THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Efficient Spatial Keyword Query Processing on Geo-textual Data

Bolong Zheng

Master of Engineering

A thesis submitted for the degree of Doctor of Philosophy at

The University of Queensland in 2017

School of Information Technology & Electrical Engineering

Abstract

The past decades have witnessed a transformation from a desktop-based web to a predominantly mobile web, where more often than not, users access the web from mobile devices. As a result, a huge volume of geo-textual web objects that have both geographical location and textual description have been generated. In literature, there have been lots of efforts in enabling efficient processing large-scale geo-textual data under a variety of problem settings. In spite of the remarkable progress in this field, unsolved challenges remain. In this PhD thesis, I investigate a few interesting but challenging problems of this area. The contribution of this thesis can be summarized in three aspects. First, a set of new query predicates has been defined with a target of more diversified data types (e.g., activity trajectories), underlying spaces (e.g., road network) and query semantics, by which users can acquire their interested results more easily and effectively. Second, from a technical point of view, I have developed I/O efficient indexing structures and search algorithms to assure that a query can be answered within a reasonably small amount of time especially when dealing with massive geo-textual objects, which is not uncommon in real application scenarios. Last but not least, extensive empirical studies have been conducted based on real and large-scale datasets, which uncovered interesting patterns, rules and trends. These insights have shown directions for further improving my methodologies and also shed lights on future research. Below is a brief description of the contributions:

First, I study the keyword-oriented queries on activity trajectories (*KOAT*). The activity trajectory is semantically enriched trajectory data, which embeds the information about behaviors of the moving objects. Therefore, searching activity trajectory is able to support a variety of applications for a better quality of location-based services. This work aims to return k trajectories that contain the most relevant keywords to the query and yield the least travel effort in the meantime. The main difference between this work and conventional spatial keyword query is that no query location exists in *KOAT*, which means the search area cannot be localized. To improve the query performance, a spatial-textual ranking function is first proposed between query keywords and activity trajectories. Then a best-first search algorithm based on a hybrid index structure is developed by applying effective pruning rules and efficient refinement strategies.

Second, the problem of keyword-aware continuous k nearest neighbour search on road networks is studied, which computes the k nearest vertices that contain the query keywords issued by a moving object, and maintains the results continuously as it is moving on the road network. In particular, a

framework, called labeling approach for continuous query (*LARC*), is proposed for processing the query with both low computation and communication overheads. First, a keyword-based pivot tree (*KP-tree*) is proposed to improve the efficiency of the static k nearest neighbour query by avoiding massive network traversals and sequential probe of keywords. Then, the concepts of dominance interval and region on road network are developed, which share the similar intuition with safe region in Euclidean space but are more complicated with a dedicated design.

Finally, a novel type of query called clue-based route search (*CRS*) is investigated, which allows a user to provide clues on keywords and spatial relationships along the route. These personalized requirements make the route search become distance-sensitive such that the distances between PoIs along the route must be as close as possible to the user specified distance. To improve efficiency, a branch-and-bound algorithm is proposed that prunes unnecessary vertices in query processing. In order to quickly locate candidate, an *AB-tree* is proposed that stores both the distance and keyword information in a tree structure. To further reduce the index size, a *PB-tree* is constructed by utilizing the virtue of 2-hop label index to pinpoint the candidate.

Declaration by Author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis.

Publications during candidature

Journal paper:

- Bolong Zheng, Han Su, Kai Zheng, Xiaofang Zhou. Landmark-based Route Recommendation with Crowd Intelligence. *Data Science and Engineering* (DSE) 2016, 1(2), 86-100.
- Kai Zheng, Bolong Zheng*, Jiajie Xu, Guanfeng Liu, An Liu, Zhixu Li. Popularity-aware Spatial Keyword Search on Activity Trajectories. *World Wide Web Journal* (WWWJ) 2016, 1-25.

Conference paper:

- Bolong Zheng, Kai Zheng, Xiaokui Xiao, Han Su, Hongzhi Yin, Xiaofang Zhou, Guohui Li. Keyword-Aware Continuous kNN Query on Road Networks. In *Proceedings of IEEE International Conference on Data Engineering* (ICDE) 2016, Helsinki, 871-882.
- Yaguang Li, Han Su, Ugur Demiryurek, Bolong Zheng, Kai Zeng, Cyrus Shahabi. PerNav: A Route Summarization Framework for Personalized Navigation. In *Proceedings of ACM International Conference on Management of Data* (SIGMOD) 2016, San Francisco, 2125-2128. (DEMO)
- Bolong Zheng, Nicholas Jing Yuan, Kai Zheng, Xing Xie, Shazia Sadiq, Xiaofang Zhou. Approximate Keyword Search in Semantic Trajectory Database. In *Proceedings of IEEE International Conference on Data Engineering* (ICDE) 2015, Seoul, 975-986.
- Kai Zheng, Han Su, Bolong Zheng, Shuo Shang, Jiajie Xu, Jiajun Liu, Xiaofang Zhou. Interactive Top-k Spatial Keyword Queries. In *Proceedings of IEEE International Conference on Data Engineering* (ICDE) 2015, Seoul, 423-434.
- Haozhou Wang, Kai Zheng, Jiajie Xu, Bolong Zheng, Xiaofang Zhou, Shazia Sadiq. SharkDB: An In-memory Column-oriented Trajectory Storage. In *Proceedings of ACM International Conference on Information and Knowledge Management* (CIKM) 2014, Shanghai, 1409-1418.
- Bolong Zheng, Kai Zheng, Mohamed A Sharaf, Xiaofang Zhou, Shazia Sadiq. Efficient Retrieval of Top-k Most Similar Users from Travel Smart Card Data. In *Proceedings of IEEE International Conference on Mobile Data Management* (MDM) 2014, Brisbane, 259-268.

- Jiping Wang, Kai Zheng, Hoyoung Jeung, Haozhou Wang, Bolong Zheng, Xiaofang Zhou. Cost-Efficient Spatial Network Partitioning for Distance-Based Query Processing. In *Proceedings of IEEE International Conference on Mobile Data Management* (MDM) 2014, Brisbane, 13-22.

Publications included in this thesis

Bolong Zheng, Nicholas Jing Yuan, Kai Zheng, Xing Xie, Shazia Sadiq, Xiaofang Zhou. Approximate Keyword Search in Semantic Trajectory Database. ICDE 2015, Seoul, 975-986. - incorporated as Chapter 3.

Contributor	Statement of contribution
Bolong Zheng (Candidate)	Designed algorithm (50%) Wrote the paper (60%) Designed experiments (80 %) Proofreading the paper (30 %) Joined the discussion (30 %)
Nicholas Jing Yuan	Designed algorithm (30%) Wrote the paper (30%) Designed experiments (10 %) Proofreading the paper (20 %) Joined the discussion (20 %)
Kai Zheng	Designed algorithm (20%) Wrote the paper (10%) Designed experiments (10 %) Proofreading the paper (20 %) Joined the discussion (20 %)
Xing Xie	Proofreading the paper (10 %) Joined the discussion (10 %)
Shazia Sadiq	Proofreading the paper (10 %) Joined the discussion (10 %)
Xiaofang Zhou	Proofreading the paper (10 %) Joined the discussion (10 %)

Bolong Zheng, Kai Zheng, Xiaokui Xiao, Han Su, Hongzhi Yin, Xiaofang Zhou, Guohui Li. Keyword-Aware Continuous kNN Query on Road Networks. ICDE 2016, Helsinki, 871-882. - incorporated as Chapter 4.

Contributor	Statement of contribution
Bolong Zheng (Candidate)	Designed algorithm (50%) Wrote the paper (60%) Designed experiments (80 %) Proofreading the paper (20 %) Joined the discussion (20 %)
Kai Zheng	Designed algorithm (20%) Wrote the paper (30%) Designed experiments (10 %) Proofreading the paper (20 %) Joined the discussion (20 %)
Xiaokui Xiao	Designed algorithm (30%) Wrote the paper (10%) Designed experiments (10 %) Proofreading the paper (20 %) Joined the discussion (20 %)
Han Su	Proofreading the paper (10 %) Joined the discussion (10 %)
Hongzhi Yin	Proofreading the paper (10 %) Joined the discussion (10 %)
Xiaofang Zhou	Proofreading the paper (10 %) Joined the discussion (10 %)
Guohui Li	Proofreading the paper (10 %) Joined the discussion (10 %)

Contributions by others to the thesis

In all of the presented research in this thesis, Prof Xiaofang Zhou, as my principal advisor, and Dr Kai Zheng, as my associated advisor, have provided technical guidance for formulating the problems, refinement of ideas as well as reviewing and polishing the presentation.

Statement of parts of the thesis submitted to qualify for the award of another degree

None.

Acknowledgments:

I would like to give my sincere thanks to people who have supported and helped me during my PhD years. It is a great pleasure to convey my gratitude to them all in my humble acknowledgment.

First and foremost, I would like to give my earnest appreciation to my principal advisor, Prof. Xiaofang Zhou, who not only brought me into the exciting area of spatial-temporal database, but also has been remarkably helpful in various stages of my research. Furthermore, I acknowledge him for his valuable suggestions in each group discussion, and sparing his precious time to read my drafts and give constructive comments about them.

I am deeply grateful to my associate advisor, Dr. Kai Zheng, who has been supportive since the first day I came to The University of Queensland, and has oriented and guided me with patience and care throughout my PhD study. His passion for research sets an example for me, and encourages me to continue my career as an academic researcher in computer science. He is also a generous friend, which I always appreciate from my heart. Without his consistent and impressive kindness, all my papers that have been published or submitted, as well as this thesis, would not have been completed or written.

I am also sincerely grateful to Prof. Xiaokui Xiao for providing me two research internships at Nanyang Technological University, where he taught me useful research skills and gave insightful comments from rich experience on my work. Acknowledge also goes to Dr. Nicholas Jing Yuan, who provided me an internship at Microsoft Research Asia and helped to improve my work.

The DKE group has provided a very nice working environment for the past three years. I have benefited greatly from the discussions with a number of faculty members in the group. It is my pleasure to specially acknowledge Prof. Yufei Tao, Dr. Mohamed Sharaf, Prof. Shazia Sadiq, Dr. Helen Huang, Prof. Hengtao Shen, Prof. Xue Li, Dr. Hongzhi Yin, and Dr. Sen Wang. In my daily work I have been blessed with a friendly and cheerful group of fellow students. A big thank you goes to Dr. Han Su, Dr. Haozhou Wang, Dr. Jialong Han, Mr. Peng Wang, Dr. Wen Hua and Ms. Ruojing Zhang, I really couldn't remember how many times exactly we have been together to explore and enjoy the delicious food in Brisbane, especially the favorite hotpot. I will also give thanks to Mr. Junhao Gan, Mr. Lei Li, and Mr. Xingzhong Du for our great "lunch routine", in which various thoughts and ideas are inspired and shared quite often. Thanks also go to my office mates, Mr. Abdullah Albarrak and Mrs. Weiqing Wang, who offer me delightful mood to work in every single

day.

Of course, great appreciation is also owed to the Chinese Scholarship Council, which has funded me all the way through my PhD study.

Last but not least, my deepest gratitude goes to my family. They have constituted the most wonderful family I can ever imagine, and being a part of it is the luckiest possession in my life. My parents who raised me with caring and gently love deserve special mention for their inseparable support. It is their support and love that encourage me to pursue my dreams.

Keywords

spatial keyword query, spatio-temporal database, trajectory database, point-of-interest, nearest neighbour, shortest path, road network, algorithm, performance

Australian and New Zealand Standard Research Classifications (ANZSRC)

ANZSRC code: 080604, Database Management, 100%

Fields of Research (FoR) Classification

FoR code: 0806, Information Systems, 100%

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, mobile devices such as smartphones and tablets are gradually predominating the transformation of the web from desktop-based age to mobility-based age, resulting in a large-scale collection of movement data. The increasing development of GPS-based technologies enables to provide accurate geo-location and time information to a GPS receiver equipped in most mobile devices. Another important development on geo-positioning technologies is the construction of the communication infrastructures, such as 3G, 4G and WiFi, used by some mobile devices. By sending the radio signals, the real-time location of the device is reported to the base stations or cell towers. This type of technology can be applied in both indoor and outdoor environment but suffers the problem of less accuracy in positioning than GPS does. With such proliferation of geo-positioning technologies, accurate user positioning is becoming more and more available. Since space is one of the most important aspects of all real-world phenomena, its nature can be exploited by any application that tries to model entities in the real world, especially the movements of human beings.

With the developments outlined above, the data collected from mobile devices offers an unprecedented amount of information that can be used to help us understand the behavior of moving objects, which hastens the emergence of research on spatial (temporal) database in the past decades. Effective and efficient technologies to manage such spatial data are in high demand and can be found in a great number of important applications, such as trip planning, transportation management, geographical information systems, moving object tracking, sensor networks, environment monitoring, just to name a few.

Recent years have also witnessed the proliferation of location-based services, there is a clear trend that an increasing amount of spatial data associated with the textual information is available in many applications. Such objects contain information on both spatial and textual dimensions are called the geo-textual objects. To provide better user experience, the location-based services maintain such geo-textual objects to answer user queries w.r.t. user-specified location. For example, a person wants to find a restaurant within 10 minutes walking distance. With the foundation of research achievements on spatial database, such queries, known as spatial keyword queries, which find the top-k objects of interest in terms of both spatial proximity and textual relevance to the query, have been studied under a variety of problem settings.

1.1 Background

1.1.1 Geo-textual Data

The web objects have an associated geo-location and a textual description are called the geo-textual data. For example, the location information as well as concise textual descriptions of some businesses (e.g., restaurants, hotels) can be easily found in online local search services (e.g., yellow pages). Another example is the GPS navigation system, where a PoI (Point-of-Interest) is a specific point location that someone may find useful or interesting, and is usually annotated with textual information (e.g., descriptions and users' reviews). By marking a PoI as destination on the map, users are able to plan a trip with suggestions. Moreover, in many social network services (e.g., Facebook, Flickr), a huge number of photographs are accumulated everyday that are geo-tagged by users. These uploaded photographs are usually associated with multiple text labels. This leads to the fact that massive amounts of objects become available on the web. Such geo-textual objects include restaurants, petrol stations, shopping malls, parking slots, universities, hotels, entertainment services, public transport, etc. Formally, a geo-textual object is in the form of $o = (l, \Phi)$, where l is the location of o (represented by a point or shape such as a rectangle) and Φ is the textual description on o (represented by a set of keywords).

The source of geo-textual data can be mainly classified into two categories. First, they could be obtained from location-based services. Such objects are called the static geo-textual objects since

most of them do not update quickly. For example, in DianPing, many PoIs are being associated with users' descriptions and reviews. Second, they can also be extracted from geo-tagged web contents. These objects are called the streaming geo-textual objects since they are accumulated everyday by geo-tagging the user generated contents. For example, the geo-tagged photos in Flickr and the geo-located tweets in Twitter. It is worth to note that these location-based services allow people to check-in at the location of PoI. As a result, the traditional trajectory databases are redefined and enriched by attaching activity or semantic meanings. Such check-in sequences of geo-textual objects that contain the information about the semantic meanings of user behaviour (e.g., activity or place name) at particular places are called activity trajectories, which also find various applications in location-based services.

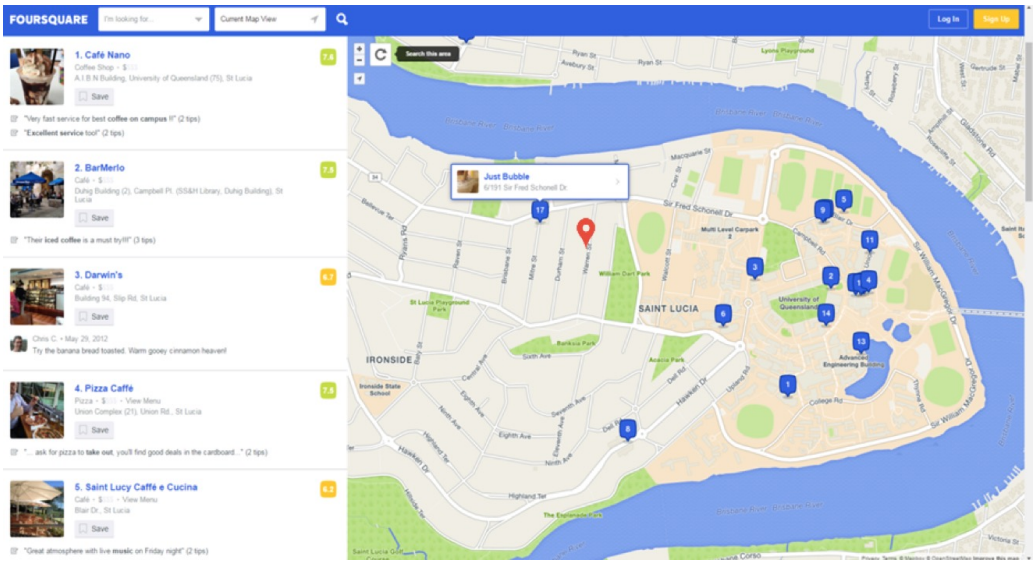


FIGURE 1.1: Example of Spatial Keyword Queries

1.1.2 Spatial Keyword Queries

The availability of a substantial amount of geo-textual objects leads to that a considerable fraction of web queries on search engines have local intent and target on these objects. The prototype spatial keyword query takes a location and a set of keywords as arguments and returns objects that are spatially and textually relevant to these arguments. It is easy to see the location in query represents the local intent, and the keywords describe user's attentive. For example in Figure 1.1, the user wants to

find a cafe shop near to his home. First, he inputs the location of his home and types in the keyword "cafe", then the service just returns all the nearby cafe shops to him as reference and reports the "Just Bubble" as the closest one.

Due to the importance of geographical space to our daily lives, spatial keyword queries are receiving increasing interest in the research community where a range of techniques have been proposed for efficient query processing. Four types of spatial keyword queries, classified based on their way of specifying the spatial and textual predicates, are receiving particular attention, namely the Boolean kNN query, the top-k kNN query, the Boolean range query and the top-k range query [23, 125].

- **Boolean kNN query**: It aims to retrieve the k objects nearest to the query location (represented by a point) and each object's text description contains all the query keywords.
- **Top-k kNN query**: It proposes to retrieve the k objects with the highest ranking scores, measured by a weighted combination of their distances to the query location and the textual similarity between their textual descriptions and query keywords.
- **Boolean range query**: It aims to retrieve all objects whose textual description contains all the query keywords and whose location is within the query region.
- **Top-k range query**: It proposes to retrieve the k objects whose location is within the query region and has the highest textual relevance to the query keywords.

In spite of the remarkable progress on spatial keyword queries, unsolved problems, such as the queries with a target of more diversified data types (e.g., activity trajectories), underlying spaces (e.g., road network) and query semantics, still remain and are open to study.

1.2 Problem Statement

With the rapid development of GPS-enabled smart mobile devices and location-based services, there is a clear trend that objects are increasingly being geo-tagged. To provide better user experience, these services maintain location-related information to answer user queries w.r.t. user-specified location. In addition to the spatial characteristics, a user may also have a specific requirement on the description

of the object such as “restaurant”, “hotel”, “petrol station”, etc. For example, a person wants to find a restaurant within 10 minutes walking distance. Such queries, known as spatial keyword queries, have been extensively studied in recent years [30, 59, 64, 80, 83, 121, 125, 131]. Typically, given a set of geo-textual objects, a spatial keyword query takes a location and a set of keywords as arguments and returns top-k objects that are spatially and textually relevant to these arguments. In this thesis, a set of new query predicates has been defined with a target of more diversified data types (e.g., activity trajectories), underlying spaces (e.g., road network) and query semantics, by which users can acquire their interested results more easily and effectively.

As activity trajectory contains the information about user historical behaviors, searching them by keywords is able to provide the users with reasonable suggestions on trip planning. Therefore, in the first work, I study the keyword-oriented query processing on activity trajectories. As follows, if a user is moving on a route suggested by trip planning applications, it is necessary to support the spatial keyword queries in a continuous manner. Thus I investigate the keyword-aware continuous kNN query processing on road networks in the second work. Finally, I explore the clue-based route search problem as a supplement for the trip planning application, which allows the users to provide clues for searching the intended route.

1.2.1 Keyword-oriented Queries on Activity Trajectories

In many location-based social network applications, increasing volumes of geo-textual objects are becoming available on the web that represent Point-of-Interest (PoIs). These applications allow people to check-in at these PoIs, each having a spatial location and a semantic description. As a result, the traditional trajectory databases are redefined and enriched by attaching activity or semantic meanings. In this thesis, the term *activity trajectory* is used to represent this check-in sequence of geo-textual objects that contain the information about the semantic meanings of user behaviour (e.g., activity or place name) at particular places. However, most existing work mainly focuses on querying the PoIs, only a few work considers trajectory data. To this end, I study the problem of searching activity trajectories only by keywords, which is very useful in many location-based services such as intelligent tourist guide and trip planning. This work proposes to support efficient processing keyword-oriented queries on activity trajectory, wherein given a set of query keywords, the output is the top-k trajectory

segments with "closely matched" keywords and short travel distances.

