbors for sequential multiple object types is to find the shortest path which goes through multiple object types in pre-defined sequence. In this case, the sequence is crucial to the user and the user wants to pass these object types in a certain order as in example 3.3.2.

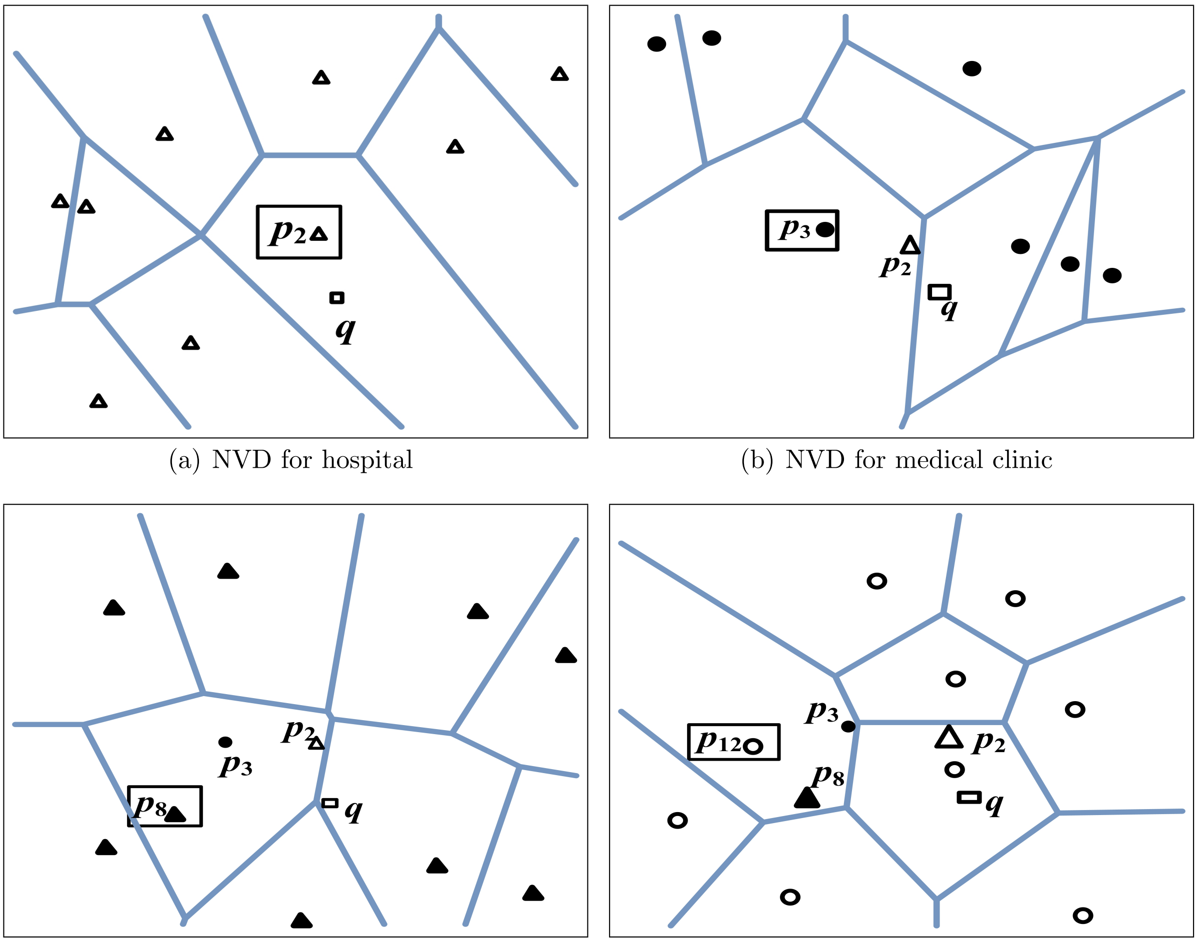
Algorithm 2 M_NN(k, query point)

1: Generate Voronoi diagram using all interest points
2: RL (Result List) $= \{(type_1, \emptyset), (type_2, \emptyset), ..., (type_k, \emptyset)\}$
3: $p_i = 1^{st}$NN = contain (q)
4: $type_i$ = Check_type (p_i)
5: RL (Result List) $= \{(type_1, \emptyset), (type_2, \emptyset), ..., (type_i, P_i), ..., (type_k, \emptyset)\}$
6: Initial NP (Neighbor point) = {p_i's adjacent generator}
7: Expand q within this polygon & record distance from q to border point.
8: Calculate distance from q to each p in NP & sort them in ascending distance order. $NP = \{(p_1$, dist(q, p_1)),..., p_i, dist(q, p_i)}
9: Pop out the first p in NP, suppose it is p_j
10: $type_j$ = Check_type (p_j)
11: **if** in RL, $type_j$ has null values **then**
12: update RL as ($type_j$, p_j)
13: **else**
14: ignore p_j
15: **end if**
16: **if** all type has values in RL **then**
17: terminate algorithm
18: **else**
19: Add p_j's adjacent neighbor into NP & go to step 8
20: **end if**

From the example, we can tell that the sequence of object types is crucial, in other words, the routine should begin at home then pass the hospital, the medical checkup clinic, the GP office and end at one pharmacy. It is not hard to see that the approach performs in the following steps: when the path reaches the interest point, it is treated as the new query point. Then continue to search the nearest neighbor of the next type until all object types have been found. There are two ways in which we can solve this query, naming as one NVD for each object type and one NVD for all objects.

One NVD for each object type: This method firstly generates NVD for the 1^{st} object type and finds the nearest one of this type. Then it generates NVD for the 2^{nd} object type and finds the nearest one of this type considering 1^{st} NN as the query point. Continue to do so for the following object types until nearest neighbors for all types have been found.

Fig. 3.10 shows the processing steps based on example 3.3.2. The result is automatically shown in the figures: Shortest path starts at q, firstly goes to hospital p_2,



(a) NVD for hospital

(b) NVD for medical clinic

(c) NVD for GP office

(d) NVD for pharmacy

Figure 3.10: Example 3.3.2 - One NVD for each object type

then heads to checkup clinic p_3, after that, towards to GP office p_8, finally arrives pharmacy p_{12} for medicine.

One NVD for each object type is actually dividing this query into multiple 1_NN queries. There is no reason to doubt its correctness. But concerning its efficiency, it becomes infeasible because if there are too many different kinds of interest points, loading different NVDs will consume most of the processing time. **One NVD for all object types:** This method generates just one NVD for all object types, including not only the type of user concerns but also other object types. It saves time and storage. The following steps illustrate how it works.

Firstly, one NVD is generated considering all objects as polygon generators. Then invoke the "contain" function to get the first nearest neighbor. Check whether it is the 1^{st} type the user wants. If yes, go to the 2^{nd} step; otherwise check the adjacent neighbors of this interest point until find the nearest neighbor of 1st type.

Algorithm 3 iM_kNN(k, query point)

1: Generate Voronoi diagram using all interest points within given types
2: RL (Result List) = $(type_1, \emptyset), (type_2, \emptyset), ..., (type_k, \emptyset)$
3: Initial M =1
4: p_i = 1^{st}NN = contain(q)
5: $type_i$ = Check_type (p_i)
6: **if** $type_i = type_m$ **then**
7:     update $type_m$'s values as p_i&M=M+1
8: **else**
9:     Expand q within this polygon&record distance from q to border
10:     Initial NP (*Neighbor point) = p_i's adjacent generator*
11:     Calculate distance from q to each p in NP & sort them in ascending order.
    NP = {(p_1, dist(q, (p_1))),...,(p_1, dist(q, (p_1))) }
12:     Pop out the first p in NP, suppose it is p_j. $type_j$ = Check_type (p_j)
13:     **if** $type_j = type_m$ **then**
14:         update $type_m$'s values as p_j&M=M+1
15:     **else**
16:         update NP by adding p_j's adjacent neighbor into NP&go to step 11
17:     **end if**
18: **end if**
19: **while** $M \leq k$ **do**
20:     Suppose p_{m-1}= $type_{m-1}$'s values in RL
21:     Initial NP(Neighbor point)=p_{m-1}'s adjacent generator
22:     Calculate distance from p_{m-1} to each p in NP and sort them in ascending order.
    NP = {(p_1, dist(q, p_1)),..., (p_i, dist(q, p_i))}
23:     Pop out the first p in NP, suppose it is p_n. $Type_n$ = Check_type (p_n)
24:     **if** $Type_n = type_m$ **then**
25:         update $type_m$'s values as p_n&M=M+1
26:     **else**
27:         update NP by adding p_n's neighbor into NP&go to step 22
28:     **end if**
29: **end while**
30: **return** NP

Secondly, consider the 1^{st} NN as query point; find its nearest neighbor of 2^{nd} type using the pre-computed distance to its adjacent neighbors.

Thirdly, do these operations iteratively until all object types have been found.

Finally, a shortest path comes out which begins at the query point, passes multiple object types in user defined sequence until it reaches the last object. That is the optimum path of this kind of query.

The algorithm can be expressed in Algorithm 3.

A case study based on example 3.3.2 fully illustrates the approach. In this case, the user concerns 4 object types ($k = 4$) because they want to pass the hospital, checkup clinic, GP office and pharmacy one by one. The processing steps are as follows:

- Generate NVD as Fig. 3.11. White triangle, black dot, black triangle and white dot indicate hospital, checkup clinic, GP office and pharmacy respectively.
- Initial RL={(hospital, ∅), (checkup clinic, ∅), (GP office, ∅), (pharmacy, ∅)} & $M = 1$
- Use $contain()$ function to locate p_1 which is the 1^{st}NN of q.
- Expand q within p_1's polygon and record all distances from q to borders.
- As type(p_1) = pharmacy $\neq type_m$, Initial $NP = \{p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$ by adding all p_1's adjacent into NP.
- Calculate the distance from q to each p in NP and sort them in ascending order. Update NP as NP={(p_2,2),(p_3,4),(p_4,6),(p_8,8), (p_7,9),(p_5,12),(p_6,16)}
- Pop out p_2. As Type (p_2) = hospital = $type_m$, update RL as RL = (hospital, p_2), (checkup clinic, ∅), (GP office, ∅), (pharmacy, ∅). $M = 2$
- As $M < k$, $type_{m-1}$'s value is p_2. Initial $NP = \{p_1, p_3, p_4, p_9\}$ by adding all p_2's adjacent into NP.
- Calculate the distance from p_2 to each p in NP and sort them in ascending distance to p_2. Update NP, suppose NP={(p_1,3),(p_3,6),(p_9, 8),(p_4,9)}
- Then pop out p_1. As type(p_1)=pharmacy $\neq type_m$=checkup clinic, ignore p_1. Add p_1's adjacent neighbors into NP and update NP. Suppose the distance is NP={(p_3,6),(p_9,8),(p_4,9),(p_8,13),(p_7,15),(p_5,18),(p_6,21)}
- Pop out p_3. As type(p_3) = checkup clinic = $type_m$, update RL as RL = {(hospital, p_2), (checkup clinic, p_3), (GP office, ∅), (pharmacy, ∅)}. $M = 3$

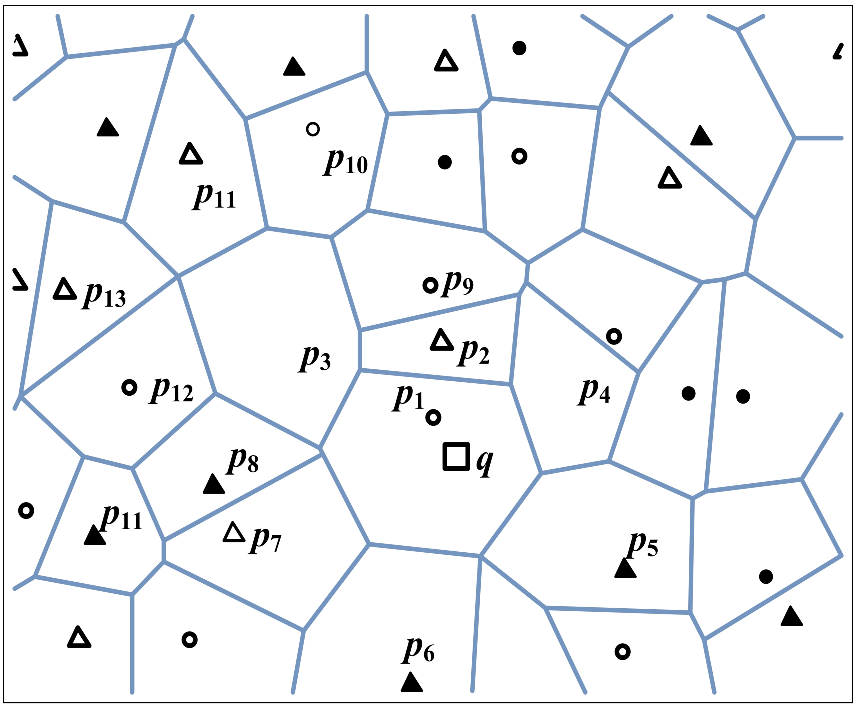


Figure 3.11: Example 3.3.2 - One NVD for all objects

- As $M < k$, $type_{m-1}$'s value is p_3. Initial NP = {(p_8, 12), (p_1, 14), (p_2, 17), (p_9, 23), (p_{12}, 25), (p_{13}, 27), (p_{11}, 30), (p_{10}, 31)}.
- Then pop out p_8. As type(p_8) = GP office = $type_m$, update RL as RL = {(hospital, p_2), (checkup clinic, p_3), (GP office, p_8), (pharmacy, $\emptyset$)}. $M = 4$
- As $M = k$, $type_{m-1}$'s value is p_8. Initial NP = {(p_{12}, 15), (p_1, 16), (p_7, 17), (p_{14}, 26)}.
- Then pop out p_{12}. As type(p_{12})=pharmacy=$type_m$, update RL as RL = {(hospital,p_2), (checkup clinic,p_3), (GP office,p_8), (pharmacy,p_{12})}. $M = 5$
- As $M > k$, algorithm terminates.

Results are: The optimum path firstly goes to hospital p_2, then heads to checkup clinic p_3, GP office p_8 and finally arrives at pharmacy p_{12} for medicine.

Optimum Path Multiple-object-type Nearest Neighbors (PM_NN)

Optimum path for multiple object types' query is similar with the 2_{nd} query except that object types can be passed in any sequence. In this query, the length of whole path is the criterion of assessment. As multiple 1_NN cannot guarantee the final path is the shortest one, this approach is different with IM_NN approach in the last section. More details can be told based on example 3.3.3.

In example 3.3.3, the sequence of interest points is unimportant because posting letter, depositing cheque and so on are independent tasks and it does not matter which task the user does first. In addition, the objective of this query is to make the whole path short not to find any nearest object. There may be an instance that after choosing the nearest post office, the path to other place will become farther. Maybe choosing the second or even third nearest post office is better. In addition, how to arrange the sequence of interest points is another issue needed to be solved. The following steps illustrate the process of the approach.

Firstly, generate NVD considering all objects as polygon generators. Then invoke "contain" function to get the nearest generator P. Check its type and record P as the first object type that the user will visit. For all P's adjacent neighbors, sort them in the ascending sequence of their distance to P. Check their types one by one, if the path has not visited that object type, record it as the next P. From now on, start from this P, do the same operation as the first P until all types have been found and the path is completed. The obove operation cannot guarantee this path is shortest but it did set a boundary for the query (d_{max}) which means once expansion is over this boundary, it should be terminated.

Secondly, every object whose distance to q is smaller than d_{max} can be treated as potential first interest point. Sort them in a queue by their distance to q.

Thirdly, for each interest point in the queue, pop it out, find its closest neighbor whose type has not been covered and then from that neighbor do the same things until all types of interest points have been covered. If in the process of the expansion, the distance is over the boundary, terminate it directly. If the path is completed, compare its path length with the boundary and update the boundary if it is smaller.

Terminate the algorithm when there is no interest point in the queue. The optimum path shows how the user can pass multiple object types in random sequence.

Algorithm 4 PM_NN(k, query point)

1: Initial TS=$\{type_1,...,type_k\}$ R=$\{dist_q, \emptyset_1, \emptyset_2, ..., \emptyset_k\}$,$d_{max}$=$\infty$,$RL = \emptyset$,$S = \emptyset$
2: p_1=1^{st}NN=contain(q), t_{p1}=Check_type(p_1)
3: Suppose $type_i = t_{p1}$, remove it from TS
4: R=$\{dist_q, p_1, \emptyset_2, ..., \emptyset_k\}$
5: Initial NP(Neighbor point)=p_1's adjacent generator
6: Calculate distance from p_1 to each P in NP in ascending order.
NP = {(p_1, dist(q, (p_1))),...,(p_i, dist(q, (p_i)))}
7: Pop out the first P in NP, suppose it is p_j
8: **if** t_{pj}=Check_type(p_j) is in TS **then**
9: update t_{pj} in R & remove t_{pj} from TS
10: **if** TS is not $\emptyset$ **then**
11: add p_j's neighbor into NP & go to step 7
12: **else**
13: **if** $dist_q < d_{max}$ **then**
14: update $d_{max} = dist_q$ & $RL = R$
15: **else**
16: ignore it
17: **end if**
18: **end if**
19: **else**
20: add p_j's adjacent neighbor into NP & go to step 7
21: **end if**
22: Expand q within this polygon & record distance from q to border point
23: Update S={all objects *(dist) to* $q < d_{max}$ *sort in ascending distance order*}
24: **for** each P in S **do**
25: Pop out the first P & Initial TS=$\{type_1,type_2,...,type_k\}$
26: t=Check_type(P)
27: R=$\{dist_q,P,\emptyset_2,...,\emptyset_k\}$
28: Initial NP(Neighbor point)=P's adjacent generator
29: Calculate distance from P to each p_i in NP & Wipe out P whose $dist(q, P) > d_{max}$
NP={(p_1,dist(q,(p_1))),...,(p_i,dist(q,(p_i)))}
30: **if** $NP \neq \emptyset$ **then**
31: Pop out the first p in NP, suppose it is p_j
32: **else**
33: go to step 23
34: **end if**
35: t_{pj}=Check_type(p_j)
36: **if** t_{pj} is in TS **then**
37: update t_{pj} in R & remove t_{pj} from TS
38: **if** TS is not $\emptyset$ **then**
39: add p_j's neighbor into NP & go to step 23
40: **else**
41: **if** $dist_q < d_{max}$ **then**
42: update $d_{max} = dist_q$ & $RL = R$ & wipe off p in S $dist(P)$>d_{max}
43: **end if**
44: go to step 29
45: **end if**
46: **end if**
47: add p_j's adjacent neighbor into NP & go to step 29
48: **end for**

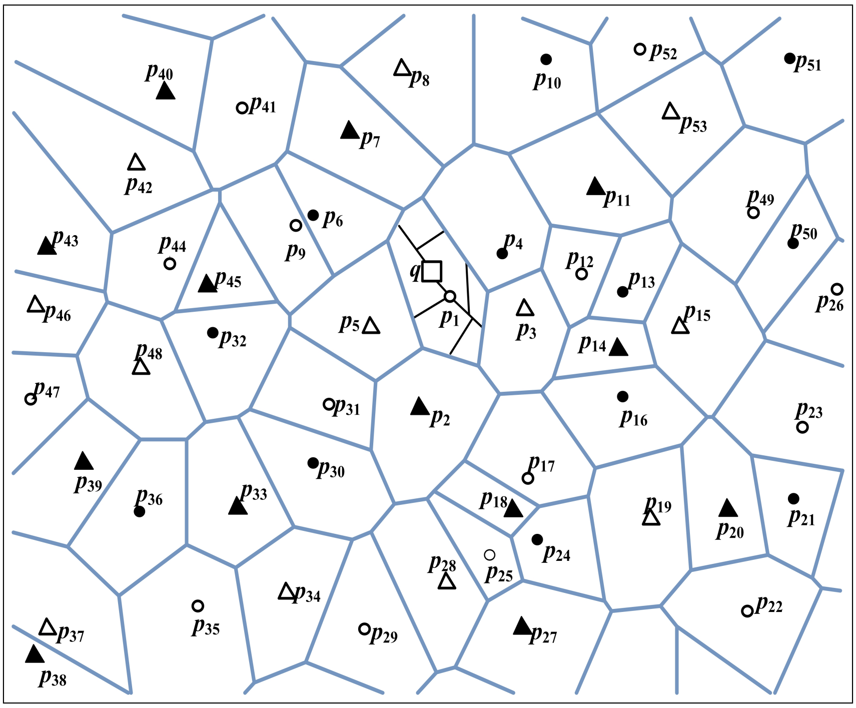


Figure 3.12: Example 3.3.3 - One NVD for all objects

The algorithm can be express in Algorithm 4.

To clarify the algorithm, a case study will fully illustrate how it works.

- Generate NVD as in Fig. 3.12. White triangle, black dot, black triangle and white dot indicate post office, bank, shop and dry cleaner respectively.

- Initial $d_{max} = \infty$, TS = {post office, bank, shop, dry cleaner}, R= $\{dist_q$, $\emptyset_1$, $\emptyset_2$,..., $\emptyset_k\}$, $RL = \emptyset$

- Use $contain()$ function to locate p_1 which is the 1_{st}NN of q.

- As Type(p_1)=dry cleaner, update TS={post office, bank, shop} by removing it from TS & $R = \{1, P_1, \emptyset_2, ..., \emptyset_k\}$// suppose p_1 to p_q's distance is 1

- From p_1, find nearest neighbor whose type in TS. Suppose p_4. As Type(p_3) = bank, update TS = {post office, shop} & $R = \{5, P_1, P_4, ..., \emptyset_k\}$// suppose p_4 to p_1's distance is 4

- Do the same to p_4 as the step above until $TS = \emptyset$. Suppose R = {15, p_1, p_4, p_{11}, p_{53}} Update d_{max} =15

- Expand q within p_1's polygon and record all distances from q to borders.
- Initial $S = \{p_4, p_3, p_6, p_5, p_7, p_2...p_n\}$ // whose distance to q within d_{max}
- Pop out p_4&update TS={post office,dry cleaner,shop} & R= {3,p_4,$\emptyset_2$,...,$\emptyset_k$}. Search p_4's nearest neighbor whose type in TS and do the same for the rest interest points iteratively until $TS = \emptyset$. Suppose $R = \{12, p_4, p_3, p_{12}, p_{11}\}$, then update d_{max}=12. Wipe out p in S $dist_q > 12$.
- Do the same operation to every P in S until S is empty. In the process, once the distance is over d_{max}, terminate expansion for this path.

The result comes out finally $R = \{10, p_3, p_{12}, p_{13}, p_{14}\}$. The user firstly goes to post office p_3, then heads to dry cleaner p_{12}, after that, towards bank p_{13} and finally arrives at shop p_{14} and the length of the final path is 10.

3.4 Performance Evaluation

In this section, we evaluate these two methods using different data and environment settings.

3.4.1 Voronoi based Continuous kNN

Melbourne city map and Geelong map in Victoria, Australia, are chosen in the experimentations from the "whereis" website [Cor] to represent high-density and low-density scenarios of interest points. All interest points, network links and intersect nodes are real-world data. We analyze the behavior of our approach in the aspects such as segment division in different path or point of interest density by DAR/IE and VCkNN and runtime with various lengths of path and the values of k.

Segment division

Firstly, we aim at finding the differences in the number of segments divided along the path in different path densities. The Melbourne city map is used to indicate high path density, in other words, more network intersections along the path (2.1 intersection/km). Correspondingly, the Geelong city map is used to indicate low path density (1 intersection/km). Interest points are distributed at 10.93/km2 on two different maps. From fig.3.13, we can draw several conclusions:

- Segment increases show a nearly linear trend;
- In the VCkNN algorithm, paths are divided into the same segment no matter whether the path density is high or low;
- DAR/IE algorithm divides into more segments in high path density than in low path density;
- VCkNN always generates less segments than DAR/IE no matter the path density.

Secondly, we aim at finding the differences of number of segments divided along the path in different point of interest point density. Restaurants in the Melbourne city map indicate a high point of interest density (23/km2), whereas petrol stations indicate a low point of interest density (1.8/km2). Path density is about 1.2 intersection/km. From fig.3.14, several conclusions lists below:

- In the VCkNN algorithm, more segments occur if the objects of interest are distributed in high density then in low density;
- DAR/IE algorithms remain the same no matter the points of interest are in low or high density;
- VCkNN always generates less segment than DAR/IE no matter the density of objects of interest.

Run time

Firstly, we report our experimentation results on runtime of different density of points of interest. We use 20 points of interest to represent a low-density sample

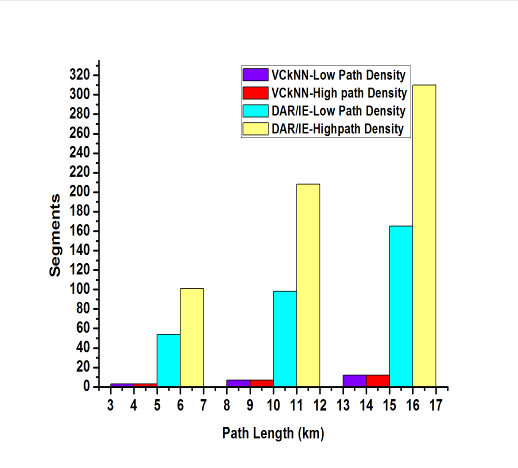


Figure 3.13: Segment in different path density

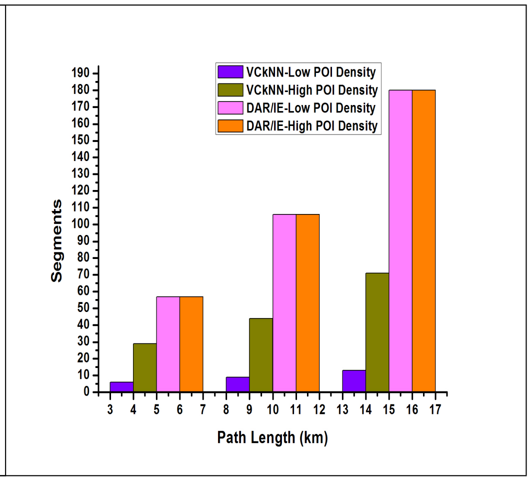


Figure 3.14: Segment in different POI density

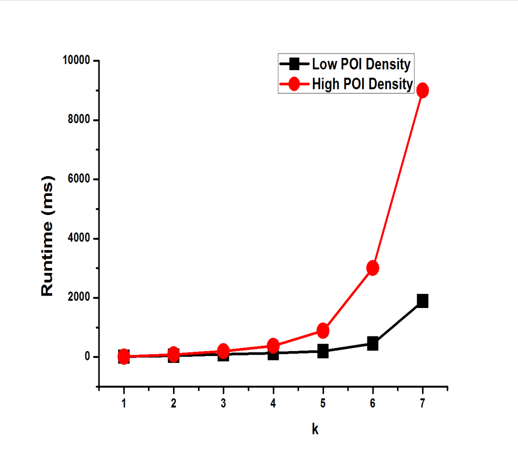
and 100 interest points to represent a high density sample. Also we test 20 different query positions to get the average runtime based on k from 1 to 7.