1.2.2 Keyword-aware Continuous *k*NN Queries on Road Networks

With the rapid development of GPS-enabled smart mobile devices and location-based services, there is a clear trend that objects are increasingly being geo-tagged. Many real-world applications have the requirements to support the continuous k nearest neighbour (CkNN) queries, or also known as moving k nearest neighbour queries. Most of the existing work adopts the idea of "safe region" where all the inside points share the same kNN results, thus reducing the query processing cost in terms of both computation and communication. In this thesis, I study the *keyword-aware continuous k nearest neighbour* (𝕂CkNN) on road networks, which computes the kNN results that contain the query keywords and maintains the results in a continuous manner. For high frequency keywords, a *window sliding approach* is adopted to build a dominance interval with low costs. For low frequency keywords, a *path-based dominance updating approach* is proposed to resolve the dominance region on road network, which guarantees the validity of the current 𝕂kNN results and significantly reduces the computation and communication costs.

1.2.3 Efficient Clue-based Route Search on Road Networks

With the development of location-based services, keyword queries are also combined with travel planning in commercial applications such as GPS navigation systems or online map services. The existing solutions (e.g., [16, 63, 93]) for trip planning or route search are dealing with the scenarios when a user wants to visit a sequence of PoIs, each of which contains a user specified keyword. In this thesis, I investigate the problem of clue-based route search (CRS), which allows a user to provide clues on textual and spatial context within the route. Formally, a CRS query is defined over a road network G, and the input to the query consists of a source vertex v_q and a sequence of clues $C = \{\mu_i\}$, where each clue μ_i contains a query keyword w_i and a user expected network distance d_i. The query returns a path $\mathcal{P}$ in G starting at v_q, such that $\mathcal{P}$ passes through vertices (PoIs) that contain all the query keywords and comply the same keyword order as in C. In the meantime, it has the minimum matching distance, which is defined as the degree of satisfaction of the user to $\mathcal{P}$. To find an optimal route such that it covers a set of query keywords in a given specific order, and the matching distance

is minimized, several efficient algorithms and index structures are proposed to speed up the search process.

1.3 Challenges and Contributions

Different with traditional queries on spatial databases, processing spatial keyword queries on large scale geo-textual data can be more challenging due to the following reasons:

- In most work (e.g., [30, 33]), the spatial keyword search requires the users to specify an exact location in their queries such that the returned geo-textual objects are close to these locations. However, we observe that in many circumstances a user does not always have a preferred location in advance. This is usually the case when a tourist plans a trip to a city and has not decided where to live. Without a query location, the search area cannot be localized such that processing the query can be even more challenging. The only known proposals (e.g., [42, 118]) that do not take query location into consideration are called m closest queries, which aim to find m closest objects that match the query keywords and their distance diameter is minimized. Therefore, more such queries are still open to study.

- Many real-world applications have the requirements to support the continuous queries, but existing techniques (e.g., [59, 131]) for the static spatial keyword query are not directly applicable for continuous spatial keyword query. Although some existing works adopt safe region technique to reduce the query processing cost [62, 74], they fall short either in the region construction overhead or validation overhead. Therefore, on-going efforts are required to meliorate the user experience by improving the continuous spatial keyword query processing efficiency.

- The distance metric for evaluating the goodness of geo-textual objects on spatial dimension also differs across existing studies. Many existing proposals assume geo-textual objects locate in Euclidean space, which might be inappropriate especially in urban areas where the movements of users are constrained by the road network. However, computing the network distance between objects in scalable networks is a complicated problem, as well as finding the k nearest neighbors. Therefore, novel indexing and querying solutions for spatial keyword queries on road networks should be invented to keep the computation cost tractable.

In the following, I will briefly describe our contributions in addressing the above challenges.

1.3.1 Keyword-oriented Queries on Activity Trajectories

In this thesis, I study the problem of searching activity trajectories by keywords. Given a set of query keywords, the *keyword-oriented query for activity trajectory* (KOAT) returns k trajectories that contain the most relevant keywords to the query and yield the least travel effort in the meantime. The main difference between KOAT and conventional spatial keyword query is that no query location exists in KOAT, which means the search area cannot be localized. To capture the travel effort in the context of query keywords, a novel score function, called *spatio-textual ranking function*, is first defined. Then a hybrid index structure called GiKi is developed to organize the trajectories hierarchically, which enables pruning the search space by spatial and textual similarity simultaneously. Finally an efficient search algorithm and fast evaluation of the value of *spatio-textual ranking function* are proposed. The results of our empirical studies based on real check-in datasets demonstrate that our proposed indices and algorithms can achieve good scalability.

This research [121] was published in *IEEE International Conference on Data Engineering* (**ICDE**) 2015.

1.3.2 Keyword-aware Continuous kNN Queries on Road Networks

In this thesis, I study the problem of *keyword-aware continuous k nearest neighbour* (𝕂CkNN) search on road networks, which computes the k nearest vertices that contain the query keywords issued by a moving object and maintains the results continuously as the object is moving on the road network. This work proposes a framework, called a Labelling AppRoach for Continuous kNN query (*LARC*), on road networks to cope with 𝕂CkNN query efficiently. First, a keyword-based pivot tree index is built to improve the efficiency of boolean spatial keyword queries by avoiding massive network traversals and sequential probe of keywords. To reduce the communication cost, the concepts of dominance interval and region are developed on road network, which share the similar intuition with safe region for processing continuous queries in Euclidean space but are more complicated with a dedicated design. The empirical studies of our experiments have verified the superiority of our proposed solution in all aspects of index size, communication cost and computation time.

This research [123] was published in *IEEE International Conference on Data Engineering* (**ICDE**) 2016.

1.3.3 Efficient Clue-based Route Search on Road Networks

In this thesis, I investigate the problem of clue-based route search (CRS), which allows a user to provide clues on keywords and spatial relationships. First, a greedy algorithm and a dynamic programming algorithm are proposed as baselines. To improve efficiency, a branch-and-bound algorithm is developed that prunes unnecessary vertices in query processing. In order to quickly locate candidate, an AB-tree is proposed that stores both the distance and keyword information in tree structure. To further reduce the index size, a PB-tree is constructed by utilizing the virtue of 2-hop label index to pinpoint the candidate. Extensive experiments are conducted and verify the superiority of our algorithms and index structures.

1.4 Thesis Outline

The rest of this thesis is organized as follows: In Chapter 2 I review the related work. Chapter 3 introduces the keyword-oriented queries on activity trajectories. I discuss the keyword-aware continuous k nearest neighbor queries on road network in Chapter 4. In Chapter 5 I describe the problem of processing clue-based route search on road networks. Finally, I conclude this thesis in Chapter 6.

Chapter 2

Literature Review

In this chapter, we introduce an overview of some important types of queries, including spatial queries and spatial keyword queries, which are related to this thesis with various settings. First of all, we provide the related work on spatial queries in Section 2.1, since the techniques of spatial database are the basis of spatial keyword queries. More specifically, we review the literature on kNN queries, trajectory query processing and shortest path and distance queries. In Section 2.2, we provide a brief description of the existing techniques for various spatial keyword queries, where we review the related work on top-k spatial keyword queries, continuous spatial keyword queries and travel route search.

2.1 Spatial Queries

In this section, we introduce several classical spatial queries, including kNN queries, trajectory similarity search and shortest path and distance queries.

2.1.1 Nearest Neighbour Queries

Queries on Euclidean Space

In recent years, a number of work has been proposed for efficient processing k nearest neighbor queries on Euclidean space. Given a set of objects and a query point, this query finds the k nearest objects to this given point in space. Most of the methods adopt a multi-dimensional index structures, such as R-tree and its variants [27, 46, 84].

N. Roussopoulos et al. [84] present an efficient branch-and-bound R-tree traversal algorithm to find the nearest neighbor object to a query point, and then generalize it to finding the k nearest neighbors. In this work, two important metrics for an optimistic and a pessimistic search ordering strategy as well as for pruning are discussed. The optimistic metric, which is called *minDist*, is defined as the minimum possible distance between the query point and its NN. On the other hand, the pessimistic metric, called *minmaxDist*, is defined as the furthest possible distance where the NN of the query point can reside. The heuristics used in the algorithm are based on orderings of the *minDist* and *minmaxDist* metrics, and a depth first traversal is adopted to find the NN to a query point in an R-tree.

K. Cheung et al. [27] propose an improved nearest neighbor search algorithm on the R-tree and its variants. The improvement lies in the removal of two heuristics that use the metrics *minDist* and *minmaxDist*, since the calculation of *minmaxDist* is computationally expensive and has a complexity of $O(d)$. Moreover, it turns out that these two metrics do not actually increase the pruning power, so the calculation of *minmaxDist* is indeed not necessary. Instead, an improved algorithm that does not make use of *minmaxDist* is proposed, which is shown to be at least as powerful as previous one in the pruning capability.

G. Hjaltason et al. [46] propose a best first search paradigm to process the nearest neighbor query. Different with depth first traversal, the next MBR or subtree to be expanded is always the one with smallest *minDist* to the query among all those to be visited. Therefore, a priority queue is kept to maintain the entries of MBRs to be expanded with their *minDist* as the sorting key. Initially, the root is pushed into the priority queue. At each step, the top element in this priority queue is popped for processing, and its child MBRs or objects are either pushed into this priority queue, or taken as candidates. This search process terminates when the distance of the top element in priority queue is greater than current k-th candidate, or the queue is empty. This algorithm is shown to be efficient since it only visits necessary MBRs. However, this method suffers from the buffer thrashing if the heap becomes larger than available memory.

Queries on Road Networks

Nearest neighbor queries have received significant attention in spatial database community in the past decade. Recently, research focus is also extended to a road network scenario by taking the network distance as distance metric. Normally, a road network is modeled as a graph $G = (V, E)$, where a vertex $v \in V$ denotes a road intersection, an edge $e \in E$ denotes the road segment between

two intersections, and the weight of each edge is the network distance. Given a query node and a road network, the k nearest neighbor query on road network finds k nodes that are closest to the query node in terms of network distance [26, 47, 51, 58, 76, 92, 94, 109].

Shahabi et al. [92] propose graph embedding techniques to deal with nearest neighbor queries. In this work, a road network is transformed into a high-dimensional Euclidean space such that the techniques used in Euclidean space can be well applied. They show that the distance in the embedding space is a good approximation of the actual distance.

Papadias et al. [76] propose an architecture that integrates network and Euclidean information, capturing pragmatic constraints. Based on this architecture, they develop a Euclidean restriction and a network expansion framework to efficiently prune the search space by taking advantage of location and connectivity.

Jensen et al. [51] propose a general spatial-temporal framework for NN queries in a road network which is represented by a graph. In this framework, some algorithms similar to Dijkstra's algorithm are used in order to perform online computation of network distance from a query node to an object.

Kolahdouzan et al. [58] propose VN^3 to partition the network into cells by the Network Voronoi Diagram by pre-computation. They index these cells by an R-tree in the Euclidean space, thus the problem of finding first NN is reduced to a point location problem. For kNN queries, they also pre-compute the distances between border points of adjacent cells.

Yiu et al. [109] study the problem of aggregate nearest neighbor queries, which returns the object that minimizes an aggregate distance function with respect to a set of query points. They consider alternative aggregate functions and techniques that utilize Euclidean distance bounds, spatial access methods, and/or network distance materialization structures.

Hu et al. [47] propose an approach that indexes the network topology based on a novel network reduction technique. It simplifies the network by replacing the graph topology with a set of interconnected tree-based structures, therefore a new NN algorithm is developed on these tree-based structures following a predetermined tree path to avoid costly network expansion.

Chen et al. [26] address the problem of monitoring the k nearest neighbors to a dynamically changing path in road networks. Given a destination where a user is going to, this query returns the kNN with respect to the shortest path connecting the destination and the users current location, and thus provides a list of nearest candidates for reference by considering the whole coming journey.

In addition, nearest neighbor search on high dimensional space has also been extensively studied. Jagadish et al. [50, 110] present an efficient method, called iDistance, for K-nearest neighbour search in a high dimensional space, which partitions the data and selects a reference point for each partition. The data points in each partition are transformed into a single dimensional value based on their similarity with respect to the reference point. By applying iDistance, the kNN search is performed by using one-dimension range search.

2.1.2 Trajectory Query Processing

Trajectory Storage and Indexing

The most popular and classical data structure for spatial data is R-tree [43]. However, directly applying R-tree on spatial dimension and temporal dimension for trajectory data is not good enough. Therefore, many optimizations are proposed to make the R-tree based structures support the trajectory data. TB-tree [78] uses a hybrid tree structure to store and index both spatial and temporal information, but is not adequate to process long trajectories, which can make the bounding rectangles very large. TPR-tree [95] and TPR*-tree [99] invoke the predication model to predict the future positions of moving objects. On the other hand, partitioning trajectories into segments become a new way to improve the query performance. Rasetic et al. [81] derive an analytical cost model to control the splitting process for a trajectory into segments based on given query. SETI [19] stores trajectory segments in a 3D R-tree for their spatial information. Meanwhile, SETI indexes the temporal information by using one dimensional time lines to increase the search performance. PIST [13] partitions the sample points rather than partitioning the trajectories. Similarly, the TrajStore [32] propose a new adaptive storage system that indexes the trajectory data based on quad-tree index and clustering methods. These algorithms or systems are designed based for disk based systems, which means I/O cost between hard disk to memory is the main concern. However, there is no I/O cost in in-memory based systems. Thus these algorithms and systems are not feasible for in-memory systems.

Trajectory Similarity Search

Given a query trajectory and a trajectory database, the trajectory similarity search returns trajectories with smallest distances to the query trajectory. The distance between the query trajectory and target trajectories is measured by a distance function [4, 24, 25, 36, 100, 108].

Discrete Fourier Transform (DFT) [4] is the pioneering work of this area, which transforms trajectories to multi-dimensional points, and then computes their Euclidean distances in feature space. Faloutsos et al. [36] extend DFT to support subsequence matching. However, these methods require the trajectories to have the exact same length.

Dynamic Time Warping (DTW) [108] is another well-known algorithm for finding similar trajectory patterns between two trajectories. It removes the restriction of DFT by allowing time-shifting in the comparison of trajectories. DTW uses a recursive method to search all possible point combinations between two trajectories for the one with minimal distance, and can be easily converted to dynamic programming.

Longest Common Subsequence (LCSS) [100] is a robust similarity measure for processing low quality trajectory. To detect matching points, a threshold is applied, and if the threshold is greater than the distance between two points, these two points are considered to be a match. The intuition of LCSS is that it allows some unmatched sample points to be matched under some flexibilities.

Chen et al. [24] propose the ERP distance, which utilizes L_1-norm as distance measure. Therefore, efficient pruning can be well deployed by using the metric properties, which is a significant advantage over DTW and LCSS. Chen et al. [25] propose another edit distance based similarity measure EDR, which is similar to LCSS in using a threshold to determine if two points are matched when considering penalties to gaps.

Despite the research on trajectory similarity measure, some applications based on trajectory similarity have also been proposed [122, 128, 130].

Zheng et al. [130] aim to mine interesting locations and classical travel sequences by using the user generated GPS trajectories. They regard an individual's visit to a location as a link from the individual to the location, and weight these links in terms of users' travel experiences in various regions. A HITS-based model is proposed to infer a user's travel experience and the interest of a location, and then they detect the classical travel sequences in a specified region using location interests and user's travel experience.

Zheng et al. [128] represent the uncertainty of the objects moving along road networks as time-dependent probability distribution functions. Given a set of uncertain trajectories, they construct an index called uncertan trajectory hierarchy to organise the trajectories. Based on this index, they propose efficient algorithms for processing spatio-temporal range queries.

Yuan et al. [112, 113] propose a smart driving direction system to leverage the intelligence of experienced drivers. This system employs GPS-equipped taxis as mobile sensors to probe the traffic rhythm of a city and taxi drivers intelligence in choosing driving directions in the physical world. They propose a time-dependent landmark graph to model the dynamic traffic pattern as well as the intelligence of experienced drivers so as to provide a user with the practically fastest route to a given destination at a given departure time. Then, a Variance-Entropy-Based Clustering approach is devised to estimate the distribution of travel time between two landmarks in different time slots. Based on this graph, they design a two-stage routing algorithm to compute the practically fastest and customized route for end users.

Zheng et al. [122] propose to model the users' trajectories in public transportation systems. The goal is to estimate the similarity between users' travel patterns according to their travel smart card data. The core of this proposal is that they define a travel spatial-temporal similarity function to measure the spatial range and temporal similarity between users. In addition, they also propose a hybrid index structure, which integrates inverted files and cluster-based partitioning, to allow for efficient retrieval of the top-K most similar users.

Travel spatial pattern similarity could be analogically viewed as a problem of weighted set-based string similarity. Such related research has been conducted in [8,20,44,45,96,105], Marios et al. [44] concentrate on weighted similarity functions like TF/IDF, and introduce variants that are well suited for set similarity selections in a relational database context.

Besides, travel temporal pattern similarity could also be treated as problem of measuring distance between distributions. [67,77,85,86] investigate the properties of a metric between two distributions, the Earth Movers Distance (EMD), which is based on the minimal cost that must be paid to transform one distribution into the other. EMD is more robust than histogram matching techniques, in that it can operate on variable-length representations of the distributions that avoid quantization and other binning problems typical of histograms. When used to compare distributions with the same overall mass, the EMD is a true metric.

2.1.3 Shortest Path and Distance Queries

The shortest path and distance query is a significant problem that finds applications in various commercial navigation products and map services. Given a graph G, the input of the query are a source s and a destination t, the output is the shortest path between s and t. [3, 9–11, 34, 38, 39, 70, 82, 87, 88, 90, 101, 103, 104, 120, 133]

The classic solution for this problem is the Dijkstra's algorithm [34]. For a given source node in the graph, the algorithm finds the shortest path between that node and every other node. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. Given two vertices in graph, the bidirectional Dijkstra's algorithm [79] proceeds two instances of Dijkstra's algorithm simultaneously. Each instance traverses the vertices in graph in ascending order of the distance to these two query nodes, and maintains a minimum spanning tree for every visited vertex. These two traversal terminate when they meet at a vertex in the middle, then the corresponding shortest path and network distance are reported.

Bast et al. [10,11] introduce a concept of transit nodes, which is used as a means for preprocessing a road network, with each node given coordinates and each edge given a travel time or length, in order that the shortest path queries can be answered fast. It first imposes a grid on the road networks, and then precomputes the shortest paths from within each grid cell to a set of vertices that are considered important for the cell. With the pre-computed distances, it can efficiently compute the distance between any two vertices in road networks.

CH [38] is an indexing technique for graph that imposes a total order on the nodes in graph according to their relative importance. A hierarchy is constructed by iteratively contracting the least important node that CH replaces the shortest paths by shortcuts. Then CH pre-computes the distances between various nodes based on the total order. CH then utilizes the pre-computed distances to accelerate shortest path and distance queries by applying a bidirectional dikjstra's algorithm.

Samet et al. [87, 89] propose a framework called Spatially Induced Linkage Cognizance (SILC) that uses path coherence between the shortest paths and the spatial locations of nodes in the network, thus resulting in an encoding that is compact in representation and fast in path and distance retrievals. SILC first precomputes the allpair shortest paths and then stores them in a concise form in order that

the query can be answered efficiently.

Sankaranarayanan et al. [90] propose a technique called Path-Coherent Pairs Decomposition (PCPD). PCPD shares the similar intuition with SILC that precomputes and stores all shortest paths among the nodes in the graph. PCPD introduce three approximate oracles for spatial networks that are able to answer distance queries in an approximate manner.

To the best of our knowledge, one of the most notable recent developments is the emergence of practical *2-hop labeling* methods [1, 2, 5, 6, 54] for $\mathcal{DO}$ on large networks. It constructs labels for vertices such that a distance query for any vertex pair u and v can be answered by only looking up the common labels of u and v.

Abraham et al. [1] propose a hub-based labeling algorithm by using the contraction hierarchies algorithm [38] for preprocessing. They introduce two label compression techniques, a distance oracle to accelerate long-range (and random) queries, and the index-free variant of the algorithm.

PLL [6] proposes an efficient method for exact shortest path and distance queries based on distance labeling to vertices. The algorithm conducts breadth-first search (BFS) from all the vertices with pruning strategies. Though the algorithm is simple, the pruning surprisingly reduce the search space and the labels, resulting in fast preprocessing time, small index size and fast query time.

HopDB [54] proposes a novel 2-hop labeling indexing method for P2P distance querying on unweighted directed graphs, and have developed I/O efficient algorithms for index construction when the given graph and the index cannot fit in main memory. With scalable indexing complexities, this method performs well on different types of scale-free networks and can handle graphs many times larger than existing methods.

Akiba et al. [5] propose a new framework called highway-based labelings and an algorithm named pruned highway labeling for preprocessing. It exploits the highway structure in road networks, decomposes a graph into shortest paths and stores distances from each vertex to the shortest paths in each label. In addition, it computes small labels for highway-based labelings.

2.2 Spatial Keyword Queries

Searching geo-textual objects with query location and keywords has gained increasing attention recently due to the popularity of location-based services. A prototypical spatial keyword query takes a

set of keywords and a location as input and finds geo-textual objects that are spatially and textually relevant. In literature, a wide range of work has already been proposed that study different aspects of spatial keyword search [15, 17, 18, 23, 30, 33, 37, 42, 61, 64, 116, 118, 119]. In this section, we introduce three types of spatial keyword queries: top-k spatial keyword queries, continuous spatial keyword queries and travel route search.

2.2.1 Top-k Spatial Keyword Queries

Top-k query problem is an enduring research point [35, 49, 68, 97]. Ilyas et al. [49] describe and classify top-k processing techniques in relational databases. The NRA algorithm [35] [68] tries to compute a "range" of possible scores for each object since the lack of random access prevents computing an exact score for each seen object. By allowing random access to the underlying data sources it triggers the need for cost models to optimize the number of random and sorted accesses. On top of techniques for top-k queries, we introduce top-k spatial keyword queries on Euclidean space and road networks.

Queries on Euclidean Space

In Euclidean space, IR^2-tree [33] integrates signature files and *R-tree* to answer boolean keyword queries. In IR^2-tree, a signature is added to each node of this tree to represent the textual content of all spatial objects in the subtree. An efficient incremental algorithm is presented to answer top-k spatial keyword queries using the IR^2-Tree by accessing a minimal portion of the tree nodes.

IR-tree [30] is an R-tree augmented with inverted files that supports the ranking of objects based on a score function of spatial distance and text relevancy. It proposes several hybrid indexing approaches and encompasses algorithms that utilize the proposed indexes for computing the top-k query, and it is capable of taking into account both text relevancy and location proximity to prune the search space at query time.

Cao et al. [17] proposes a location-aware top-k prestige-based text retrieval (LkPT) query, to retrieve the top-k spatial web objects ranked according to both prestige-based text relevance (PR) and location proximity. They develop two baseline algorithms and propose two new algorithms to process the LkPT query efficiently.

Chen et al. [23] provide an all-round survey of 12 state-of-art geo-textual indices and proposes

a benchmark that enables the comparison of the spatial keyword query performance. They consider the support for three fundamental kinds of geo-textual queries: Boolean kNN query, top-k kNN query and boolean range query.

Zhang et al. [118, 119] proposes the m closet keyword query (mCK query) which aims to find the closest objects that match the query keywords and their distance diameter is minimized. They introduce a new index called the bR*-tree, which is an extension of the R*-tree. Based on bR*-tree, they exploit a priori-based search strategies to effectively reduce the search space. They also propose two monotone constraints, namely the distance mutex and keyword mutex, as a priori properties to facilitate effective pruning. Recently, Guo et al. [42] first prove that mCK is NP-hard. Then they propose approximation algorithms to solve the mCK query with a ratio of $(\frac{2}{\sqrt{3}} + \epsilon)$.

Cao et al. [18] propose a collective spatial keyword query, in which a different semantics is taken such that the group of objects in the result covers the query keywords and has the lowest cost. They study two particular instances of the problem, both of which are NP-complete. They develop approximation algorithms with provable approximation bounds and exact algorithms to solve the two problems.

Li et al. [64] study the problem of direction-aware spatial keyword search, which aims at finding the k nearest neighbors to the query that contain all input keywords and satisfy the direction constraint. They devise novel direction-aware indexing structures to prune unnecessary directions. They further develop effective pruning techniques and search algorithms to efficiently answer a direction-aware query. As users may dynamically change their search directions, they propose to incrementally answer the queries.

Zheng et al. [124] study the problem of activity trajectory similarity query (ATSO). It returns k trajectories that cover the query activities and yield the shortest minimum match distance from given a sequence of query points. In this query, each point contains several activities such as sports, shopping etc. They developed a hybrid grid index structure, called GAT, to organise the trajectory segments and activities hierarchically. By using GAT, the pruning speed can be increased significantly, which means both I/O and CPU remain low. Finally, the authors extend their method to support ordersensitive ATSO queries.

Zheng et al. [127] propose a top-k spatial keyword query for activity trajectories, with the objective to find a set of trajectories that are not only close geographically but also meet the requirements of

the query semantically. They provide a novel similarity function, hybird indexing structure, efficient search algorithm and further optimizations to answer the query efficiently.

Queries on Road Networks

ROAD [59, 60] is proposed for spatial object search on road networks. It is extensible to diverse object types and efficient for processing various location-dependent spatial queries, as it maintains objects separately from an underlying network and adopts an effective search space pruning technique. Based on network traversal and object lookup, ROAD organizes the road network as a hierarchy of subgraphs, and connects them by adding shortcuts to accelerate network traversals and provide quick object lookups. To manage those shortcuts and object abstracts, two cooperating indices, namely, Route Overlay and Association Directory are devised. By using network expansion, the subgraphs without intended object are pruned out.

Rocha et al. [83] introduce top-k spatial keyword queries on road networks. Given a query location and a set of query keywords, a top-k spatial keyword query on road networks returns the k best spatio-textual objects ranked in terms of both textual similarity to the query keywords and shortest path to the query location. They present a basic approach to process the queries by combining state-of-the-art techniques. Then, they present an enhanced approach that indexes the edges of the road network, and permits identifying and retrieving the objects relevant to the query efficiently. Finally, they propose an overlay approach that groups objects in regions, taking in account the textual similarity among the objects, and permits computing an upper-bound score for all objects in the region. Consequently, regions whose the upper-bound score is smaller or equal the score of the k-th object already found can be pruned, improving the performance.

G-tree [131, 132] adopts a graph partitioning approach to form a height-balanced and scalable index namely G-tree. Within each subgraph, a distance matrix is kept, and for any two subgraphs, the distances between all borders of them are stored as well. Based on these distances, it efficiently computes the distance between query vertex and target vertices or tree nodes. The basis for this framework is an assembly-based method to calculate the shortest-path distances between two vertices. Based on the assembly-based method, efficient search algorithms to answer k nearest neighbor queries and keyword-based k nearest neighbor queries are developed.

Zhang et al. [117] study the problem of diversified spatial keyword search on road networks

which considers both the relevance and the spatial diversity of the results. They propose an efficient signature-based inverted indexing technique to facilitate the spatial keyword query processing on road networks. Then they further develop an efficient diversified spatial keyword search algorithm by taking advantage of spatial keyword pruning and diversity pruning techniques.

Luo et al. [69] develop a distributed solution to answering spatial keyword queries on road networks. They propose an operation for answering spatial keyword queries and reduce the problem of answering a query into computing a function of such operations. They propose a new distributed index that enables each machine to independently evaluate the operation on its network fragment in a distributed environment.

Jiang et al. [53] adopt 2-hop label for handling the distance query for kNN problem on large networks. For low frequent keywords, they propose a forward search component by utilizing the inverted lists. For high frequent keywords, the propose a forward backward search component by constructing a 2-hop label backward index and a keyword lookup tree index. Finally, they adopt a hybrid approach to combine the forward search and forward backward search together.

2.2.2 Continuous Spatial Keyword Queries

There are quite a number of studies on CkNN/MkNN queries. YPK-CNN [111], CPM [71] and GMA [72] study the problem of finding nearest moving objects (e.g., taxies) to a location and focus on dealing with frequent updates of moving objects.

YPK-CNN [111] propose two efficient and scalable algorithms using grid indices, one for indexing objects and the other for queries. For each approach, a cost model is developed, and a detailed analysis along with the respective applicability are presented.

CPM [71] investigates the problem of monitoring continuous NN queries over moving objects. The goal of the query processor is to constantly return the results of all queries, as location updates by from both the objects and the queries. It proposes an efficient algorithm based on a conceptual partitioning of the space around each query in order to restrict the result maintenance and nearest neighbor computation to objects that lie in the vicinity of the query. The core idea is to retrieve the first-time results of incoming queries, and the new results of existing queries that change location. It produces and stores book-keeping information to facilitate fast update handling.

GMA [72] studies k-NN monitoring on road networks, where the network distance between a query and an object is determined by the length of the shortest path connecting them. It proposes two methods that can handle arbitrary object and query moving patterns, as well as fluctuations of edge weights. The first one maintains the query results by processing only updates that may invalidate the current nearest neighbor sets. The second method follows the shared execution paradigm to reduce the processing time. In particular, it groups together the queries that fall in the path between two consecutive intersections in the network, and produces their results by monitoring the nearest neighbor sets of these intersections.

CkNN [98] finds the kNN for every single point on a predefined linear trajectory. This is achieved by identifying all influence points on the trajectory. It first deal with continuous nearest neighbor then propose query processing methods using R-trees as the underlying data structure. However, it is limited that the query trajectory must be known at query time.

UNICONS [28] deals with nearest neighbor queries as well as continuous NN queries in the context of moving objects databases. It precomputes and stores mNN results for each vertex in road network, and incorporates the use of precomputed NN lists into Dijkstras algorithm for nearest neighbor queries. For CkNN query, UNICONS computes the valid intervals of the query path. However, if k is large the massive network traversals still can not be avoid to obtain kNN results. Moreover, UNICONS is poor for handling sparse objects due to frequent recomputation of valid intervals.

Existing work [48, 62, 74, 102] adopts the concept of safe region which maintains a kNN set and an associated "safe region" where the query object can move freely without invalidating this kNN set. The query processor only needs to process a kNN query when the query object moves out of the safe region. Thus, both the computation and communication cost between the query object and the query processor are reduced.

Different with previous safe-region-based techniques, V*-Diagram [74] exploits the knowledge of both the query location and data objects. First it uses R-tree to obtain $k + x$ NN results. With the auxiliary x results, it constructs the safe region by narrowing down the known region. By combining with a finxed-rank region, it finally generates an integrated safe region.

MkSK [102] studies the problem that considers both spatial locations and keywords and maintains a safe zone that guarantee the validity for CkNN query. It develops two solutions for computing a safe region. The first is an early stop algorithm. The other is an advanced algorithm that prunes subtrees

of objects that do not contribute to the safe region and applies two optimizations to further reduce the search space and communication cost. However, MkSK is only limited to Euclidean space and cannot be well deployed.

Instead of computing a safe region, INS [62] uses a small set of influential neighbour objects, which shares the similar functionality with safe region. As long as the current kNN results are closer to the query object than the influential neighbour objects, the current kNN results stay valid and no recomputation is required. Thus the high cost of safe region recomputation is avoid. They also prove that the region defined by the safe guarding objects is the largest possible (optimal) safe region. This means the recomputation frequency of this method is minimized.

2.2.3 Travel Route Search

The travel route search problem has been substantially studied for decades [16, 22, 55, 56, 63, 65, 93, 107, 114, 115]. Traveling Salesman Problem (TSP) [29] is the most classic problem in route planning. TSP aims to find the round trip that has the minimum cost from a source point to a set of targets and finally returns to the source point.

Li et al. [63] study the problem of Trip Planning Query (TPQ) in spatial databases, where each object is associated with a location and a category. With a starting point S, a destination E and a set of categories C, TPQ retrieves the best trip that starts at S passes through at least one point from each category, and ends at E. The difficulty of this query lies in the existence of multiple choices per category. TPQ can be considered as a generalization of Travelling Salesman Problem (TSP), thus two approximation algorithms are proposed.

M Sharifzadeh et al. [93] study the problem of optimal sequenced route (OSR), which aims to find a route of minimum length starting from a source point and passing through a number of typed locations in a specific sequence imposed on the types of the locations. The OSR problem is first transformed into a shortest path problem on a large planer graph, which turns out that the classic algorithm such as Dijkstra's algorithm is impractical for most real-world scenarios. They propose a light threshold-based iterative algorithm LORD, which utilizes various thresholds to prune the locations that cannot belong to the optimal route. They also propose an extension algorithm R-LORD, which uses R-tree to examine the threshold values more efficiently.

H. Chen et al. [22] study the problem of multi-rule partial sequence route (MRPSR), which provides a unified framework that subsumes the well-known trip planning query (TPQ) [63] and the optimal sequenced route (OSR) [93] query and aims to find an optimal route with minimum distance under some partial category order rules defined in the query. They first prove that MRPSR is NP-hard and then propose three heuristic algorithms to search for near-optimal solutions for the MRPSR query.

Kanza et al. [56] study route-search queries by suggesting three semantics for such queries and deals with the problem of efficiently answering queries under the different semantics. The shortest-route semantic requires the answer to be the shortest pre-answer, the most-profitable-route semantic asks that the answer is the pre-answer that has the highest total score among the pre-answers whose length dost not exceed a given length, and the most-reliable-route demands that the answer is the pre-answer with the highest minimal score among the pre-answers whose length dose not exceed a given length. They propose a greedy algorithm to find a route whose length is smaller than a specified threshold while the total text relevance of this route is maximized.

Kanza et al. [55] study the problem of finding a route that visits at least one satisfying entity of each type in an interactive approach. In each step, a candidate is given to user to provide a feedback specifying whether the entity satisfies her. They present heuristic algorithms for interactive route search for two cases, depending on whether the constraints define a complete order or a partial one. The main challenge of this work is to use the feedback in order to find a route that is shorter and has a higher degree of success, compared to routes that are computed in non-interactive approaches.

Yao et al. [107] study the problem of multi-approximate-keyword routing (MAKR) query, which complements the standard shortest path search with multiple keywords and an approximate string similarity function. For each keyword, the matching point is supposed to have an edit distance smaller than a given threshold. They first prove MAKR is NP-hard and then propose an exact algorithm and three approximate algorithms to answer the query efficiently.

Cao et al. [16] define the problem of keyword-aware optimal route query, which is to find an optimal route such that it covers a set of user-specified keywords, a specific budget constraint is satisfied, and the objective score of the route is optimized. The problem of answering KOR queries is proved to be NP-hard. They first propose an approximation algorithm OSScaling with provable approximation bounds. Based on this algorithm, another more efficient approximation algorithm BucketBound is also proposed. Finally, they design a greedy approximation algorithm.

Li et al. [65] propose two different solutions, namely backward search and forward search, to deal with the general optimal route query without a total order. Given a set of spatial objects, each of which is associated with categorical information, the optimal route query finds the shortest path that starts from the query point, and covers a user-specified set of categories. The user may also specify partial order constraints between different categories.

Zhang et al. [115] propose the problem of personalized trip recommendation, which aims to find the optimal trip that maximizes users' experiences for a given time budget constraint and also takes the uncertain traveling time into consideration.

Zeng et al. [114] study the problem of optimal route search for keyword coverage, which takes into account the weighted user preferences in route search, and also presents a keyword coverage problem, which finds an optimal route from a source location to a target location such that the keyword coverage is optimized and that the budget score satisfies a specified constraint.

Chapter 3

Keyword-oriented Queries on Activity Trajectories

Driven by the advances in location positioning techniques and the popularity of location sharing services, semantic enriched trajectory data, which is called activity trajectory, have become unprecedentedly available. In this chapter, we study the problem of searching activity trajectories by keywords. Given a set of query keywords, a *keyword-oriented query for activity trajectory* (KOAT) returns k trajectories that contain the most relevant keywords to the query and yield the least travel effort in the meantime. The main difference between this work and conventional spatial keyword queries is that there is no query location in KOAT, which means the search area cannot be localized. To capture the travel effort in the context of query keywords, a novel *spatio-textual ranking function*, is first defined. Then we develop a hybrid index structure called GiKi to organize the trajectories hierarchically, which enables pruning the search space by spatial and textual similarity simultaneously. Finally an efficient search algorithm and fast evaluation are proposed. In addition, we extend the proposed techniques of KOAT to support range-based query and order sensitive query, which can be applied for more practical applications. The results of our empirical studies based on real check-in datasets demonstrate that our proposed index and algorithms can achieve good scalability.

This chapter is organized as follows. We give an introduction in Section 3.1 and define the necessary concepts and formulate the query in Section 3.2. Section 3.3 presents the baseline methods. Proposed index structure and solution for KOAT are discussed in Section 3.4 and Section 3.5. Section

3.6 describes the enhanced algorithm. Section 3.7 reports the experimental observations. Finally, we conclude this chapter in Section 3.8.

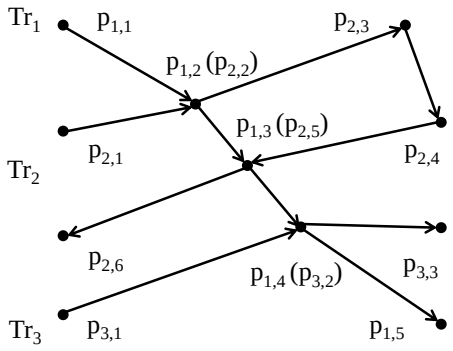
3.1 Introduction

Mobility devices such as smartphones and tablets now are gradually predominating the transformation of the web from desktop-based age to mobility-based age, resulting in large-scale collection of movement data. Such data recording the motion history of moving objects, known as *trajectories*, play an essential role in a variety of well-established application areas (e.g., tracking, urban planning, traffic management, geo social networks). Representative work includes designing effective trajectory indexing structures [14, 78], trajectory query processing [25, 100], uncertainty management [126, 128], and mining knowledge/patterns from trajectories [52, 129]. In the meantime, increasing volumes of geo-textual objects are becoming available on the web that represent Point-of-Interest (PoIs). Applications (e.g., Facebook[1], Foursqure[2] and Flickr[3]) allow people to check-in at these PoIs, each having a spatial location and a semantic description. By virtue of semantic labelling, the traditional trajectory databases are redefined and enriched by attaching activity or semantic meanings. In this work, we use the term *activity trajectory* to represent this check-in sequence of geo-textual objects that contain the information about the semantic meanings of user behaviour (e.g., activity or place name) at particular places.

Cong et al. [17, 18, 30] have extensively studied spatial keyword queries with different problem settings. However, these work are based on geo-textual objects, rather than trajectory database. Only a few proposals on activity trajectory have been published that aim to return relevant geo-textual objects in response to a query's activity interests and geographical preference [31, 124]. Zheng et al. [124] study the activity trajectory similarity query (ATSQ) that accounts for covering the query activities and yielding the shortest minimum match distance. A major limitation of the ATSQ is that it only supports exact keyword search condition. However, in many real application scenarios, approximate keyword search is more desirable since users may not know well enough about the data

[1] https://www.facebook.com
[2] https://foursquare.com
[3] https://www.flickr.com

to type accurate query keywords. Another limitation of previous work is that they require users to specify one or more locations in their queries so that the returned PoIs or trajectories are close to these locations. However, we observe that in many circumstances a user does not have a preferred location in advance. This is usually the case when a tourist plans a trip to a city and has not decided where to live. Searching for the travel histories based on the desired activities will help her to choose the suitable location.



Object	Associated Keywords
$p_{1,1}$	*Restaurant, Club, Shop*
$p_{1,2}$ ($p_{2,2}$)	*Cafe*
$p_{1,3}$ ($p_{2,5}$)	*Park*
$p_{1,4}$ ($p_{3,2}$)	*Theater, Pub*
$p_{1,5}$	*Market, Station*
$p_{2,1}$	*Restaurant, Gym*
$p_{2,3}$	*School, Pizza, Spa*
$p_{2,4}$	*Bakery, Hotel, Mall*
$p_{2,6}$	*Theatre, Bar*
…	…

FIGURE 3.1: Running example for KOAT query

To this end, we study the problem of searching activity trajectories with keywords, which is very useful in many location-based services such as intelligent tourist guide and trip planning. Consider the example shown in Figure 3.1. A tourist Q plans to visit q_1 : **Restaurant** for dinner and then q_2 : **Theatre** to watch a movie. She would like to check the travelling histories of other people that are relevant to her intended activities for reference, since she is new to this city. Exact keyword match would only return trajectory Tr_2 from $p_{2,1}$ to $p_{2,6}$ as result, but ideally the places with keyword **Theater** should also be considered. If we adopt some string similarity functions (e.g., *edit distance*) that allows for approximate keyword match, then both trajectory Tr_1 from $p_{1,1}$ to $p_{1,4}$ and Tr_2 from $p_{2,1}$ to $p_{2,6}$ may be considered. Moreover, Tr_1 from $p_{1,1}$ to $p_{1,4}$ is an even better choice since it obviously requires less travel effort.

Towards this direction, our previous work [121] aims to support efficient process of keyword-oriented search on activity trajectory database, wherein given a set of query keywords, the output is the top-k trajectory segments with "closely matched" keywords and short travel distances. Specifically, we propose a novel score function for activity trajectory by incorporating both keyword similarity

and travel distance into the distance measure. However, answering this new query turns out to be a challenging problem since just making use of either location or keyword information for search space pruning will result in bad query performance. In addition, evaluating this query calls for an exploration of huge numbers combinations of geo-textual objects within each trajectory. Therefore, we propose a novel hybrid index structure, called GiKi, to develop a tighter lower bound of spatio-textual ranking function for all "unseen" trajectories in the database, and also propose an efficient algorithm to compute the scores of candidates. In addition, we propose a trajectory segmentation method to partition trajectories into segments, and then propose an enhanced search algorithm based on it. Besides, though this query offers some flexibility, sometimes the user may be more interested in the trajectories which locate in a specific region or whose activity order is defined at query time. To sum up, we make the following major contributions in this chapter.