For the runtime factor, we can easily tell that if k increases, runtime increases sharply, and in a high-density scenario it is even more time consuming. Figure 16 shows the trend of these two scenarios.

From fig.3.15, we can also conclude that the runtime increases sharply after k>5. This is because too many operations on small intervals and too many operations and checkings need to be executed for the candidate interest points. The high density will do more looping and the runtime consequently goes up.

Secondly, we aim at finding the differences of runtime between shorter and longer query paths. We put 50 interest points on each map to compare the runtime. We choose 20 query paths (all equal to 20km) to get the average runtime in the Melbourne city map based on k=3. For these 20 moving paths, we record the runtime every time when query point moves 1km and after query point moves 5km. As the shorter distance query point moves out, the less time is consumed as less polygon is checked and less expansion is involved.

Fig.3.13 presents the average runtime and it can tell that the line is nearly linear which means that every part of the query path is generally independent. With the increase of query path's length, the runtime will definitely increase.



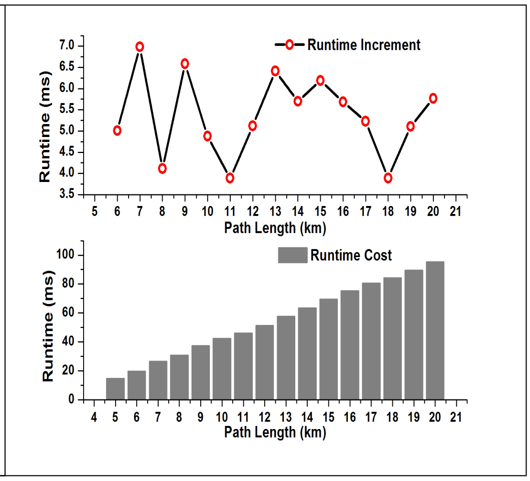


Figure 3.15: Runtime in high and low density of interest points

Figure 3.16: Runtime in different query path lengths

Split nodes number between different point of interest density

In this section, we use the same experimental conditions as the previous ones to compare the split nodes number in different point of interest density. It is known that normally for the same map, if the interest point density is low, the split nodes will be less than the high density one, because there is less chance that another interest point will be found.

From fig.3.17, because the density decides the polygon average area, the same query path will go across fewer polygons in the low density map than in the high density map. So when k=1, the low density performance is better than the high density performance. While we cannot conclude that for the same k and query path, the low density one has less split nodes than that of high density. At the same time, we can draw a conclusion that for the same map and same query path, split nodes will increase or decrease with k but the increasing amount is not constant.

3.4.2 Voronoi based Multiple types kNN

In the experimentations, Melbourne city map and Frankston map in Australia are chosen from the "whereis" website [wM06]. In these maps, shops and restaurants represent a high-density scenario of interest points, on the other hand, hospitals and shopping centers represent low-density scenario of interest points. All interest points

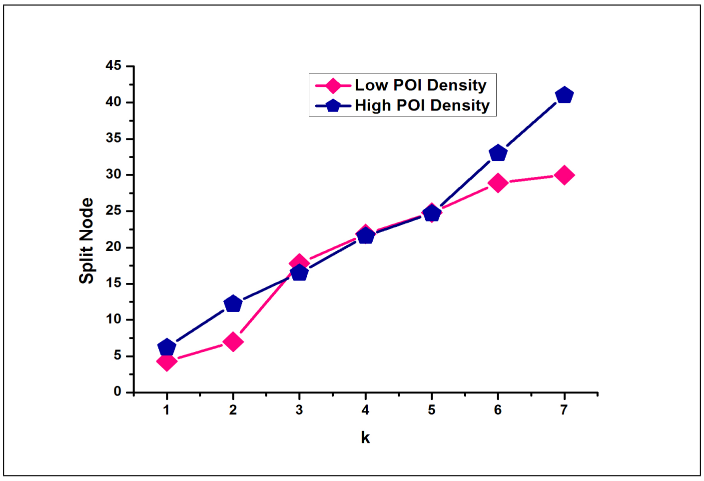


Figure 3.17: Split nodes in high and low density of interest points

are real-world data. The performance of our approaches is analyzed in runtime aspect in different diversity of interest point or in different interest points' density.

For the nearest neighbor for multiple object types, the processing time is increasing with the number of object types (M). In Fig. 3.18(a), the dash line indicates the performance of one NVD for each object type approach and the solid line indicates the performance of one NVD for all objects approach. In this case, we use different types of shops as candidate types and the average density is $5/km^2$. From Fig. 3.18(a), we can easily tell that one NVD for each object type performs better than one NVD for all objects if objects types are small, especially smaller than 4. Otherwise, one NVD for all objects is a better choice because it saves time for generating NVD. We can also tell that with the increasing object types, the processing time increases sharply because more polygon expansions will be invoked and more NVDs should be generated.

For incremental nearest neighbors for sequential multiple object types query, the processing time is increasing with the number of object types (M). Here a definition is introduced: density relative rate (DRR). DRR is ratio of the highest density to lowest density of all object types. As a result, DRR is not smaller than 1. The closer to 1 DRR is, the more evenly objects distribute. For example, if the user concerns 4 object types and their densities are $5.5/km^2$, $3.6/km^2$, $2.5/km^2$ and $1.1/km^2$ respectively. So this scenario's DRR is $5.5/km^2$ (highest) $1.1/km^2$(lowest)=5. In

Fig. 3.18(b), the first two bars indicate the processing time of one NVD for each object type approach and the last two bars indicate the processing time of one NVD for all objects approach. The first and third bars are operating in low DRR scenario (DRR=1) and the second and forth bars are in high DRR scenario (DRR=10).

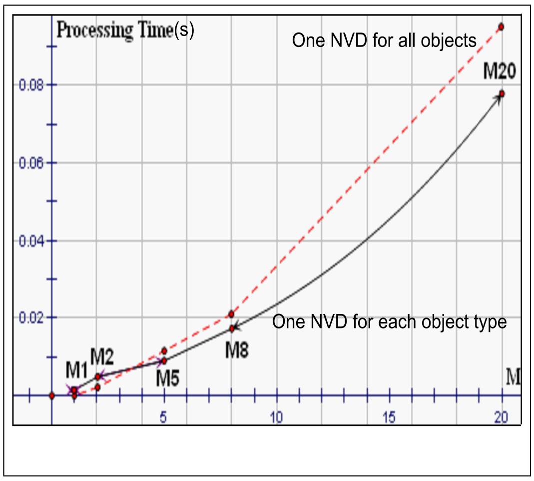
Fig. 3.18(b)illustrates that processing time will increase if DRR increases. In addition, the higher DRR is, the closer two approaches (one NVD for each & one NVD for all) performs. In addition, generally, one NVD for all interest points performs better than one NVD for each object types because generating and loading NVD are time consuming tasks.

The processing time for optimum path for multiple object types query, the processing time is increasing with the object types (M). In Fig. 3.18(c), the dash line indicates the performance of the query when density relative rate (DRR) = 1 and the solid line indicates the performance of the query when density relative rate (DRR) = 5.

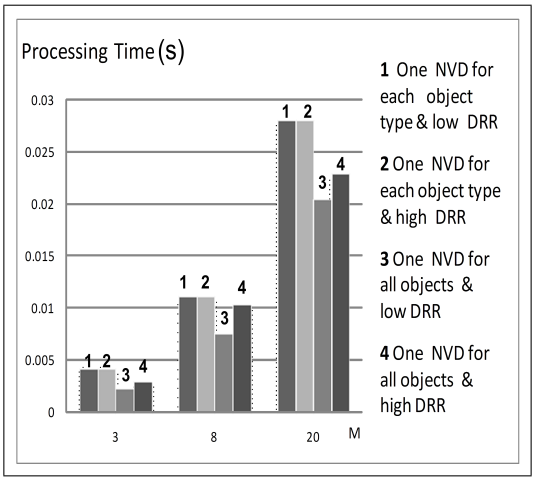
From Fig. 3.18(c), with the increasing object types, the processing time increases sharply because more polygon expansions will be invoked. Moreover, DRR is another critical factor for the performance of the approach. The processing time increases more sharply if DRR increases from 1 to 5.

3.5 Summary

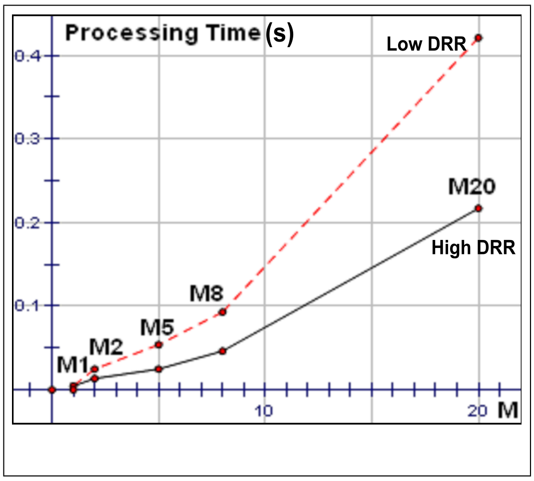
In this chapter, firstly we present a novel approach of Voronoi-based continuous k nearest neighbor search based on network distance, which we call VCkNN. The basis of VCkNN is using network expansion within each polygon and a drawing line for every border point. VCkNN gives users the split nodes as k increases and there is no need to perform kNN processing for any node on the path. In addition, VCkNN does not consider the segment between every intersection. This feature improves the performance because finding split nodes segment by segment is not efficient, especially if there are too many intersections on the query path. We have performed several experiments to measure the performance of VCkNN in different



(a) M_NN



(b) iM_NN



(c) PM_NN

Figure 3.18: Processing Time Comparison

network conditions. In general, our algorithm performs better if the density is low, especially in segment division mechanism. If the number of interest objects is smaller than 5, the performance is acceptable no matter how complex the road condition is. However, as expected, if k is greater than 5, the runtime increase sharply. Also the runtime is related to the length of the query path and the polygon it goes across. On average, high density of interest points and more crossing polygons will let the runtime and expansion step increase. If k is large, the runtime will increase sharply. When comparing VCkNN with other approaches, we can conclude that the advantage of VCkNN becomes obvious if the interest points are highly density distributed.

Secondly, inspired by novel kNN search involving multiple object types, we discussed another 3 set of queries using the Voronoi Diagram. The first query (nearest neighbor for multiple object types) provides a solution if the user wants to get 1_NN for each category of interest points. The second query (incremental nearest neighbors for sequential multiple object types) helps users to find the shortest path to pass through multiple object types in pre-defined sequence. The last query (optimum path for multiple object types) provides an optimum path for users if they want to pass multiple object types without any sequential constrain. These approaches investigate novel kNN in multiple object types using a network Voronoi Diagram which enriches the content of our mobile navigation system and gives more benefits to mobile users.

To sum up, both approaches can solve their corresponding queries efficiently when the Voronoi Diagram is utilized compared to Network Expansion.

Chapter 4

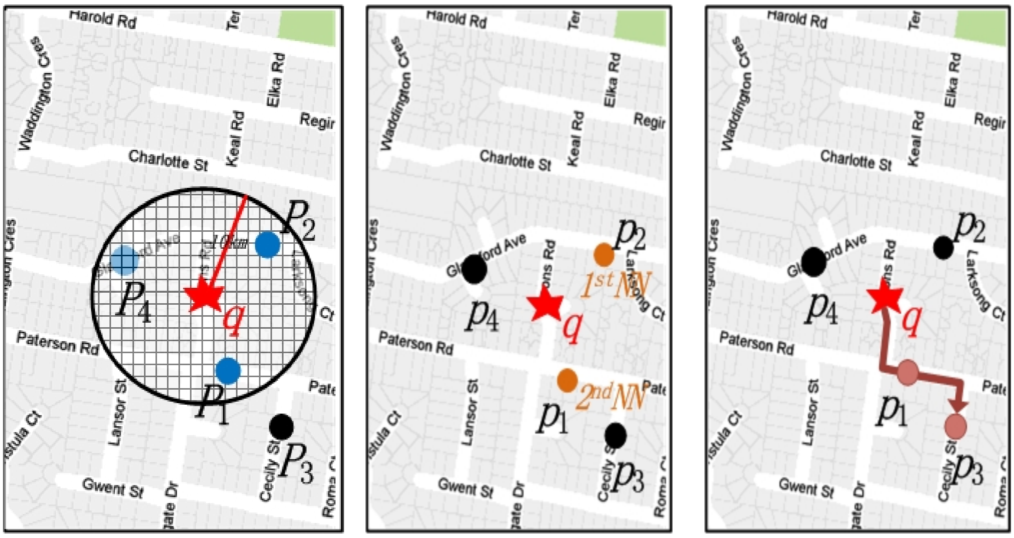
Route and Path related kNN Queries[1]

4.1 Introduction

Traditional query in spatial databases are range search [PZMT03, JT05] and k nearest neighbor search (kNN) [?, RKV95, Saf05]. Range search is to find all interest objects within a predefined range, while kNN is to find k interest objects which are closest to a query point. Both range and kNN searches provide users the candidate set of interest points and allow users to choose any one in the set because they have been previously filtered by user's conditions. From the description, we call tell the traditional range search and k nearest neighbor search are retrieving discrete points. Motivated by this, we propose 3 approaches in this chapter which brings path into the input or/and output of spatial queries.

Firstly, a possible query that a user may invoke is as follows: A market researcher may want to do a survey on restaurants and the sample size should be 10. The question is to find the shortest path for the user to visit all the 10 restaurants one by one. Range search cannot be used as there is no fixed range. kNN search cannot be used either, as after we visit the first interest point, the user may not want to

[1]Part of this chapter has been published in Zhao, G., Xuan, K., and Taniar, D. Path kNN query processing in Digital Ecosystems, IEEE Transactions on Industrial Electronics (TIE) 2011.



(a) Range search (b) Traditional kNN (c) Proposed pkNN

Figure 4.1: Result comparisons

return to query point and go to the second one. In this case, the user wants to continue to go to the second location from the first, and so on. This is a typical path based k nearest neighbor query (pkNN). The difference between range search, traditional kNN, and our proposed pkNN is highlighted in Fig.4.1 in section 4.2.

pkNN is described as given a set of candidate interest objects, a query point and the number of objects k, find the shortest path which starts from the query point and goes through k interest objects. By following this path, a user can visit all k interest objects one by one, and furthermore, this path has the shortest distance among all other possible paths.

Secondly, another query comes into our minds that is called path branch point(PBP) and which is discussed in section 4.3. PBP can be defined as: given a set of candidate interest objects and a pre-defined path which starts at S and end at E, find a path which starts at S, via an interest point p and ends at E. This path should overlap with the pre-defined path as much as possible with acceptable distance increment. This is a novel query which is motivated by users' common requirements because most users have ad hoc paths in their daily travel and they can tolerate a longer driving distance to some extent if they can drive on a familiar path. In this proposed approach, an **Adjust Score** is calculated for each path which is determined by overlapping distance and increased distance cost. The following example explains the query.

Fig.4.6 is an example of a path branch points query. The pre-defined path is marked as a red line in Fig.4.6 which starts at S and ends at E. Our aim is to find another path which starts from S, via one p and ends at E. This path should overlap the pre-defined path as much as possible under the condition that the driving distance increment is acceptable. As an example, take two paths which go through p_1 and p_2 respectively. One of the possible paths via p_1 ($Path_1$) is $S \rightarrow p_1 \rightarrow n_2 \rightarrow n_6 \rightarrow E$ and one of the possible paths via p_2 ($Path_2$) is $S \rightarrow n_2 \rightarrow n_{10} \rightarrow p_2 \rightarrow E$. How to determine which path is more suitable (optimal) to the user's requirement is the main target of this chapter.

This section makes three main contributions. First, the path branch point (PBP) query problem is defined. Second, we incorporate kNN search query and route search algorithms to process our PBP query using the network distance metric and three factors, Distance Cost (DC), Overlap Factor (OF) and Adjust Score (AS) are introduced and defined. By using these three factors, we can scale whether the path is the one the user wanted or not. For the third contribution, we evaluate this approach using experiments under different interest point distributions. Our experiments verified the applicability of the proposed approach to solve the queries, which involve finding the optimal path branch points.

Thirdly, as route search has been extended to include locations to be visited along the planned route [KSSD08, YS05, HJ04, ZXTS08, ZXR+, TBPM05, KSS09, KSSD08, Zha08]. The aim was to find the shortest distance, and sometimes the most reliable route, that covers all user-defined locations or places. Although this is certainly useful, it is often impractical, due to a couple of reasons: (i) each location or place, which are normally a spatial business entity (e.g. bank, dry cleaner, supermarket) has the opening hours - this implies that when this place is visited, it must be during their business hours; and (ii) the traveling time from one location to another needs to be considered, as in many cases, traveling time is more useful than the distance alone. Hence, in order to make route planning over visiting locations, one must take into account these two constraints. In section 4.4, we refer to these constraints as Time Constraints. Therefore, our chapter focuses on route search over multiple locations taking into consideration time constraints.

It is therefore imperative to assume that the route or path that arrives on the location outside the operation hour is considered as an invalid path. This problem exists in daily life, whereby we sometime have to choose a longer path to go back and forth places just to meet the business hours of one location before its closing time. Hence, we need to draw time constraints into our proposed methods.

Route search over multiple locations is often assumed to be the problem of kNN or continuous kNN in spatial and mobile databases [?]. There is a huge distinction between kNN and route search. kNN finds spatial objects that are closest located to the query point, without considering the path that needs to be established as the user has to visit each objects in the query result. Because of this, existing work on kNN is inapplicable to solve route search problems. Our previous work on incremental kNN (called ikNN) [ZXTS08] attempts to solve route search problem whereby it could find the shortest distance to cover k number of homogenous type of locations - however, it does not consider time constraints, nor heterogenous multiple types of locations.

In section 4.4, we focus on two problems of route search over multiple heterogenous locations: one for fixed locations, and the other for flexible locations. Fixed locations refer to predetermined locations by the user, such as Citibank on a specific location, Pharmore pharmacy on a specific location, etc. In this case, not only a specific business entity is specified, such as Citibank and not any bank, or Pharmore pharmacy and not any pharmacy, but also the specific location, such as Citibank on 180 High Street, or Pharmore pharmacy on 25 Cure Road, etc. Hence, a Route Search over Fixed Locations (our proposed algorithm is then called $RFix$) finds the most efficient route to visit the user-defined fixed locations in a non-predefined order.

Flexible locations, on other hand, refer to predetermined location types, are not the exact location itself. For example, if user wants to visit a pharmacy, which can be the pharmacy anywhere; or to visit Citibank, but can be in any branch. So, a route search over flexible locations for example is to find the most efficient route to visit Citibank, a pharmacy, etc, in a non-predefined order. Our proposed algorithm for Route Search over Flexible Locations is abbreviated as $RFlex$. Both $RFix$ and $RFlex$ use the travel time network to estimate the travel time between any two locations, as well as using the time constraints imposed by not only the operating hours of each location, but also the traveling time itself.

To sum up, chapter.4 is the second main chapter of this thesis, which includes 3 approaches of path/route based k Nearest neighbor search. More specific descriptions are:

- Section 4.2 proposes a query that is called path based k nearest neighbor search. It aims at providing a path that visits k objects and the length of the path that is the shortest.

- Section 4.3 explains a query which is called path branch point route search. By given the query path and an object type, path branch point route search retrieves the optimal path that balances the overlap ratio of query path and the length of result path.

- Section 4.4 describes a novel route research which adds time constraint into the search. In addition, a user may define the objects visiting sequence as sequential or random.

Let us begin the main part of these three approaches with the performance evaluation followed after.

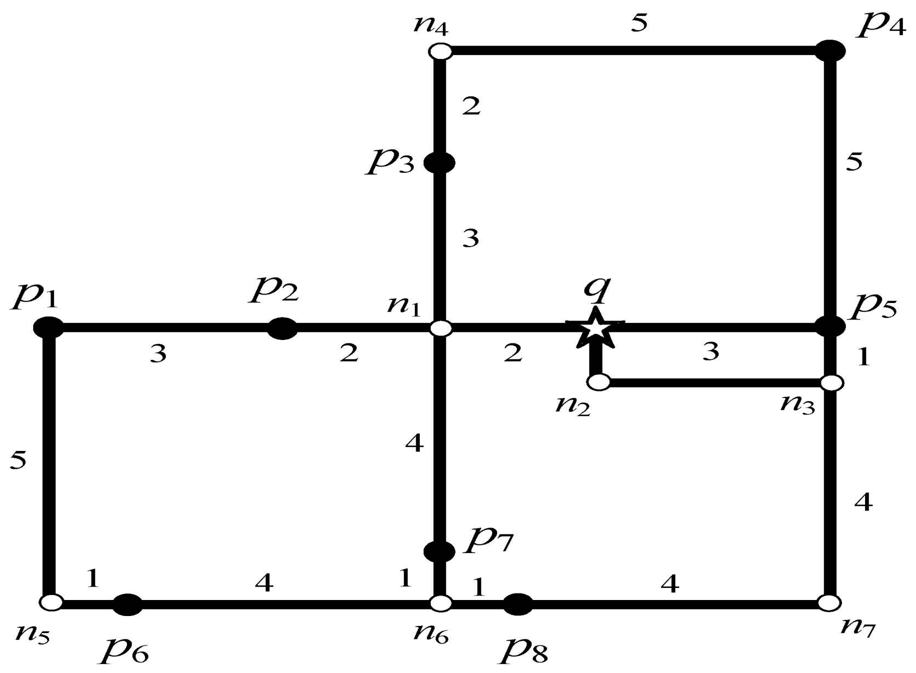


Figure 4.2: An example of road networks

4.2 Approach 1: Path based kNN Search Queries

Nowadays, most queries in mobile databases take the road networks into consideration because Euclidean distance, in most situations, cannot reflect the real connections between interest objects. The driving distance, or time cost, are determined by the shortest distance in road connection between objects. As a result, firstly, we define the method that how a weighted map is constructed. Also in this section, we introduce the data structures which is going to be used in the approach.

4.2.1 Definition of road network elements

Fig.4.2 is an example of road networks, in which query point q, road network intersections n_1-n_7 (white points), and interest points p_1-p_8 (black points) are vertices and the solid lines connecting these vertices are edges. The number on each edge represents the shortest distance, in other word, the weight of the edge.

Definition 4.2.1. ***(Expansion)*** *Expansion is the traversal from a vertex v_i to all of its adjacent vertices.*

In Fig.2.1, from q, three q's adjacent vertices n_1, n_2 and p_5 will be expanded.

4.2.2 Data structure

In the approach processing progress, couple of data structures are going to be used, such as *pk*NN tuple, Result Set, Expanded Set, Boundary Set and distance. The following definitions defines them in details.

Definition 4.2.2. *(pk**NN tuples)*** *t=(v, $d_{net}(v,q)$, VP) where $v \in V$,* ***visited point set*** *VP=(p_1, p_2, ..., p_m) where $m \leq k$. Each tuple represents a path which starts from q to v.*

In Fig.2.1, t_1=(n_3, 4, $\{p_5\}$) and t_2=(n_3, 4, $\{\emptyset\}$) are two paths. t_1 starts from q to n_3 via p_5, in other word, $q \to p_5 \to n_3$ while t_2 is the path via null object of interest which is $q \to n_2 \to n_3$.

Definition 4.2.3. ***(Result Set** RS)* *RS=$\{t|t.v$ is a current expansion node $\bigcap$ $t.d_{net}(v,q) \leqslant d_{max}\}$. RS holds the pkNN tuples after expansion and sorted by $t.d_{net}(v,q)$.*

Definition 4.2.4. ***(Expanded Set** ES)* *ES=$\{t|t$ is the expanded pkNN tuple $\bigcap$ $t.d_{net}(v,q) \leqslant d_{max}\}$. ES is designed to hold all expanded pkNN tuples to do further pruning.*

See example shown in Fig.2.1, after expanded from q, *RS*=$\{(n_2, 1, \{\emptyset\}), (n_1, 2, \{\emptyset\}), (p_5, 3, \{p_5\})\}$. And the *pk*NN tuple which has been expanded is moved into *ES*. As result, *ES*= $\{(q, 0, \{\emptyset\})\}$.

Definition 4.2.5. ***(Boundary Set and distance)*** *Boundary distance d_{max}= min ($t.d_{net}(v,q)$) where $\forall t.VP$=(p_1, p_2, ..., p_k). Boundary Set BS=$\{t|t_i.VP$=(p_1, p_2,*

..., p_k) $\bigcap$ $t.d_{net}(v,q) = d_{max}$}. BS stores all candidate shortest path based on current d_{max}.

Suppose the query of example in Fig.2.1 is to find $2k$NN, in the query processing, d_{max} is 8 because one complete path is found, which is $q \to p_5 \to p_4$ and the the path length is 8. As a result, BS={p_4,8,{p_5,p_4}}. Although at last, another shorter path is found as optimal path, at this stage, BS holds the candidate result path.

4.2.3 Proposed Method

In this subsection, the pkNN approach will introduce basic expansion, pruning conditions, accelerated approach and special issues in turn.

Basic Expansions

The approach of pkNN performs network expansion, which is similar with INE (Incremental Network Expansion). The expansion starts from query point q to all adjacent vertex and store the pkNN tuples into RS. Every time pop out one pkNN tuple to do further expansion until boundary set (BS) is found. In the processing progress, using pruning conditions to prune some redundant pkNN tuples to speed up the expansion. Then keep updating the boundary distance (d_{max}) until RS has been cleared. In INE, the visited nodes will not be expanded during the expansion, while in pkNN, all adjacent nodes are expanded to no matter whether the nodes have been visited or not.

Specifically, pkNN first initials ES and RS as {(q, 0, {$\emptyset$})}. Secondly, as q is on the top of the RS, we pop it out and retrieve all adjacent nodes of q (n_1, n_2 and p_5), expand to each of them and put their pkNN tuples into RS. As a result, RS={(n_2, 1, {$\emptyset$}), (n_1, 2, {$\emptyset$)}, (p_5, 3, {p_5})}. (p_5, 3, {p_5}) tuple means the path starts from q, ends at p_5 with distance 3 via interest objects p_5. Then following the former steps, pop (n_2, 1, {$\emptyset$}) out, add it into ES and do expansion to all n_2's adjacent nodes. Update boundary set BS and d_{max} until at least one completed route has been found. Keep updating BS and d_{max} until RS is empty.

Lemma (Pruning conditions)

As we stated before, in every expansion, all adjacent nodes are expanded which will cause a lot of redundant *pk*NN tuples because we allow go-and-back expansion. As a result, pruning conditions can speed up the algorithm by reducing useless *pk*NN tuples.

Lemma 4.2.1. *Given* t_x, $t_y \in RS$, *if* $t_x.v_x = t_y.v_y$, $t_x.VP = t_y.VP$ *and* $t_x.d_{net}(v_x, q)$ $\leq t_y.d_{net}(v_x, q)$, *then* t_y *needs to be pruned. Summarized as* **prune path with vain distance**.

Proof. Given an *pk*NN query, all candidate interest points are in Set POI, suppose t_x, $t_y{\in}RS$,$t_x.v_x{=}t_y.v_y$, $t_x.VP{=}t_y.VP$, $t_x.d_{net}(v_x, q){\leq}t_y.d_{net}(v_x, q)$, we should prove that t_y can not lead the optimal path under this assumption.

As $t_x.v_x{=}t_y.v_y$, $t_x.VP{=}t_y.VP{=}\{p_1,...,p_m\}$, then this becomes another ik'NN query where $k'{=}k\text{-}m$ and $POI'{=}POI\text{-}\{p_1,...,p_m\}$. Suppose $Path'$ is the optimal path of ik'NN which starts from $t_x.v_x$.

Under this condition $t_x.d_{net}(v_x, q) \leq t_y.d_{net}(v_x, q)$, then $t_x.d_{net}(v_x, q) + Path' \leq t_y.d_{net}(v_x, q) + Path'$. *pk*NN is to find the shortest path which starts from q and goes through k interest points, given $t_x.d_{net}(v_x, q){+}Path'$ and $t_y.d_{net}(v_x, q){+}Path'$ are two candidate *pk*NN path, we should prune the longer path which is $t_y.d_{net}(v_x, q){+}Path'$. So t_y needs to be pruned. □

Example 4.2.1. *As* $t_3{=}(p_4,8,\{p_5,p_4\})$ *and* $t_4{=}(p_4,10,\{p_5,p_4\})$, *because* $t_3.v = p_4 = t_4.v$, $t_1.VP = \{p_5,p_4\} = t_2.VP$ *and* $t_3.d_{net}(v, q) = 8 \leq t_2.d_{net}(v, q){=}10$, *so* t_4 *is pruned. In other words,* t_3 *and* t_4 *represent two paths with some start and end node as well as the visited nodes are identical (includes both interest nodes name and visiting sequence). The path with longer distance is pruned because it costs vain driving distance.*

Algorithm 5 PrunCond1_Lemma1(RS)

```
for i=0,i<Size(RS),i++ do
    for j=0,j<Size(RS),j++ do
        if d_net(v_i,q)≤d_net(v_i,q)||v_i.VP=v_j.VP then
            RS=RS-t_j
        end if
    end for
end for
return RS
```

Lemma 4.2.2. *Given t_x, $t_y \in RS$, if $t_x.v_x = t_y.v_y$, $t_x.VP \supset t_y.VP$ and $t_x.d_{net}(v_x, q) \leq t_y.d_{net}(v_y, q)$, then t_y needs to be pruned. Summarized as* **prune path with less efficiency**.

Proof. Given an pkNN query, all candidate interest points are in Set POI, suppose if $t_x.v_x$=$t_y.v_y$, $t_x.d_{net}(v_x, q) \leq t_y.d_{net}(v_y, q)$ and $t_x.VP \supset t_y.VP$, we should prove that t_y can not lead the optimal path under this assumption.

Given the conditions $t_x.v_x$=$t_y.v_y$, $t_x.VP \supset t_y.VP$, suppose $t_x.VP$=$\{p_1,...,p_n,...,p_m\}$, $t_y.VP = \{p_1,...,p_n\}$. Then from $t_x.v_x$, how to find optimal path becomes another ik'NN query where k'=k-m and POI'=POI-$\{p_1,...,p_m\}$, and from $t_y.v_y$, how to find optimal path becomes another ik''NN query where k''=k-n and POI''=POI-$\{p_1,...,p_n\}$.

Suppose $Path''$ is the optimal path of pk''NN which starts from $t_{x/y}.v_{x/y}$, so $Path_{t_y}$+$Path''$ is one candidate pkNN path.

If $Path''$ contains some interest points say $0 \leq i \leq m - n$ of $t_x.VP$-$t_y.VP$, then $Path_{t_x}$+$Path''$ contains m+k-n-i>k interest points. As $t_x.d_{net}(v_x, q) \leq t_y.d_{net}(v_y, q)$, so $Path_{t_x}$+$Path'' \leq Path_{t_y}$+$Path''$. Suppose $Path_{sub}$ is the sub-path of $Path_{t_x}$+$Path''$ which goes through k interest points, so $Path_{sub} \leq Path_{t_x}$+$Path'' \leq Path_{t_y}$+$Path''$. As a result, t_y needs to be pruned. □

Example 4.2.2. *Suppose t_1=(n_3, 4, $\{p_1,p_5\}$) and t_2=(n_3, 4, $\{p_1\}$), because $t_1.v = n_3 = t_2.v$, $t_1.d_{net}(v, q)$ =4 $\leq t_2.d_{net}(v, q)$=4 and $t_1.VP$=$\{p_1,p_5\} \supset t_2.VP$=$\{p_1\}$, as a result, t_2 is pruned. To sum up, t_1 and t_2 represent two paths with some start and end node. t_1 visits more interest objects than t_2, in other word, t_1 visits more objects* *after visiting all $t_2.VP$ in same sequence. Moreover, t_1's distance is not longer than t_2's distance. We can conclude that t_2 is a path with less efficiency, so prune it.*

Algorithm 6 PrunCond2_Lemma2(RS)

```
for i=0,i<Size(RS),i++ do
    for j=0,j<Size(RS),j++ do
        if d_net(v_i,q)≤d_net(v_i,q)||v_i.VP⊃v_j.VP then
            RS=RS-t_j
        end if
    end for
end for
return RS
```

Lemma 4.2.3. *Given $t_x \in RS$, if $t_x.d_{net}(v_x, q) > d_{max}$, then t should to be pruned. Summarized as* **shrink** *RS* **by Boundary Distance** (d_{max})*.*

Proof. Suppose $Path$ is the full path which contains t_x, then $length_{Path} > t_x.d_{net}(v_x, q)$. Because $t_x.d_{net}(v_x, q) > d_{max}$, then $length_{Path} > d_{max}$, in other word, $Path$ is not the optimal path of pkNN. So t_x is pruned. □

Example 4.2.3. *If $d_{max}=7$, there is a tuple $t_5=(n_1, 8, \{p_3\})$ in RS. It is pruned because $t_5.d_{net}(n_1, q)=8>d_{max}$. This lemma is summarized as following: the pkNN tuples in RS represent uncompleted path but with longer distance than the completed path has been found. So there is no chance this tuple can be the optimal path, this tuple is pruned. As a result, RS shrink using d_{max}.*

Algorithm 7 PrunCond3_Lemma3(RS,d_{max})

```
for i=0,i<Size(RS),i++ do
    if d_net(v_i,q)>d_max then
        RS=RS-t_j
    end if
end for
return RS
```

Lemma 4.2.4. *Tuples in ES can prune pkNN tuples in RS using Lemma 1 and 2. Summarized as ES* **can prune** *RS* **by lemma 1 and 2***.*

Proof. Def. 4.2.4 tells RS is to keep the expansion history. Suppose $t_1 \in RS$, $t_2 \in ES$, t_1 and t_2 fit Lemma 1 and 2, t_1 can be pruned. The reason are the same as proof of Lemma 1 and 2. □

Example 4.2.4. *As stated before, after first expansion from* q*, tuple* t_i*=(*q*, 0, {∅}) is in* ES*. Then after couple of expansions, there is tuple* t_j*=(*q*, 2, {∅}) in* RS*.* t_i *and* t_j *fit Lemma 1, so* t_j *in* RS *is pruned which accelerates the approach as well. This is main reason we use* ES *to keep expansion history.*

Algorithm 8 PrunCond4_Lemma4(RS,ES)

for i=0,i<$Size(ES)$,i++ **do**
    for j=0,j<$Size(RS)$,j++ **do**
        if $d_{net}(v_i,q) \leq d_{net}(v_i,q) || v_i.VP = v_j.VP$ **then**
            RS=RS-t_j
        end if
        if $d_{net}(v_i,q) \leq d_{net}(v_i,q) || v_i.VP \supset v_j.VP$ **then**
            RS=RS-t_j
        end if
    end for
end for
return RS

Most redundant pkNN tuples can be pruned by lemma 1-4 which accelerates the processing time of pkNN approach. There is no order between four pruning conditions. For each tuple in RS, if it fits one pruning condition, just prune it. RS keeps the tuples which are unfit for all four pruning conditions. Tuples which fits one pruning conditions commonly exist after each expansion. As a result, the performance can be accelerated by using four pruning lemmas after each expansion. The following accelerated approach algorithm (Algorithm 9) is produced based on these pruning conditions.

Special Issues

After addressing the approach, there are three important issues need to be clarified in this section.

Algorithm 9 pkNN(q,k)

Initial P={p|all interest objects}, N={n|all intersection nodes}, $V=q\cup P\cup N$
Initial $RS=\{(q, 0, \{\emptyset\})\}$, $ES=\emptyset$, $BS=\emptyset$, $d_{max}=\infty$
Initial pkNN tuple structure t=(v, $d_{net}(v,q)$, $VP\subset P$, when $Size(VP)=k$, the path is complete.
Do{
De-queue the top t in RS and $ES=ES\cup t$
Find all t's adjacent nodes in V
for Each t's adjacent node nd **do**
  if $nd\in P$ and $nd\notin t.VP$ **then**
    if $Sizet.VP< k$-1 **then**
      $RS=RS\cup(nd,d_{net}(v,q)+d_{net}(v,nd),\{t.VP\cup nd\})$
    else
      if $d_{net}(v,q)+d_{net}(v,nd)< d_{max}$ **then**
        $d_{max}=d_{net}(v,q)+d_{net}(v,nd)$
        $BS=\{(nd,d_{net}(v,q)+d_{net}(v,nd),\{t.VP\cup nd\})\}$
      else
        if $d_{net}(v,q)+d_{net}(v,nd)=d_{max}$ **then**
          $BS=BS\cup(nd,d_{net}(v,q)+d_{net}(v,nd),\{t.VP\cup nd\})$
        end if
      end if
    end if
  else
    $RS=RS\cup(nd,d_{net}(v,q)+d_{net}(v,nd),\{t.VP\})$
  end if
end for
PrunCond1_Lemma1(RS)
PrunCond2_Lemma2(RS)
PrunCond3_Lemma3(RS,d_{max})
PrunCond4_Lemma4(RS,ES)
} **While** ($RS\neq\emptyset$)

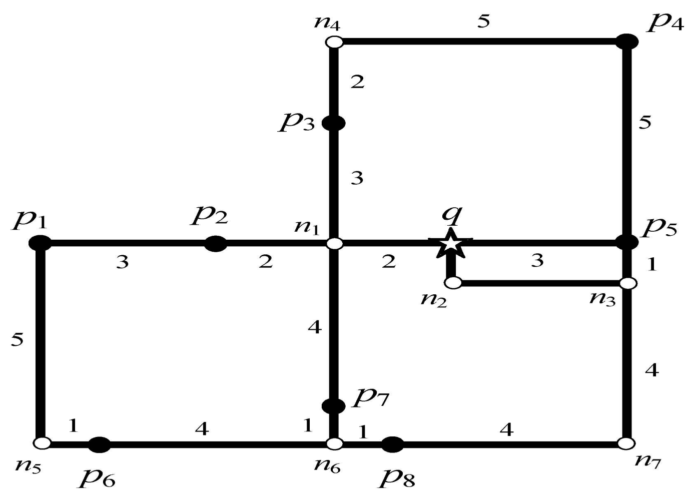


Figure 4.3: An example of Local minima scenario

Local minima: One main question from the pkNN queries is that whether we can use multiple 1_NN, instead of pkNN to answer the same query? For example, if we want to find p2NN, can we use 1_NN first in query point to find nearest p_i and then use 1_NN on p_i to find nearest p_j ? The answer is NO. See example in Fig.4.3, using multiple 1_NN, we will get a path $q \rightarrow p_5 \rightarrow p_4$ because p_5 is closest to q and p_4 is closest to p_5. Whereas using p2NN, the path would be $q \rightarrow p_2 \rightarrow p_1$. And actually distance $dist(q \rightarrow p_2 \rightarrow p_1) = 7$, which is shorter than $dist(q \rightarrow p_5 \rightarrow p_4) = 8$. This simple example shows that it is impossible to use multiple 1_NN technique to answer pkNN queries.

Looping scenario: In pkNN, we do a full expansion, which means we expand to all adjacent neighbor nodes. It is different from INE and Dijistra's algorithm because will full expansion, we allow go-and-back path. As a result, a looping scenario is well performed in pkNN. But lemma 1 and 2 can prune the redundant looping paths.

See example in Fig.4.4. Starting from q, after few expansion to n_1, n_2, p_4,n_3,n_4 in turn, the path ends at n_4. Then the next expansion will go to q and this is a typical looping scenario. This looping tuple can be pruned by lemma 1 because the second round will cost longer distance than the first round.

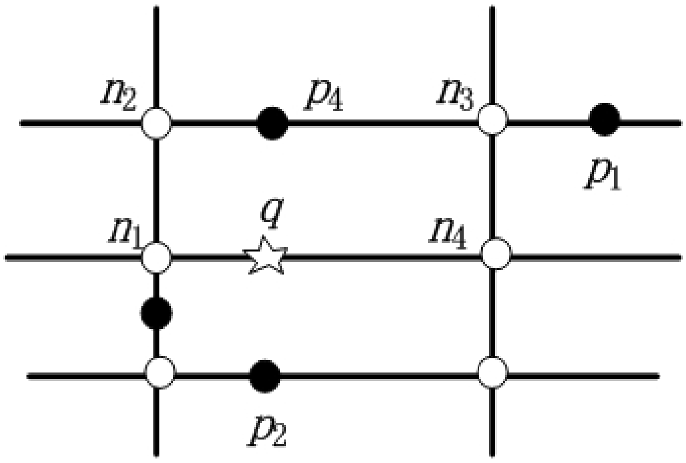


Figure 4.4: An example of looping scenario

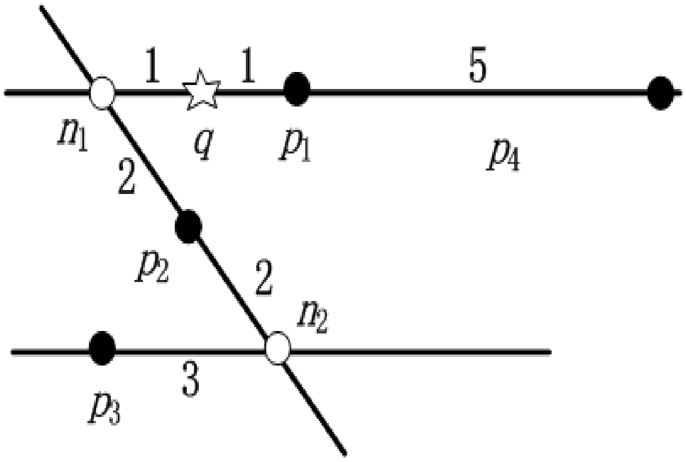


Figure 4.5: An example of U-Turn scenario

U-Turn scenario: As we expand the path to all adjacent nodes, we allow the user to go back anytime. Fig.4.5 shows that this condition should be allowed in the algorithm; otherwise the answer of the query can be wrong.

Suppose the query is 2pNN. After expansion from q, we got the following tuples in $RS = \{(p_1,\ 1,\ \{p_1\}),\ (n_1,\ 1,\ \{\emptyset\})\}$. When we expand p_1, if we do not allow U-Turn, we will not add q into RS, so we cannot find n_1, then p_2 in future, so the result will be $q \rightarrow p_1 \rightarrow p_4$. Actually, the shortest path should be $q \rightarrow p_1 \rightarrow p_2$ because $dist(q \rightarrow p_1 \rightarrow p_2) = 5$ is smaller than $dist(q \rightarrow p_1 \rightarrow p_4) = 6$.

Now we can summarize pkNN algorithm in general, our pkNN approach essence is full expansion with four pruning conditions to improve the performance evaluation, in other word, accelerate the query processing, at the same, three issues (local minima, looping and U-turn scenarios) are been emphasized to ensure the correctness of the pkNN result.

4.3 Approach 2: Path Branch Point based kNN Search

4.3.1 Preliminaries

Euclidean distance, which has been widely used in most queries in mobile database, cannot reflect the real connections between interest points. The measure which is investigated in this approach is network distance. The network distance depends on the underlying road network which links the interest points, while Euclidean distance reflects the relative positions of interest points. So at the beginning of this section, the road network and network distance are defined.

In addition, the two factors DC (distance cost) and OF (overlap factor) are introduced. By using DC and OF, a formula can distinguish one path from the others.

Finally, the data structure is also illustrated to clarify the algorithm in section 4.3.2.

Definitions

In the query of our approach, one of the given condition is pre-defined path. Def. 4.3.1 defines the notation of pre-defined path.

Definition 4.3.1. ***(Pre-defined Path)*** *is one condition in the query, which starts at point S and ends at point E. Pre-defined path can be notated as $Path_{pre}$.*

Definition 4.3.2. ***(Overlap Segment)*** *If both of $Path_a$ and $Path_b$ contain segment s_i, s_i is defined as $Path_a$ and $Path_b$'s overlap segment, marked as $OS_{a,b}$.*

Definition 4.3.3. ***(Separation)*** *and* ***(Regression Point)*** *If n_in_j is one of the overlap segment $OS_{a,b}$ of $Path_a$ and $Path_b$ and according to the direction of $Path_{pre}$,*

n_i is antecedent point while n_j is the succedent point, n_i and n_j are defined as a pair of Separation Point ($SP_{a,b}$) and Regression Point ($RP_{a,b}$) respectively.

Factors and Formula

Definition 4.3.4. ***Overlap Factor(OF)*** *of $path_n$ is the percentage of $OS_{n,pre}$ out of $dist(Path_n)$, if $path_n$ starts from S, via p_i and ends at E, $p_i \in P$.*

$$OF_{path_n} = \frac{\sum_{i=1}^{n} dist(OS_{n,pre})}{dist(Path_n)} \tag{4.1}$$

From Def. 4.3.4 and equation(4.1), we can see that the OF is the factor of overlap distance divide $Path_n$. For example, if the overlap part of $Path_n$ and $Path_{pre}$ is 7km and $Path_n$ is 11km, we can calculate that the OF of $Path_n$ is $\frac{7}{11}$.

Definition 4.3.5. ***Distance Cost(DC)*** *of $path_m$ is the percentage of $dist(Path_{pre})$ compared to $dist(Path_m)$, if $path_m$ starts from S, via p_i and ends at E, $p_i \in P$.*

$$DC_{path_m} = \frac{dist(Path_{pre})}{dist(Path_m)} \tag{4.2}$$

From Def. 4.3.5 and equation(4.2), we can see that is the DC is the percentage of $dist(Path_{pre})$ compared to $dist(Path_m)$. For example, if the $Path_{pre}$ is 10km, while $Path_m$ is 11km, the DC is $\frac{10}{11}$.

Definition 4.3.6. ***Adjust Score(AS)*** *is determined by the result of Overlap Decrement (OF) multiple Distance Cost (DC). The closer to 1, the better the path is.*

$$AS_{path_i} = OF_{path_i} * DC_{path_i} \tag{4.3}$$

After clarifying the Defs. 4.3.5-4.3.6 and equations 4.2-4.3, we can summarize as follows: factor DC examines the distance cost of path, factor OF represents the overlap, while OF multiplied by DC, which is factor AS examines whether the

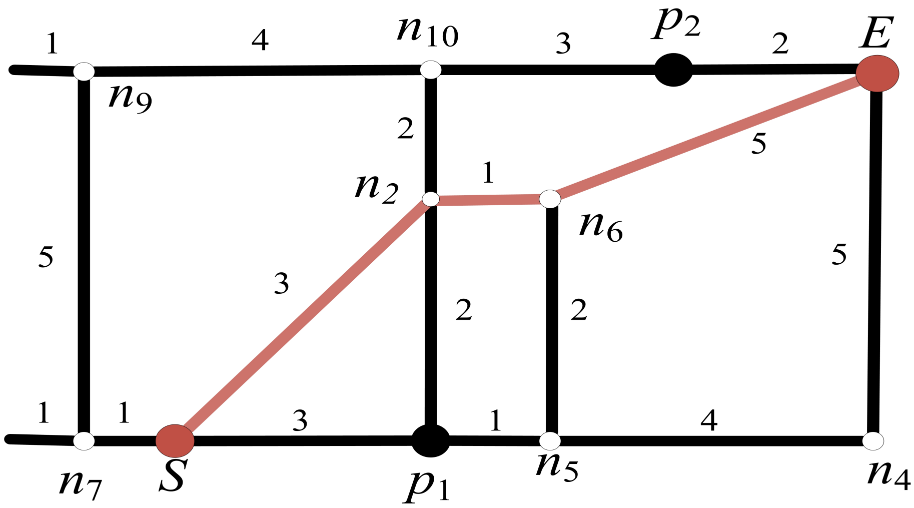


Figure 4.6: An example of path branch points query

balance between OF and DC is optimal. The closer to 1, the better the path is. To continue the example above, the AS is $\frac{7}{11} * \frac{10}{11} = \frac{70}{121} \approx 0.579$.

Definition 4.3.7. ***(Path branch points)*** *Given $Path_{pre}$ and $p_i \in P$, a Path branch point query finds the optimal $Path_a$ (with largest AS) and the pair of $SP_{a,pre}$ and $RP_{a,pre}$. $Path_a$ should start at S, via p_i and end at E.*

Example 4.3.1. *Fig.4.6 is an example of a path branch points query. The predefined path is marked as a red line in Fig.4.6 starting at S and ending at E. Taking p_1 and p_2 as an example, one possible path through p_1 ($Path_1$) is $S \rightarrow p_1 \rightarrow n_2 \rightarrow n_6 \rightarrow E$ and one possible path through p_2 ($Path_2$) is $S \rightarrow n_2 \rightarrow n_{10} \rightarrow p_2 \rightarrow E$. The factors of $Path_1$ and $Path_2$ are calculated as follows.*

$OF_{Path_1} = \frac{1+5}{3+2+1+5} = \frac{6}{11}$.

$DC_{Path_1} = \frac{3+1+5}{3+2+1+5} = \frac{9}{11}$.

$AS_{Path_1} = OF_{Path_1} * DC_{Path_1} = \frac{6}{11} * \frac{9}{11} = \frac{54}{121} \approx 0.446$.

$OF_{Path_2} = \frac{3}{3+2+3+2} = 0.3$.

$DC_{Path_2} = \frac{3+1+5}{3+2+3+2} = 0.9$.

$AS_{Path_2} = OF_{Path_1} * DC_{Path_1} = 0.3 * 0.9 = 0.27$.

From this example, we call see that $Path_1$ *is better than* $Path_2$ *even* $Path_1$*'s distance is longer than* $Path_2$*. As* $Path_1$ *has more overlap with the pre-defined path, which means the user can drive further on the familiar path,* $Path_1$ *is determined to be a better path than* $Path_2$ *which fits our motivation as well.*

4.3.2 Proposed Approach

In this part, the path branch points approach (PBP) will be introduced, including lemmas, pruning conditions, algorithms and the process will be outlined.

PBP query

In this part, the query of our proposed path branch points (PBP) approach is described.

In daily life, most people have a preferred route if the start point and destination are given. For example, when traveling from the office to home, a person generally takes the same route each day, which is usually the shortest or fastest. Users are disinclined to change the route, even when they need to visit another destination along the route. Most users agree that if the driving distance is not too much greater, they prefer to keep as much as possible to the same route. How to balance the increment of the driving path with the overlapping percentage of the pre-defined path is the motivation of this chapter.

Given a pre-defined path which starts at S and ends at E, a user decides to visit one specific type of interest point along the path. Choosing the best path which not only intersects with the user's pre-defined path, but also overlaps with it as much as possible, providing the increase in the driving distance is acceptable, is a general description of the PBP query. In fig.4.6, our approach is to determine which is the better path from $S \rightarrow p_1 \rightarrow n_2 \rightarrow n_6 \rightarrow E$ and $S \rightarrow n_2 \rightarrow n_{10} \rightarrow p_2 \rightarrow E$, and to finally find the optimal path of all possible paths.

Lemmas

Section 4.3.1 has already clarified the factors which determine the optimal path. In this section, lemmas are illustrated based on an in depth analysis of OF, DC and AS factors.

Lemma 4.3.1. *If dist(Path$_a$)* $= \min\{\forall$*dist(Path)—Path starts at S and ends at E*$\}$ *and dist(Path$_b$)* $= \min\{\forall$*dist(Path) —Path starts at S, via P and ends at E,*$p_i \in P\}$*, dist(Path$_a$)* $\leq$ *dist(Path$_b$), in other words,* $\forall$ *path$_b$,* $0 \leq DC_{path_b} \leq 1$.

Proof. Lemma 4.3.1 can be proven by contradiction.

Suppose $dist(Path_a) > dist(Path_b)$, which means $Path_b$ is shorter than $Path_a$. Because $Path_b$ starts at S and ends at E, which satisfies the condition of $Path_a$ as well, and $dist(Path_a) > dist(Path_b)$, $Path_a \neq \min\forall dist(Path)$—$Path$ starts at S and ends at E. This is against the given condition.

As a result, we can conclude that $dist(Path_a) \leq dist(Path_b)$ under given conditions.

So $0 \leq DC_{path_m} = \dfrac{dist(Path_{pre})}{dist(Path_m)} \leq 1$. □

Lemma 4.3.2. $\forall$ *path$_c$ — path$_c$ starts at S, via P and ends at E,* $p_i \in P$*, a conclusion can be drawn:* $0 \leq OF_{path_c} \leq 1$.

Proof. Because $OF_{path_n} = \dfrac{\sum_{i=1}^{n} dist(OS_{n,pre})}{dist(Path_n)}$, according to Lemma 4.3.1, $dist(Path_{pre}) \leq dist(Path_n)$.