- We introduce a spatio-textual ranking function to take both the travel effort and textual proximity into consideration. In addition, we propose a novel index structure called GiKi to organize the trajectories in a hierarchical manner. On top of that, a best-first search strategy is developed to prune a large number of disqualifying trajectories by keyword similarity and travel distance minimization simultaneously.

- We also propose a trajectory segmentation method to partition trajectories into segments by considering spatial, temporal and semantic features. Based on this, an enhanced search algorithm based on trajectory segmentation is developed to answer the query more efficiently.

- We propose a range-based query and an order-sensitive query on activity trajectories, which are extended from KOAT, to support more practical applications.

- We conduct extensive experimental study based on real check-in datasets. The experimental results show the scalability of our proposed solution.

3.2 Problem Statement

In this section, we formally define the *keyword-oriented query for activity trajectories* (KOAT). Table 3.1 summarizes the major notations used in this chapter.

Table 3.1: Summary of notations

Notation	Definition
Q	A set of query keywords
Tr	A semantic trajectory
p	A geo-textual object with l, t and Φ
$Tr[s, e]$	A sub-trajectory of Tr starts from s to e
$Q_{km}(Tr)$	A keyword mapping from Q to Tr
$Q_{mkm}(Tr)$	Minimum keyword mapping from Q to Tr
$D_{td}(Q, Tr)$	Textual distance between Q and Tr
$D_{tr}(Tr)$	Travel distance of Tr
$D_{md}(Q, Tr[s, e])$	Matching distance between Q and $Tr[s, e]$
$R_{st}(Q, Tr)$	Spatio-textual ranking function

Definition 3.1 (Activity Trajectory)**.** *An activity trajectory Tr is defined as a sequence of geo-textual objects, i.e. $Tr = \{p_1, p_2, \ldots, p_{|Tr|}\}$, where each object $p_i = (l, t, \Phi)$ contains a location $p_i.l$, a timestamp $p_i.t$ and a set of keywords $\Phi(p_i) = \{w_j\}$ denoting the semantic/textual description associated with the location. A sub-trajectory $Tr[s, e] \subseteq Tr$ is a segment of Tr from p_s to p_e.*

In order to estimate the relevance between the query and ideal trajectories, we proceed to consider ones with high textual similarity and less travel distances. First, we introduce a concept of keyword matching that maps query keywords to objects within a trajectory.

Definition 3.2 (Keyword Matching)**.** *Given a set of query keywords $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$, and a trajectory Tr or a sub-trajectory $Tr[s, e]$. A keyword matching from Q to Tr is a set of objects $Q_{km}(Tr)$, where the contained keywords may have high textual similarity with Q. In addition, the keyword matching with highest textual similarity w.r.t. Q is defined as the minimum keyword matching, i.e., $Q_{mkm}(Tr)$. In other words, $Q_{mkm}(Tr)$ contains the keywords that are most similar to query keywords among all keywords in Tr.*

We adopt the *edit distance* metric to measure the textual relevance. Formally, we define **textual distance** as the sum of minimum normalized *edit distances* between each $q_i \in Q$ and keywords $w_j \in Q_{mkm}(Tr).\Phi$.

$$D_{td}(Q, Tr) = \frac{1}{|Q|} \sum_{q_i, w_j} \min\{\frac{d_e(q_i, w_j)}{\max\{|q_i|, |w_j|\}}\} \tag{3.1}$$

We define **travel distance** as the length of the trajectory or sub-trajectory. In addition, we adopt Sigmoid function to normalize it, since the value of Sigmoid function changes more quickly when the variable is small, which matches the intuition that users' satisfactory is usually more sensitive when the travel distance is short. Further, we use φ as distance adjusting parameter, which can be easily computed by the maximum length of trajectories in database:

$$D_{tr}(Tr) = \frac{2}{1 + e^{-\varphi \cdot len(Tr)}} - 1 \tag{3.2}$$

It is worth noting that users only concern about the sub-trajectories from which they could get reference, thus we adopt a concept of **matching distance** to incorporate the textual relevance and travel distance together. Formally, given Q and $Tr[s, e]$, the matching distance, denoted as $D_{md}(Q, Tr[s, e])$, is defined as follows:

$$D_{md}(Q, Tr[s, e]) = \alpha \cdot D_{td}(Q, Tr[s, e]) + (1 - \alpha) \cdot D_{tr}(Tr[s, e]) \tag{3.3}$$

where α is a user-specified parameter and used to adjust the relative importance of the textual relevance and travel distance.

Definition 3.3 (Spatio-textual Ranking Function)**.** *Given Q and Tr, the spatio-textual ranking function is defined as the minimum matching distance for all $Tr[s, e] \subseteq T$:*

$$\mathcal{R}_{st}(Q, Tr) = \min_{Tr[s,e] \subseteq Tr} \{D_{md}(Q, Tr[s, e])\} \tag{3.4}$$

Definition 3.4 (Keyword-oriented Query of Activity Trajectory)**.** *Given an activity trajectory database $\mathcal{T}$, a set of query keywords $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$, and k. A keyword-oriented query of activity trajectory (KOAT) returns k distinct trajectories that have the minimum score of spatio-textual ranking function. Intuitively, the KOAT will return the parts of some activity trajectories that contain most similar keywords w.r.t. the query and yield shortest travel distance.*

3.3 Existing Approaches

In this section, we propose two baseline algorithms that explore the possibility of extending existing techniques to solve KOAT.

3.3.1 Probe based Algorithm

To compute *edit distance* between two strings s_1 and s_2, the Wagner-Fischer algorithm [73] is an exact but costly operation, while n-gram [40] is one of the most popular techniques that is able to quickly estimate *edit distance*. For a string s, its n-grams are produced by sliding a window of length n over the characters of s. For example, $p_{1,4} \in Tr_1$ contains the keyword "*Pub*", the 3-gram of "*Pub*" is {*##P, #Pu, Pub, ub$, b$$*}. The principle of n-gram similarity between two strings is that the more n-grams they share, the more similar they are expected to be. The probe based algorithm (PBA) is the brute-force approach that traverses each trajectory Tr in order to find the sub-trajectory $Tr[s, e]$ that $D_{md}(Q, Tr[s, e])$ is minimized. At the first step, we retrieve all the trajectories $Tr \in \mathcal{T}$ whose objects share at least one common 3-gram with each query keyword $q_i \in Q$ and treat them as candidates. Given Tr and a query keyword q_i, we proceed to compute the minimum *edit distance* between each object $p \in Tr$ and q_i, and store them in an array. For each candidate Tr, we utilize two probes pb_s and pb_e pointing to two objects that construct a sub-trajectory $Tr[pb_s, pb_e]$. By moving the position of two probes, we compute the matching distance $D_{md}(Q, Tr[pb_s, pb_e])$, thus obtain the value of spatio-textual score function $R_{st}(Q, Tr)$. It is easy to see, for each Tr, the space cost is $O(|Q| \cdot |Tr|)$ and the time cost is $O(|Tr|^2)$. Therefore, we know PBA is impractical except for trajectories with short length.

3.3.2 Inverted List based Algorithm

The inverted list based algorithm (ILA) utilizes the inverted list as the index structure to prune the search space. For each keyword w, let the set of n-grams contained in w be G_w. Analogously, for object p, we have $G_p = \cup_{w \in p.\Phi} G_w$ and for trajectory Tr, $G_{Tr} = \cup_{p \in Tr} G_p$. Therefore, we use inverted lists for the grams of all trajectories to answer KOAT. For each gram σ, we have a posting list l_σ containing the IDs of trajectories whose grams contain σ. Given Q, we need to scan the inverted lists in order to obtain the trajectories which share common grams with Q as candidates. We first compute

G_{q_i} for each keyword q_i. It is straightforward that we merge the TID lists l_{σ_j}s ($\sigma_j \in G_{q_i}$) by *Heap Algorithm* [91] to get a candidate list for each q_i. In each list, trajectories are sorted in descending order by $|G_{q_i} \cap G_{Tr}|$. Therefore, trajectories have no common grams with Q are pruned.

Lemma 3.1. *[40] For string s_1 and s_2 of length $|s_1|$ and $|s_2|$, if $d_e(s_1, s_2) = \tau$, then $|G_{s_1} \cap G_{s_2}| \geq \max(|s_1|, |s_2|) - 1 - (\tau - 1) * n$. Through transformation, we obtain*

$$\begin{aligned} \tau &\geq \frac{1}{n} \cdot (\max(|s_1|, |s_2|) - 1 - |G_{s_1} \cap G_{s_2}|) + 1 \\ &\geq \frac{1}{n}(|s_1| - 1 - |G_{s_1} \cap G_{s_2}|) + 1 \end{aligned} \tag{3.5}$$

Therefore, for each Tr, the lower bound of textual distance w.r.t. q_i is computed by

$$D_{td}(\{q_i\}, Tr)_L = \frac{|q_i| - |G_{q_i} \cap G_{Tr}| + n - 1}{n \cdot \max(|q_i|, maxLen)}. \tag{3.6}$$

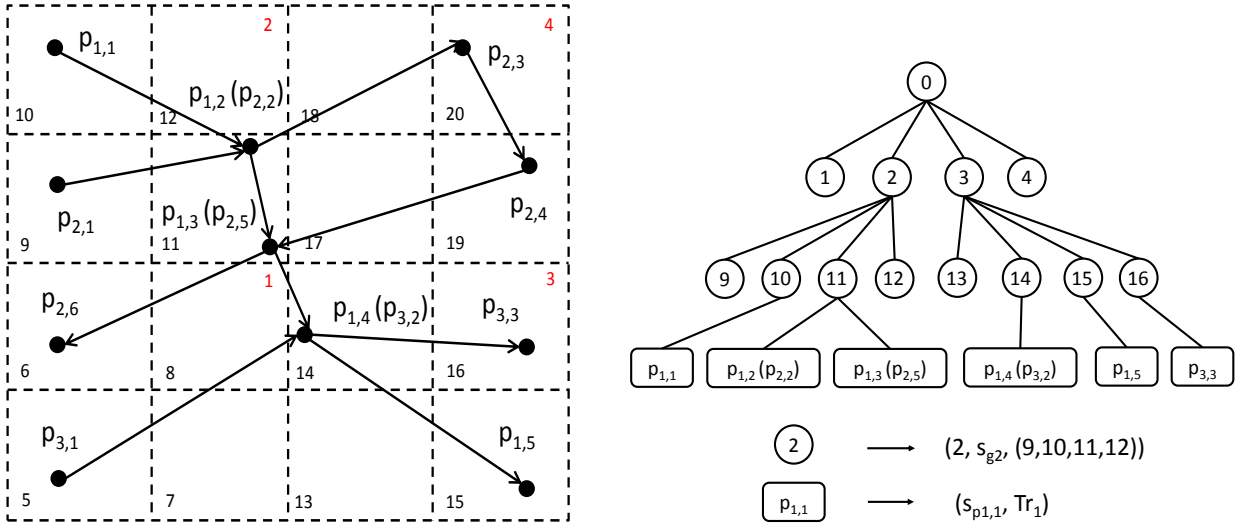
where *maxLen* is the maximum length of keywords contained by Tr. Therefore, we are easy to have $D_{td}(Q, Tr)_L = \sum_{q_i \in Q} D_{td}(\{q_i\}, Tr)_L$. As we know, the best case is that the travel distance equals to 0, i.e., all q_is are matching to the same object of trajectory Tr. Therefore, $R_{st}(Q, Tr)_L = \alpha \cdot D_{td}(Q, Tr)_L$. We sequentially examine trajectories in order of their lower bounds and compare with the k-th best result. This process terminates when the lower bound of next candidate exceeds k-th result. By this means, this baseline is expected to examine fewer trajectories than PBA and achieve better efficiency.

3.4 Hybrid Index Structure

In this section, we propose a novel index structure, namely Grid-Keyword index (GiKi), for trajectories by incorporating both spatial and textual information to enable pruning search space, as shown in Figure 3.2. Specifically, we construct a d-Grid by dividing the entire space into quad grids by building 1-Grid, ..., $(d-1)$-Grid, d-Grid, which forms a hierarchy, as shown in Figure 3.2(a). In particular, GiKi consists of two components: 1) Activity Grid-Tree index (AG-Tree); 2) Keyword-Reference index (K-Ref).

3.4.1 Activity Grid Tree Index

Based on the grid division, we build AG-Tree to index trajectories together with their keywords. As shown in Figure 3.2(b), where leaf nodes correspond to the geo-spatial objects attached with



(a) Grid partition of space (b) AG-Tree index

Figure 3.2: Grid keyword index overview

keywords, and non-leaf nodes are MBRs bounding these objects. For non-leaf node g, which is actually an MBR, contains three elements (GID, s_g, $\{g' \in g.sub\}$):

1. **Grid ID**. It denotes the ID of grid node g;
2. **Keyword signature**. We use all n-grams of objects that contained in g to compute the keyword signature s_g on the purpose of textual distance computation, and the computation process will be detailed later;
3. **Sub entries**. A sequence of pointers pointing to sub grids or objects $g' \in g.sub$.

For leaf node, $p \in Tr$ contains two elements (s_p, TID), where s_p is the keyword signature of keywords in $p.\Phi$, and TID denotes the ID of $Tr \in \mathcal{T}$.

Keyword signature generation. In order to obtain the keyword signature, a straightforward solution is to compute the n-grams of keywords and store them in the index. However, this approach requires storing all grams in the tree structure, which is too space consuming. In order to reduce space cost and improve computation efficiency, we adopt *MinHash* [106] method to generate keyword signature for each node. Basically, *MinHash* is initially used to detect duplicate web pages for searching. To implement the *MinHash* scheme and estimate the set resemblance, some unbiased estimators are proposed to estimate the size of a set by repeatedly assigning random ranks to the universe, and keeping

the minimal rank of a set. The minimum values from the permuted ranks of a set kept for each permutation are called the signature and can be used to estimate the set resemblance. Formally, consider a set of random permutations $\mathbb{F} = \{\pi_1, \pi_2, \ldots, \pi_{|\mathbb{F}|}\}$ and a universe of items U. For any set $A \subseteq U$, let $\min\{\pi_i(A)\}$ denote $\min\{\pi_i(x)|x \in A\}$, which is the minimum random rank assigned by permutation π_i. For any subset $A \subseteq U$, any $x \in A$ and all $\pi_i \in \mathbb{F}$, $\mathbb{F}$ is called *min-wise independent permutations* if it satisfies that $Pr(\min\{\pi_i(A)\} = \pi_i(x)) = \frac{1}{|A|}$. To estimate the set similarity γ between set A and B ($A, B \subseteq U$), the signature of A is constructed as $s_A = [\min\{\pi_1(A)\}, \ldots, \min\{\pi_{|\mathbb{F}|}(A)\}]$ which is similar to B. Therefore, γ can be approximated by Jaccard Similarity [40], i.e., $\gamma(s_A, s_B) \approx \frac{|s_A \cap s_B|}{|s_A \cup s_B|}$. In KOAT, we use all the n-grams generated from keywords in $\mathcal{T}$ to construct U. Then we compute the keyword signature for all leaf nodes and non-leaf nodes in AG-Tree, i.e., $s_g = [\min\{\pi_1(G_g)\}, \ldots, \min\{\pi_{|\mathbb{F}|}(G_g)\}]$, where G_g is the n-grams of g.

3.4.2 Keyword Reference Index

As we know, exactly computing *edit distance* during query processing is infeasible in terms of computation time. Therefore, we offline construct K-Ref to index *edit distance* within each trajectory. For each $Tr \in \mathcal{T}$, we choose a set of reference keywords $R(Tr) = \{w_r\}$ to index the *edit distance* between keywords contained in Tr and reference keywords in $R(Tr)$. Given Tr, our objective is to partition the keywords into N clusters and select an reference keyword w_{r_n} for each cluster, such that the mathematical expectation of *edit distance* within each cluster is minimized [122]. Needless to say, it is obviously the K-means problem with a different optimization objective. Therefore, each object p_i in Tr is indexed by a B^+-Tree with an index key $y(p_i)$, which is computed based on the *edit distance* between keyword w_i^j and its corresponding reference keyword w_{r_n}, i.e., $y(p_i) = d_e(w_{r_n}, w_i^j) + n \cdot c$ ($0 \leq n < N$), where $c = \max\{d_e(\cdot)\} + 1$. In addition, for each cluster, a lower bound distance $LB(w_{r_n})$ and an upper bound distance $UB(w_{r_n})$ are also kept.

3.5 Keyword-oriented Query Processing

In this section, we introduce a dynamic programming algorithm DPA for KOAT. First, we retrieve a set of candidate trajectories, which contain some similar keywords w.r.t. query keywords by utilizing

AG-Tree. Then, we take advantage of K-Ref to compute the lower bound for each candidate trajectory. Finally, we validate the trajectories by computing spatial-textual ranking value. During this process, we keep track of k-th smallest value of $\mathcal{R}_{st}(Q, Tr)^k$ found so far and a lower bound $\mathcal{R}_{st}(Q, Tr)_L$ for all "unseen" trajectories. Once we have $\mathcal{R}_{st}(Q, Tr)^k < \mathcal{R}_{st}(Q, Tr)_L$, the algorithm can terminate safely. Otherwise we will incrementally fetch more candidates and repeat the above process again. The basic structure of our proposed search algorithm is introduced by Algorithm 1.

Algorithm 1: Outline for KOAT

Input: Trajectory database $\mathcal{T}$, query Q

Output: top-k result set RS

```
1  while true do
2      CS ← ∅;
3      CS ← retrieve at least λ candidate;
4      R_st(Q, Tr)_L ← update the lower bound;
5      for each Tr ∈ CS do
6          if Tr is a valid candidate then
7              Compute R_st(Q, Tr) and put Tr into RS;
8              Update R_st(Q, Tr)^k and R_st(Q, Tr)_L;
9      if R_st(Q, Tr)^k < R_st(Q, Tr)_L then
10         break;
11 Keep the top-k results in RS;
12 return RS;
```

3.5.1 Candidate Retrieval

A candidate is a trajectory that is possible and seemingly promising to become a result for the query. Since we intend to find the trajectories with "closely matched" keywords w.r.t. query, which means the similarity between the signatures generated by Q and Tr should be as greater as possible, we first propose to find the grids with higher similarity to query. In order to obtain the candidate set CS, we adopt the best-first paradigm to search the AG-Tree and process the query keywords one by one. It

Algorithm 2: Candidate Retrieval

Input: Trajectory database $\mathcal{T}$, query Q and AG-Tree constructed from $\mathcal{T}$

Output: CS

```
1  CS ← ∅; g ← root of AG-Tree;
2  for each q_i in Q do
3      Insert g into PQ; Generate s_{q_i};
4      while PQ ≠ ∅ do
5          Dequeue PQ;
6          if γ(s_{q_i}, s_g) > 0 and g.SubGrids ≠ 0 then
7              Enqueue all g ∈ g.SubGrids into PQ;
8          if γ(s_{q_i}, s_g) > 0 and g.SubGrids = 0 then
9              for p ∈ g.SubPoints do
10                 Compute γ(s_{q_i}, s_p);
11                 if γ(s_{q_i}, s_p) > 0 then
12                     Update γ(s_{q_i}, s_{Tr})_max;
13 Check entries Tr in CS w.r.t. all q_i ∈ Q;
14 Compute Σ_{q_i∈Q} γ(s_{q_i}, s_{Tr});
15 Retrieve at least λ candidates;
16 return CS;
```

is worth to notice that they are processed in ascending order of the frequency of their n-grams, i.e., keywords with infrequent n-grams are processed first, since they are more likely to prune trajectories, and we thus obtain less candidates to deal with.

For non-leaf node g, we maintain a priority queue PQ with entries in the form of $(\gamma(s_g, s_{q_i}), \text{GID}, q_i)$, where $\gamma(s_g, s_{q_i})$ is the signature similarity between g and q_i, and is used as the key to sort the entries in PQ. The process starts to enqueue the root node of AG-Tree to PQ, then we dequeue the top entry of PQ and compute $\gamma(s_{g'}, s_{q_i})$ for all its child grids or objects $g', p \in g.sub$. If $\gamma(s_{g'}, s_{q_i}) > 0$, we enqueue the entry of g' or p into PQ, otherwise g' is pruned. For leaf node p, we retrieve the information of trajectories that $p \in Tr$ and push them into CS. Note that, for each candidate Tr, the entry is in form of $(\gamma(s_{q_i}, s_{Tr})_{max}, \text{TID}, q_i)$, where $\gamma(s_{q_i}, s_{Tr})_{max}$ is the maximum value of signature similarity among all

$p \in Tr$ w.r.t. q_i and updated during the process of tree traversal for each candidate Tr. This process is repeated until there is no grids left in PQ. For other keywords q_j after processing q_i, if there is no entry existing for Tr w.r.t q_j, Tr will be pruned from CS. Finally, the candidates are sorted in descending order of $\sum_{q_i \in Q} \gamma(s_{q_i}, s_{Tr})$, and we retrieve at least λ candidates for further validation. The candidate retrieval algorithm is shown in Algorithm 2.