With reference to Def. 4.3.2, we can conclude $\sum_{i=1}^{n} dist(OS_{n,pre}) \leq dist(Path_{pre})$.

As $\sum_{i=1}^{n} dist(OS_{n,pre}) \leq dist(Path_{pre}) \leq dist(Path_n)$, $OF_{path_n} = \dfrac{\sum_{i=1}^{n} dist(OS_{n,pre})}{dist(Path_n)}$ ≤ 1. □

Lemma 4.3.3. *Given dist(Path$_{pre}$)*$=\min$ *(*$\forall$*dist(Path)—Path at from S and ends at E), then* $0 \leq AS \leq 1$.

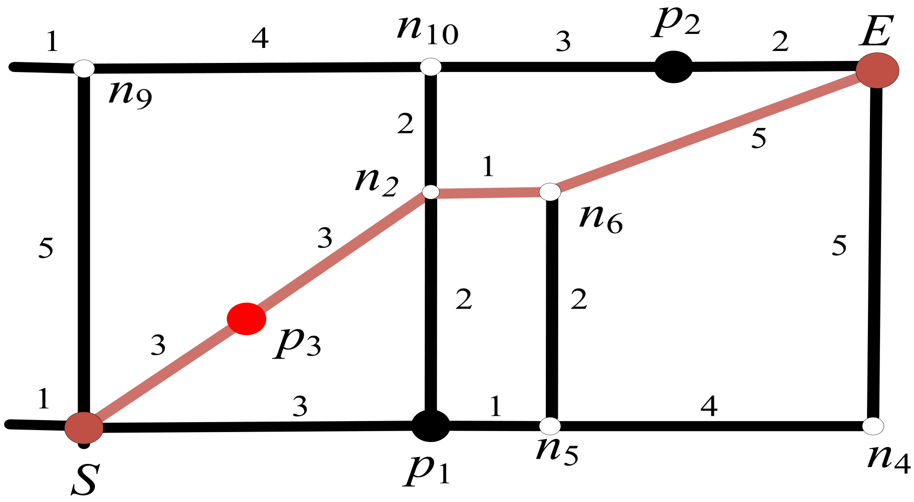


Figure 4.7: An example of Lemma 4.3.4

Proof. According to Lemma 4.3.1 and 4.3.2, $0 \leq OF_{path_i} \leq 1$ and $0 \leq DC_{path_c} \leq 1$, as a result, $0 \leq AS_{path_i} = OF^*DC \leq 1$. □

By deep analysis of Lemma 4.3.1, 4.3.2 and 4.3.3, we can conclude that the closer AS to 1, the more optimal the path.

Lemma 4.3.4. *If there is any interest point on the $Path_{pre}$, the optimal path is $Path_{pre}$ with no branch point.*

Proof. If there is any interest point on the $Path_{pre}$, we can conclude that $dist(Path_k) = dist(Path_{pre})$ as well as $\sum_{i=1}^{n} dist(OS_{k,pre}) = dist(Path_{pre})$.

Consequently $OF_{path_k} = \frac{\sum_{i=1}^{n} dist(OS_{k,pre})}{dist(Path_k)} = 1$ and $DC_{path_k} = \frac{dist(Path_{pre})}{dist(Path_k)} = 1$. So $AS_{path_k} = DC^*OF = 1$.

This is the optimal path because it has the largest AS value. □

Example 4.3.2. *Fig.4.7 is an example of Lemma 4.3.4. In Fig.4.7, there is an interest point p_3 located on the pre-defined path. According to Lemma 4, it is the optimal path with no branch point. We can demonstrate this using the following calculations.*

Suppose $path_i$ is $S \rightarrow P_3 \rightarrow n_2 \rightarrow n_6 \rightarrow E$. $OF_{path_i} = \frac{12}{12} = 1$ and $DC_{path_k} = \frac{12}{12} = 1$. So $AS_{path_i} = DC^*OF = 1$.

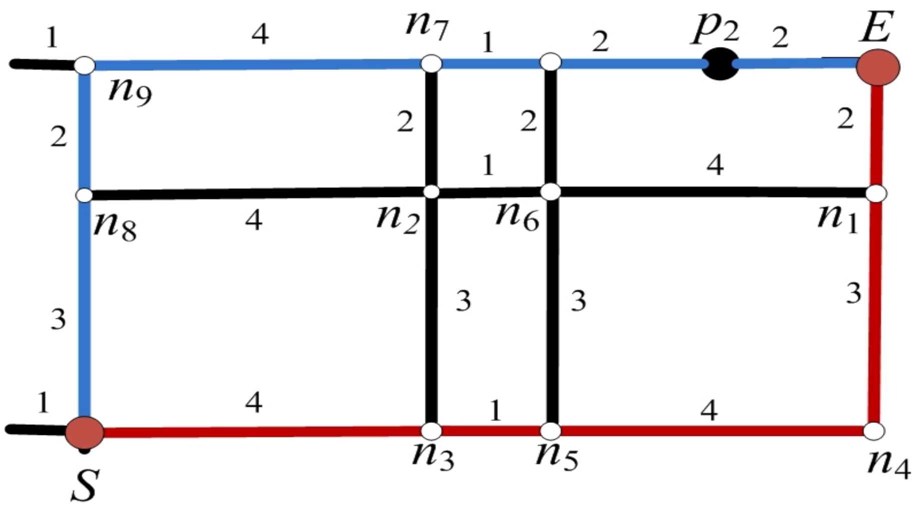


Figure 4.8: An example of Lemma 4.3.5

$path_i$ is the optimal path as it already reaches the top boundary of AS. It also fits our motivation because this path will allow the user to travel an a familiar route for as long as possible (in this case, the whole path is familiar to the user) with an acceptable increase in the driving distance (no distance increase in this case). It is clear that this is the optimal path.

Lemma 4.3.5. *If $\forall Path_j$—$\sum_{i=1}^{n} dist(OS_{j,pre})$=0, AS_{path_j}=0 and should be discarded.*

Proof. If $\sum_{i=1}^{n} dist(OS_{j,pre})$=0

$$\text{obviously } OF_{path_k} = \frac{\sum_{i=1}^{n} dist(OS_{j,pre})}{dist(Path_k)} = 0,$$

so AS_{path_j}=DC*OF=0.

This is the worst path because it has the smallest AS value. □

Example 4.3.3. *Fig.4.8 is an example of Lemma 4.3.5. In Fig.4.8, the red path is $Path_{pre}$ and the blue path ($Path_{blue}$) is the candidate path which starts at S, via p_2 and ends at E. According to Lemma 4.3.4, it is the worst path because there is no overlap between $Path_{pre}$ and $Path_{blue}$. We can demonstrate this using the following calculations.*

As $\sum_{i=1}^{n} dist(OS_{blue,pre})$=0 *obviously* $OF_{path_{blue}} = \dfrac{\sum_{i=1}^{n} dist(OS_{blue,pre})}{dist(Path_{blue})} = 0,$

so $AS_{path_{blue}} = DC * OF = 0.$

From lemma 4.3.5, we can see that although $path_{blue}$ *has no driving distance increment (*$dist(path_{blue}) = dist(path_{pre})$*), it is still considered as the worst path because the entire path is not familiar to the user. The user may want to drive further provided most of the path is familiar to him.*

Implementing the PBP query

In this section, the steps involved in implementing the PBP query are described.

Firstly, initial $BoundaryAS = 0$, $ResultList = \emptyset$, array $Intersections = [\emptyset]$, $DistBoun = \infty$, $ResultList = \emptyset$.

Secondly, find all interest points along $Path_{pre}$. If any, terminate the algorithm by returning these interest points and $Path_{pre}$. According to Def.3, this is the optimal path of this query.

Thirdly, consider S and E as group kNN and find all group NN results then group them into a set $CandidateSet$ sorted by the sum distance to S and E. Whenever $BoundaryAS$ is updated, update the $DistBoun$ using the following formula:

$DistBoun = \sqrt[2]{\dfrac{dist(Path_{pre})^2}{BoundaryAS}}.$

Then pop out the first p_i from $CandidateSet$. Find the shortest path $path_l$ from S via p_i to E. Let n_x be the separation point of $path_l$ and $path_{pre}$ ($sp_{l,pre}$) and n_y the regression point of $path_l$ and $path_{pre}$ ($rp_{l,pre}$). Calculate AS_{path_l}. If $AS_{path_l} > BoundaryAS$, set $BoundayAS = AS_{path_l}$, and replace $ResultList$ as $path_l$, n_x, n_y. If $AS_{path_l} = BoundayAS$, add $path_l$, n_x, n_y into $ResultList$. If $AS_{path_l} < BoundaryAS$, put all intersections between n_x and n_y into $Intersections$ including n_x and n_y in the visiting sequence. For each pair of n_i and n_j in $Intersections$ and n_i is forehead of n_j, form a $path_i$ as $S \rightarrow n_i \rightarrow p_i \rightarrow n_j \rightarrow E$. If $dist(path_i) \leq DistBoun$, calculate AS_{path_i} and follow the comparison above.

Algorithm 10 PBP ($Path_{pre}$,P)

1: Initial $BoundaryAS$=0
2: Initial $ResultList$=$\emptyset$
3: Initial array $Intersections$=[$\emptyset$]
4: Initial $DistBoun$=∞
5: Initial $ResultList$=$\emptyset$
6: $OnPath$=p—p_i on $Path_{pre}$
7: **if** Size($OnPath$)¿0 **then**
8:     **Return** $OnPath$
9: **end if**
10: $CandidateSet$=GkNN(S,E) sorted by dist(S,p_i)+dist(p_i,E).
11: Pop out first p_i in $CandidateSet$
12: Find the shortest path $path_l$ from S via p_i to E
13: **if** $dist$($path_l$)¡$DistBoun$ **then**
14:     n_x=$sp_{l,pre}$ and n_y=$rp_{l,pre}$
15:     Calculate AS_{path_l}
16:     **if** AS_{path_l}¿$BoundaryAS$ **then**
17:         Set $BoundayAS$=AS_{path_l}, $ResultList$=$\{path_l$, n_x, $n_y\}$
18:         Update $DistBoun = \sqrt[2]{\frac{dist(Path_{pre})^2}{BoundaryAS}}$
19:     **end if**
20:     **if** AS_{path_l}=$BoundaryAS$ **then**
21:         $ResultList$=$ResultList$$\cap$$\{path_l$, n_x, $n_y\}$
22:     **end if**
23:     **if** AS_{path_l}¡$BoundaryAS$ **then**
24:         $Intersections$=$\{n$—n_x,intersections between n_x and n_y, $n_y\}$ in visiting sequence
25:         **for** Each pair of n_i, n_j, i¡j **do**
26:             Form a $path_i$ as $S$$\rightarrow$$n_i$$\rightarrow$$p_i$$\rightarrow$$n_j$$\rightarrow$$E$
27:             **if** $dist$($path_i$)<$DistBoun$ **then**
28:                 Calculate AS_{path_i}
29:                 **if** AS_{path_i}¿$BoundaryAS$ **then**
30:                     Set $BoundayAS$=AS_{path_i}, $ResultList$=$\{path_i$, n_i, $n_j\}$
31:                     Update $DistBoun = \sqrt[2]{\frac{dist(Path_{pre})^2}{BoundaryAS}}$
32:                 **end if**
33:                 **if** AS_{path_i}=$BoundaryAS$ **then**
34:                     $ResultList$=$ResultList$$\cap$$\{path_l$, n_x, $n_y\}$
35:                 **end if**
36:                 Break
37:             **end if**
38:         **end for**
39:     **end if**
40:     Break
41: **end if**
42: **if** $Size$($CandidateSet$)¿0 **then**
43:     Go to Line 11
44: **end if**
45: Return $ResultList$

Next, continue pop out p_i from $CandidateSet$ and follow the previous steps until the sum distance to S and E in $CandidateSet$ is larger than $DistBoun$. Lastly, the optimal path is in $ResultList$. Algorithm 10 is produced to find the path branch points.

4.4 Approach 3: Time Constraint Route Search

4.4.1 Preliminaries

Some variations of route search have been investigated in earlier works [KSSD08, YS05, HJ04, ZXTS08, TBPM05, KSSD08, Zha08, EL05, PG98]. In [YS05, HJ04], they try to find the route with smallest deviation to visit a new point when a user travels a pre-defined route. [ZXTS08, TBPM05] have single type of interest points and no time constraint involved. Considering the inaccuracy and incomplete issues, some works assigned scores or probabilities to each locations and the result path should pass the locations with high probabilities [KSS09, KSSD08]. In our query, the criteria is the path with shortest travel time which is different from shortest path [ZXT+09b]. In addition, in some route search, the path can go through multiple locations of the same type [KSSD08, KSS09] while our path only visits one location for each type. Moreover, even if a lot of different constraints have been studied, there is no paper which draws time constraint into Route Search query. Although our proposed methods are significantly different from existing works [EL05], they are still worth to be reviewed as they become our motivation.

Route Search Query

Route search query was proposed by Yaron Kanza et. al in 2008 [KSSD08]. There are three semantics covered in this chapter, such as given start point, end point and all types of user interests, the first semantic is to return the shortest route that goes via all relevant entities. A second semantic is to find the most-profitable route, which is the route having the highest accumulative relevance and the length of the route is within the given limit. A third semantic is to compute the most-reliable route, which goes through as much higher relevant entities as possible and the length of the route is within the given limit.

Route search query uses greedy insertion from both start point and end point to find the final path. In route search, similar as our methods, multiple location types are concerned and the final path must pass one location of each type in any order.

Outstanding Problems

- Route search query chooses multiple criteria such as shortest distance, highest relevance or more relevant entities with higher relevance. While we choose the shortest travel time because we draw time constraint into the query and shortest distance cannot guarantee the shortest travel time.

- Route search query has a pre-defined path length limit while our methods are indifferent to travel distance.

- Route search query pre-defined the start and end point while our methods only give the start point.

- Route search query does not concern arriving time for any interest point while the optimum path of our methods should meet the time constraint of each type.

Path Based kNN

Path Based k nearest neighbor (PkNN) [ZXTS08] is given a set of candidate interest points to find the shortest path which starts at query point and goes through k interest points. For example, find the shortest path which goes through 3 restaurants from q. It is a novel kNN because the result is the shortest path and the interest points are visited one by one. While it is not a Route Search query neither because all candidate interest points are single type, while our Route Search query involves multiple types which needs access to different data structures such as multiple Network Voronoi Diagram [ZXT+09b, KS04, OBSC00] and the result path must cover every type.

PkNN uses network expansion as Incremental Network expansion (INE). In the process of network expansion, PkNN records all expansion branches until one path is full of k interest points. The path is set as the boundary. Continue to do the expansion, once there is a path shorter than the boundary; shrink it until all possible branches are expanded out of boundary. PkNN is similar with our proposed methods, such as both methods have a boundary identifier (D/T_{max}) to shrink the expansion scope and numbers of points to be visited are given.

Outstanding Problems

Single vs. Multiple types: PkNN considers all interest points as single type. In reality, the user may want to process the k nearest neighbor search in a multiple type objects environment.

Time Constraint: PkNN does not concern arriving time constraint for any interest point while the optimum path of our methods should meet the time constraint.

Additional Specifications

Before we move on to the proposed methods, more specifications are described and compared with other existing works:

- *Why cannot kNN solve Route Search query?*

  kNN cannot guarantee all locations to be fully covered, since the distance from query point to all interest points is the only criteria, and not the complete path.

- *What are the differences between incremental kNN and Route Search query?*

  In path based kNN, all interest points are considered as single type while in route search query, there are multiple types of interest points and the optimal path should go through all types of interest points.

- *Can multiple 1_NN find all interest points?*

  If we use 1_NN to find the nearest neighbor until all types have been found, there is high possibility that the final path is not the most optimum one. The first nearest interest point may lead to a further distance to other interest points, as a result, it does not yield an optimum path. This is a common local minimal problem in scheduling.

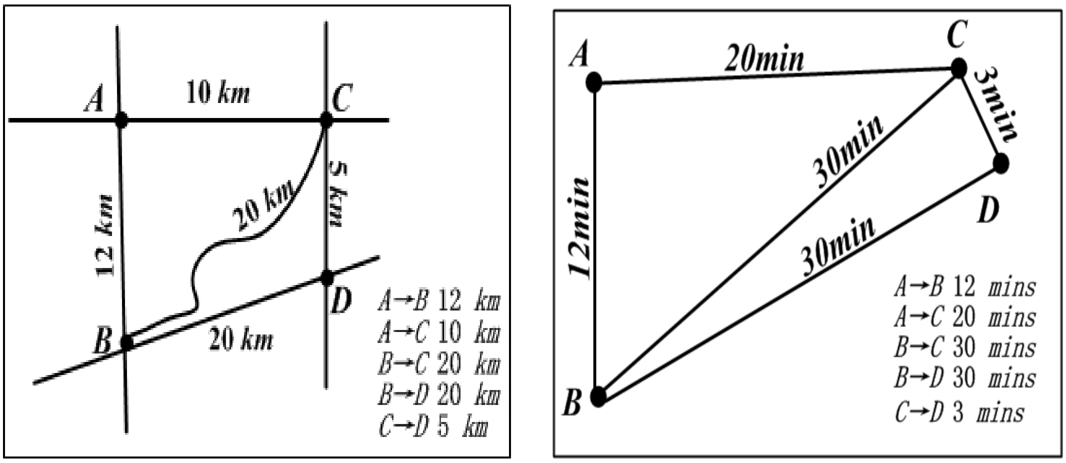
- *Why does the shortest time need to be used instead of the shortest distance?*

  Since the problem of answering Route Search queries is a generalization of the traveling salesman problem, it is unlikely to have an efficient solution, i.e., there is no polynomial-time algorithm that solves the problem (unless P=NP) [KSSD08]. Hence, as a solution, this chapter incorporates time constraint in order to prune as many expansion branches as possible and makes the query more realistic. If we use time constraint to prune the expansion branch, choosing traveling time as criteria is straightforward and can be adjusted to different travel time period.

4.4.2 Proposed Methods

In this section, our proposed Route Search for fixed locations (*RFix*) and flexible locations (*RFlex*) are described. In each method, the query is given first followed by detailed explanations of the key issues, then after listing the algorithm, an example will illustrate the processing steps. As travel time is chosen as criteria, *travel time network* needs to be introduced first.

In Travel time network [KZWW05], the measurement between nodes is the travel time 4.9(b) instead of network distance 4.9(a). This is often more desirable because under certain conditions travel time is more meaningful than network distance, such as whether the path arriving at locations within their operating hours depends on the travel time, not travel distance. In this chapter, we use the average travel speed in routine profile to estimate the approximate travel time. Fig. 4.4.2 gives an



(a) Road Network Denotation (b) Travel Time Network Denotation

Figure 4.9: Road network vs. travel time network

example of a travel time network. We assume that the average travel time for each road segment is read from the traffic profile.

Route Search for fixed locations ($RFix$)

When the query is a route search for fixed locations, this query can be categorized as Route Search for fixed locations ($RFix$). Example 4.4.1 illustrates a $RFix$ query. It can be expressed as follows: Start at q at 4:30pm, find the optimum path whose travel time is shortest and this path should visit A, B, C and D between 9:00am–5:00pm, 9:00am–5:30pm, 4:30pm–5:40pm and 6:00pm–6:30pm respectively. Now we can treat q, A, B, C and D as locations and invoke our proposed method $RFix$ to find an optimum path. The pruning conditions are explained as follows:

Example 4.4.1. *Secretary will leave her office at 4:30pm. She has a plan to do:*

♢ *Fetch a suit from dry cleaner A and A's trading time is 9:00am–5:00pm.*

♢ *Fetch a contract from Company B and B's open hours are 9:00am–5:30pm.*

♢ *Send a report to manager's apartment C and he is at home 4:30pm–5:40pm.*

♢ *Pick up her son from kindergarten D and D's pick up period is 6:00pm–6:30pm.*

$RFix$ Definition

The $RFix$ query can be formally defined like this:

Definition 4.4.1. *RFix is a route search query consisting of:*

Input: Type Set $T=\{t_1,t_2,\ldots,t_n\}$, *Locations set* $P=\{p_1,p_2,\ldots,p_n\}$, $\forall\ p_i \in t_i$.

Output: A Path l *which goes through all ps in* P *and distance*$_l$ *is the shortest.*

Pruning Conditions Since the problem of answering Route Search queries is a generalization of the traveling salesman problem, it is unlikely to have an efficient solution, hence an efficient pruning method is crucial. With the prune conditions, the candidate permutation is greatly reduced and that is the basis of our solutions. Two pruning conditions are discussed in this section. Firstly, definition for *Invalid Path* and *Valid Path* are introduced here.

Definition 4.4.2. *A path is invalid when at least one location's arriving time followed by this path is out of its operating hours; otherwise if it visits all user defined location types, it is a valid path.*

$$\left.\begin{array}{l} \exists P_i \\ Path(q \rightarrow P_1 \rightarrow \ldots \rightarrow P_i \rightarrow \ldots \rightarrow P_j) \\ T(q \rightarrow P_1 \rightarrow \ldots \rightarrow P_i) \notin OperatingHour(P_i) \end{array}\right\} \Rightarrow Invalid(q \rightarrow P_1 \rightarrow \ldots \rightarrow P_i \rightarrow \ldots \rightarrow P_j) \tag{4.4}$$

Pruning condition 1: non-reversible visiting sequence. If $q \rightarrow p_i \rightarrow p_j$ is a valid path while $q \rightarrow p_j \rightarrow p_i$ is an invalid path as in Equation (1), (p_i, p_j) have non-reversible visiting sequence ($q \rightarrow p_i \rightarrow p_j$). Hence, any solution visiting p_j before p_i should be pruned.

$$\left.\begin{array}{l} valid(q \rightarrow P_i \rightarrow P_j) \\ Invalid(q \rightarrow P_j \rightarrow P_i) \\ Type(P_i), Type(P_j) \subseteq LocationTypelist \end{array}\right\} \Leftrightarrow P_{i\nleftarrow}^{\;\rightarrow} P_j \tag{4.5}$$

Proposition 2. *Given a query point* q *and two locations* p_1 *and* p_2

$$\left.\begin{array}{l} StartTime > OpenTime(P_1) \\ StartTime > OpenTime(P_2) \\ CloseTime(P_1) < CloseTime(P_2) \\ Invalid(q \rightarrow P_1 \rightarrow P_2) \end{array}\right\} \not\Rightarrow Invalid(q \rightarrow P_2 \rightarrow P_1) \tag{4.6}$$

Proof. If $CloseTime(p_1) < CloseTime(p_2)$ which means p_1 closes earlier than p_2, it is possible that (p_1, p_2) have a non-reversible visiting sequence $q \rightarrow p_1 \rightarrow p_2$ because if we visit p_2 first, when we arrive p_1, it is already closed. It is self-evidence.

If $CloseTime(p_1) < CloseTime(p_2)$ and $q \rightarrow P_1 \rightarrow P_2$ is invalid, we will prove that we can not conclude $q \rightarrow P_2 \rightarrow P_1$ is invalid by contrapositive.

$$\left.\begin{array}{l} T(q, P_1) + T(P_1, P_2) \approx 2 * T(q, P_1) > CloseT(P_2) \Rightarrow Invalid(q \rightarrow P_1 \rightarrow P_2) \\ T(q, P_2) + T(P_2, P_1) \approx T(q, P_1) < CloseT(P_1) \Rightarrow Valid(q \rightarrow P_2 \rightarrow P_1) \end{array}\right\} \Leftrightarrow \exists P_2 \underset{\not\leftarrow}{\rightarrow} P_1 \tag{4.7}$$

$\square$

Pruning condition 2: Invalid sub-paths make the entire path invalid. Any permutation containing invalid sub-path should be pruned. See Proposition 2.

Proposition 3.

$$\left.\begin{array}{l} Invalid(q \rightarrow P_i \rightarrow P_j) \\ StartTime > Max(OpenTime(P_i), OpenTime(P_j)) \end{array}\right\} \Rightarrow Invalid(q \rightarrow ... \rightarrow P_i \rightarrow ... \rightarrow P_j) \tag{4.8}$$

Proof.

$$\left.\begin{array}{l} T(q \rightarrow P_i \rightarrow P_j) > CloseTime(P_j) \\ T(q \rightarrow ... \rightarrow P_i \rightarrow P_k \rightarrow P_j) > T(q \rightarrow P_i \rightarrow P_j) \\ P_k \neq P_i \neq P_j \end{array}\right\} \tag{4.9}$$

$$\Rightarrow T(q \rightarrow ... \rightarrow P_i \rightarrow ... \rightarrow P_j) > CloseTime(P_j)$$

$$\Rightarrow Invalid(q \rightarrow ... \rightarrow P_i \rightarrow ... \rightarrow P_j)$$

$\square$

The $RFix$ method is processed in the following steps and the algorithm is shown in Algorithm 11.

Algorithm 11 RFix(q,$StartTime$,$LocationSet$,$LocationOperatingHour$)

Load routine traffic speed and calculate travel time for all segments
Initial $EntitySet = q + LocationSet$
For any two in $EntitySet$, calculate its travel time
$First$=All locations whose travel time to q within earliest $CloseTime$
For any two locations p_i and p_j in $LocationSet$, check $q \rightarrow p_i \rightarrow p_j$ is valid or not. If it is invalid, put $q \rightarrow p_i \rightarrow p_j$ in $PruneList$
$CandidatePath$= Permutations of $LocationSet$ whose first point in $First$ and contain no subpath in $PruneList$
Initial $Total\_TimeCost = \infty$
for each candidate path in $CandidatePath$ **do**
    $TimeCost$=sum up all travel time and once $TimeCost$>$Total\_TimeCost$, terminate this loop
    if at some step, $TimeCost$ is out of p_i's $LocationOperatingHour$ **then**
        ignore this path and prune all path from $CandidatePath$ whose has it as sub path
        $TimeCost = \infty$
    end if
    if $Total\_TimeCost > TimeCost$ **then**
        $Total\_TimeCost = TimeCost$
    end if
end for
Final $Total\_TimeCost$ is travel time cost and its path is the optimum path

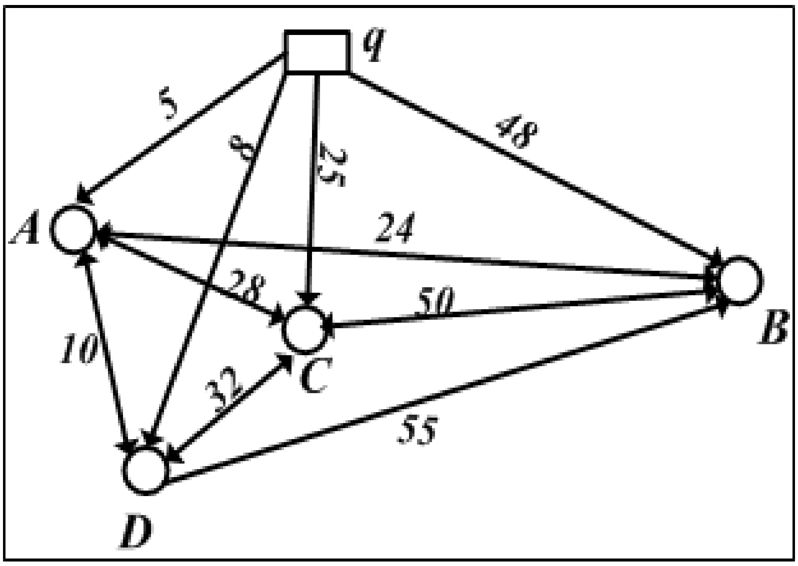


Figure 4.10: $RFix$ Example

Traversal Permu.	Action & Reason	$CandidatePath$
$q{\to}B...$	Pruned (B not in $First$)	None
$q{\to}C...$	Pruned (C not in $First$)	None
$q{\to}A{\to}B{\to}C{\to}D$		None
$q{\to}A{\to}B{\to}D{\to}C$		None
$q{\to}A{\to}D{\to}B{\to}C$	Pruned	None
$q{\to}D{\to}A{\to}B{\to}C$	Invalid subpath	—-
$q{\to}D{\to}B{\to}A{\to}C$	($q{\to}B{\to}C$)	None
$q{\to}D{\to}B{\to}C{\to}A$		None
$q{\to}A{\to}C{\to}D{\to}B$		None
$q{\to}A{\to}D{\to}C{\to}B$	Pruned	None
$q{\to}D{\to}A{\to}C{\to}B$	Invalid subpath	None
$q{\to}D{\to}C{\to}A{\to}B$	($q{\to}D{\to}B$)	None
$q{\to}D{\to}C{\to}B{\to}A$		None
$q{\to}A{\to}C{\to}B{\to}D$	Unpruned	$q{\to}A{\to}C{\to}B{\to}D$

Table 4.1: RFix Filter Process

Firstly, depending on current time period, retrieve the traffic speed and calculate the time cost between any two locations in the set which includes the query point and all fixed locations. The process is similar to Dijkstra algorithm if the weight between entities are travel time cost.

Secondly, find all locations in $First$ whose travel time to q is within the earliest close time, meaning that if the path goes to the other points first, the path already misses the location with the earliest close time. After checking its arriving time is in the operating hours, put it into $First$. As a result, $First$ holds all possible locations which can be the first visited.

Thirdly, for any two locations (p_i and p_j), calculate whether $q \to P_i \to P_j$ match i, j's operating hours or not. If not, record $q \to P_i \to P_j$ into $PruneList$. Then generate all permutations of visiting sequence whose first visiting node is in $First$ and do not include any sub path in $PruneList$.

Fourthly, for each candidate path, sum up its cost time and compare the time with time constraint. Once it exceeds the time constraint, ignore it and filter the other path who has the same sub path. E.g. if $q \rightarrow p_1 \rightarrow p_2 \rightarrow p_3$ fails to match time constraint and when query starts, p_1, p_2 and p_3 have opened already, $q \rightarrow p_1 \rightarrow p_5 \rightarrow p_2 \rightarrow p_6 \rightarrow p_3$ should be pruned as well.

Finally, compare the time cost of the paths left and choose the optimum one.

A Case Study To clarify the algorithm, a case study (see in Fig. 4.10) is presented. This case study is based on Example 4.4.1. Table 4.1 shows the $RFix$ filter process. Firstly, $q \rightarrow B \rightarrow A$ will make A over its close time, so B is not in the $First$ list. The same goes for C.

Secondly, according to Proposition 2, $q \rightarrow C_{4:55pm} \rightarrow B_{5:19pm}$ is valid and $q \rightarrow B_{5:18pm} \rightarrow C_{5:42pm}$ cannot meet C close time (5:40pm), so $B \overset{\rightarrow}{\not\leftarrow} C$. Add $q \rightarrow B \rightarrow C$ into $pruneList$.

Thirdly, the same as second step, add $q \rightarrow D \rightarrow B$ into $PruneList$.

Finally, generate the permutation whose first node is A or D (in $First$) and does not contain sub path in $PruneList$. In this case, only one path lists. After checking this path satisfy all time constraints, it is the result of this query ($q \rightarrow A_{4:35pm} \rightarrow C_{5:03pm} \rightarrow B_{5:27pm} \rightarrow D_{6:22pm}$).

Route Search for flexible locations ($RFlex$)

When the user has pre-defined the location types whereby any locations of that type can be visited, this query can be categorized as Route Search for flexible locations, see example 4.4.2. As there are no fixed locations, we should distill the location types first according to the query specification. Then the query can be summarized as finding an optimum path which goes through these types within the time constraint.

Example 4.4.2. *Secretary will leave her office at 4:30pm. She has a plan to do:*
◇ *Deposit a cheque in any bank and all banks' trading hour is 10:00am–5:00pm.*

♢ *Buy a printer in any shop and all shops' operating hours is 11:00am–5:30pm.*

♢ *Post a letter in any post office and posts' trading hour is 10:00am–5:40pm.*

♢ *Buy some medicine in any pharmacy and all pharmacies' trading hour is 10:00am–6:30pm.*

RFlex **Definition** The *RFlex* query can be formally defined like this:

Definition 4.4.3. *RFlex is a route search query consisting of:*
Input: Type Set $T = \{t_1, t_2, \ldots, t_n\}$*, Locations set* $P = \{p_1, p_2, \ldots, p_m\}$*,* $m > n$*.* *Type*$(p_i, \ldots, p_j) = t_k \in T$
Output: A Path l *which goes through ps and ps cover all types in* T*. Also distance*$_l$ *is the shortest.*

Pruning Conditions Traditional Route Search query is an NP complete problem and the focus of this section is how to use time constraint to prune most of expansion branches. Basically, our pruning conditions prune the path which leads to *unreachable point* or which is *out of time.*

Proposition 4. *(Pruning condition 3): If* P *satisfies Equation 7,* P_NN *holds* $P\acute{s}$ *nearest NN of all types in unvisited_type,*P *leads to a unreachable point.* P *will be pruned out candidate_next set. See Algorithm 12.*

$$\left.\begin{array}{l} TimeCost(P, P\_NN(Type_i)) + TimeCost(q, P) > CloseTime(Type_i) \\ Type_i \in unvisited\_type \end{array}\right\} \tag{4.10}$$
$$\Rightarrow Unreachable(P)$$

Proof.

$$\left.\begin{array}{l} CloseTime(Type_i) < TimeCost(P, P\_NN(P_k) \leq TimeCost(P, \forall P_i) \\ P_i \& P_k \in Type_m \in unvisited\_type \end{array}\right\} \tag{4.11}$$
$$\Rightarrow TimeCost(P, \forall P_i) > CloseTime(Type_i) \Rightarrow Unreachable(P)$$

□

Example of Pruning condition 3: Fig. 4.11 is a case study of Example 4.4.2. Suppose p_1 is the *candidate_next* point, according to Proposition 3, whether it

Algorithm 12 Untouch($candidate\_next$,$visited\_type$,T,$TimeCost$)

for each p in $candidate\_next$ **do**
  p_type=$get\_Location\_type()$
  $visited\_type = visited\_type + P_type$
  $unvisited\_type = unvisited\_type - P_type$
  Initial $boolean$=0
  Find pś nearest NN of all types in $unvisited\_type$ and put in p_NN
  $p\_TimeCost$=$get\_POI\_TimeCost()$
  $TimeCost$=$TimeCost$+$p\_TimeCost$
  for each NN in p_NN **do**
    NN_type=$get\_POI\_type(NN)$
    $NN\_TimeCost$=$get\_Location\_TimeCost(NN)$
    if $NN\_TimeCost + TimeCost > NN\_type$'s close time **then**
      $boolean$=1
      Return
    end if
  end for
  if $boolean$=1 **then**
    Delete this p from $candidate\_next$
  end if
end for

should be pruned or not depends on the data in Table 4.2. p_i is the nearest neighbor of p_1 in $Type_i$. As p_1's type is T_1, then $unvisited\_type = T_2, T_3, T_4, T_5$. Find p_1's NN for each type in $unvisited\_type$ (Column 1 in Table 2). We can easily tell that p_2 is out of $OperatingHours$ of $Type_2$, in other words, p_1 leads unvisited type T_2 unreachable. Consequently p_1 cannot be the $candidate\_next$ point.

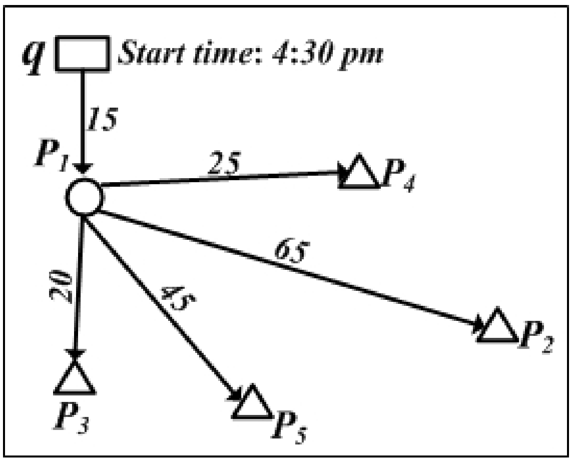


Figure 4.11: Pruning Cond. 3

p_1 NN	Type	Oper.Time	Arriving Time
p_2	T_2	5:00pm	$StartT$+15+65=5:20pm
p_3	T_3	5:30pm	$StartT$+15+45=5:00pm
p_4	T_4	5:40pm	$StartT$+15+25=4:40pm
p_5	T_5	6:30pm	$StartT$+15+20=4:35pm

Table 4.2: Proposition 3 Demo for $RFlex$

Proposition 5. *(Pruning condition 4): Locations whose types are in* $unvisited\_Type$ *within* $TimeCon$ *(see equation 9) can be in* $candidate\_next$*. Equation 10 which collects the* $candidate\_next$ *set can prune lots of interest points.*

$$StartTime > \max_{\forall i \in n}(OperatingHours(P_i)) \Rightarrow TimeCon = \max(CloseTime(unvisited\_Type)) \tag{4.12}$$

$$candidate\_next(P) = \begin{cases} \forall P_i \\ TimeCost(P, P_i) < TimeCon \\ Type(P_i) \in unvisited\_Type \end{cases} \tag{4.13}$$

Proof. Suppose when we start the query, all locations are open, $TimeCon$ should be set as the earliest close time in $unvisited\_Type$ as if the earliest close time type has not been visited, the locations which are going to be visited must be finished ahead of the earliest close time, otherwise when expanding to the earliest close time type, the arriving time is already out of the time constraint. □

Algorithm 13 CandidateNext(p,$visited\_type$,T,$TimeCost$)

1: $unvisited\_type = T - visited\_type$
2: $TimeCon$ = Earliest $CloseTime$ of $unvisited\_type$
3: $Candidate\_next$ = all points whose type in $unvisited\_type$ and travel time within $TimeCon$

Example of Pruning condition 4:Referring to Fig. 4.12 and Table 3, suppose p_3 of T_3 is the current expansion point. In Table 4.2, each line represents one scenario as the $unvisited\_Type$ is different. Take the first line as an example, if the only visited type is T_3, then the $unvisited\_Type = \{T_1, T_2, T_4, T_5\}$ and $TimeCon = T_1$'s

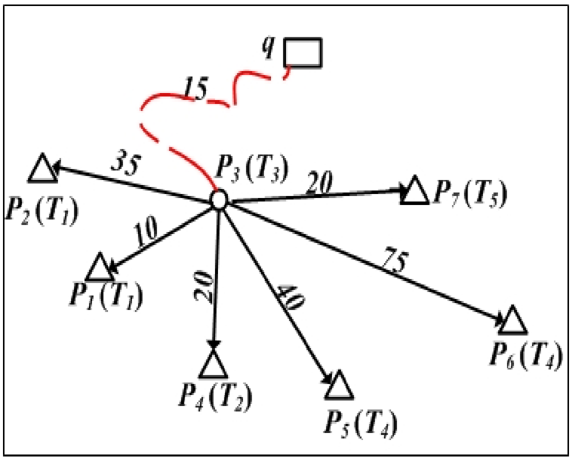


Figure 4.12: Pruning Cond. 4

close time = 4:30pm. Only p_1 can be the *candidate_next* point because all of the other points are out of $TimeCon$ according to Proposition 4. Algorithm 13 shows the process of how to find *candidate_next* points which satisfies pruning conditions 4.

Visit_Type	Unvisit_Type	$TimeCon$	Cand_Next
T_3	T_1, T_2, T_4, T_5	5:00pm	p_1
T_1, T_3	T_2, T_4, T_5	5:30pm	p_4, p_5, p_7
T_2, T_3	T_1, T_4, T_5	5:00pm	p_1
T_1, T_2, T_3	T_4, T_5	5:40pm	p_5, p_7

Table 4.3: Proposition 4 Demo for $RFlex$

With these two pruning conditions, most expansions branches have been pruned. Although the execution time could potentially be exponential in the worst case because the pruning strategy is only heuristic, the pruning conditions do improve the performance significantly.

The detailed steps are shown as follows.

Firstly, initialize T_{max} as ∞ and it is the boundary identifier which will hold the final result. Initialize the *visited_type* as $\emptyset$ which is the collection of the location types that have been visited. Also, initialize T as all location types of user interest.

Algorithm 14 RFlex(q,$StartTime$,T,$LocationTtable$)

```
if q= start point then
    Load routine traffic speed in current period
    Initial T_max=∞
    visited_type = ∅
    unvisited_type = T
    Static TotalTimeCost=0
    TimeCost=0
end if
CandidateNext(q,visited_type,T,TimeCost)
Untouch(candidate_next,visited_type,T,p_TimeCost,TimeCost)
for each p in candidate_next do
    TotalTimeCost = TotalTimeCost+TimeCost
    StartTime = StartTime+TimeCost
    if visited_type = T then
        if TimeCost ≤ T_max then
            Update T_max=TotalTimeCost and record its path tree
        end if
        Break
    else
        RFlex(p,StartTime,T − visited_type,LocationTtable)
    end if
end for
Final T_max is Time Cost and its path is optimum path
```

Secondly, according to the current time period, load routine traffic speed and get all interest points around q whose travel time to q is within $TimeCon$. Collect them in a set call *candidate_next* and prune these points using pruning condition 3.

Thirdly, for any point in *candidate_next*, do the following. Remove any p in *candidate_next*, add p's type into *visited_type*. Then repeat the second step until all types have been visited. Once this path's cost time is shorter than T_{max}, replace T_{max} with its cost time.

Finally, T_{max} is the shortest travel time and its path is the optimum path.

The algorithm of $RFlex$ is shown in Algorithm 14.

4.5 Performance Evaluation

In this section, we evaluate these three methods using different data and environment setting.

4.5.1 Path based kNN search

In the experimentations, different simulation data (stored as several tables) are chosen to represents high (45 interest points), medium (20 interest points) and low density of interest points (10 interest points) respectively. In addition, a netlike map is created to represent more looping map (136 links), a general map with couples of loops (127 links) and an emanative map with few loops (101 links) are chosen to represent the medium and less looping maps in our performance evaluation.

Moreover, we examined pkNN for different values of k. All interest points, network links and intersect nodes are simulated data. The experiments were performed on a Mac with Intel Core 2 Duo processors, 2GB of RAM, and MS Access as our database. We analyze the behavior of our approach in the aspects such as expansion steps and run time with different densities of interest points and the values of k. The looping in the map has also been evaluated in expansion steps and run time

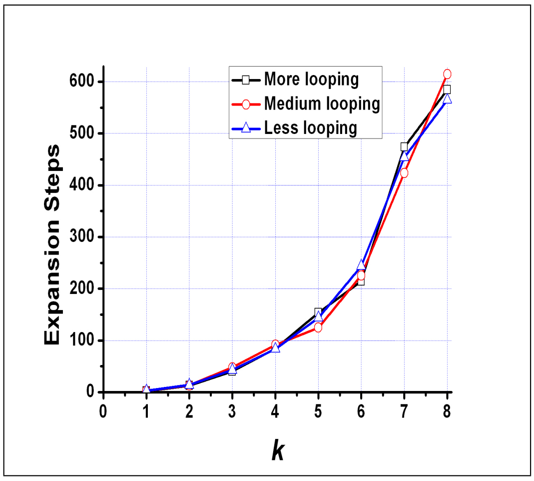


Figure 4.13: Expansion steps for different loops in maps

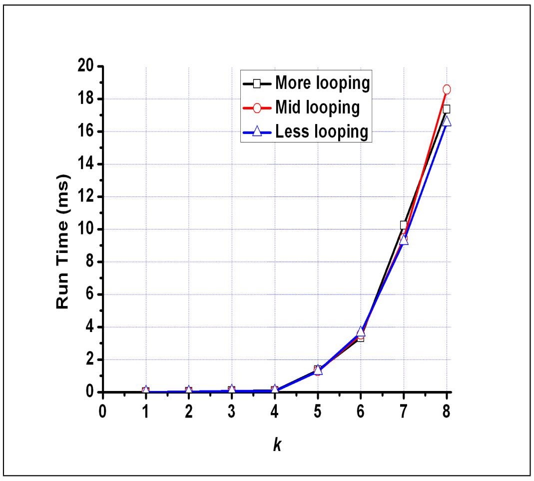


Figure 4.14: Runtime for different loops in maps

using different values of k. We also compare the performance of our approach in expansion steps and run time with pruning and without pruning.

Looping map

In this section, we aim at finding the differences of expansion steps and runtime if less or more loops are involved in the searching map. A simulate map like Melbourne city is chosen as more looping map because it is a netlike map and in the expansions, more looping scenarios accrue there. A simulate map like Malvern suburb is chosen as normal which consists of couples of loops. A simulate map like Peninsula map is chosen as less looping map because it is emanative map. Also 50 different query positions which were generated randomly are tested to get the average expansion steps and runtime based on k which from 1 to 8. Less looping maps will give us a better performance because less expansion will be acted and runtime will be less.

Fig.4.13 and Fig.4.14 show the trend of the expansion steps and runtime if k increases and gives the comparison among less looping map, medium looping map and more looping map.

From Fig.4.13 and Fig.4.14, we can draw a conclusion that the expansion steps and runtime go up with the increasing value of k. This can be easily understood as more k requires more computations. At the same time, we found there is no

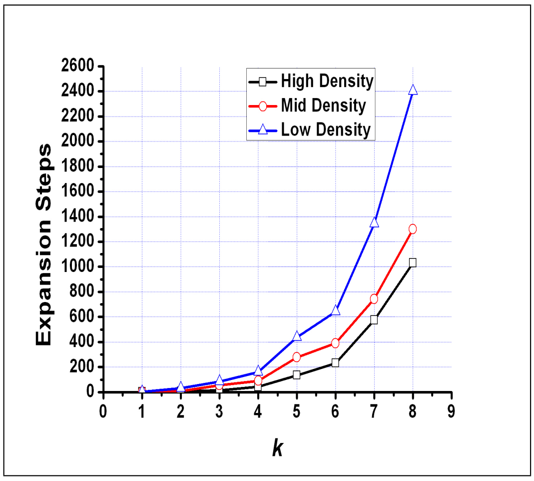


Figure 4.15: Expansion steps of different POI densities

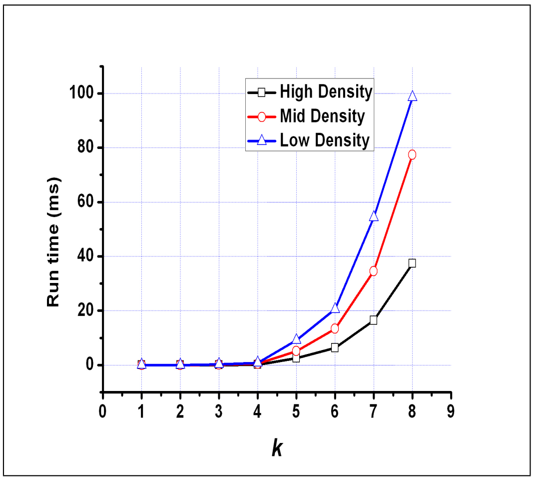


Figure 4.16: Runtime of different POI densities

significant difference in expansion steps and runtime under different conditions of loops. In other words, the performance of pkNN performs at the same scale in a netlike map or an emanative map. This may surprise us because according to common sense, more loops mean more redundant cost. This outcome is due to 4 pruning conditions proposed by us.

Density of interest points

In this section, we aim at finding the difference of expansion steps and runtime with various densities of interest points. The performance is evaluated based on the different interest points distributions, such as high density distributed objects (42 interest points), medium density distributed objects (20 interest points), and low density distributed objects (9 interest points). In addition, 50 different query positions are tested to get the average performance result based on different values of k. It is known that for the same map, if the interest point density is low, the runtime will increase because more time will be a cost to do expansion. Also if k increases, the runtime will increase because more time will be spend on trying to find more interest points. Fig.4.15 and Fig.4.16 demonstrate the trend of each scenario.

From Fig.4.15 and Fig.4.16, we can conclude that with the increment of k, pkNN performs exponential growth in expansion steps as well as runtime. Another factor

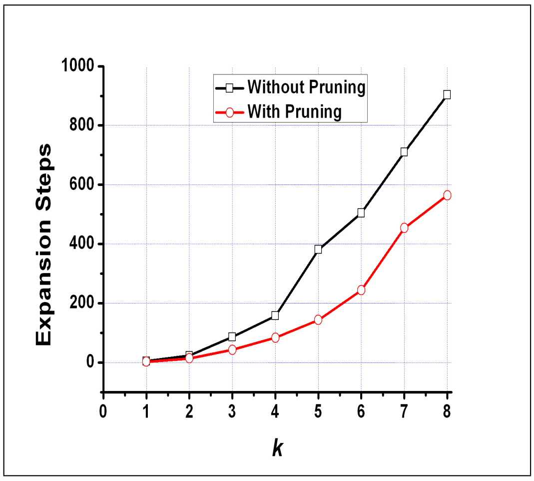


Figure 4.17: Expansion steps with or without Pruning conditions

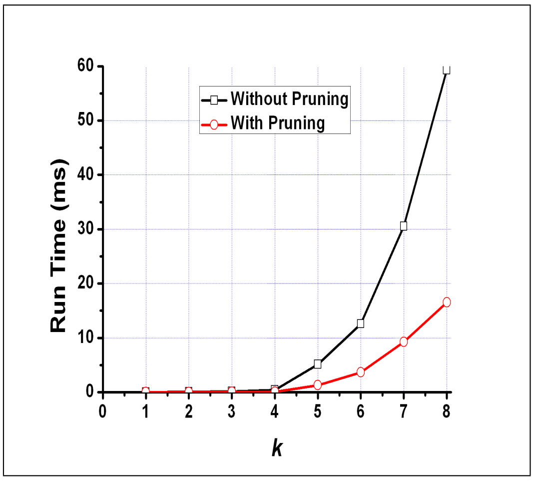


Figure 4.18: Runtime with or without Pruning conditions

which effects expansion steps and runtime is the density of interest points. The higher the density is, the lower cost in expansion and runtime, in other words, the better performance. The performance result coincides with our prediction because the more k required, the more operations are in the approach. Also the lower density distributed of the interest points, the more cost to find the next node on the route.

Pruning Conditions

The pruning conditions are the bright spots of our pkNN approach. Instead of computing all possible permutations, full expansion is inducted. In addition, in the process of expansion, the expansion history is stored into a list in order to optimize the expansion as well as accelerate the processing efficiency. Fig.4.17 and Fig.4.18 show the expansion steps and runtime improvements respectively between algorithms without and with pruning conditions.

Fig.4.17 and Fig.4.18 illustrate how pruning conditions improves the performance. With the increasing values of k, the expansion steps and runtime go up in linear growth instead of exponential growth. With the pruning condition, it is possible to implement pkNN with larger values of k. To sum up, pruning conditions not only improve the performance, but also enhance the feasibility of our pkNN algorithm.

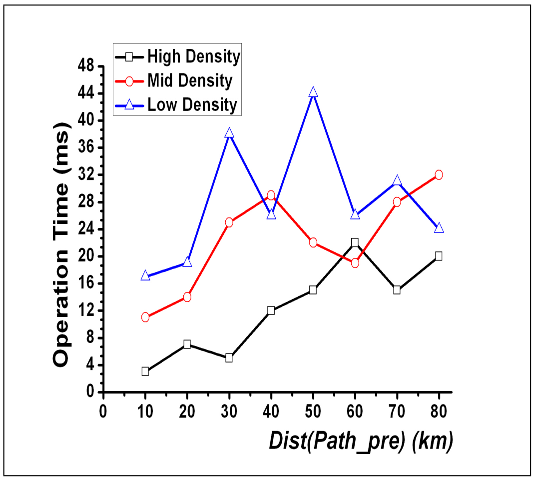


Figure 4.19: Operation time of different POI densities

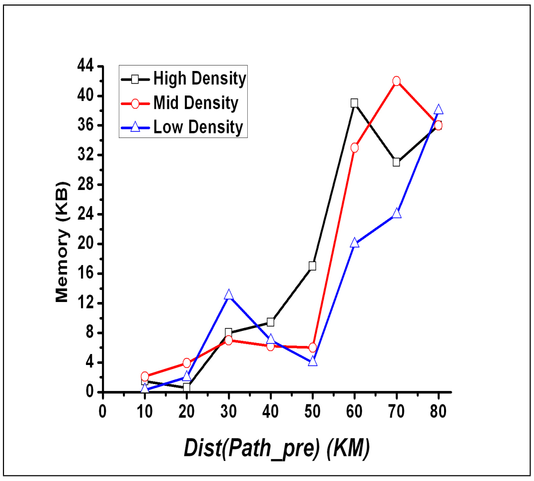


Figure 4.20: Memory size of different POI densities

4.5.2 Path Branch Point based kNN Search Queries

In these experiments, Sydney city map, Canberra city map and Hobart city map from the Whereis website (www. whereis.com) were chosen to represent high, medium and low density of interest points. All interest points, network links and intersect nodes are real-world data. We analyze the behavior of our approach in aspects such as operation time, memory size and AS values with different densities of interest points and the length of $Path_{pre}$.