In the process of candidate retrieval, memory and computation time are the two major factors determining the cost. Sequentially checking all $Tr \in \mathcal{T}$ to obtain $\gamma(s_{Tr}, s_{q_i})$ requires a large volume of memory to store signatures of Tr, and the computation time is unimaginable high as well. However, making use of AG-Tree obviously reduces the need of memory volume. Moreover, the proper chosen d of AG-Tree is supposed to achieve a better efficiency because coarse grain grid incurs more computation inside grids while fine grain grid induces more grids to consider. Analogously, a reasonable number of permutation, i.e., $|\mathbb{F}|$, is more likely to improve the accuracy of estimation, thus reduces the size of CS at the sacrifice of memory for signature storage.

3.5.2 Lower Bound Computation

Another important task during the process of top-k trajectory retrieval is to maintain a lower bound for all $Tr \in CS$. From the insight of Equation 3.4, we have the following lemma:

Lemma 3.2. *The lower bound of spatio-textual ranking value between Q and Tr, i.e., $\mathcal{R}_{st}(Q, Tr)_L$, is the minimum lower bound of matching distance between Q and $Tr[s, e] \subseteq Tr$, i.e., $D_{md}(Q, Tr[s, e])_L$,*

$$\mathcal{R}_{st}(Q, Tr)_L = \min\{D_{md}(Q, Tr[s, e])_L\}. \tag{3.7}$$

Proof. As we know, $D_{md}(Q, Tr[s, e])_L$ is the lower bound for $Tr[s, e]$, thus $\min\{D_{md}(Q, Tr[s, e])_L\}$ is the lower bound for all $Tr[s, e] \subseteq Tr$. Therefore, Equation 3.7 is proven. □

From lemma 3.2, for each candidate trajectory Tr, we need to find the ideal keyword match that constructs a sub-trajectory $Tr[s, e]$ whose $D_{md}(Q, Tr[s, e])_L$ is minimum among all possible keyword matches. As we know, the matching distance consists of textual and travel distances, so the matching distance between Q and minimum keyword matching $Q_{mkm}(Tr)$ is not necessarily the minimum matching distance among all possible keyword matches, because we are still possible to find keyword matches with smaller matching distances than $Q_{mkm}(Tr)$. Nevertheless, in order to compute

$\mathcal{R}_{st}(Q, Tr)_L$, we initially determine $Q_{mkm}(Tr)$, and incrementally update keyword matching $Q_{km}(Tr)$ and corresponding matching distance until its minimum value is reached. Let $\mathtt{p}_i$ be the matched object w.r.t. q_i, we then introduce how to determine and update keyword match. Since the *edit distance* obeys triangle inequality, we have

$$d_e(q_i, w) \geq |d_e(w_{r_j}, w) - d_e(w_{r_j}, q_i)|. \tag{3.8}$$

which computes $d_e(q_i, w)_L$ by measuring the difference between $d_e(w_{r_j}, w)$ and $d_e(w_{r_j}, q_i)$. In other words, we need to find $\mathtt{p}_i$ whose contained keywords have smallest *edit distance* w.r.t. q_i. Therefore, we make use of the *edit distance* between Q and $R(Tr)$ to locate $\mathtt{p}_i$ by accessing K-Ref as shown in Algorithm 3. To determine $\mathtt{p}_i$, we first compute $d_e(q_i, w_{r_j})$ between q_i and each reference keyword $w_{r_j} \in R(Tr)$. Then for each keyword cluster w.r.t. w_{r_j}, we are easy to determine $\mathtt{p}_i$ by using the preserved $LB(w_{r_j})$ and $UB(w_{r_j})$. If the value of $d_e(q_i, w_{r_j})$ falls in between $LB(w_{r_j})$ and $UB(w_{r_j})$, we apply a binary search on K-Ref to find the closest object $\mathtt{p}_i$ in terms of *edit distance*. Otherwise, $\mathtt{p}_i$ is the bounding object. After processing all w_{r_j}s, the minimum $d_e(q_i, w)_L$ as well as $\mathtt{p}_i$ are obtained.

Let $Tr[s, e]_{mkm}$ be the sub-trajectory built by the minimum keyword matching $Q_{mkm}(Tr)$, it is obvious that the textual distance between Q and $Tr[s, e]_{mkm}$ is a lower bound for $\mathcal{R}_{st}(Q, Tr)$ as mentioned in baseline ILA. However, it is still too loose to be effective in practice. In addition, we cannot guarantee that $D_{md}(Q, Tr[s, e]_{mkm})_L$ is a lower bound for $\mathcal{R}_{st}(Q, Tr)$, since it is possible that there exists another sub-trajectory $Tr[s, e] \subseteq Tr$ yields a tighter lower bound satisfying that $D_{md}(Q, Tr[s, e])_L < D_{md}(Q, Tr[s, e]_{mkm})_L$.

Lemma 3.3. *Given $Tr[s, e] \subseteq Tr$, if $D_{md}(Q, Tr[s, e]) < D_{md}(Q, Tr[s, e]_{mkm})$, there must be that $len(Tr[s, e]) < len(Tr[s, e]_{mkm})$.*

Proof. We know $Tr[s, e]_{mkm}$ is built by minimum keyword matching $Q_{mkm}(Tr)$. If $Tr[s, e]$ has a longer length than $Tr[s, e]_{mkm}$, $D_{md}(Q, Tr[s, e])$ cannot be smaller than $D_{md}(Q, Tr[s, e]_{mkm})$. □

Based on the elaboration above, we propose a **dynamic programming** algorithm (DPA) to compute the lower bound as shown in Algorithm 4. We define the DP state as (v) which represents the lower bound $\mathcal{R}_{st}(Q, Tr)_L$ at state v. For the initial state (1), we determine $Tr[s, e]_{mkm}$ by Algorithm 3 and compute the initial $D_{md}(Q, Tr[s, e]_{mkm})_L$. Let $Tr[s_c, e_c]$ be the sub-trajectory that holds current lower bound for state (v). For the state transition function (v), we propose to update $Tr[s_c, e_c]$ by

Algorithm 3: Determining $\mathtt{p}_i$

Input: q_i, K-Ref of Tr

Output: $\mathtt{p}_i, d_e(q_i, \mathtt{p}_i)_L$

```
1  Initial d_e(q_i, p_i)_L ← 0;
2  for each w_{r_j} in R(Tr) do
3      Compute d_e(q_i, w_{r_j});
4      if d_e(q_i, w_{r_j}) falls in between bounding range then
5          Search K-Ref to obtain p_i;
6          d_e(q_i, p_i)_L ← |d_e(q_i, w_{r_j}) − d_e(p_i, w_{r_j})|;
7      else
8          p_i is the bounding object;
9          d_e(q_i, p_i)_L ← |d_e(q_i, w_{r_j}) − UB(w_{r_j})|;
10         or d_e(q_i, p_i)_L ← |d_e(q_i, w_{r_j}) − LB(w_{r_j})|;
11 p_i ← argmin_{p_i}{d_e(q_i, p_i)_L};
12 d_e(q_i, p_i)_L ← min{d_e(q_i, p_i)_L};
13 return p_i, d_e(q_i, p_i)_L;
```

replacing a current object $\mathtt{p}_i$. Intuitively, we aim to choose a new $\mathtt{p}_i$ that yields minimum marginal gain. Therefore, $D_{td}(Q, Tr[s_c, e_c])_L$ will be relaxed but still smaller than all the rest possible keyword matches. Note that, from Lemma 3.3, we know only sub-trajectories with smaller length than $Tr[s_c, e_c]$ are considered during the state transition process. Thus, we have the following state transition function:

$$(v) = \min \begin{cases} (v-1) \\ D_{md}(Q, Tr[s_c, e_c])_L \end{cases} \tag{3.9}$$

Let $\triangle D_{td}$ be the minimum marginal gain of updating $Tr[s_c, e_c]$, this process terminates when $\triangle D_{td} > \frac{1-\alpha}{\alpha} \cdot D_{tr}(Tr[s_c, e_c])_L$ since in this case there dost not exist a sub-trajectory $Tr[s, e]$ whose $D_{md}(Q, Tr[s, e])_L$ is smaller than current lower bound. The DPA algorithm shows how to compute lower bound distance of trajectory Tr, and the value of termination state (v) is $\mathcal{R}_{st}(Q, Tr)_L$.

Algorithm 4: Computing Lower Bound

Input: Q, K-Ref of Tr

Output: $R_{st}(Q, Tr)_L$

```
for each q_i ∈ Q do
    Determine p_i;
Determine Tr[s, e]_mkm formed by all p_is;
Compute D_td(Q, Tr[s_c, e_c])_L, D_tr(Tr[s_c, e_c]) and D_md(Q, Tr[s_c, e_c])_L;
while true do
    Determine new Tr[s_c, e_c];
    Compute △D_md(Q, Tr[s_c, e_c]);
    if △D_md(Q, Tr[s_c, e_c]) > ((1−α)/α) · D_tr(Tr[s_c, e_c]) or all Tr[s, e]s have been processed then
        break;
    else
        Update Tr[s_c, e_c];
        Compute D_md(Q, Tr[s_c, e_c])_L;
R_st(Q, Tr)_L ← D_md(Q, Tr[s_c, e_c])_L;
return R_st(Q, Tr)_L;
```

3.5.3 Candidate Validation

After computing lower bound, we proceed to validate candidates and compute $\mathcal{R}_{st}(Q, Tr)$ for candidates. Generally, if the lower bound of candidate trajectory Tr next to process is less than current k-th best value in result set RS, then trajectory Tr needs to validate. Otherwise, the search process terminates and k results are returned.

As mentioned in Algorithm 4, the updated lower bounds $D_{md}(Q, Tr[s, e])_L$ between Q and possible sub-trajectories $Tr[s, e]$ have been computed in the intermediate states. A heap H_{st} is kept to store these lower bounds in ascending order, then the exact matching distances $D_{md}(Q, Tr[s, e])$ between $Tr[s, e]$ and Q are computed in the same heuristic. It is worth to notice that the computation of $\mathcal{R}_{st}(Q, Tr)$ is actually the top-1 query in terms of matching distance, therefore the top-1 result is returned as exact $\mathcal{R}_{st}(Q, Tr)$.

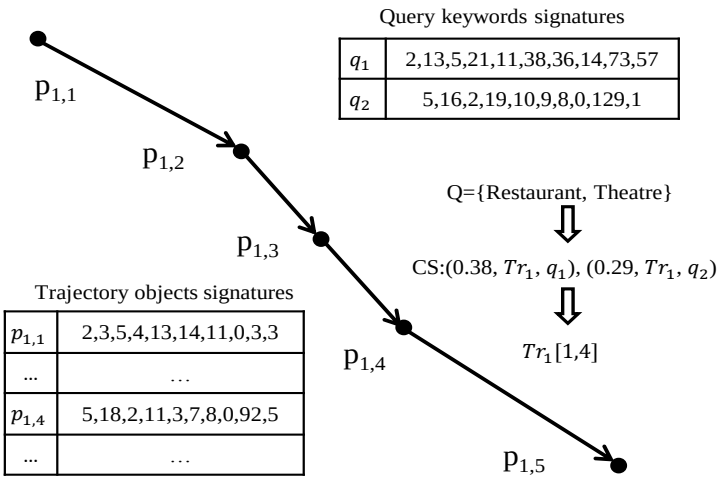


FIGURE 3.3: Query processing of KOAT.

Example 3.1. *For the query processing on Tr_1, objects with similar keyword signature w.r.t. query are obtained in the candidate retrieval phase. For Tr_1, we have two entries corresponds to q_1 and q_2 respectively. For lower bound computation, $p_{1,1}$ and $p_{1,4}$ are determined by accessing K-Ref. Therefore, $Tr_1[s_c, e_c] = Tr_1[1,4]$, and $D_{td}(Q, Tr_1[1,4])_L = 0, D_{tr}(Tr_1[1,4]) = 0.499$ and $D_{md}(Q, Tr_1[1,4])_L = 0.250$. In the process of updating $Tr_1[s_c, e_c]$, it is easy to see that the next possible sub-trajectory $Tr_1[4,5]$ is disqualified and $Tr_1[1,4]$ bounds Tr_1. Finally, the exact value of spatio-temporal ranking function, i.e., $\mathcal{R}_{st}(Q, Tr_1) = 0.426$, is computed in candidate validation.*

3.6 Enhanced Query Processing

In this section, we propose an enhanced search algorithm for KOAT query based on trajectory segmentation. Generally, the activity trajectories usually suffer a data sparsity problem due to users irregular check-in behaviours, and a trajectory always includes several trip experiences. However, users only concern about the trajectory segments from which they could get references. In addition, considering such a scenario that the length of $Tr[s, e]_{mkm}$, which is built by $Q_{mkm}(Q, Tr)$ in the initial state, is unaccepted long, the computation can still be very costly due to redundant amount of keyword matches. Hence, for the goal of improving search efficiency, we propose to partition the trajectory into segments and keep them as compact as possible and in some sense of homogeneous, and also introduce the enhanced algorithm based on such segmentation.

3.6.1 Trajectory Segmentation

Given Tr, the trajectory segmentation aims to partition Tr into κ sub-trajectories and obtain $\mathbb{TS}$ ($\kappa = |\mathbb{TS}|$) such that

1. Each sub-trajectory $Tr[s_u, e_u] \in \mathbb{TS}$ is made up of contiguous objects, i.e., $Tr[s_u, e_u] = \{p_{s_u}, \ldots, p_{e_u}\}$;
2. $\bigcup_{Tr[s_u,e_u]\in\mathbb{TS}} Tr[s_u, e_u] = Tr, 1 \leq u \leq \kappa$;
3. $\bigcap_{Tr[s_u,e_u]\in\mathbb{TS}} Tr[s_u, e_u] = \emptyset, 1 \leq u \leq \kappa$.

Clearly, a brute-force approach to segment a trajectory is to simply keep the size of each segment be M, e.g., $M = 3$. Thus, $|Tr|/M$ segments are generated for each trajectory Tr. However, this method is poorly effective since it neglects the inner relation between objects within each trajectory. Based on this intuition, we propose a trajectory segmentation algorithm by leveraging spatial, temporal and semantic features together.

Spatial Feature. Spatial twist is a special property of individual travel pattern, thus generating a large number of roundabout trajectories. However, as travel distance is a major factor considered in KOAT, roundabout trajectories may lead to needless computing. As shown in Figure 3.1, if both $p_{1,2}(p_{2,2})$ and $p_{1,3}(p_{2,5})$ contain the query keywords, $Tr_2[2, 5]$ is obviously needless to process and not the best choice due to longer travel distance than $Tr_1[2, 3]$. Therefore to keep spatial homogeneous, we define roundabout distance to measure the degree of twists and turnings within a trajectory.

Definition 3.5. ***(Trajectory Roundabout Distance)*** *Given Tr, suppose the projection points of $p_2, \cdots, p_{|Tr|-1}$ onto the straight line $p_1 \rightarrow p_{|Tr|}$ are $p_2^{\perp}, \cdots, p_{|Tr|-1}^{\perp}$, respectively. And $d(p_i, p_i^{\perp})$ is the euclidean distance between p_i and $p_i^{\perp}$, for $1 < i < |Tr|$. The spatial feature value, i.e., roundabout distance $f_{rd}(Tr)$ is defined as follows:*

$$f_{rd}(Tr) = \sum_{1<i<|Tr|} d(p_i, p_i^{\perp}) \tag{3.10}$$

Temporal Feature. Generally, the timestamps in a trajectory could be very sparse, whereas they could be more dense within a single trip contained by the trajectory. On the purpose of extracting such single trips from trajectory, we propose to segment a trajectory by taking temporal feature into consideration. Hence, we define time variance of a trajectory to obtain dense time distributions to keep temporal homogeneous within a trajectory.

Definition 3.6. ***(Trajectory Time Variance)*** *Given $Tr = \{p_1, \ldots, p_{|Tr|}\}$ and $p_i.t$ is the timestamp of p_i. The mean time is $\bar{t} = p_1.t + \frac{\sum_{1<i\leq|Tr|} p_i.t - p_1.t}{|Tr|-1}$. The temporal feature value, i.e., time variance is defined as follows:*

$$f_{tv}(Tr) = \frac{1}{|Tr|} \sum_{1\leq i\leq|Tr|} (p_i.t - \bar{t})^2 \tag{3.11}$$

Semantic Feature. In activity trajectory database, PoI is a specific point location that someone may find useful or interesting. According to users different travel purposes, PoIs can be roughly classified into 10 categories as shown in Table 3.2. With the similar observation from temporal feature, different trips in a single trajectory can suffer a problem of PoI category duplication, which increases the redundant computation of keyword distance. Therefore to keep the categories of PoI contained in sub-trajectory as diverse as possible, we define the information entropy of a trajectory as follows:

Table 3.2: Category of PoIs

	Category	**Typical Keywords**
1	Home	Apartment, house
2	Work	Government, office, building
3	Education	School, training center
4	Food	Restaurant
5	Entertainment	Museum, theater, club
6	Outdoor	Park, sports field
7	Shopping	Shop, mall, outlet
8	Transportation	Airport, railway, bus station
9	Health care	Hospital, medical center, pharmacy
10	Others	Other keywords

Definition 3.7. ***(Trajectory Information Entropy)*** *Given a trajectory $Tr = \{p_1, \ldots, p_{|Tr|}\}$, with $p_i.\Phi = \{w_i^1, \ldots, w_i^{|p_i.\Phi|}\}$. For each w_i^j, $C(w_i^j)$ is the corresponding category w_i^j belongs to. Thus, the semantic feature value, i.e., information entropy is defined by*

$$f_{ie}(Tr) = -\sum_{\forall i,j} p(w_i^j) \log p(w_i^j) \tag{3.12}$$

where $p(w_i^j)$ is the proportion of a category $C(w_i^j)$ in the PoI collection.

Objective Function. To partition the trajectory Tr into sub-trajectories $Tr[s_u, e_u] \in \mathbb{TS}$, we first define the weight average feature value of Tr based on its segmentation $\mathbb{TS}$ w.r.t. each feature $\delta \in \{rd, tv, ie\}$, denoted as $WA_\delta(\mathbb{TS}, Tr)$:

$$WA_\delta(\mathbb{TS}, Tr) = \sum_{Tr[s_u,e_u]\in\mathbb{TS}} \frac{|Tr[s_u, e_u]|}{|Tr|} f_\delta(Tr[s_u, e_u]) \tag{3.13}$$

As we know, different segmentations would result in different degrees of decrease in feature values. The decrease in feature value of Tr resulted by segmentation $\mathbb{TS}$ is defined as:

$$\triangle\mathbb{D}_\delta(\mathbb{TS}, Tr) = \frac{f_\delta(Tr) - WA_\delta(\mathbb{TS}, Tr)}{f_\delta(Tr)} \tag{3.14}$$

In addition, each feature has a different requirement on segmentation granularity, which indicates how fine-grained the trajectory is partitioned and described. With aforementioned three features, we propose a combination score function, $\triangle\mathbb{D}(\mathbb{TS}, Tr)$, to take all these three features into account, which is defined as follows:

$$\triangle\mathbb{D}(\mathbb{TS}, Tr) = \sum_{\delta} \lambda_\delta \cdot \triangle\mathbb{D}_\delta(\mathbb{TS}, Tr) \tag{3.15}$$

where $\sum \lambda_\delta = 1$ and λ_δ is chosen by the weight of feature importance, which could have an effect on the search performance. It is worth to notice that $\lambda_\delta = 0$ means the feature δ is not considered.

As mentioned before, the goal of trajectory segmentation is to keep the trajectory as compact as possible and in some sense of homogeneousness. Therefore, we aim to find an "optimal" segmentation such that number of segments $\kappa = |\mathbb{TS}|$ is minimum, and for all segments $Tr[s_u, e_u] \in \mathbb{TS}$, the maximum decrease of feature value of its binary segmentation is less than threshold θ, i.e., $\max\{\triangle\mathbb{D}(\mathbb{TS}_{Tr[s_u,e_u]}, Tr[s_u, e_u])\} < \theta$ with $|\mathbb{TS}_{Tr[s_u,e_u]}| = 2$. Therefore, the objective function is:

$$\begin{gathered} \text{Minimize } \kappa \\ \text{Subject to } \max\{\triangle\mathbb{D}(\mathbb{TS}_{Tr[s_u,e_u]}, Tr[s_u, e_u])\} < \theta \\ \forall Tr[s_u, e_u] \in \mathbb{TS}, |\mathbb{TS}_{Tr[s_u,e_u]}| = 2 \end{gathered} \tag{3.16}$$

Greedy Trajectory Segmentation. For trajectory segmentation, we propose a greedy algorithm to partition the trajectory in a binary fashion. In each iteration, Tr is partitioned as $\mathbb{TS} = \{Tr[1, i], Tr[i + 1, |Tr|]\}$. We first compute $\triangle\mathbb{D}(\mathbb{TS}, Tr)$ for each object p_i in Tr, and the algorithm "greedily" chooses the current best split object p_i with $max\{\triangle\mathbb{D}(\mathbb{TS}, Tr)\}$, thus Tr is partitioned into $Tr[1, i]$ and $Tr[i + 1, |Tr|]$. Then we come into next iteration and repeat the computation. It's worth to notice that this process terminates when $max\{\triangle\mathbb{D}(\mathbb{TS}_{Tr[s_u,e_u]}, Tr[s_u, e_u])\}$ is less than the threshold θ.