Interest Point distribution density

In this section, we aim to find the differences in runtime, memory size and AS values between low, medium and high density interest points. We use restaurants in Sydney to represent a high density sample, parks in Canberra represent a medium density sample and hospitals in Hobart to represent a low density sample. Also we test 20 different query positions to obtain the average runtime, memory size and AS values based on the distance of the pre-defined path from 10 to 80.

From Fig.4.19, the conclusion can be drawn that the run time will increase if the density decreases. This is because the lower the density, the less chance that the interest point is close to the path. As a result, lower density will cause more

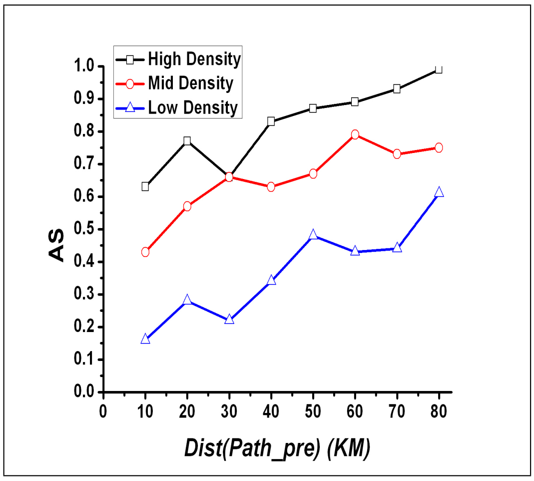


Figure 4.21: *AS* values of different POI densities

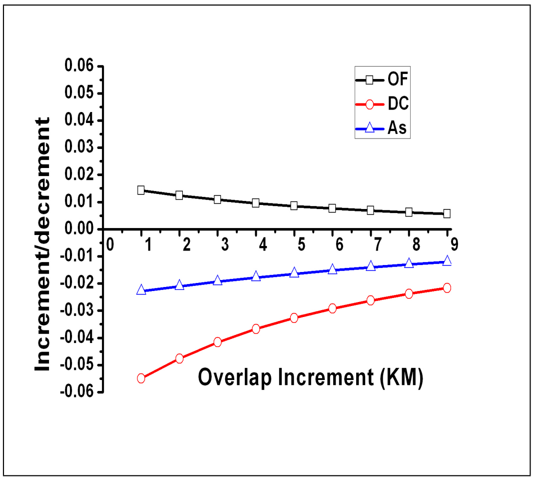


Figure 4.22: Factor change based on different overlap increment-AS all negative

comparisons which will cause the delay of the operation. Also, we can see that there is no fixed relation between run time and the length of the pre-defined path.

From Fig.4.20, the conclusion can be drawn that the memory size will increase when the length of the pre-defined path increases, at the same time, the density of interest points and memory size seem to be unrelated according to our experiments. This is because the shorter the pre-defined path, the shorter the boundary distance. In other word, there is only a small chance that this will be the optimal path. At the same time, most interest points are pruned by boundary distance, which saves memory size.

From Fig.4.21, the conclusion can be drawn that the *AS* value is closer to 1 when either the density of interest points or the length of the pre-defined path increases. Both these two factors increase the possibility that there are some interest points on the path or around the path. In other words, with an increase in the density of the interest points and the length of the pre-defined path, the greater the possibility that the path is more suitable to the user.

Factor Increment/decrement

In this section, we aim to find the factor's increment or decrement when the path becomes longer in order to have more overlapping distance. From Figs.4.22, 4.23 and

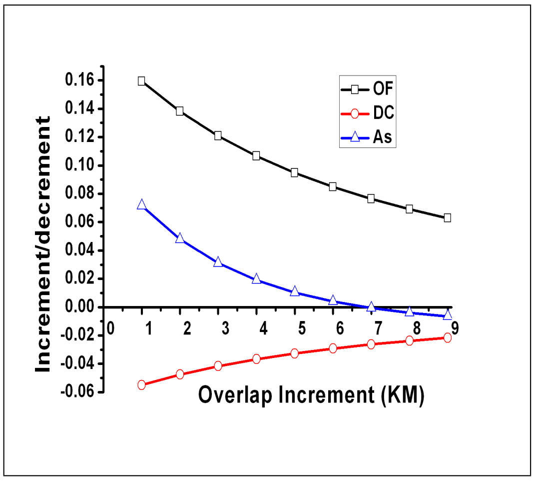


Figure 4.23: Factor change based on different overlap increment-AS partial negative and partial positive

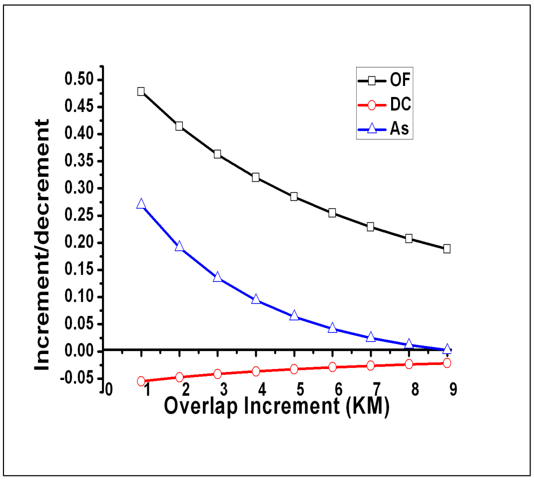


Figure 4.24: Factor change based on different overlap increment-AS all positive

4.24, we can tell that the DC becomes increasingly smaller because the increment is always negative, while the OF becomes increasingly larger because the increment is already negative. After the calculation, we can see that there are three possible results: the AS increases, the AS decreases or the AS increases then decreases.

Fig.4.22 shows an example where the AS decreases. In this case, the shortest path is the one which has chance to be optimal, after the increment in the driving distance, although it may take overlap in return, but the overlap increment cannot increase the AS.

Fig.4.23 shows an example where the AS increases then decreases. The turning point is where the optimal path is and this shows that the shortest path is not always the optimal path.

Fig.4.24 shows an example where the AS increases. In this case, there is a chance that by increasing the driving distance to increase the section of the path that overlaps, the path becomes better than the shortest path.

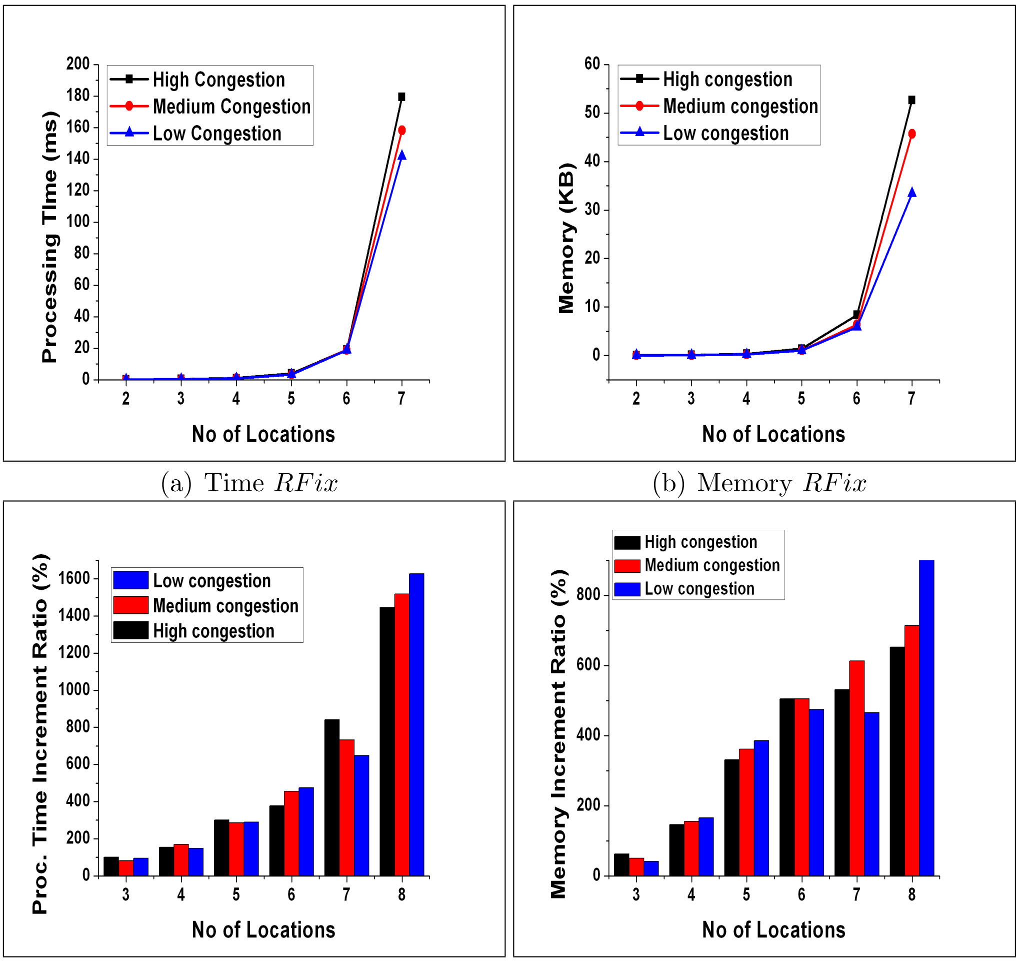
4.5.3 Time Constraint Route Search over Multiple Locations

We used network and interest points data in Los Angeles in our experiments. We extracted 8 different types of interest points to simulate different location types, including 15 parks, 29 coffee lounges, 31 bank branches, 54 hotels, 78 post offices, 158 pharmacies, 283 shops and 597 restaurants and all interest points are normally distributed. In our experiments, we varied the following parameters: the number of location types, the congestion level (speed), density of interest points and the average time interval between locations to observe their effects on average processing time, memory as well as their improvement compared with the exhaustive traversal of all permutation approach.

Experimental Results of *RFix*

Since number of locations highly influences our method's performance, we test the processing time (Fig. 4.25(a)) and memory (Fig. 4.25(b)) for 2 to 8 locations based on 3 different traffic status (low, medium and high congestion). From Fig. 4.25(a) and Fig. 4.25(b), we can easily tell that with the increasing number of locations, the processing time and memory are directly proportional to the number of locations. In addition, when the congestion level increases from low to high, the speed decreases at the same time and it causes a slight increase of the processing time and memory. When it is extremely congested, the processing time and memory will cost approximately 8% to 14% more than the lower level. While adding one more location into query list, there will be exponential growth in processing time and logarithmic growth in memory size required, referring to Fig. 4.25(c) and Fig. 4.25(d) when location number is greater 8, the performance scale increases sharply.

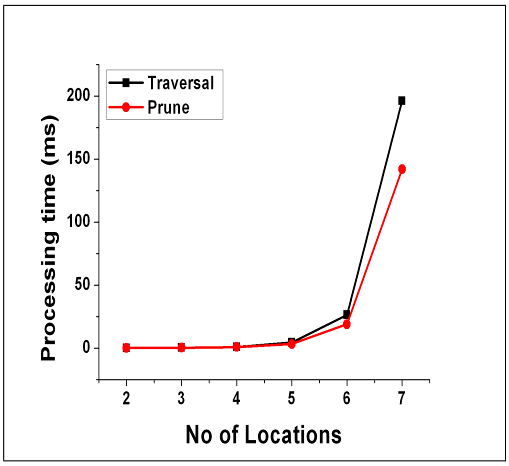
Without using our method, *Route Search for fixed locations* query can be solved by traversing all permutations at the cost of processing time and memory size. To improve its performance, we have proposed two pruning conditions in this chapter. Fig. 4.26(a) and Fig. 4.26(b) give the performance comparison in processing time



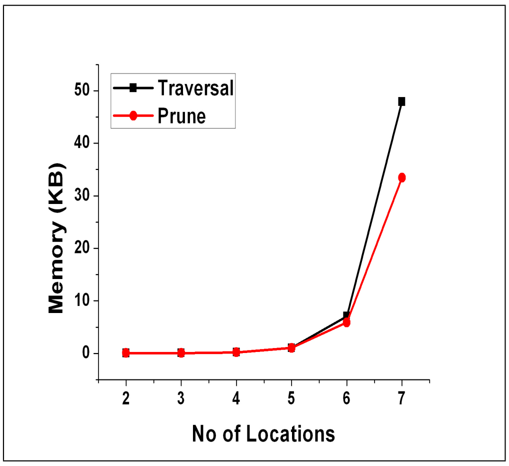
(a) Time *RFix* (b) Memory *RFix*

(c) Time Incr-Ratio (d) Mem. Incr-Ratio

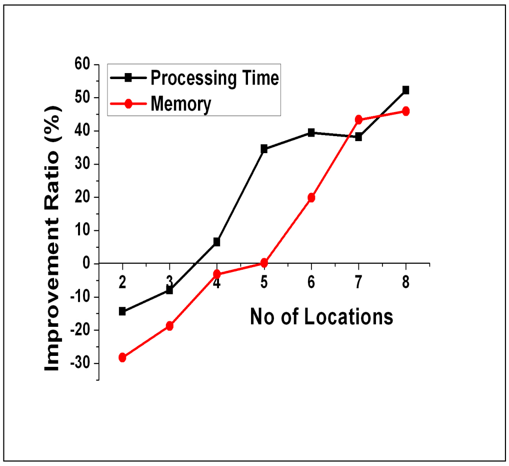
Figure 4.25: Time and memory comparison between different number of locations and traffic status in *RFix* and Time and memory incremental ratio when adding more locations



(a) Proc. time-Prun. cond.

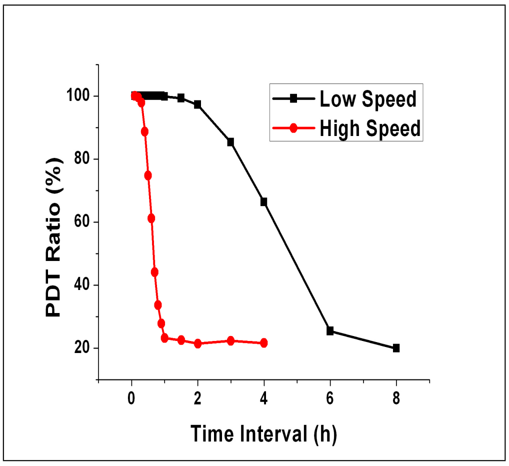


(b) Memory-Prun. cond.

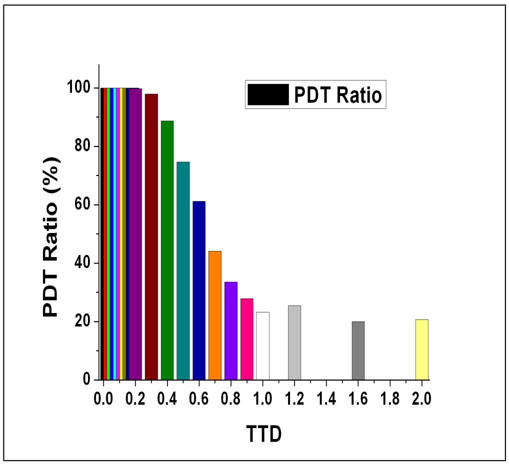


(c) Time&Memory Improv.

Figure 4.26: Proc. time and memory comparison between $RFix$ and traversal methods



(a) PDT-speed



(b) PDT-TTD

Figure 4.27: PDT is optimum ($RFix$) ratio

and memory between with and without pruning conditions. Fig. 4.26(c) highlights the advantages of our pruning conditions.

To sum up, with the increasing number of locations, our methods with pruning conditions outperform the traversal methods in both processing time and memory aspects especially when the location number is greater than 3.

In section 4.4, we include time constraint into Route Search query, normally the user will follow rules: *earliest close, first visit*. In other words, the path is sorted by the close time sequence and this path is abbreviated as PDT. PDT ratio is the possibility that PDT is the optimum path (Equation 11). In this section, we analyze factors that will affect the visiting path, see Fig. 4.27(a) and Fig. 4.27(b).

$$PDT\ Ratio = \frac{number\ of\ times(PDT = optimum\ path)}{n\ times\ experiments} \tag{4.14}$$

Before analyzing our experiment results, first we define a factor called *Travel distance span in average Time interval to objects Distribution region* (TTD). Object Distribution Region (ODR) is the size of the region that user can arrive within the last location close time. TTD represents the coverage percentage of ODR in average time interval between locations.

$$ODR = \pi((\max CloseTime_i - StartTime) * \bar{t})^2 \tag{4.15}$$

$$TTD = \frac{(\sum_{i=1}^{n}(CloseTime_n - CloseTime_{n-1})/n) * \bar{t}}{ODR} \tag{4.16}$$

Fig. 4.27(a) shows that low travel speed will lead to a high possibility that an optimum path is PDT until the average interval increases to 2 hours or more, while high travel speed leads more possibility that optimum path is different from PDT when the average interval is greater than 0.4 hour. This result is in conformity with common sense as if the average time interval between locations is small and speed is relatively slow, visiting locations along the close time sequence has a higher possibility to meet the time constraint because if we visit the later close time location, there will be a high possibility that we cannot catch the earlier location, and vice versa. Fig. 4.27(b) illustrates that if TTD increases, which means its coverage percentage in average time interval increases, the possibility that PDT is the optimum path drops. The percentage remains stable until TTD increases to around 1.

Experimental Results of $RFlex$

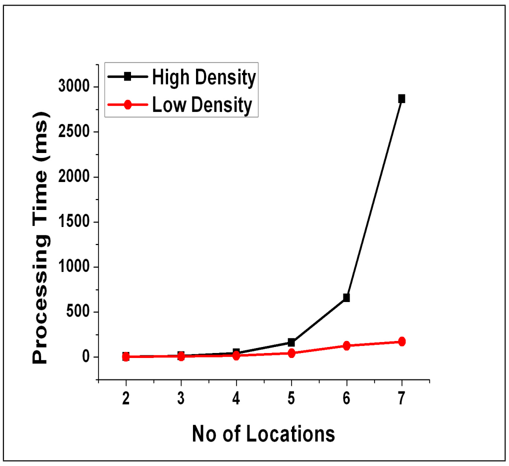
For *Route Search for flexible locations* query, the average time interval between locations is a factor which affects the visiting sequence. People generally believe that if the average time interval between locations is large, the performance falls badly. While Fig. 4.29(a) and Fig. 4.29(b) prove that this conjecture is not correct because our $RFlex$ goes down to get the first path and this path is set as boundary. As a result, the processing time is nearly linear and the memory decreases a little with the increasing average time interval between locations. As a result, our approach performs well even if there is no time constraint when the number of locations remains constant.

The processing time and memory will steeply increase with the increasing number of locations and this is already proven in $RFix$. In this section, we compare the differences between high density distributed objects (e.g. restaurant, density = 0.09375) and low density distributed objects (e.g. parks, density = 0.0024). In our experiments we test the processing time and memory for locations numbers from 2 to 7 refer to Fig. 4.28(a) and Fig. 4.28(b) for low and high densities of locations. Fig. 4.28(c) and Fig. 4.28(d) show that if Route Search query involves more low density objects, with the increasing number of locations, the processing time and memory do increase, but much slower than the high density objects.

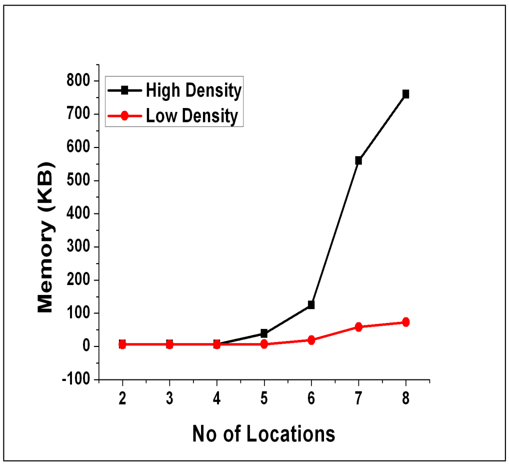
4.6 Conclusion

This chapter discusses the route and path related kNN queries. The motivation is to bring route/path into the input or output of spatial queries.

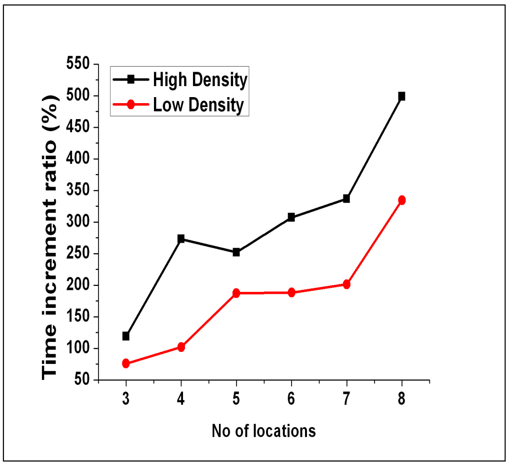
In this chapter, a novel approach called path based k nearest neighbor based on network distance on road network is been introduced firstly. The basis of pkNN is network expansion. The proposed approach, pkNN, gives users correct paths, even



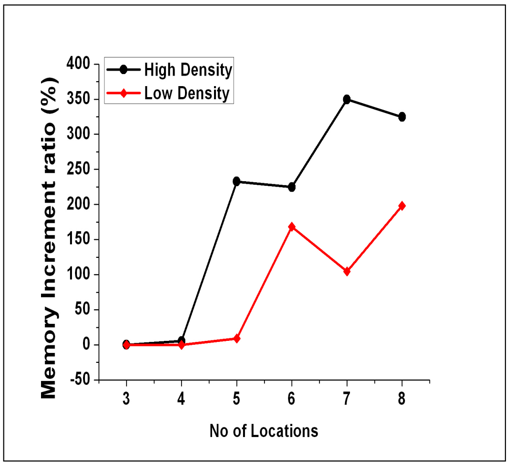
(a) Time-Density



(b) Time-Density

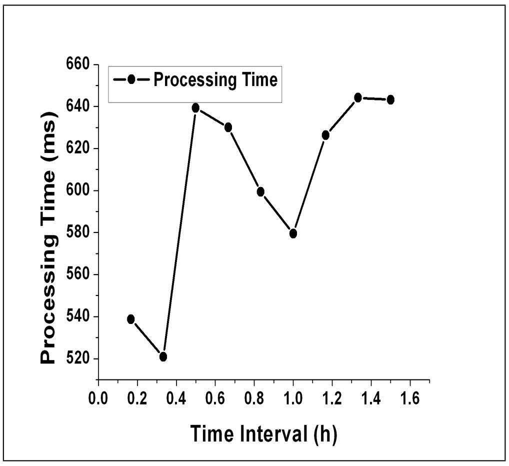


(c) Time-Density

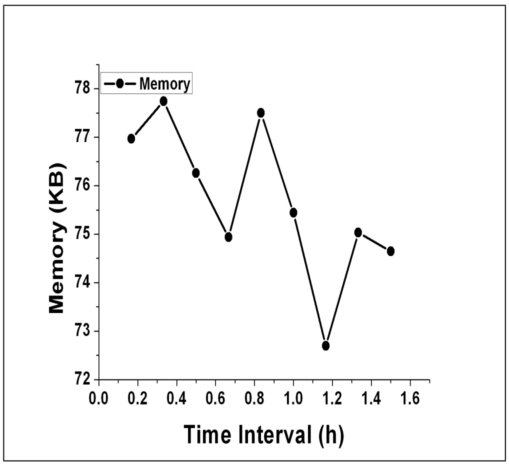


(d) Time-Density

Figure 4.28: Proc. time and memory comparison for different object densities in *RFlex* and Proc. time and memory incremental ratio when adding more locations



(a) Proc. Time



(b) Memory

Figure 4.29: *RFlex* in different time intervals

when the route is complex like that in real world. We have also taken care of complex circumstances involving local minimum, loops, and U-Turns. We performed several experiments to measure the performance of pkNN in various network conditions. In general, our algorithms performs well if the density is high and the number of interest objects is smaller than 7. However, as expected, if the density of the interest objects is low and number of interest objects is large, the performance of pkNN will degrade sharply. On average, if k is given, lower density of interest points will let the runtime and expansion step increase. If k is large, the runtime and expansion step will increase sharply. Furthermore, there is no significant increase if more loops are involved on the underlying map because pruning conditions are proposed in this chapter. With pruning conditions, even the expansion steps and runtime will increase with the increasing values of k, it makes the increase in linear growth instead of exponential growth.

Secondly, we defined the path branch path query and proposed an approach which can scale the path into fit or unfit user requirement categories. We aim at using a mathematical formula to quantify the percentage of fitness to the user's requirement. In this chapter, we examine several special scenarios such as an interest point on the pre-defined path, no overlap with the pre-defined path, and a scenario where distance cost is negligible. We performed several experiments to measure the performance of PBP in different interest point distributions. In general, our algorithms perform well if the density is high. However, as expected, if the density of the interest objects is low and the pre-defined path is too short, there will be a high possibility that the path is not optimal because it is likely that there is no interest point around the path and driving path to it will not have any overlap with the pre-defined path. The more interest points around the path, the longer the pre-defined path, hence the more optimal the path.

The third part of this chapter proposes novel Route Search methods with time constraint involving multiple object types. Route Search for fixed locations provides a solution to users if they want to find the shortest travel time path for multiple location types and the locations of these types are fixed. Route Search for flexible locations helps users to find the shortest travel time path if the locations of these types are flexible. Both queries do not concern visiting sequence of objects subject to the final path as long as they arrive at each location within its operating hours. In our method, the network Voronoi Diagram is used to find the candidate next visiting point within certain time range and it enriches the content of our mobile navigation system and gives more benefit to mobile users as well. We performed several experiments to measure the performance of $RFix$ and $RFlex$ in different network conditions and object distributions. In general, our algorithm performs better if the number of locations is small. If the number of locations is smaller than 7, the performance is acceptable no matter how complex the road condition is and how objects are distributed. However, as expected, if the number of locations is greater than 7, the processing time and memory increase sharply. In addition, if the average location close time interval is large, our optimum path has a high possibility that it is not PDT which means discarding PDT and using our methods can give users a better path choice. Lastly when comparing $RFix$ with the traditional traversal permutation method, it performs better and the advantage becomes obvious when the number of locations increases up to 4.

To sum up, all approaches bring route into input or/and output of spatial queries which highly enrich the type of spatial queries contents. These approaches have been proven that they can solve their corresponding queries efficiently.

Chapter 5

Conclusion

5.1 Contributions

In this thesis, we presented efficient techniques to deal with Voronoi Diagram based k Nearest Neighbor Search queries and route based k nearest neighbor queries search under different settings. Chapter 3 presents some variants of k nearest neighbor search respectively in section 3.2, section 3.3. Chapter 4 presents the novel category of k nearest neighbor search which is based on route/path. Three different approaches are discuss respectively in section 4.2, section 4.3, section 4.4. At last, section 5 summarizes this thesis as well as pointing out the future work.

- **Contribution 1:** Chapter 3 does the optimization by utilizing Voronoi Diagram to merge the road segments into polygons in order to replace the Network Expansion. Two approaches are proposed in the chapter, summarized as follows:

  In section 3.2, we proposed an alternative approach for Continuous k Nearest Neighbor query processing, which is based on Network Voronoi Diagram (we call our proposed method VCkNN, for Voronoi CkNN). This approach avoids the weakness of existing work [GR03, GR99] and improves the performance by utilizing the Voronoi diagram. VCkNN ignores intersections on the query path; instead, it uses Voronoi polygons to subdivide the path. Our proposed

VCkNN approach is based on the attributes of the Voronoi diagram itself and using a piecewise continuous function to express the distance change of each border point. Our experiment verified the applicability of VCkNN approach to solve CkNN queries and demonstrated that it outperforms existing algorithms.

Section.3.3 presents new approaches on three different queries involving multiple object types using a network Voronoi Diagram, including: a) query to find nearest neighbor for multiple types of interest point (or 1NN for each object type), b) query to give the shortest path to cover multiple-object-types in a pre-defined sequence, and c) query to find an optimum path for multiple object types that gives the shortest path that covers the required interest objects in a random sequence. In these queries, more than one object type is considered and the query result is highly related with the object types. Every object belongs to one of the category and there is no overlap between categories. That is the basic property of *multiple-object-type query.* Our experiment verified the applicability of our approach to solve k nearest neighbor over a multiple type of objects.

- **Contribution 2:** Chapter 4 opens up new route search queries which are able to bring path into the input or/and output of spatial queries. The tradition spatial queries use discrete points as input and output. So this chapter is mainly doing exploring a new area. Three approaches are proposed in the chapter, summarized as follows:

  Section 4.2 investigates a novel route based k nearest neighbor query which is called Path based kNN Search Query. Path bases k nearest neighbor search is to find the shortest path which goes through k objects. In general, the overall distance of the path becomes the selection criteria. We propose an efficient algorithm and present several pruning conditions to do optimization that significantly reduces the overall computation cost and processing time. The efficiency of our proposed approach is demonstrated using real data sets and simulated data sets.

Section 4.3 brings a novel query which is called path branch point(PBP). PBP can be defined as: given a set of candidate interest objects and a pre-defined path which starts at S and end at E, find a path which starts at S, via an interest point P and ends at E. This path should overlap with the pre-defined path as much as possible with acceptable distance increment. This is a novel query which is motivated by users' common requirements because most users have ad hoc paths in their daily travel and they can tolerate a longer driving distance to some extent if they can drive on a familiar path when they want to visit a certain type of objects on the way. In this proposed approach, an Adjust Score is calculated for each path which is determined by overlapping distance and increased distance cost.

Section 4.4 introduces time constraint into route search over multiple locations. Each spatial business entity has its own valid time which implies the time constraint of the route. Moreover, instead of finding the **shortest** path, the aim is to find the path with shortest time cost. Meanwhile, all spatial entities are visited within their valid time frame. The query definitions are clearly stated at first, followed by two types of query scenarios: route search over objects with flexible locations and route search over objects with fixed locations. Extensive experiments demonstrate the efficiency of our proposed algorithms.

To sum up, the main contribution of the thesis can be summarized as the following two aspects in general: 1) optimization using Voronoi Diagram 2) exploring new area in spatial query search which brings route into it.

5.2 Open Problems and Future Work

Spatial query processing has been studied over decades, but we still see that possible extensions can be made in the future.

Firstly, a novel route based k nearest neighbor query is proposed which is called Path based kNN Search Query. Our approach has been proven that it can give the user precise results. But when the k increases over 10, the efficiency of this approach drops dramatically. One open problem leaves to us is how to improve the efficiency with small preciseness sacrifice.

Secondly, we bring a novel query which is called path branch point(PBP). In this chapter, I defined a function to judge the length increment cost as well as the length overlap percentage. There might be the other proposals on defining the concepts. With the different cost modeling, the query will produce different routes as result because some users might want to sacrifice the driving distance but prefer the familiar road, while other users might lay emphasis on shortest distance and are tolerant of unfamiliar paths.

Thirdly, time constraint is introduced into route search over multiple locations. We present two different processing procedures when the user wants to visit flexible or fixed locations. Another open problem which can be investigated in future work might be the various visiting sequences assigned by users. In other words, the visiting sequence of the objects can be sequential, random or partial sequential and random. The objects location can be flexible, fixed or partial flexible and fixed.

To sum up, there are still a few open problems for us to investigate in the future work. Our algorithm performs well under some settings. We may introduce more parameters into the spatial queries. By solving these open problems, the queries can assist user's lives. The following are several possible directions for future work.

- **Query processing in P2P and Ad-hoc networks:** In this thesis, all algorithms and queries are based on 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 the most of circumstances, we conjecture that Voronoi based algorithms in P2P or Ad-hoc networks will outperform existing techniques as well. We also would like to investigate whether it is possible to let each peer to 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 to cooperate to manage the relative positions in the P2P or Ad-hoc networks.

- **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 target ever in the past few years [STX08, XSP09]. It is an interesting problem to investigate how the higher dimensional Voronoi diagram or other geometric theories would help to improve the performance of spatial query processing.

- **Join up spatial query results**: All the existing spatial queries are univocal. But some of the user queries might contain more than one query types, e.g. joining k nearest neighbor search with range search. In the future, we may find that the joining result of two categories might represent users's special requirements.

- **Incorporate some intelligent features in mobile navigation** In the future, it can be expected that mobile navigation incorporates some intelligent features, such as extracting movement patterns of mobile users [GT04b], and these can be adopted for mobile navigation. There have also been some successful works in incorporating Voronoi diagrams and network Voronoi diagrams in mobile navigation [XZTS08] as well as the use of ontology in query expansion [WST03,WST04]. Further, there are two important issues in mobile navigation, scalability and performance. Data broadcasting has been known to be able to address the scalability issues and indexing can be used to speed up performance. Further investigations on these two issues in mobile navigation can be useful [TR04, TR02] and the k nearest neighbor algorithm can be broadly investigated in various area of research, such as digital ecosystems [MP07, LZL08, BX11, CLLP11].

Appendix A

Simulation Source Code

A.1 kNN Implementation

Point.Java

```java
package algorithm;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public final class Algorithm {
    public Algorithm(final List<Point> all, final int maxK) {
        this.all = all;
        this.maxK = maxK;
        this.nearest = new HashMap<Point, List<Point>>();
        this.findNearestAtMaxK();
    }

    @SuppressWarnings({ "null", "boxing" })
    public Map<Integer, List<Cluster>> findClusters() {
        final Map<Integer, List<Cluster>> clusters = new HashMap<Integer, List<Cluster
            >>();
```

```

        clusters.put(0, new ArrayList<Cluster>());
        for (final Point p : this.all) {
            clusters.get(0).add(new Cluster(p));
        }
        for (int i = 1; i <= this.maxK; ++i) {
            final Map<Point, List<Point>> pairs = this.findPairs(i);
            final List<Cluster> previousClusters = clusters.get(i - 1);
            for (final Cluster c : previousClusters) {
                if (c.isMerged()) {
                    continue;
                }
                final Cluster mergedCluster = new Cluster();
                final List<Cluster> mergingCluster = new ArrayList<Cluster>();
                mergedCluster.getPoints().addAll(c.getPoints());
                c.setMerged(true);
                int j = 0;
                double mergingMinStableFactor = c.getStableFactor();
                while (mergedCluster.getPoints().size() > j) {
                    final Point p = mergedCluster.getPoints().get(j);
                    if (pairs.containsKey(p)) {
                        for (final Point p2 : pairs.get(p)) {
                            if (!c.getPoints().contains(p2)) {
                                Cluster p2Cluster = null;
                                for (final Cluster whereIsP2 : previousClusters) {
                                    if (whereIsP2.getPoints().contains(p2)) {
                                        p2Cluster = whereIsP2;
                                        break;
                                    }
                                }
                                for (final Point p2Point : p2Cluster.getPoints()) {
                                    if (!mergedCluster.getPoints().contains(p2Point)) {
                                        mergedCluster.getPoints().add(p2Point);
                                    }
                                }
```

```
                mergingCluster.add(p2Cluster);
                p2Cluster.setMerged(true);
                if (p2Cluster.getStableFactor() < mergingMinStableFactor) {
                    mergingMinStableFactor = p2Cluster.getStableFactor();
                }
            }
        }
    }
    ++j;
}
if (!clusters.containsKey(i)) {
    clusters.put(i, new ArrayList<Cluster>());
}
int connections = 0;
for (final Point p : mergedCluster.getPoints()) {
    if (pairs.containsKey(p)) {
        connections += pairs.get(p).size();
        System.out.println(p + "\t" + pairs.get(p).size());
    }
}
System.out.println(mergedCluster.getPoints().size() + " merged points");
final double newStableFactor = connections
        / Math.pow(mergedCluster.getPoints().size(), 2);
System.out.println(newStableFactor + "  is newStableFactor");
System.out.println(mergingMinStableFactor
        + "  is mergingMinStableFactor");

if (newStableFactor > mergingMinStableFactor
        || Math.abs(newStableFactor - mergingMinStableFactor) < 0.00001) {
    clusters.get(i).add(mergedCluster);
    mergedCluster.setStableFactor(newStableFactor);
    System.out.println("!!! MERGED");
} else {
    clusters.get(i).add(c);
    clusters.get(i).addAll(mergingCluster);
```

```
                System.out.println("!!! ORIGINAL CLUSTERS RETAINED");
            }
        }
        for (final Cluster c : clusters.get(i)) {
            c.setMerged(false);
        }
    }
    return clusters;
}

public Map<Point, List<Point>> findPairs(final int k) {
    final Map<Point, List<Point>> pairs = new HashMap<Point, List<Point>>();
    for (final Point p1 : this.all) {
        for (final Point p2 : this.nearestTo(p1, k)) {
            if (this.nearestTo(p2, k).contains(p1) && p1 != p2) {
                if (!pairs.containsKey(p1)) {
                    pairs.put(p1, new ArrayList<Point>());
                }
                pairs.get(p1).add(p2);
            }
        }
    }
    return pairs;
}

@SuppressWarnings("boxing")
private void findNearestAtMaxK() {
    for (final Point p : this.all) {
        final Map<Double, Point> distances = new HashMap<Double, Point>();
        for (final Point p2 : this.all) {
            if (p == p2) {
                continue;
            }
            distances.put(p.distanceTo(p2), p2);
        }
```

```
        final List<Double> sortedDistances = new ArrayList<Double>(
            distances.size());
        sortedDistances.addAll(distances.keySet());
        Collections.sort(sortedDistances);
        final List<Point> nearestPoints = new ArrayList<Point>(100);
        for (final Double d : sortedDistances.subList(0, this.maxK)) {
            nearestPoints.add(distances.get(d));
        }
        this.nearest.put(p, nearestPoints);
      }
    }

    private List<Point> nearestTo(final Point p, final int k) {
      return this.nearest.get(p).subList(0, k);
    }

    private final List<Point> all;
    private final int maxK;

    private final Map<Point, List<Point>> nearest;
}
```

A.2 kNN Demo Code

Point.Java

```
package demo;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.WindowConstants;


import algorithm.Algorithm;
import algorithm.Cluster;
import algorithm.Point;

public final class DemoFrame extends JFrame {
    public DemoFrame() {
        this.setTitle("KNN Demo");
        this.setSize(1024, 600);
        this.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        this.setLayout(new BorderLayout());
        this.canvas = new Canvas();
        this.controlPanel = new ControlPanel(this.canvas);
        this.getContentPane().add(this.controlPanel, BorderLayout.NORTH);
        this.getContentPane().add(this.canvas, BorderLayout.CENTER);
    }

    private final Canvas canvas;
```

```

    private final ControlPanel controlPanel;

    public static void main(final String[] args) {
        new DemoFrame().setVisible(true);
    }

    private static final long serialVersionUID = -2297653670935822145L;
}

@SuppressWarnings("serial")
final class Canvas extends JPanel {
    public Canvas() {
        this.addMouseListener(new MouseAdapter() {
            @Override
            public final void mousePressed(final MouseEvent e) {
                for (final Point p : Data.points) {
                    if (p.distanceTo(new Point(e.getX(), e.getY())) < 10) {
                        double sum = 0;
                        double previous=0;
                        double giniIndex=0;
                        for (int i = 1; i <= Data.clusterK; ++i) {
                            final Algorithm a = new Algorithm(Data.points, i);
                            final Map<Point, List<Point>> pairs = a.findPairs(i);
                            if (pairs.containsKey(p)) {
                                sum += pairs.get(p).size()+previous;
                                JOptionPane.showMessageDialog(null, " (" + (int)p.getX()+ " , "+(int)p.getY()+") " +"When K = "+i+", the MKNN number is "+ pairs.get(p).size());
                                previous=pairs.get(p).size();
                            }
                            else
                                JOptionPane.showMessageDialog(null, " (" + (int)p.getX()+ " , "+(int)p.getY()+") " +"When K = "+i+", the MKNN number is 0");
                        }
```

```
                    giniIndex=1-sum/(Data.clusterK * Data.clusterK);
                    JOptionPane.showMessageDialog(null, "GINI-index: " + giniIndex);
                    break;
                }
            }
        }
    });
}

@SuppressWarnings("boxing")
@Override
public final void paintComponent(final Graphics g) {
    ((Graphics2D) g).setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
    g.setColor(new Color(0, 3, 97));
    g.fillRect(0, 0, this.getWidth(), this.getHeight());

    g.setColor(Color.white);
    for (final Point p : Data.pairs.keySet()) {
        for (final Point to : Data.pairs.get(p)) {
            g.drawLine((int) p.getX(), (int) p.getY(), (int) to.getX(),
                    (int) to.getY());
        }
    }

    g.setColor(Color.yellow);
    for (final Point p : Data.points) {
        g.fillOval((int) p.getX() - 2, (int) p.getY() - 2, 5, 5);
        g.drawString(p.toString(), (int) p.getX(), (int) p.getY());
    }
    if (Data.showClusters) {
        final Random r = new Random();
        if (Data.clusters.containsKey(Data.clusterK)) {
            for (final Cluster c : Data.clusters.get(Data.clusterK)) {
                g.setColor(new Color(r.nextFloat(), r.nextFloat(), r.nextFloat()));
```

```
                    for (final Point p : c.getPoints()) {
                        g.fillOval((int) p.getX() - 5, (int) p.getY() - 5, 10, 10);
                    }
                }
            }
        }
    }
}

@SuppressWarnings("serial")
final class ControlPanel extends JPanel {

    public ControlPanel(final Canvas canvas) {
        this.numberTextField.setText("15");
        this.nearDistanceTextField.setText("50");
        this.nearPercentageTextField.setText("20");
        this.kTextField.setText("3");
        // this.clusterKTextField.setText("1");
        this.add(this.numberLabel);
        this.add(this.numberTextField);
        this.add(this.kLabel);
        this.add(this.kTextField);
        // this.add(this.clusterKLabel);
        // this.add(this.clusterKTextField);
        this.add(this.nearDistanceLabel);
        this.add(this.nearDistanceTextField);
        this.add(this.nearPercentageLabel);
        this.add(this.nearPercentageTextField);
        this.add(this.viewClusterButton);
        this.add(this.randomButton);
        this.add(this.runButton);
        this.add(this.clearButton);
        this.randomButton.addActionListener(new ActionListener() {
            @SuppressWarnings({ "boxing", "synthetic-access" })
            @Override
```

```
            public final void actionPerformed(final ActionEvent e) {
                ControlPanel.clear();
                final Random r = new Random();
                final double width = canvas.getWidth(), height = canvas.getHeight();
                final int nearDistance = Integer
                        .valueOf(ControlPanel.this.nearDistanceTextField.getText()),
nearPercentage = Integer
                        .valueOf(ControlPanel.this.nearPercentageTextField.getText());
                Point previous = null;
                for (int i = 0; i < Integer.valueOf(ControlPanel.this.numberTextField
                        .getText()); ++i) {
                    if (previous == null) {
                        previous = new Point(r.nextDouble() * width, r.nextDouble()
                                * height);
                    } else {
                        double x, y;
                        do {
                            double d;
                            final double angle = r.nextDouble() * 2 * Math.PI;
                            if (r.nextInt(101) <= nearPercentage) {
                                d = r.nextDouble() * nearDistance;
                            } else {
                                d = r.nextDouble() * width;
                            }
                            x = previous.getX() + d * Math.cos(angle);
                            y = previous.getY() + d * Math.sin(angle);
                        } while (x < 0 || x > width - 1 || y < 0 || y > height - 1);
                        final Point p = new Point(x, y);
                        previous = p;
                    }
                    Data.points.add(previous);
                }
                canvas.repaint();
            }
        });
```

```
        this.runButton.addActionListener(new ActionListener() {
            @SuppressWarnings({ "boxing", "synthetic-access" })
            @Override
            public final void actionPerformed(final ActionEvent e) {
                final Algorithm a = new Algorithm(Data.points, Integer
                        .valueOf(ControlPanel.this.kTextField.getText()));
                Data.pairs = a.findPairs(Integer.valueOf(ControlPanel.this.kTextField
                        .getText()));
                Data.clusterK = Integer.valueOf(ControlPanel.this.kTextField.getText());
                Data.showClusters = false;
                canvas.repaint();
            }
        });
        this.viewClusterButton.addActionListener(new ActionListener() {
            @SuppressWarnings({ "boxing", "synthetic-access" })
            @Override
            public final void actionPerformed(final ActionEvent e) {
                Data.clusterK = Integer.valueOf(ControlPanel.this.kTextField.getText());
                final Algorithm a = new Algorithm(Data.points, Integer
                        .valueOf(ControlPanel.this.kTextField.getText()));
                Data.pairs = a.findPairs(Integer.valueOf(ControlPanel.this.kTextField
                        .getText()));
                Data.clusters = a.findClusters();
                Data.showClusters = true;
                canvas.repaint();
            }
        });
        this.clearButton.addActionListener(new ActionListener() {
            @SuppressWarnings("synthetic-access")
            @Override
            public final void actionPerformed(final ActionEvent e) {
                ControlPanel.clear();
                canvas.repaint();
            }
        });
```

```
    }

    private final JLabel numberLabel = new JLabel("Number of points: "),
            kLabel = new JLabel("K:"), clusterKLabel = new JLabel("Cluster K:"),
            nearPercentageLabel = new JLabel("Near%: "),
            nearDistanceLabel = new JLabel("Near distance: ");
    private final JTextField numberTextField = new JTextField(4),
            kTextField = new JTextField(2), nearPercentageTextField = new JTextField(
                    2), nearDistanceTextField = new JTextField(3),
            clusterKTextField = new JTextField(2);
    private final JButton randomButton = new JButton("Random"),
            runButton = new JButton("Draw the link"), clearButton = new JButton("
        Clear"),
            viewClusterButton = new JButton("Show Clusters");

    private static void clear() {
        Data.points.clear();
        Data.pairs.clear();
        Data.clusters.clear();
    }
}

final class Data {
    static {
        Data.points = new ArrayList<Point>();
        Data.pairs = new HashMap<Point, List<Point>>();
        Data.clusters = new HashMap<Integer, List<Cluster>>();
    }
    public static int clusterK;
    public static Map<Integer, List<Cluster>> clusters;
    public static Map<Point, List<Point>> pairs;
    public static List<Point> points;
    public static boolean showClusters;
}
```

A.3 Path kNN Query Search Simulation

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Data.Odbc;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Collections;

namespace IKNN
{
    public partial class IKNNMain : Form
    {
        private DataTable dtRoutine;
        private DataTable dtPassedRoutine;
        private DataTable dtSegment = new DataTable();
        private DateTime StartTime;

        private int intDmax;

        private struct Query
        {
            public string strSP;
            public int intSPDistance;
            public string strEP;
            public int intEPDistance;
            public int intKpoints;
        }

        public IKNNMain()
        {
```

```
            InitializeComponent();

            LoadSegmentData();
        }

        private void btnRun_Click(object sender, EventArgs e)
        {
            dtPassedRoutine = new DataTable();
            dtRoutine = new DataTable();
            intDmax = int.MaxValue;
            Query myQ = new Query();

            myQ.strSP = txtA.Text.Split(',')[0];
            myQ.intSPDistance = Convert.ToInt32(txtA.Text.Split(',')[1]);
            myQ.strEP = txtB.Text.Split(',')[0];
            myQ.intEPDistance = Convert.ToInt32(txtB.Text.Split(',')[1]);
            myQ.intKpoints = Convert.ToInt32(txtK.Text);

            DataColumn dcNode = new DataColumn("Node", Type.GetType("System.String"
            ));
            dtRoutine.Columns.Add(dcNode);
            DataColumn dcDistance = new DataColumn("Distance", Type.GetType("System.
            Int32"));
            dtRoutine.Columns.Add(dcDistance);

            for (int i = 0; i < myQ.intKpoints; i++)
            {
                DataColumn dcWaypoint = new DataColumn("WP" + i, Type.GetType("
            System.String"));
                dtRoutine.Columns.Add(dcWaypoint);
            }

            DataRow myQueryRowA = dtRoutine.NewRow();

            myQueryRowA["Node"] = myQ.strSP;
```

```
            myQueryRowA["Distance"] = myQ.intSPDistance;

            dtRoutine.Rows.Add(myQueryRowA);

            DataRow myQueryRowB = dtRoutine.NewRow();

            myQueryRowB["Node"] = myQ.strEP;
            myQueryRowB["Distance"] = myQ.intEPDistance;

            dtRoutine.Rows.Add(myQueryRowB);

            int intDminRow = −1;
            int intK = 1;

            StartTime = DateTime.Now;

            intDminRow = FindMinDistanceRow();
            GetLinkedSegments(intDminRow, myQ.intKpoints);


            while (intDminRow == −1 || intDmax > Convert.ToInt32(dtRoutine.Rows[
         FindMinDistanceRow()]["Distance"]) || intDmax == int.MaxValue)
            {
                intDminRow = FindMinDistanceRow();
                GetLinkedSegments(intDminRow, myQ.intKpoints);
                intK++;
            }

            lstOutput.Items.Add(intK + " Time: " + DateTime.Now.Subtract(StartTime));

            //for (int i = 0; i < dtRoutine.Rows.Count; i++)
            //{
            //    lstOutput.Items.Add(dtRoutine.Rows[i]["Node"] + " " + dtRoutine.Rows[i]["
         Distance"] + dtRoutine.Rows[i]["WP0"]);
            //}
```

```

            dtRoutine.Clear();
            dtPassedRoutine.Clear();
            dtRoutine.Dispose();
            dtPassedRoutine.Dispose();
        }

        private void GetLinkedSegments(int intMinRowId, int intKpoints)
        {
            for (int i = 0; i < dtSegment.Rows.Count; i++)
            {
                DataRow myRow = dtRoutine.NewRow();

                if (dtSegment.Rows[i]["StartPoint"].Equals(dtRoutine.Rows[intMinRowId]["
        Node"]))
                {
                    if (!isPassedRoutine(intMinRowId, intKpoints, i))
                    {
                        myRow["Node"] = dtSegment.Rows[i]["EndPoint"];
                        UpdateWayPoints(myRow, intMinRowId, intKpoints, i);
                    }
                }
                else if (dtSegment.Rows[i]["EndPoint"].Equals(dtRoutine.Rows[intMinRowId][
        "Node"]))
                {
                    if (!isPassedRoutine(intMinRowId, intKpoints, i))
                    {
                        myRow["Node"] = dtSegment.Rows[i]["StartPoint"];
                        UpdateWayPoints(myRow, intMinRowId, intKpoints, i);
                    }
                }
            }
            //lstOutput.Items.Add(dtRoutine.Rows[intMinRowId]["Node"].ToString() + " " +
            //      dtRoutine.Rows[intMinRowId]["Distance"].ToString() + " " +
```

```
            //      (dtRoutine.Rows[intMinRowId]["WP0"].Equals(DBNull.Value) ? "Null" :
        dtRoutine.Rows[intMinRowId]["WP0"].ToString()));

            if (dtPassedRoutine.Rows.Count <= 0)
                dtPassedRoutine = dtRoutine.Clone();

            DataRow myPassedRow = dtPassedRoutine.NewRow();
            for (int i = 0; i < dtRoutine.Columns.Count; i++)
            {
                myPassedRow[i] = dtRoutine.Rows[intMinRowId][i];
            }
            dtPassedRoutine.Rows.Add(myPassedRow);

            dtRoutine.Rows.RemoveAt(intMinRowId);

            FilterDuplicateRoutines(intKpoints);
        }

        private void UpdateWayPoints(DataRow myRow, int intMinRowId, int intKpoints, int
          i)
        {
            myRow["Distance"] = Convert.ToInt32(dtRoutine.Rows[intMinRowId]["Distance"
        ]) + Convert.ToInt32(dtSegment.Rows[i]["Length"]);

            for (int j = 0; j < intKpoints; j++)
            {
                myRow["WP" + j] = dtRoutine.Rows[intMinRowId]["WP" + j];
            }

            int intTmp;
            if (!int.TryParse(myRow["Node"].ToString(), out intTmp))
            {
                for (int k = 0; k < intKpoints; k++)
                {
                    if (myRow["WP" + k].Equals(DBNull.Value))
```

```
                    {
                        myRow["WP" + k] = myRow["Node"];
                        if (intKpoints == k + 1)
                        {
                            if (intDmax > Convert.ToInt32(myRow["Distance"]))
                            {
                                intDmax = Convert.ToInt32(myRow["Distance"]);
                            }
                        }
                        break;
                    }
                    else
                    {
                        if (myRow["WP" + k].Equals(myRow["Node"]))
                            break;
                    }
                }
            }
            dtRoutine.Rows.Add(myRow);
        }

        private bool isPassedRoutine(int intMinRowId, int intKpoints, int i)
        {
            if (dtPassedRoutine.Rows.Count > 0)
            {
                for (int m = 0; m < dtPassedRoutine.Rows.Count; m++)
                {
                    if (dtPassedRoutine.Rows[m]["Node"].Equals(dtSegment.Rows[i]["EndPoint
        "]))
                    {
                        bool isNotSame = false;
                        for (int n = 0; n < intKpoints; n++)
                        {
                            if (!dtPassedRoutine.Rows[m]["WP" + n].Equals(dtRoutine.Rows[
        intMinRowId]["WP" + n]))
```