3.6.2 Search with Segmented Trajectories

Based on trajectory segmentation, we propose an enhanced algorithm to process KOAT query more efficiently, and we only highlight some important steps and changes of our method for simplification.

In AG-Tree, we modify the entry of leaf node p in form of $(s_p, \text{SID}, \text{TID})$, where SID is ID of sub-trajectory $Tr[s_u, e_u]$, and the structure of non-leaf node remains the same. In K-Ref, we apply the reference keywords selection on each sub-trajectory $Tr[s_u, e_u]$, thus we obtain $|\mathbb{TS}|$ reference keyword sets. Similarly, each object $p \in T[s_u, e_u]$ is indexed in a B$^+$-Tree. In addition, the lower bound $LB(w_{r_n}^{Tr[s_u,e_u]})$ and upper bound $UB(w_{r_n}^{Tr[s_u,e_u]})$ for each reference keyword are given as well.

Candidate retrieval. Since the format of entries kept in AG-Tree is changed, each candidate entry in CS is kept in the form of $(\gamma(s_{q_i}, s_{Tr[s_u,e_u]}), \text{SID}, \text{TID}, q_i)$. The procedure of candidate retrieval is the same as the previous algorithm.

Computing lower bound. Computing lower bound is the most critical procedure in this algorithm. Once the candidate set CS is obtained, we will be faced with one of these two situations: 1) Inner sub-trajectory computing; 2) Cross sub-trajectory computing. Next we will introduce how these two situations are solved and how to combine them to compute the lower bound for each candidate trajectory.

1) **Inner sub-trajectory computing.** For sub-trajectory $Tr[s_u, e_u] \in \mathbb{TS}_{Tr}(1 \leq u \leq \kappa)$, the lower bound needs to be computed by inner sub-trajectory approach, if and only if $Tr[s_u, e_u]$ is a "complete" candidate, which means the entries stored in CS related to $Tr[s_u, e_u]$ must be corresponding to all query keywords. It is worth to notice that the goal of trajectory segmentation is to extract single travel experience from multiple ones, and hopefully we only need to consider such "complete" sub-trajectories as query candidates.

In the inner sub-trajectory computing, $Tr[s_u, e_u]$ is considered as an independent trajectory. As $Tr[s_u, e_u]$ has a comparable smaller size than Tr, the number of states (v) in DPA algorithm would obviously decreas and achieve much better efficiency in computing $R_{st}(Q, Tr[s_u, e_u])_L$. For all "complete" $Tr[s_u, e_u] \in \mathbb{TS}_{Tr}$, we keep a lower bound $IN(Q, Tr) = \min\{R_{st}(Q, Tr[s_u, e_u])_L\}$.

2) **Cross sub-trajectory computing.** If there are more than one sub-trajectories contained in CS for a single candidate, we need to consider the cross sub-trajectory computing. Determining the value of spatio-temporal textual ranking function among sub-trajectories is complicated due to the variances

of both textual distances and lengths of cross sub-trajectories, hence we create a "virtual" object $p_{Tr[s_u,e_u]}$ regarding all the keywords in $Tr[s_u, e_u]$ as its keywords for each sub-trajectory $Tr[s_u, e_u]$.

By adopting the virtual object, we follow the same direction of DPA algorithm to choose intermediate state $Tr[s_c, e_c]$, where the difference is that the objects now contained in $Tr[s_c, e_c]$ are virtual objects representing sub-trajectories w.r.t. Q.

For each state (v), we aim to compute the lower bound of matching distance for current $Tr[s_c, e_c]$. For lower bound of travel distance, $len(Tr[s_c, e_c])_L$ is computed by *euclidean distance* between the closest objects locating in the furthest sub-trajectories, which are represented by virtual objects $\mathtt{p}_i$s determining $Tr[s_c, e_c]$. On the other hand, $D_{td}(Q, Tr[s_c, e_c])_L$ is the sum of lower bound *edit distances* between virtual objects $\mathtt{p}_i$ and $q_i \in Q$. As mentioned before, we index all the keywords in sub-trajectories based on their *edit distances* to each reference keyword $w_{r_n}^{Tr[s_u,e_u]}$. Therefore, we can easily obtain $D_{td}(Q, Tr[s_c, e_c])_L$ by accessing K-Ref. When DPA algorithm terminates, we keep a cross sub-trajectory lower bound $CR(Q, Tr) = (v) = D_{md}(Q, Tr[s_c, e_c])_L$. Finally, we have $R_{st}(Q, Tr)_L = \min\{IN(Q, Tr), CR(Q, Tr)\}$.

3.7 Experiments

In this section, we conduct extensive experiments on real datasets to study the performance of proposed index structures and algorithms.

3.7.1 Experimental Settings

Algorithms evaluated. We study the performance of two baseline algorithms probe based algorithm (PBA), inverted list based algorithm (ILA), and our dynamics programming algorithm (DPA) for query processing, enhanced trajectory segmentation optimization algorithm (TSA). All these algorithms were implemented in Java on Window 7 and run on an Intel(R) CPU i7-4770 @3.4GHz and 16G RAM.

Data and queries. We use a real activity trajectory dataset by crawling the online check-in records of Foursquare within the areas of Los Angeles (LA) and New York City (NYC) [124]. Each check-in record of Foursquare contains the user ID, venue with geo-location (PoI), time of check-in, and tips

written in English. The detailed statistics of dataset are given in Table 3.3.

TABLE 3.3: Statistics of dataset

	LA	NYC
#trajecotry	31,557	49,027
#POIs	3,164,124	2,056,785
#query objects per trajectory	51.83	38.27
#keywords per object	2.84	2.23

3.7.2 Efficiency Measurement

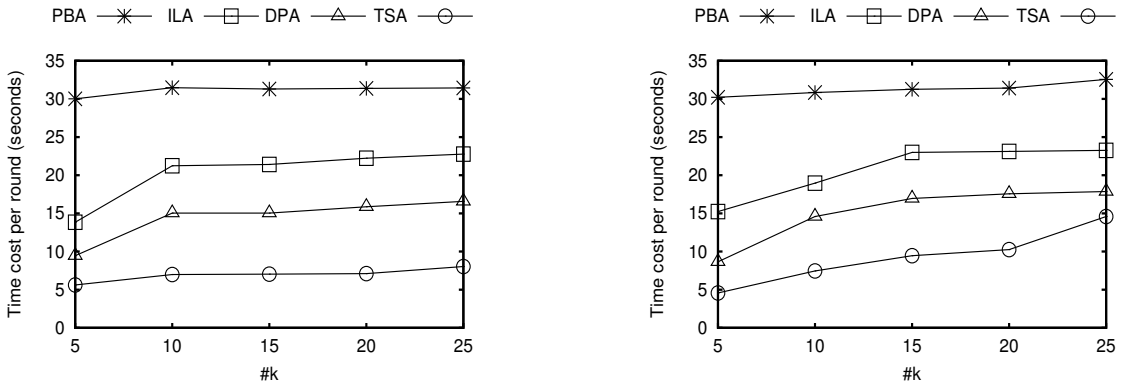
For efficiency measurement, we will first compare time costs of these four algorithms under different constraints, such as the number of results k, the number of query keywords $|Q|$, the cardinality of dataset $|D|$. Then we will study the effect of partition granularity of GiKi on proposed algorithm DPA and enhanced algorithm TSA, which compare the time cost and memory cost. Finally, we will study the effect of trajectory segmentation TSA algorithm resulted by different weights of features, which compare the time costs and pruning rates. By default, we build a d-Grid with $d = 8$ for activity trajectory dataset, which means the entire space is partitioned into $2^8 \times 2^8$ grids. The default values for other parameters are summarized in Table 3.4.

TABLE 3.4: Parameter settings

#results k	10
#query keywords $\|Q\|$	3
#α	0.7
#d	8
#λ_δ	$\{0.4, 0.3, 0.3\}$

Effect of k. In the first set of experiments, we study the effect of the intended number of results k by plotting the average time costs of PBA, ILA, DPA and TSA on both LA and NYC datasets. As shown in Figure 3.4, our proposed indexing approach, GiKi, significantly outperforms inverted indexing method on both datasets. In particular, PBA is the algorithm without any index structure, and ILA uses inverted list to retrieve candidates, which is faster than PBA. For DPA and TSA utilizing

GiKi, it is 2-3 times faster than PBA and ILA. Since ILA finds all the trajectory candidates and then compute the smallest value of spatio-textual utility function by applying method of PBA, the running time remains constant for all the values of k. The other two proposed methods incur higher cost as k increases, since k-th smallest keyword travel distance becomes greater and more candidates need to be retrieved and refined.

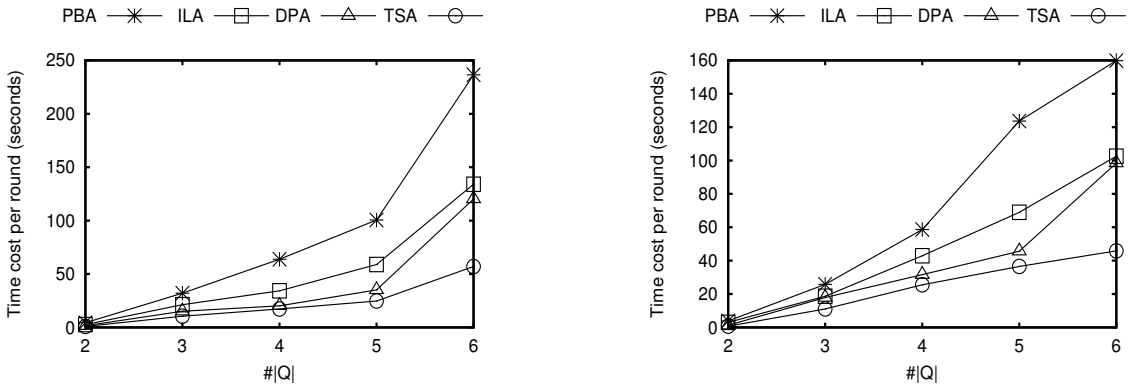


(a) LA (b) NYC

FIGURE 3.4: Effect of k for KOAT query

Effect of $|Q|$. Next we study the query performance when the number of query keywords $|Q|$ varies. The results are presented in Figure 3.5. Again, our proposed algorithms has superior performance than all baseline algorithms. All these four methods incur more time cost with the increase of query keywords, this tendency becomes extremely obvious when $|Q|$ becomes 4. This is because computing *edit distance* is a costly operation, the more keywords we get, the more time it takes. In addition, more query keywords lead to more different objects combinations, which can result in an exponential increase in execution time. It is worth to notice that DPA does not show a greater increase in performance than ILA, since even if we prune more candidates, the computation of *edit distance* still requires a lot of time. However, by applying TSA, trajectories are partitioned into small pieces, which highly reduces the operation time of computing both *edit distance* and travel distance within sub-trajectories or between sub-trajectories.

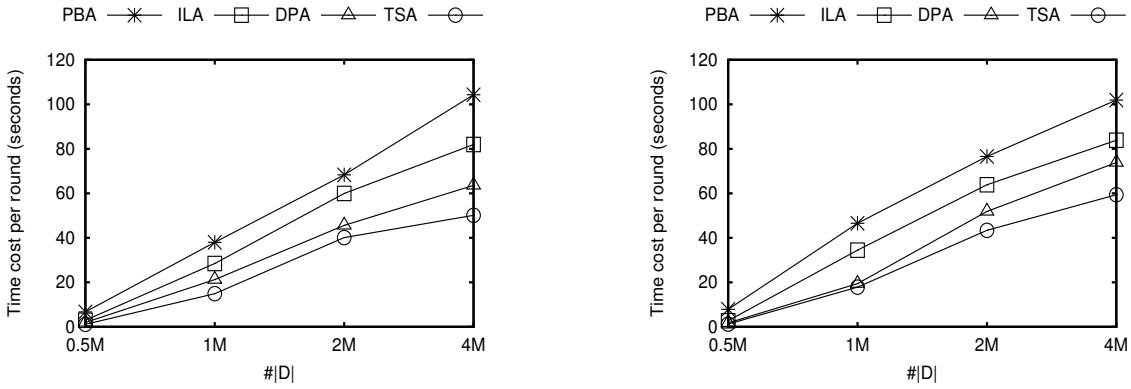
Effect of $|D|$. Then we investigate the query performance w.r.t. the cardinality of datasets on both LA and NYC. The results are shown in Figure 3.6. Without surprise, in such scalability experiments, we can see that the time costs of all four methods increase linearly w.r.t. the size of datasets, i.e., 0.5M, 1M, 2M and 4M, this is because the algorithms are faster when the datasets are small. Moreover, we



(a) LA (b) NYC

FIGURE 3.5: Effect of $|Q|$ for KOAT query

also can see that the computation times are high when the size of the datasets are 2M and 4M, this is because they get more candidates to process even if they have index structures to prune trajectories. In addition, it is worth to notice that our proposed methods, i.e., DPA and TSA, scale much better than baseline methods on both LA and NYC datasets since our algorithms and optimization highly reduce the number of candidate trajectories need to be considered.

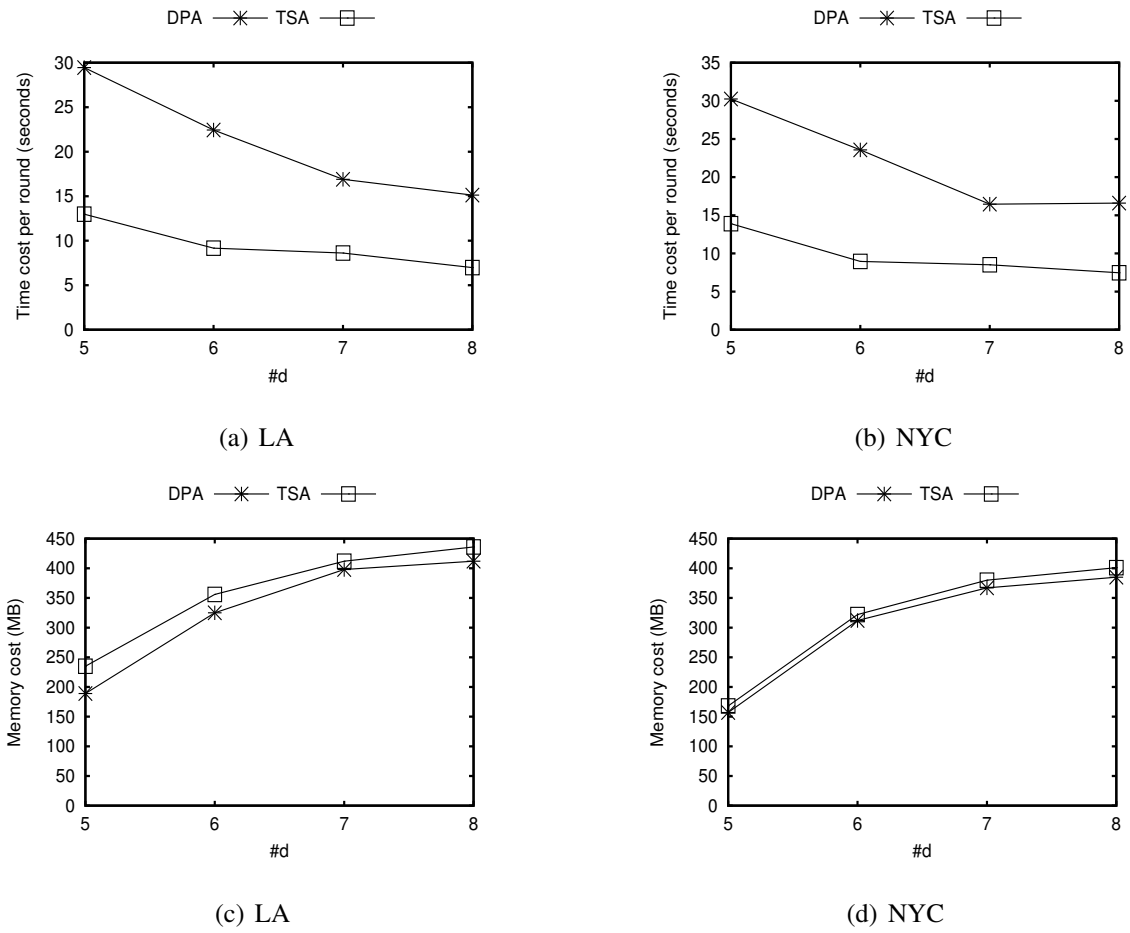


(a) LA (b) NYC

FIGURE 3.6: Effect of $|D|$ for KOAT query

Effect of d. We now proceed to examine the effect of the partition granularity of AG-Tree. Recall that by default we partition the entire space into 256×256 grids ($d = 8$). In this set of experiments, we set the number of partitions to $32 \times 32(d = 5)$, $64 \times 64(d = 6)$, $128 \times 128(d = 7)$ and $256 \times 256(d = 8)$ and record the running time and memory cost of DPA and TSA on both LA and NYC datasets, since only these two proposed methods use AG-Tree for indexing. The results are shown in Figure 3.7.

Generally, better performance will be achieved for both DPA and TSA by using GiKi index with finer granularity since tighter lower bound keyword travel distance can be derived with smaller sized grids. As mentioned before, coarse grain grid incurs more computation inside grid and more storage cost, just as the experiment results show that the time cost decrease and memory cost increase with the enlarging of d. But we also can see that, these decreasing and increasing patterns are slow down when we enlarge d from 7 to 8. This is because larger value of d incurs more grids need to consider from up to down in AG-Tree. Besides, the memory costs of DPA and TSA are very close, since the two methods use the same index structure and TSA only utilizes a little extra memory to store sub-trajectory information.



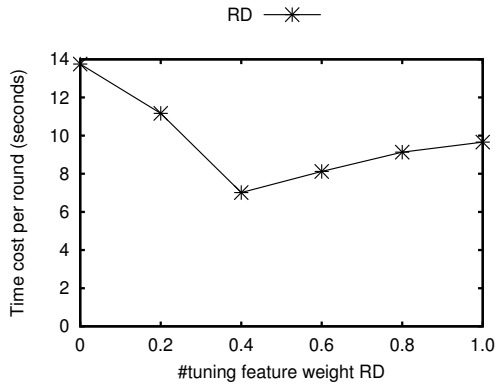
(a) LA (b) NYC

(c) LA (d) NYC

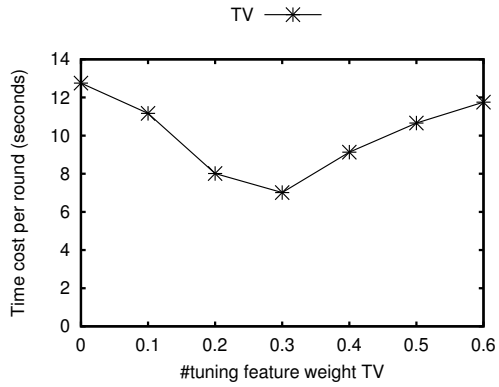
FIGURE 3.7: Effect of d for KOAT query

Effect of λ. We study the effect of trajectory segmentation on query performance by applying TSA, which is resulted by different weights of feature importance. As we know, different segmentation

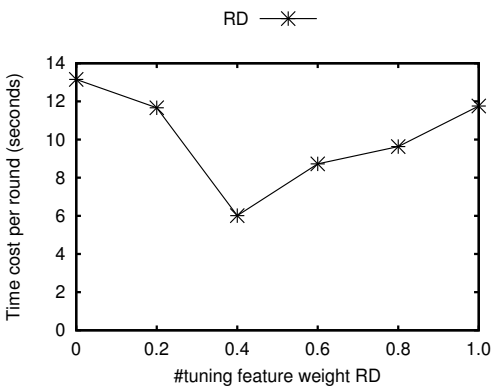
ways may lead to total different computation time, and the weights of different feature value directly affect the segmentation results. In this experiment, we study the best weight, i.e., λ_{rd}, λ_{tv} and λ_{ie}, that achieves highest efficiency of query processing. Figure 3.8 shows the experiment results, we first tune the value of λ_{rd} and keep the rest $\lambda_{tv} = \lambda_{ie}$. We can see that when the value of λ_{rd} is 0.4, the corresponding time costs achieve a high performance. If we continue to enlarge the value of λ_{rd}, the time cost increases. Therefore, we choose $\lambda_{rd} = 0.4$ in both the datasets of LA and NYC. After tuning λ_{rd}, we continue to tune the value of λ_{tv}. In LA dataset, we can see that when we set $\lambda_{tv} = 0.3$, i.e., $\lambda_{tv} = \lambda_{ie} = 0.3$, the time cost is minimized. In NYC dataset, it is easy to see that when $\lambda_{tv} = 0.4$ the time cost achieves the best performance, and $\lambda_{ie} = 0.2$. This because when we compute the value of spatio-temporal ranking function, the spatial feature is mainly concerned, which means spatial feature plays a more important role in distance computing.