```
                {
                    isNotSame = true;
                    break;
                }
            }

            if (!isNotSame)
            {
                if (Convert.ToInt32(dtPassedRoutine.Rows[m]["Distance"]) <=
    Convert.ToInt32(dtRoutine.Rows[intMinRowId]["Distance"]) + Convert.ToInt32(
    dtSegment.Rows[i]["Length"]))
                {
                    return true;
                }
            }
        }
    }
}
return false;
}

private void FilterDuplicateRoutines(int intKpoints)
{
    for (int i = 0; i < dtRoutine.Rows.Count; i++)
    {
        for (int j = i + 1; j < dtRoutine.Rows.Count; j++)
        {
            if (dtRoutine.Rows[i]["Node"].Equals(dtRoutine.Rows[j]["Node"]))
            {
                bool isNotSame = false;
                for (int k = 0; k < intKpoints; k++)
                {
                    if (!dtRoutine.Rows[i]["WP" + k].Equals(dtRoutine.Rows[j]["WP" +
    k]))
                    {
```

```
                        isNotSame = true;
                        break;
                    }
                }

                if (!isNotSame)
                {
                    if (Convert.ToInt32(dtRoutine.Rows[i]["Distance"]) >= Convert.ToInt32(dtRoutine.Rows[j]["Distance"]))
                    {
                        dtRoutine.Rows.RemoveAt(i);
                        i = i > 0 ? i-- : i;
                        j = i + 1;
                    }
                    else if (Convert.ToInt32(dtRoutine.Rows[i]["Distance"]) < Convert.ToInt32(dtRoutine.Rows[j]["Distance"]))
                    {
                        dtRoutine.Rows.RemoveAt(j);
                    }
                }
                else
                {
                    for (int l = 0; l < intKpoints; l++)
                    {
                        if (!dtRoutine.Rows[i]["WP" + l].Equals(dtRoutine.Rows[j]["WP" + l]))
                        {
                            if (dtRoutine.Rows[i]["WP" + l].Equals(DBNull.Value))
                            {
                                if (Convert.ToInt32(dtRoutine.Rows[i]["Distance"]) >= Convert.ToInt32(dtRoutine.Rows[j]["Distance"]))
                                {
                                    dtRoutine.Rows.RemoveAt(i);
                                    i = i > 0 ? i-- : i;
                                    j = i + 1;
```

```
                                }
                            }
                            else if (dtRoutine.Rows[j]["WP" + l].Equals(DBNull.Value)
)
                            {
                                if (Convert.ToInt32(dtRoutine.Rows[j]["Distance"])
>= Convert.ToInt32(dtRoutine.Rows[i]["Distance"]))
                                {
                                    dtRoutine.Rows.RemoveAt(j);
                                }
                            }
                            break;
                        }
                    }
                }
            }
        }
    }
}

private int FindMinDistanceRow()
{
    int intDmin = −1;
    int intRowId = −1;

    for (int i = 0; i < dtRoutine.Rows.Count; i++)
    {
        int intTmp;

        intTmp = Convert.ToInt32(dtRoutine.Rows[i]["Distance"]);

        if (intTmp < intDmin || intDmin == −1)
        {
            intDmin = intTmp;
            intRowId = i;
```

```
                }
            }

            return intRowId;
        }

        private void LoadSegmentData()
        {
            string strSQL = "select * from segment";

            OdbcConnection dbConn = new OdbcConnection(@"Dsn=MS Access Database;dbq
=IKNN.mdb;driverid=25;fil=MS Access;maxbuffersize=2048;pagetimeout=5");
            OdbcCommand dbCmd = new OdbcCommand(strSQL, dbConn);
            OdbcDataAdapter dbAdapter = new OdbcDataAdapter(dbCmd);

            dbConn.Open();
            dbAdapter.Fill(dtSegment);
            dbCmd.Dispose();
            dbAdapter.Dispose();
            dbConn.Close();
        }
    }
}
```

A.4 Time Constraint Route Search Simulation

```

package undesignated;

import classes.Global;
import classes.Point;
import classes.QueryPoint;
import java.util.*;
import java.util.ArrayList;
```

```java

public class Main {

    public static void main(String[] args) {
     // for (int p=0; p<10; p++){
        generateRandomPoints();
        createQueryPoint();
        printPoints(Global.pointList);
        ArrayList<Object> candidateNext=candidateNext(Global.queryPoint);

        ArrayList<Integer> visitedType=new ArrayList<Integer>();
        for (int i=0; i< Global.typeList.size(); i++){
        Global.TC.add(0.00);
        }
        run(candidateNext,visitedType);
        }

    public static void printObjects(ArrayList<Object> aObjects){
        ArrayList<Point> pointSet=(ArrayList<Point>)aObjects.get(1);
        printPoints(pointSet);
    }

     public static void run(ArrayList<Object> candidateNext, ArrayList<Integer>
         visitedType){
        Global.TC.set(Global.level-1, Global.timeCost);
        ArrayList<Point> candidateNext1=(ArrayList<Point>)candidateNext.get(1);

        for(int i=0; i<candidateNext1.size(); i++){
                  if (Global.level==1){
                     if (getTravelTime(getDistance(candidateNext1.get(i),(QueryPoint)
         candidateNext.get(3)))+ Global.timeCost<Global.Tmax){
                     Global.timeCost+=getTravelTime(getDistance(candidateNext1.get(i),(
         QueryPoint)candidateNext.get(3)));}
                     else {
                         if (i==candidateNext1.size()-1){
```

```
                    Global.level--;

                    if (Global.level==0){
                    System.exit(0);
                    }
                    else{
                    Global.timeCost= Global.TC.get(Global.level-1);
                    visitedType.remove(visitedType.size()-1);
                    break;
                    }
                    }
                    else continue;
                    }
             }
            else{
            if (getTravelTime(getDistance((Point)candidateNext.get(3),candidateNext1.get(
        i)))+Global.timeCost<Global.Tmax){
                Global.timeCost+=getTravelTime(getDistance((Point)candidateNext.get(3),
        candidateNext1.get(i)));}
            else {
               System.out.println("*****TimeCost is too large go back to super
        *****");
               if (i==candidateNext1.size()-1){
               Global.level--;
               if (Global.level==0){
               System.exit(0);
               }
               else{
               Global.timeCost= Global.TC.get(Global.level-1);
               visitedType.remove(visitedType.size()-1);
               break;
               }}
               else continue;
               }
            }
```

```
        visitedType.add(candidateNext1.get(i).getType());
        Global.level++;
        for(int k=0; k<visitedType.size(); k++){
        System.out.println("****Visited Type**"+visitedType.get(k)+" ");
        }

        if(visitedType.size()==Global.typeList.size()){
            if(Global.timeCost<=Global.Tmax) Global.Tmax=Global.timeCost;
            System.out.println("Tmax: "+Global.Tmax);

            visitedType.remove(visitedType.size()-1);
            Global.level--;
            if(Global.level==1)
                Global.timeCost=0;
            else
            Global.timeCost-=getTravelTime(getDistance((Point)candidateNext.get(3),
    candidateNext1.get(i)));
            if (i==candidateNext1.size()-1){
            Global.level--;
            if (Global.level==0){
            System.out.println("The algorithm finish");
            System.exit(0);
            }
            else{
            Global.timeCost= Global.TC.get(Global.level-1);
            visitedType.remove(visitedType.size()-1);
            break;
            }}
            else continue;
        }
        ArrayList<Object> CN=new ArrayList<Object>();
        CN=candidateNext(candidateNext1.get(i), visitedType, Global.timeCost, Global.
    level);
        ArrayList<Point> temp=(ArrayList<Point>)CN.get(1);
```

```
            System.out.println("Global.level: "+Global.level);
            System.out.println("Global.typeList.size(): "+Global.typeList.size());
            if (temp.size()!=0)
            {
                run(CN,visitedType);
            }
            else
            {
             Global.level--;
             Global.timeCost= Global.TC.get(Global.level-1);
             visitedType.remove(visitedType.size()-1);
            }

            if (i==candidateNext1.size()-1){
                Global.level--;
                if (Global.level==0){
                System.exit(0);
                }
                else{
                Global.timeCost= Global.TC.get(Global.level-1);
                visitedType.remove(visitedType.size()-1);
                break;}
            }
        }
    }

    public static void generateRandomPoints(){
        int noPoints, noTypes=0;
        Random r=new Random();
        Global.mx=8;//input.nextInt();
        Global.my=8;//input.nextInt();
        noTypes=10;//input.nextInt();

        for (int i=0; i<noTypes; i++){
            int hour=0;
```

```
            double min=0;
            hour=5;//input.nextInt();
            min=0.0+(double)i*30.0;//input.nextDouble();
            Global.typeList.add(hour+min/60);
            noPoints=6;//input.nextInt();

            for(int j=0; j<noPoints; j++){
                int x, y=0;
                x=r.nextInt(Global.mx)+1;
                y=r.nextInt(Global.my)+1;
                Global.pointList.add(new Point(x,y,i));
            }
        }
    }

    public static void createQueryPoint(){
        int qx, qy, hour=0;
        double min=0;
        qx=4;//input.nextInt();
        qy=4;//input.nextInt();
        hour=4;//input.nextInt();
        min=30.0;//input.nextDouble();
        Global.queryPoint=new QueryPoint(qx, qy, hour+min/60);
        Global.speed=30.00;//input.nextDouble();
    }

    public static double getDistance(Point x, Point y){
        return Math.sqrt(Math.pow(x.getX()-y.getX(),2) + Math.pow((x.getY()-y.getY()),2))
         ;
    }

    public static double getDistance(Point x, QueryPoint y){
        return Math.sqrt(Math.pow(x.getX()-y.getX(),2) + Math.pow((x.getY()-y.getY()),2))
         ;
    }
```

```

    public static double getTravelTime(double distance){
        return distance/Global.speed;
    }

    public static void printPoints(ArrayList<Point> aSet){
        for(int i=0; i<aSet.size(); i++){
            aSet.get(i).printPoint();
        }
    }

    public static ArrayList<Object> candidateNext(Point p, ArrayList<Integer> visitedType,
         double timeCost, int level){
        ArrayList<Integer> T=new ArrayList<Integer>();
        for(int i=0; i<Global.typeList.size(); i++) T.add(i);
        ArrayList<Point> result=new ArrayList<Point>();

        ArrayList<Integer> unvisitedType=new ArrayList<Integer>();
        for(int i=0; i<T.size(); i++){
            if(visitedType.indexOf(T.get(i))==-1){unvisitedType.add(T.get(i));}
        }
        double timeCon=100;
        for(int i=0; i<unvisitedType.size(); i++){
            double time=Global.typeList.get(unvisitedType.get(i));
            if(time<timeCon)timeCon=time;
        }
        for(int i=0; i<unvisitedType.size(); i++){
            for(int j=0; j<Global.pointList.size(); j++){
                if(Global.pointList.get(j).getType()==unvisitedType.get(i)){
                    double distance=getDistance(Global.pointList.get(j),p);
                    if(getTravelTime(distance)<=(timeCon-timeCost-Global.queryPoint.
         getQueryTime())&&touch(Global.pointList.get(j), unvisitedType, distance, timeCost)){
                    if(getTravelTime(distance)<=(timeCon-timeCost-Global.queryPoint.
         getQueryTime())){
                        result.add(Global.pointList.get(j));
```

```
                    }
                }
            }
        }
        ArrayList<Object> r=new ArrayList<Object>();
        r.add(level);
        r.add(result);
        r.add(timeCost);
        r.add(p);
        printObjects(r);
        return r;
    }

    public static boolean touch(Point p, ArrayList<Integer> unvisitedType, double
         pTimeCost, double timeCost){
        for(int i=0; i<unvisitedType.size(); i++){

            double dmax=Math.sqrt(Global.mx^2+Global.my^2);
            for(int j=0; j<Global.pointList.size(); j++){
              if(Global.pointList.get(j).getType()==unvisitedType.get(i)){
                    double distance=getDistance(Global.pointList.get(j),p);
                    if(distance<dmax)dmax=distance;
              }
            }
            if(getTravelTime(dmax)>(Global.typeList.get(unvisitedType.get(i))-timeCost-
         pTimeCost-Global.queryPoint.getQueryTime())){return false;}
        }
        return true;
    }


    public static ArrayList<Object> candidateNext(QueryPoint p){
        ArrayList<Integer> unvisitedType=new ArrayList<Integer>();
        for(int i=0; i<Global.typeList.size(); i++) unvisitedType.add(i);
        ArrayList<Point> result=new ArrayList<Point>();
```

```

        double timeCon=100;
        for(int i=0; i<unvisitedType.size(); i++){

            double time=Global.typeList.get(unvisitedType.get(i));
            if(time<timeCon)timeCon=time;
        }
        for(int j=0; j<Global.pointList.size(); j++){
            double distance=getDistance(Global.pointList.get(j),p);

            if(distance<=(timeCon-p.getQueryTime())*Global.speed) {
               result.add(Global.pointList.get(j));

               }
           }


        ArrayList<Object> r=new ArrayList<Object>();
        r.add(1);
        r.add(result);
        r.add(0);
        r.add(p);
        printObjects(r);
        return r;
    }

}
```

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 *Proceeding of ACM SIGMOD*, pages 291–301. ACM Press, May 1985.

[Ber93] Marshall W. Bern. Approximate closest-point queries in high dimensions. *Inf. Process. Lett.*, 45(2):95–99, 1993.

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

[BMW07] Oliver Bohl, Shakib Manouchehri, and Udo Winand. Mobile information systems for the private everyday life. *Mobile Information Systems*, 3(3-4):135–152, 2007.

[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.

[BX11] L. Barolli and F. Xhafa. Jxta-overlay: A p2p platform for distributed, collaborative, and ubiquitous computing. *IEEE Transactions on Industrial Electronics*, 58(6):2163 – 2172, 2011.

[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.

[CCCX12] Xin Cao, Lisi Chen, Gao Cong, and Xiaokui Xiao. Keyword-aware optimal route search. *PVLDB*, 5(11):1136–1147, 2012.

[CF98] King Lum Cheung and Ada Wai-Chee Fu. Enhanced nearest neighbour search on the r-tree. *SIGMOD Record*, 27(3), 1998.

[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.

[CHC04] Ying Cai, Kien A. Hua, and Guohong Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Mobile Data Management*, pages 27–38, 2004.

[CKSZ08] Haiquan Chen, Wei-Shinn Ku, Min-Te Sun, and Roger Zimmermann. The multi-rule partial sequenced route query. In *GIS*, page 10, 2008.

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

[CLLP11] B. Choi, J. Lee, J. Lee, and K. Park. A hierarchical algorithm for indoor mobile robot localization using rfid sensor fusion. *IEEE Transactions on Industrial Electronics*, 58(6):2226 – 2235, 2011.

[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):123–137, 1979.

[Cor] Telstra Corporation. Whereis website. `http://www.whereis.com`. Accessed 10 June, 2012.

[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.

[DKD08] Nir Dolev, Yaron Kanza, and Yerach Doytsher. Efficient orienteering route search over uncertain spatial datasets. In *GIS Algorithms and Techniques*, pages 329–336, Stockholm, Sweden, 14-19 June 2008.

[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.

[EL05] Andreas Ehliar and Dake Liu. Flexible route lookup using range search. In *Communications and Computer Networks*, pages 345–350, 2005.

[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.

[Gad08] David A. Gadish. Introducing the elasticity of spatial data. *IJDWM*, 4(3):54–70, 2008.

[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.

[GL04] Bugra Gedik and Ling Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[GR99] Marina L. Gavrilova and Jon G. Rokne. Swap conditions for dynamic voronoi diagrams for circles and line segments. *Computer Aided Geometric Design*, 16(2):89–106, 1999.

[GR03] Marina L. Gavrilova and Jon G. Rokne. Updating the topology of the dynamic voronoi diagram for spheres in euclidean d-dimensional space. *Computer Aided Geometric Design*, 20(4):231–242, 2003.

[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.

[GT] John Goh and David Taniar. Mining frequency pattern from mobile users. In *Proc. of the 8th KES*, volume 3215 of *LNCS*. Springer.

[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 *KES*, pages 795–801, 2004.

[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 *Proceeding 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.

[GZCL09] Yunjun Gao, Baihua Zheng, Gencai Chen, and Qing Li. On efficient mutual nearest neighbor query processing in spatial databases. *Data Knowl. Eng.*, 68(8):705–727, 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.

[HJ04] Xuegang Huang and Christian S. Jensen. In-route skyline querying for location-based services. In *W2GIS*, pages 120–135, 2004.

[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.

[ISS03] Glenn S. Iwerks, Hanan Samet, and Kenneth P. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, pages 512–523, 2003.

[JLO07] 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.

[KGT99] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. Nearest neighbor queries in a mobile environment. In *Spatio-Temporal Database Management*, pages 119–134, 1999.

[KHK07] Jongbum Kim, B. F. Hobbs, and J. F. Koonce. Analysis of the sensitivity of decision analysis results to errors and simplifications in problem structure: Application to lake erie ecosystem management. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 37(4):505–518, 2007.

[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.

[KLH+07] H.J. Koskimaki, P. Laurinen, E. Haapalainen, L. Tuovinen, and J. Roning. Application of the extended knn method to resistance spot welding process identification and the benefits of process information. *IEEE Transactions on Industrial Electronics*, 54(5):2823 – 2830, 2007.

[KLSS09] Yaron Kanza, Roy Levin, Eliyahu Safra, and Yehoshua Sagiv. An interactive approach to route search. In *GIS*, pages 408–411, 2009.

[KM00] Flip Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD Conference*, pages 201–212, 2000.

[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, 2005.

[KSS09] Yaron Kanza, Eliyahu Safra, and Yehoshua Sagiv. Route search over probabilistic geospatial data. In *SSTD*, pages 153–170, 2009.

[KSSD08] Yaron Kanza, Eliyahu Safra, Yehoshua Sagiv, and Yerach Doytsher. Heuristic algorithms for route-search queries over geographical data. In *GIS*, page 11, 2008.

[KZWW05] Wei-Shinn Ku, Roger Zimmermann, Haojun Wang, and Chi-Ngai Wan. Adaptive nearest neighbor queries in travel time networks. In *Proceeding of ACM GIS*, pages 210–219. ACM Press, Nov 2005.

[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.

[LKSS10] Roy Levin, Yaron Kanza, Eliyahu Safra, and Yehoshua Sagiv. Interactive route search in the presence of order constraints. *PVLDB*, 3(1):117–128, 2010.

[LLL11] Wookey Lee, C.K. Leung, and J.J.H. Lee. Mobile web navigation in digital ecosystems using rooted directed trees. *IEEE Transactions on Industrial Electronics*, 58(6):2154 – 2162, 2011.

[LLT11] Eric Hsueh-Chan Lu, Chih-Yuan Lin, and Vincent S. Tseng. Tripmine: An efficient trip planning approach with travel time constraints. In *Mobile Data Management (1)*, pages 152–161, 2011.

[LNY03] King-Ip Lin, Michael Nolen, and Congjun Yang. Applying bulk insertion techniques for dynamic reverse nearest neighbor problems. In *IDEAS*, pages 290–297, 2003.

[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.

[LXWX05] Yingwei Luo, Guomin Xiong, Xiaolin Wang, and Zhuoqun Xu. Spatial data channel in a mobile navigation system. In *Proceedings of ICCSA 2005*, volume 3481 of *LNCS*, pages 822–831, Singapore, 2005. Springer.

[LZL08] Ken C. K. Lee, Baihua Zheng, and Wang-Chien Lee. Ranked reverse nearest neighbor search. *IEEE Trans. Knowl. Data Eng.*, 20(7):894–910, 2008.

[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.

[McK09] Matt McKeon. Harnessing the information ecosystem with wiki-based visualization dashboards. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1081–1088, 2009.

[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.

[MHP05] Kyriakos Mouratidis, Marios Hadjieleftheriou, and Dimitris Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD Conference*, pages 634–645, 2005.

[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.

[MMB11] Nirmesh Malviya, Samuel Madden, and Arnab Bhattacharya. A continuous query system for dynamic route planning. In *ICDE*, pages 792–803, 2011.

[MMHM09] Zoubir Mammeri, Franck Morvan, Abdelkader Hameurlain, and Nadhem Marsit. Location-dependent query processing under soft real-time constraints. *Mobile Information Systems*, 5(3):205–232, 2009.

[MP07] Kyriakos Mouratidis and Dimitris Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng.*, 19(6):789–803, 2007.

[MPBT05] Kyriakos Mouratidis, Dimitris Papadias, Spiridon Bakiras, and Yufei Tao. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *IEEE Trans. Knowl. Data Eng.*, 17(11):1451–1464, 2005.

[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.

[MVZ02] Anil Maheshwari, Jan Vahrenhold, and Norbert Zeh. On reverse nearest neighbor queries. In *CCCG*, pages 128–132, 2002.

[MXA04] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD Conference*, pages 623–634, 2004.

[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.

[OBSC00] 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.

[PG98] John Pearson and Hans W. Guesgen. Some experimental results of applying heuristic search to route finding. In *FLAIRS Conference*, pages 394–398, 1998.

[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.

[PKX08] Sunil Prabhakar, Dmitri V. Kalashnikov, and Yuni Xia. Indexing, query and velocity-constrained. In *Encyclopedia of GIS*, pages 518–523. 2008.

[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.

[PS07] 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 *Proceeding of 29th VLDB*, pages 802–813, Berlin, Germany, 2003. Morgan Kaufmann Publishers Inc.

[RDAYY11] Senjuti Basu Roy, Gautam Das, Sihem Amer-Yahia, and Cong Yu. Interactive itinerary planning. In *ICDE*, pages 15–26, 2011.

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

[SAE00] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop* *on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

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

[Saf06] Maytham Safar. Enhanced continuous knn queries using pine on road networks. In *ICDIM*, pages 248–256, 2006.

[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.

[SGZ07] Lionel Savary, Georges Gardarin, and Karine Zeitouni. Geocache: A cache for gml geographical data. *IJDWM*, 3(1):67–88, 2007.

[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.

[SK98] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD Conference*, pages 154–165, 1998.

[SKS08] Mehdi Sharifzadeh, Mohammad R. Kolahdouzan, and Cyrus Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.

[SRAE01] Ioana Stanoi, Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*, pages 99–108, 2001.

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

[SS09] 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.

[SWCD97] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.

[SX08] Shashi Shekhar and Hui Xiong, editors. *Encyclopedia of GIS*. Springer, 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.

[TFPL04] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu. Prediction and indexing of moving objects with unknown motion patterns. In *SIGMOD Conference*, pages 611–622, 2004.

[TG07] David Taniar and John Goh. On mining movement pattern from mobile users. *IJDSN*, 3(1):69–86, 2007.

[TP02] Yufei Tao and Dimitris Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD Conference*, pages 334–345, 2002.

[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 *VLDB*, pages 744–755, 2004.

[TPS02a] 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.

[TPS02b] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, 2002.

[TR02] David Taniar and J. Wenny Rahayu. A taxonomy of indexing schemes for parallel database systems. *Distributed and Parallel Databases*, 12(1):73–106, 2002.

[TR04] David Taniar and J. Wenny Rahayu. Global parallel index for multi-processors database systems. *Inf. Sci.*, 165(1-2):103–127, 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.

[VS08] Leena Vachhani and K. Sridharan. Hardware efficient prediction correction based generalized voronoi diagram construction and fpga implementation. *IEEE Transactions on Industrial Electronics*, 55(4):1558–1569, 2008.

[WCY06] Kun-Lung Wu, Shyh-Kwei Chen, and Philip S. Yu. Incremental processing of continual range queries over moving objects. *IEEE Trans. Knowl. Data Eng.*, 18(11):1560–1575, 2006.

[wM06] Telstra Corporation whereis Melbourne, Feb 2006. 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.

[WRTS11] A.B. Waluyo, W. Rahayu, D. Taniar, and B. Scrinivasan. A novel structure and access mechanism for mobile data broadcast in digital ecosystems. *IEEE Transactions on Industrial Electronics*, 58(6):2173 – 2182, 2011.

[WST03] Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar. Optimal broadcast channel for data dissemination in mobile database environment. In *APPT*, pages 655–664, 2003.

[WST04] Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar. A taxonomy of broadcast indexing schemes for multi channel data dissemination in mobile database. In *AINA (1)*, pages 213–218, 2004.

[WST05a] Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar. Research in mobile database query optimization and processing. *Mobile Information Systems*, 1(4):225–252, 2005.

[WST05b] Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar. Research on location-dependent queries in mobile databases. *Comput. Syst. Sci. Eng.*, 20(2), March 2005.

[WW06] Xiaoyuan Wang and Wei Wang. Continuous expansion: Efficient processing of continuous range monitoring in mobile environments. In *DASFAA*, pages 890–899, 2006.

[WZK06] Haojun Wang, Roger Zimmermann, and Wei-Shinn Ku. Distributed continuous range query processing on moving objects. In *DEXA*, pages 655–665, 2006.

[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+09] Kefeng Xuan, Geng Zhao, David Taniar, Bala Srinivasan, Maytham Safar, and Marina L. Gavrilova. Network voronoi diagram based range search. In *AINA*, pages 741–748, 2009.

[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 *ICPADS*, pages 361–368, 2008.

[YL01] Congjun Yang and King-Ip Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, pages 485–492, 2001.

[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.

[YS05] Jin Soung Yoo and Shashi Shekhar. In-route nearest neighbor queries. *GeoInformatica*, 9(4):117–137, 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.

[YTL11] Bin Yao, Mingwang Tang, and Feifei Li. Multi-approximate-keyword routing in gis data. In *GIS*, pages 201–210, 2011.

[Zha08] Qiong Zhang. Hierarchical route representation, indexing, and search. *IEEE Pervasive Computing*, 7(2):78–84, 2008.

[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+] Geng Zhao, Kefeng Xuan, Wenny Rahayu, David Taniar, Maytham Safar, Marina Gavrilova, and Bala Srinivasan. Voronoi-based continuous k nearest neighbor search in mobile navigation. *IEEE Transactions on Industrial Electronics*, 56(10). In press.

[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+09a] 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.

[ZXT+09b] Geng Zhao, Kefeng Xuan, David Taniar, Maytham Safar, Marina Gavrilova, and Bala Srinivasan. Multiple object types knn search using network voronoi diagram. In *Proceeding of International Conference for Computational Science and Its Applications*, Yongin, Korea, 2009.

[ZXT11] Geng Zhao, Kefeng Xuan, and David Taniar. Path knn query processing in mobile systems. *IEEE Transactions on Industrial Electronics*, 99, 2011.

[ZXTS08] Geng Zhao, Kefeng Xuan, David Taniar, and Bala Srinivasan. Incremental k-nearest-neighbor search on road networks. *Journal Of Interconnection Networks*, 9(4):455–470, 2008.

[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.