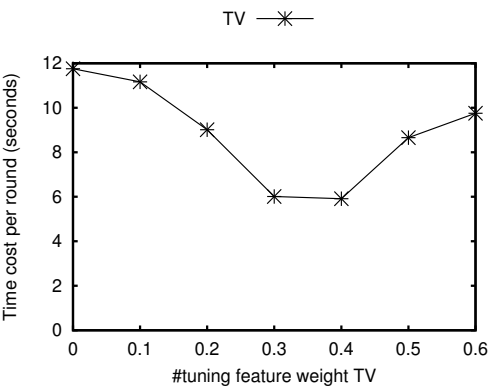
(a) LA



(b) LA



(c) NYC



(d) NYC

FIGURE 3.8: Effect of λ for KOAT query

3.8 Summary

This chapter studies the problem of searching activity trajectory database given multiple query keywords without location. The major differences between this work and exiting works are that we do not have a query location at query time and we support approximate search condition on keywords. To support efficient query processing, we develop a novel index structure, called GiKi that includes two components, i.e., AG-Tree and K-Ref, to index the activity trajectory database. Based on such index structure, we propose efficient algorithm to compute the minimum value of spatio-temporal ranking function. In the query processing, we follow the pruning and refinement paradigm to answer the query by a process of candidate retrieval, lower bound computation and candidate validation. Specifically, we propose a dynamic programming algorithm to compute the lower bound for each candidate trajectory. In addition, we propose a trajectory segmentation algorithm to partition trajectories by leveraging multiple features. Then we propose an enhanced search algorithm with such segmentation method to answer the KOAT query more efficiently. Extensive experimental results demonstrate that the proposed methods outperform baseline algorithms significantly and achieve good scalability.

Chapter 4

Keyword-aware Continuous kNN Queries on Road Networks

It is nowadays quite common for road networks to have textual contents on the vertices, which describe auxiliary information (e.g., business, traffic, etc.) associated with the vertex. In such road networks, which are modelled as weighted undirected graphs, each vertex is associated with one or more keywords, and each edge is assigned with a weight, which can be its physical length or travelling time. In this chapter, we study the problem of *keyword-aware continuous k nearest neighbour* (𝕂C*k*NN) search on road networks, which computes the *k* nearest vertices that contain the query keywords issued by a moving object and maintains the results continuously as the object is moving on the road network. Reducing the query processing costs in terms of computation and communication has attracted considerable attention in the database community with interesting techniques proposed. This work proposes a framework, called a Labelling AppRoach for Continuous *k*NN query (*LARC*), on road networks to cope with 𝕂C*k*NN query efficiently. First we build a pivot-based reverse label index and a keyword-based pivot tree index to improve the efficiency of *keyword-aware k nearest neighbour* (𝕂*k*NN) search by avoiding massive network traversals and sequential probe of keywords. To reduce the frequency of unnecessary result updates, we develop the concepts of dominance interval and region on road network, which share the similar intuition with safe region for processing continuous queries in Euclidean space but are more complicated and thus require more dedicated design. For high frequency keywords, we resolve the dominance interval when the query results changed. In

addition, a path-based dominance updating approach is proposed to compute the dominance region efficiently when the query keywords are of low frequency. We conduct extensive experiments by comparing our algorithms with the state-of-the-art methods on real data sets. The empirical observations have verified the superiority of our proposed solution in all aspects of index size, communication cost and computation time.

This chapter is organized as follows. We give an introduction in Section 4.1 and describe the problem statement in Section 4.2. Section 4.3 presents the framework of our solution, i.e., *LARC* and analyses the algorithms in detail. In Section 4.4, we introduce an enhanced algorithm *LARC++* to construct the dominance region. Our empirical observations are explained in Section 4.5. Section 4.6 concludes this chapter.

4.1 Introduction

With the rapid development of GPS-enabled smart mobile devices and location-based services, there is a clear trend that objects are increasingly being geo-tagged. To provide better user experience, these services maintain location-related information to answer user queries w.r.t. user-specified location. In addition to the spatial characteristics, a user may also have specific requirement on the description of the object such as "restaurant", "hotel", "petrol station", etc. For example, a person wants to find a restaurant within 10 minutes walking distance. Such queries, known as spatial keyword queries, which find the top-k objects of interest in terms of both spatial proximity and textual relevance to the query, have been extensively studied in recent years [30, 59, 64, 80, 83, 121, 125, 131]. All these studies have focused on static query objects whose locations are fixed throughout the query lifetime. However, many real-world applications have the requirements to support the continuous k nearest neighbour (CkNN) queries, or also known as moving k nearest neighbour queries. For instance a Uber service provider looking for potential passengers is driving on the road, the k nearest potential passengers that have requested taxis should be reported to him through the application continuously. The CkNN query can also be used to report the k nearest petrol stations continuously while a car is running low of fuel.

Existing techniques for the static kNN query are not directly applicable for the CkNN query. Therefore, on-going efforts have been made to meliorate the user experience by improving the CkNN

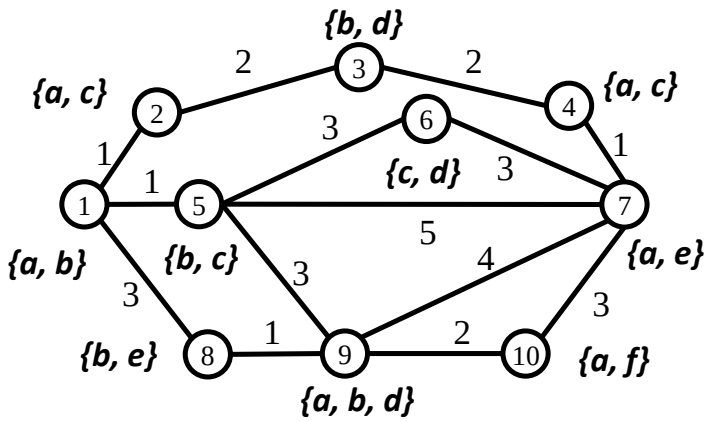


Figure 4.1: Running example for $\mathbb{K}$CkNN query

query processing efficiency [48, 62, 72, 74, 102]. Most of these works adopt the idea of "safe region" where all the inside points share the same kNN results, thus reducing the query processing cost in terms of both computation and communication. However, they assume objects are moving in free space, which might be inappropriate especially in urban areas where the movements of objects are constrained by the road network. Consider the example in Fig. 4.1, the current location of the query object is at v_6 and the query keyword is "b", thus v_3 that contains "b" would be the 1NN result in terms of Euclidean distance. However, v_5 is actually what we want instead of v_3 in terms of the network distance. Meanwhile, [41, 66, 102] study the problem of continuous top-k spatial keyword query on road networks by incorporating the spatial proximity and textual relevance to form a similarity function. However, this kind of similarity functions, which simply combine two unrelated dimensions together, usually cannot satisfy user's search intention well. As seen in more and more commercial location-based services, a more intuitive and practical query formulation for spatial keyword search is to find the objects that simply contain the query keywords. Motivated by these requirements and oversights of existing works, we study the *keyword-aware continuous k nearest neighbour* ($\mathbb{K}$CkNN) on road networks, which computes the kNN results that contain the query keywords and maintains the results in a continuous manner.

In order to process the $\mathbb{K}$CkNN query efficiently, we need to overcome several challenges. The first challenge is concerned with computing the network distances between vertices, as well as finding the *keyword-aware k nearest neighbour* ($\mathbb{K}k$NN) results efficiently. Unlike the Euclidean space, processing distance queries in scalable networks is a complicated problem. Given such a query and a

network, an obvious solution is to apply Dijkstra's search [34] from the query location to hit the target vertices or find the kNN results that contain the keyword. However, this method becomes inefficient when dealing with large-scale road networks due to massive traversals. Inspired by the 2-hop label index, which answers distance queries with small response time [1, 2, 6], we reorganize the structure of label index and build *pivot-based reverse label index* to fit our $\mathbb{K}k$NN search problem. For the keyword checking, we utilize a probabilistic data structure, i.e., the bloom filter [21], to skip sequential probe of all keywords. By combining this with the label index, we construct *keyword-based pivot tree*, by which means both the distance query and $\mathbb{K}k$NN query can be efficiently processed.

The second challenge is related to deriving a dominance interval or region as large as possible. Although some existing works adopt safe region technique to reduce the query processing cost [62, 74], they fall short either in the region construction overhead or validation overhead. In addition, the safe region in Euclidean space is just surrounded by several bisect lines, and for each object pair, there exists only one bisect line. Nevertheless, in a road network, the dominance region is determined by bisect points, and each object pair may have several bisect points since they may be connected by multiple paths, which makes the construction of dominance region even more complex and time consuming. Moreover, the area of the dominance region is highly dependent on the frequency of the query keyword. Therefore, for high frequency keywords, we adopt a *window sliding approach* to build a dominance interval with low costs. For low frequency keywords, we propose a *path-based dominance updating approach* to resolve the dominance region on road network, which guarantees the validity of the current $\mathbb{K}k$NN results and significantly reduces the computation and communication costs.

The major contributions of this chapter can be summarized as follows:

- By utilizing the labelling approach, we construct *keyword-based label index* that consists of *pivot-based reverse label* and *keyword-based pivot tree*. With such an index structure, we improve the $\mathbb{K}k$NN query efficiency by skipping the massive network traversals and sequential probe of keywords. For $\mathbb{K}\mathbb{C}k$NN query, we propose a *LARC* algorithm to resolve the dominance intervals by a *window sliding approach* for the moving object when dealing with high frequency keywords.

- For low frequency keywords, we also develop a *LARC++* algorithm that employs a *path-based*

dominance updating approach to construct an effective dominance region with low costs. This way, the frequency of communication between server and client is thoroughly reduced. In addition, a hybrid algorithm *LARC-C* that combines *LARC* and *LARC++* is introduced to cope with all cases of keyword frequency.

- Our experimental evaluation demonstrates the effectiveness and efficiency of our framework for processing the $\mathbb{K}$CkNN queries on real-world datasets. We show the superiority of our methods in answering $\mathbb{K}$CkNN queries efficiently, when compared with the state-of-the-art methods.

4.2 Problem Statement

This section formally defines the $\mathbb{K}k$NN and $\mathbb{K}$CkNN queries. Table 4.1 summarizes the major notations.

Table 4.1: Summary of notations in $\mathbb{K}$CkNN

Notation	**Definition**
$G = (V, E)$	Road network with vertex V and edge E
$\mathcal{P}(u, \ldots, v)$	A path from u to v
$\Phi(v)$	The keywords associated with v
$d_G(u, v)$	Network distance between u and v in G
$\mathtt{p}(u, v, d_s)$	A point $\mathtt{p}$ on edge (u, v) with d_s to u
$L(v)$	2-hop label of v
$PR(o)$	Pivot-based reserve label of vertex o
$KP(o)$	Keyword-based pivot tree of vertex o
$\mathcal{DI}(R)$	The dominance interval of R
$\mathcal{DR}(R)$	The dominance region of R

We model a road network as a weighted undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each edge $(u, v) \in E$ has a positive weight, i.e., physical length or travelling time, denoted as $l(u, v)$. Each vertex $v \in V$ contains a set of keywords, denoted as $\Phi(v)$. Note that the algorithms proposed in this chapter can be easily extended to the case that keywords locate on edges. Given a path between vertices u and v, denoted as $\mathcal{P}(u, \ldots, v)$, the length is the total weight of

all edges along the path. For any two vertices u and v, the distance between u and v on G, denoted as $d_G(u, v)$, is the length of the shortest path $\mathcal{SP}(u, v)$ between u and v.

Definition 4.1 (**Keyword-aware kNN**). *The Keyword-aware k Nearest Neighbour (*$\mathbb{K}$*kNN) query is defined as follows: Given a road network G, the query Q includes a query location l_q, a query keyword w_q, and a positive integer k, i.e., $Q = (l_q, w_q, k)$. Note that l_q can be either a vertex v_q or a point* $\mathtt{p}$ *on edge (u, v), denoted as $\mathtt{p_q}(u, v, d_s)$, where d_s is the distance along (u, v) between $\mathtt{p_q}$ and u. The query result consists of k vertices, denoted as $R = \{v_1, v_2, \ldots, v_k\} \subseteq V$, that are nearest to l_q among all vertices in V in terms of network distance, and each of which contains the query keyword w_q, i.e., $\forall v \in R, w_q \in \Phi(v)$.*

Definition 4.2 (**Keyword-aware CkNN**). *Given a road network G, and a moving query $Q = (l_q, w_q, k)$, a Keyword-aware Continuous k Nearest Neighbour (*$\mathbb{K}$C*kNN) query keeps returning the* $\mathbb{K}$*kNN results for every new location l_q of Q.*

Example 4.1. *As shown in Fig. 4.1, a* $\mathbb{K}$C*1NN query Q with keyword "d" is moving from v_1 to v_5. The result of Q reported at v_1 is $\{v_3\}$ and then changes to $\{v_6\}$ at $\mathtt{p}(v_1, v_5, 0.5)$. No update is needed elsewhere since the result does not change.*

4.3 Algorithm *LARC*

In this section, we introduce our labelling approach for $\mathbb{K}$CkNN query (*LARC*). Different from on-line search algorithms on graphs, e.g., Dijkstra, A* algorithms, etc, our method preprocesses the network G to construct an index structure and organizes the vertices with distance and keyword information to enhance the $\mathbb{K}k$NN search efficiency. With the help of such index, we can skip a large portion of disqualifying vertices, that either are far away from the query location or do not contain the query keyword, by looking up the information stored in the index entries. Similar to existing works on CkNN, we adopt the concept of dominance interval on road network to cope with $\mathbb{K}$CkNN query. As long as the moving object stays on the dominance interval, the $\mathbb{K}k$NN results maintain valid and no recomputation is required, thus both the computation and communication costs are reduced.

4.3.1 Keyword-based Label Index

Inspired by recent development in shortest path computation and distance query over large graphs, we make use of the 2-hop labelling technique [1] [2] [6] [54] as the base to construct our index for $\mathbb{K}$CkNN query processing.

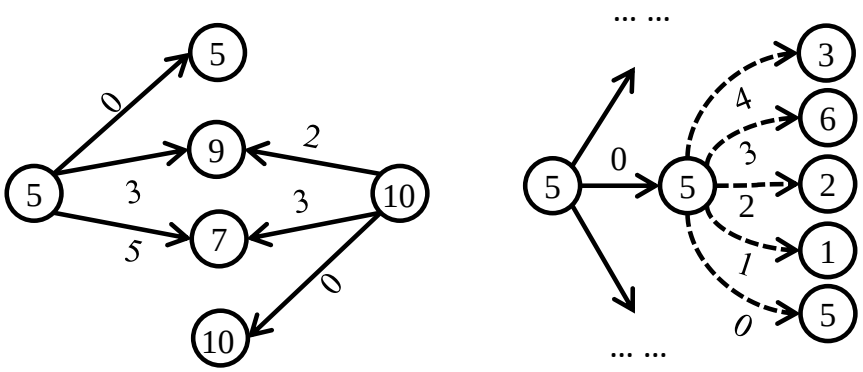
2-hop Label Index. The 2-hop label, also known as 2-hop cover, constructs labels for vertices such that a distance query for any vertex pair u and v can be answered by only looking up the common labels of u and v. For each vertex v, we precompute a label, denoted as $L(v)$, which is a set of label entries and each label entry is a pair $(o, \eta_{v,o})$, where $o \in V$ and $\eta_{v,o} = d_G(v, o)$ is the distance between v and o. We say that o is a **pivot** in label entry if $(o, \eta_{v,o}) \in L(v)$. Given two vertices u and v, we can find a common pivot o that $(o, \eta_{u,o}) \in L(u)$ and $(o, \eta_{v,o}) \in L(v)$:

$$d_G(u, v) = \min\{\eta_{u,o} + \eta_{v,o}\} \tag{4.1}$$

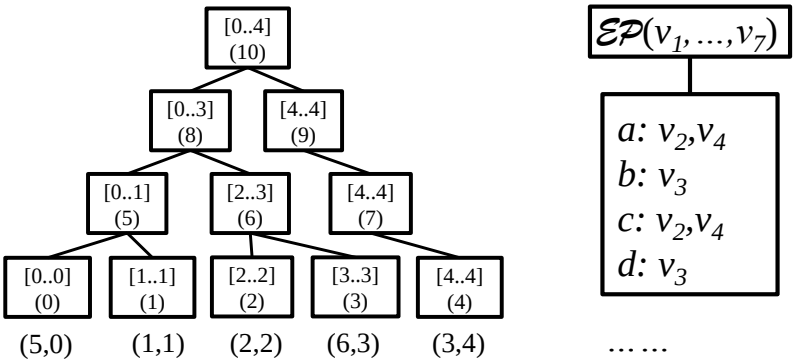
We say that the pair (u, v) is covered by o and the distance query $d_G(u, v)$ is answered by o with smallest $\eta_{u,o}+\eta_{v,o}$. As shown in Fig. 5.2, $L(v_5) = \{(v_5, 0), (v_9, 3), (v_7, 5)\}$ and $L(v_{10}) = \{(v_{10}, 0), (v_7, 3), (v_9, 2)\}$, then we have $d_G(v_5, v_{10}) = 3 + 2 = 5$.

2-hop label has been extensively studied in existing works, which can correctly answer the distance query between any two vertices in a graph, whilst keeping the size of the generated label index as small as possible. The problem of reducing label size is orthogonal to our work. We can fully utilize the state-of-the-art results to build a smaller index in our method.

Keyword-based Pivot Index. As 2-hop label possesses the nature to process distance queries with fast response time, we modify the structure of original 2-hop label to construct a *pivot-based reverse index*, i.e., *PR* index, for $\mathbb{K}k$NN query processing. In 2-hop label, the distance between any vertex pair (u, v) can be computed correctly through their common pivot o, in other words, each vertex u can reach any other vertex v in graph through a pivot o. Therefore, we store all the label entries $(o, \eta_{v,o}) \in \bigcup_{v \in V} L(v)$ regarding vertex o as pivot into the *PR* label of vertex o, i.e., $(v, \eta_{v,o}) \in PR(o)$. In $PR(o)$, we assume that all the label entries $(v, \eta_{v,o})$ are sorted in non-decreasing order of distance. Generally, given vertex u, we first find its pivot o with smaller distance in $L(u)$, then we continue to search $PR(o)$ for target vertices v containing w_q incrementally until we obtain k results. As shown in Fig. 4.2(b), given a $\mathbb{K}$1NN query $Q = (v_5, \text{“}a\text{”}, 1)$, we first search $PR(v_5)$ since $(v_5, 0) \in L(v_5)$. In sequential probe of $PR(v_5)$, we have “a”$\notin \Phi(v_5)$ and “a”$\in \Phi(v_1)$. Thus v_1 is reported as result.



(a) 2-hop label (b) PR index



(c) KP index (d) EP index

FIGURE 4.2: Overview of keyword-based label index.

However, we may still suffer a problem of inefficiency due to the keyword sparsity. Assume that the query keyword w_q is of low frequency, and only shows up in a few vertices, then we have to sequentially check the *PR* labels of many nearby pivots w.r.t. l_q until we reach vertices that contain w_q, which may incur significant traversal overheads. As we know, skipping some vertices that do not contain w_q allows for faster search. [53] proposes a forest index, which is a set of tree structures, to deal with kNN query on large graphs. As road networks are almost planar, the average number of entries in $PR(o)$, i.e., $\frac{|PR(V)|}{|V|}$, is usually small due to the limited number of vertices and edges. Therefore, we propose a *keyword-based pivot index*, i.e., *KP* index, to improve the search efficiency. For each pivot o, we take $PR(o)$ as input and simply construct a binary tree $KP(o)$. Specifically, we preserve each label entry $(v, \eta_{v,o})$ of $PR(o)$ as leaf node of $KP(o)$. For each non-leaf node of $KP(o)$, we denote the index range of leaf nodes that it covers as $[x \ldots y]$ and store the keyword information of all its sub-nodes so that the vertices containing w_q can be retrieved efficiently, as shown in Fig. 4.2(c).

For each keyword w, we utilize a hash function $H(w)$ to generate a binary code for keyword inspection. Hence, the hash code of a vertex or non-leaf node $H(X)$ is the superimposition of $H(w)$ it covers, which is generated by the bitwise operation $\vee$. A non-zero value of $H(w) \wedge H(X)$ indicates that X may contain w. However, the general hash function is unable to control the false positive rate. By contrast, while risking some space, the bloom filter [21] has a strong false positive controlling advantage for membership checking. Therefore, we adopt the bloom filter to compress the keywords instead of a general hash function in the KP index.

Enclosed Path Index. Before introducing the EP index, we first give a formal definition of the enclosed path,

Definition 4.3 (**Enclosed Path**). *We define a vertex whose degree is equal or greater than 3 as an intersection vertex. An enclosed path, denoted as $\mathcal{EP}(s, \ldots, e)$, is a path that only the starting and ending vertices are intersection ones among all vertices it passes. Formally, $\mathcal{P}(s, \ldots, e)$ is enclosed if and only if $s_{.deg} >= 3$, $e_{.deg} >= 3$ and $o_{.deg} < 3, \forall o \in \mathcal{P} \setminus \{s, e\}$.*

For each $\mathcal{EP}(s, \ldots, e)$, we construct a keyword posting list for each keyword w contained by $o \in \mathcal{EP} \setminus \{s, e\}$, which is a list of the vertices that contain w, as shown in Fig. 4.2(d). Given $\mathcal{EP}$ and query keyword w_q, the vertices in middle of $\mathcal{EP}$ that contain w can be quickly retrieved.

4.3.2 $\mathbb{K}k$NN Query Processing

With the index construction, given a query $Q = \{l_q, w_q, k\}$, we introduce the procedure of query processing for $\mathbb{K}k$NN. At the beginning, we compute $H(w_q)$ for keyword matching. In order to retrieve the $\mathbb{K}k$NN results, two steps are considered. First, we need to update the candidate pivots o whose $KP(o)$ may contain target vertices, i.e., function *UpdatePivots*. Then, we incrementally search these $KP(o)$ to obtain the target vertices, i.e., function *FindNext*. For simplicity, we only present the case that l_q is a vertex $v_q \in V$, which can be easily extended to the general cases that l_p locates on edges.

Updating Candidate Pivots. In function *UpdatePivots*, we retain a priority queue PQ_p to restore the candidate pivots o whose $KP(o)$ may contain target vertices. For each $o_i \in PQ_p$, we keep track of a candidate vertex u_i in $KP(o_i)$ that u_i contains w_q and $d_G(v_q, u_i) = \eta_{v_q,o_i} + \eta_{o_i,u_i}$ is only greater than $d_G(v_q, v_f)$ where v_f is the furthest NN result in R obtained so far. Then we determine the minimum

Algorithm 5: Keyword-aware kNN

Input: $Q = (l_q, w_q, k)$

Output: $\mathbb{K}k$NN results, i.e., $R = \{v_1, v_2, \ldots, v_k\}$

```
1  Candidate pivot queue PQ_q;
2  while R.size < k do
3      if PQ_q.size = 0 then
4          Find the first pivot o with H(w) ∧ H(root) ≠ 0;
5          (v, d) = FindNext(w, x, KP(o));
6          UpdatePivots();
7          Find the 1NN result (v, d);
8          R.push(v, d);
9      else
10         for pivot o in PQ_q do
11             Obtain the (v, d) with minimum d_min;
12         FindNext(w, x, KP(o_min));
13         UpdatePivots();
14         R.push(v, d);
15 return R;
```

distance $\min\{d_G(v_q, u_i)\}$ on the top of PQ_q, and push more pivots o' into PQ_q. Note that only o' with $\eta_{v_q,o'} < \min\{d_G(v_q, u_i)\}$ are considered, since only the vertices in such $KP(o')$ are possibly to become results. After the updating, we pop u_i with $\min\{d_G(v_q, u_i)\}$ on the top of current PQ_q into R, and continue to keep track of next u'_i in $KP(o_i)$ that contains w_q by function *FindNext*. This process is repeated until we obtain k results.

Searching KP Index. We denote the j-th leaf node in $KP(o)$ that contains w_q as (u_j, d_j). In order to obtain the next leaf node u_j in $KP(o)$ later than u_{j-1} that also contains w_q, we use function *FindNext* to search $KP(o)$ in a depth-first manner. The *FindNext* function takes w_q and the index of u_{j-1} in $PR(o)$, say x_{j-1}, as input. The search starts from the root node, and for each iteration, we compute $H(w_q) \wedge H(\sigma)$ where σ is the tree node covering $[x_\sigma \ldots y_\sigma]$. If $H(w_q) \wedge H(\sigma) \neq 0$ and $y_\sigma > x_{j-1}$, we continue to search the sub-nodes of σ. If σ is a leaf node, we examine whether $w_q \in \Phi(\sigma)$. Otherwise,

we backtrack its parent node. This process stops when we find the first vertex u_j with index $x_j > x_{j-1}$ that contains w_q.

Example 4.2. *Given a* $\mathbb{K}1NN$ *query* $Q = (v_5, \text{“}a\text{”}, 1)$*, we first search* $KP(v_5)$ *since* $(v_5, 0) \in L(v_5)$*. Then we compute* $H(\text{“}a\text{”}) \wedge H(\sigma_{10}) \neq 0$*, and* $H(\text{“}a\text{”}) \wedge H(\sigma_8) \neq 0$*. Finally, we have* $H(\text{“}a\text{”}) \wedge H(\sigma_1) \neq 0$ *and* $\text{“}a\text{”} \in \Phi(v_1)$*. Note that in this example, no pivot is updated. Thus* v_1 *is reported as result.*

Handling Multiple Keywords. For keyword-aware query, we extend the single keyword search condition into multiple keywords. We consider the keyword-aware query in AND semantic, which aims to find the vertices that contain all these query keywords. Given the $\mathbb{K}\mathrm{C}k\mathrm{NN}$ query $Q = (l_q, \Phi_q, k)$, for the keyword containment checking, we have $H(\Phi_q) = \vee_{w_q \in \Phi_q} H(w)$. For the candidate pivot o, we use $H(\Phi_q)$ to search $KP(o)$. The rest search procedure is just the same as single keyword case. Note that, the number of vertices that contain all the query keywords may be much smaller than the single keyword case, therefore there would be more false positives happening when searching the KP tree index.

Algorithm Analysis. We assume that the keywords are evenly distributed in road network. By using the bloom filter, the hash code of keyword w, i.e., $H(w)$, is a bit array of m bits. Given τ hash functions, each of which maps a keyword to one of the m array positions and set it to 1. Assume that the false positive probability is P_f. The average number of leaf nodes in $KP(o)$ is $\frac{|L(V)|}{|V|}$, and the average number of keywords that a vertex contains is $\frac{freq(V)}{|V|}$, so the average number of keywords in $KP(o)$ is $n_w = \frac{L(V) \cdot freq(V)}{|V|^2}$. From [21], we know given n_w and m, the value of τ that minimizes P_f is $\tau = \frac{m}{n_w} \cdot \ln 2 = \frac{m \cdot |V|^2 \cdot \ln 2}{L(V) \cdot freq(V)}$.

We define two cost functions $f(h)$ and $g(h)$ that $f(h)$ is the cost of searching $KP(o)$ that $KP(o)$ contains the query keyword w, and $g(h)$ is the cost of that $KP(o)$ does not contain w. We denote P_h as the probability that a tree node in $KP(o)$ with height h covers at least one query keyword w, and P as the probability that a vertex contain w. Thus, $P_h = 1 - (1 - P)^{2^h}$. For $f(h)$, we know that the left subtree is searched with $g(h-1)$ if and only if the left subtree does not contain w and the left subnode is a false positive. Therefore, the probability we search the left subtree with the cost $g(h-1)$ is $P_g = (1 - P_{h-1}) \cdot P_f$; Otherwise, we search by cost $f(h-1)$. For $g(h)$, if the subnodes are false positives, we continue to examine the subtrees. For $h = 0$, we apply a binary search to check the

containment of w in leaf node, i.e., $f(0) = g(0) = \ln \frac{freq(V)}{|V|}$. Hence, we have

$$\begin{cases} f(h) = 1 + P_g \cdot g(h-1) + (1 - P_g) \cdot f(h-1) \\ g(h) = 1 + 2P_f \cdot g(h-1) \end{cases} \tag{4.2}$$

As mentioned before, the P_f is usually small since the bloom filter is able to control the false positives well. Therefore, the Equation 4.2 can be simplified as $g(h) = O((2P_f)^h \ln \frac{freq(V)}{|V|})$ and $f(h) = O(h + \ln \frac{freq(V)}{|V|})$. The cost of *FindNext* is $O(\ln \frac{|L(V)|}{|V|} + \ln \frac{freq(V)}{|V|})$. Thus, the worst case of *UpdatePivots* is that we access each pivot $o \in L(v_q)$ for k times. Therefore, the time complexity of Algorithm 5 is $O(k \cdot \frac{|L(V)|}{|V|} \cdot \ln \frac{|L(V)| \cdot freq(V)}{|V|^2})$.

4.3.3 Dominance Interval for $\mathbb{K}$CkNN

The intuition to process $\mathbb{K}$CkNN query is that we find an interval on road network, as long as the moving object stays on such an interval, the $\mathbb{K}k$NN results maintain valid and no recomputation is required. In this section, we call such interval as dominance interval w.r.t. its corresponding $\mathbb{K}k$NN results R, i.e., $\mathcal{DI}(R)$.

Definition 4.4 (**Dominance Interval**). *Given a vertex v, a point* p *is dominated by v, i.e., $v \prec$ p, if and only if v is one of the $\mathbb{K}k$NN results w.r.t.* p. *The dominance interval $\mathcal{DI}(R)$ is a path where every point* p *locates on $\mathcal{DI}$ is dominated by $v \in R$, i.e., $R \prec \mathcal{DI}(R)$.*

Therefore, the major task of $\mathbb{K}$CkNN is to determine such dominance intervals as long as possible in order that the communication cost between server and client ends, as well as the computation cost, are thoroughly reduced.

Lemma 4.1. *Given $\mathcal{EP}(s, \ldots, e)$, let $R_{\mathcal{EP}}$ be the $\mathbb{K}k$NN query results of all points on $\mathcal{EP}$, then we have,*

$$R_{\mathcal{EP}} = R_s \cup R_e \cup_{o \in \mathcal{EP} \setminus \{s,e\}} R_o \tag{4.3}$$

Proof. The proof is straightforward so we omit it here. □

Given a moving query $Q = (l_q, w_q, k)$, we first identify the $\mathcal{EP}_q$ that l_q locates on. Based on Lemma 4.1, we compute the $\mathbb{K}k$NN results for both the vertices s and e of $\mathcal{EP}_q$ by Algorithm 5, denoted as

R_s and R_e, and also obtain all the vertices on $\mathcal{EP}_q$ that contain w_q by accessing the *EP* index of $\mathcal{EP}_q$, denoted as R_m. Finally, we proceed to divide $\mathcal{EP}_q$ into dominance intervals by a **window sliding approach**. Normally, we will be faced with one of the three possible situations: (1) $R_s = R_e$; (2) $R_s \cap R_e = \phi$; (3) $R_s \cap R_e \neq \phi$.

Case 1. $R_s = R_e$. From Lemma 4.1, it is obvious to see that $R_m \subseteq R_s(R_e)$, which means all points on $\mathcal{EP}_q$ share the same dominance interval $\mathcal{DI}(R) = \mathcal{EP}_q$, and $R = R_s = R_e$.

Case 2. $R_s \cap R_e = \phi$. We denote the set $R_s^- = R_s \setminus R_s \cap R_m$ and $R_e^- = R_e \setminus R_e \cap R_m$, and construct a result array $\mathcal{A} = \{v_1, \ldots, v_{|\mathcal{A}_{\mathcal{EP}_q}|}\}$ by concatenating R_s^-, R_m and R_e^-. As no duplicate instances exist in $\mathcal{A}$, we have $|\mathcal{A}| = |R_s^-| + |R_m| + |R_e^-|$. Note that $v_i \in R_s^-$ is sorted in descending order of $d_G(v_i, s)$ while $v_j \in R_e^-$ is sorted in ascending order of $d_G(v_j, e)$. Then we employ a sliding window $\mathcal{W}$ with size k to form the $\mathbb{K}k$NN results by covering continuous vertices in $\mathcal{A}$ from v_1 to $v_{|\mathcal{A}|}$

Lemma 4.2. *The adjacent dominance intervals $\mathcal{DI}(R_i)$ and $\mathcal{DI}(R_{i+1})$ are dominated by $R_i = \{v_i, \ldots, v_{i+k-1}\}$ and $R_{i+1} = \{v_{i+1}, \ldots, v_{i+k}\}$, respectively, $v_i \in \mathcal{A}$. The bisect point p_i between $\mathcal{DI}(R_i)$ and $\mathcal{DI}(R_{i+1})$ is determined by the median point of v_i and v_{i+k} on $\mathcal{EP}_q$.*

Proof. Mapping all the NN results into $\mathcal{A}$ deduces this problem to an order-k voronoi diagram on one dimension. Thus, each dominance interval $\mathcal{DI}(R_i)$ is dominated by k continuous results in current $\mathcal{W}$, i.e., $R_i = \mathcal{W}$. When the moving object moves into $\mathcal{DI}(R_{i+1})$ from $\mathcal{DI}(R_i)$, $\mathcal{W}$ pushes v_{i+k} and pops v_i. Therefore, the bisect point p_i is the point on $\mathcal{EP}_q$ where $d_G(\mathrm{p}_i, v_i) = d_G(\mathrm{p}_i, v_{i+k})$. □

From Lemma 4.2 we know the number of dominance intervals is $n = |\mathcal{A}| - k + 1$, and

$$\mathcal{DI}(R_i) = \begin{cases} \mathcal{P}(s, \ldots, \mathrm{p}_i), & i = 1 \\ \mathcal{P}(\mathrm{p}_{i-1}, \ldots, \mathrm{p}_i), & 1 < i < n \\ \mathcal{P}(\mathrm{p}_{i-1}, \ldots, e), & i = n \end{cases} \tag{4.4}$$

Case 3. $R_s \cap R_e \neq \phi$. We need to deliberate the cases $v \in R_s \cap R_e$. If v locates on $\mathcal{EP}_q$, i.e., $v \in R_m$, we only insert one instance of v into $\mathcal{A}$. Further, if s locates on $\mathcal{SP}(v, e)$ or e locates on $\mathcal{SP}(s, v)$, i.e., $d_G(v, e) = d_G(v, s) + d_G(s, e)$ or $d_G(v, s) = d_G(v, e) + d_G(s, e)$, we also keep one instance of v in R_s^- or R_e^-. For other cases, we retain two duplicate instances of v in $\mathcal{A}$ that one in R_s^- while the other in R_e^-.

Given current sliding window $\mathcal{W}$ with k instances from $\mathcal{A}$, i.e., $\mathcal{W} = \{v_i, \ldots, v_{i+k-1}\}$, if there exists two instances $v_x = v_y$, $v_x, v_y \in \mathcal{W}$ or $v_x \in \mathcal{W}$ is same with v_{i+k}, we extend the sliding window $\mathcal{W}$ to include one more instance but $\mathcal{W}$ still contains k distinct vertices.

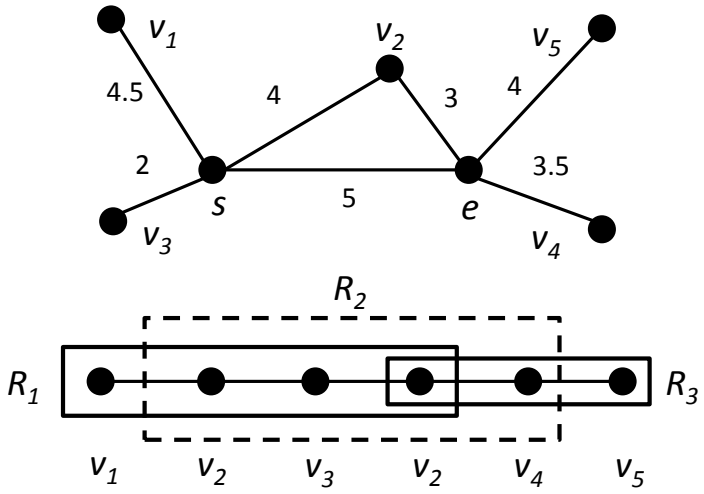


FIGURE 4.3: Dominance interval

Lemma 4.3. *Given sliding window* $\mathcal{W} = \{v_i, \ldots, v_{v_{i+k}}\}$ *that has been extended to* $k+1$ *instances. If* v_i *is the duplicate instance, the bisect point* $\mathtt{p}_i$ *between* $\mathcal{DI}(R_i)$ *and* $\mathcal{DI}(R_{i+1})$ *is the median point between* v_{i+1} *and* v_{i+k+1} *on* $\mathcal{EP}_q$.

Proof. If v_i is duplicated, and $v_{i+1} = v_i$. Thus, the next sliding window $\mathcal{W}$ pops both v_i and v_{i+1} and pushes v_{i+k+1}. So the number of vertices in $\mathcal{W}$ is still kept k, and the bisect point $\mathtt{p}_i$ is determined by v_{i+1} and v_{i+k+1}. If v_i is duplicated, and $v_{i+1} \neq v_i$. Thus, the next sliding window $\mathcal{W}$ pops both v_i and v_{i+1} and pushes v_{i+k+1} as well. Note that $v_x \in \{v_{i+2}, \ldots, v_{v+k}\}$, $v_x = v_i$ is still covered by $\mathcal{W}$, so the number of vertices in $\mathcal{W}$ is still kept k. Therefore, $\mathtt{p}_i$ is determined by v_{i+1} and v_{i+k+1}. □

For example in Fig. 4.3, $R_s = \{v_1, v_2, v_3\}$, $R_e = \{v_2, v_4, v_5\}$ and $\mathcal{A} = \{v_1, v_2, v_3, v_2, v_4, v_5\}$. Obviously, R_1 contains two duplicate instances of v_2, and $\mathtt{p}_1 = \mathtt{p}(s, e, 2)$ is the bisect point between v_1 and v_4 on $\mathcal{P}(s, \ldots, e)$. Thus, the dominance interval $\mathcal{DI}_1 = \mathcal{P}(s, \ldots, \mathtt{p}(s, e, 2))$. Consequently, $R_2 = \{v_2, v_3, v_4\}$. Note that v_2 is the duplicate instance, thus $\mathtt{p}_2 = \mathtt{p}(s, e, 3.5)$ is determined by v_3 and v_5 from Lemma 4.3. Therefore, $\mathcal{DI}_2 = \mathcal{P}(\mathtt{p}(s, e, 2), \ldots, \mathtt{p}(s, e, 3.5))$.

Following to the resolution of dominance intervals, on condition of that the moving object Q stays in $\mathcal{DI}(R_i)$, R_i is reported as $\mathbb{K}k$NN results and no communication takes place. Once Q moves out of $\mathcal{EP}_q$, a new round of dominance interval computation is issued. The pseudocode can be found in Algorithm 6.

Example 4.3. *As shown in the running example Fig. 4.1, given* $\mathbb{K}$*CkNN query* $Q = (v_2, \text{“}a\text{”}, 3)$.

First we find the enclosed path that v_2 locates on, i.e., $\mathcal{EP}(v_1, \ldots, v_7)$. Then we compute $R_{v_1} = \{v_1, v_2, v_9\}$, $R_{v_7} = \{v_7, v_4, v_2\}$ and $R_m = \{v_2, v_4\}$. Thus we have $\mathcal{A} = \{v_9, v_1, v_2, v_4, v_7\}$. The resolved dominance intervals are $\mathcal{DI}(R_1) = \mathcal{P}(v_1, \ldots, \mathrm{p}(v_1, v_2, 0.5))$ w.r.t. $R_1 = \{v_9, v_1, v_2\}$, $\mathcal{DI}(R_2) = \mathcal{P}(\mathrm{p}(v_1, v_2, 0.5), \ldots, v_3)$ w.r.t. $R_2 = \{v_1, v_2, v_4\}$ and $\mathcal{DI}(R_3) = \mathcal{P}(v_3, \ldots, v_7)$ w.r.t. $R_3 = \{v_2, v_4, v_7\}$.

Algorithm Analysis. Algorithm 6 involves two phases of computation, the first phase is the $\mathbb{K}k$NN query processing and the other is dominance interval resolution.

Theorem 4.1. *The expected time complexity of Algorithm 6 is $O(k \cdot \frac{|L(V)|}{|V|} \cdot \ln \frac{|L(V)| \cdot freq(V)}{|V|^2} + \ln |W| + n)$. The expected communication cost is $O(\frac{|E|}{\sum_{e \in E} l_e})$.*

Proof. First, we issue two $\mathbb{K}k$NN queries for s and e which takes $O(k \cdot \frac{|L(V)|}{|V|} \cdot \ln \frac{|L(V)| \cdot freq(V)}{|V|^2})$. To obtain the vertices on $\mathcal{EP}_q$ that contain w_q, we apply a binary search to locate the keyword posting list for w_q which takes $O(\ln |W|)$ for the worst case. For the dominance interval resolution, we compute n bisect points which takes $O(n)$. Therefore, the time complexity of Algorithm 6 is $O(k \cdot \frac{|L(V)|}{|V|} \cdot \ln \frac{|L(V)| \cdot freq(V)}{|V|^2} + \ln |W| + n)$. The average length of dominance interval is $O(\frac{\sum_{e \in E} l_e}{|E|})$. As the communication cost is inversely proportional to the average length of dominance interval, thus we have the expected cost $O(\frac{|E|}{\sum_{e \in E} l_e})$. □

4.4 Algorithm *LARC++*

Intuitively, the dominance intervals of frequent keywords are usually short so that *LARC* is able to handle the frequent cases well. As a contrast, infrequent keywords always hold a relative large region without the need of recomputation of $\mathbb{K}k$NN results. Accordingly, simply applying *LARC* on these cases may incur unnecessary communication and computation costs due to the limited length of dominance interval. Therefore, we introduce an enhanced algorithm *LARC++* to cope with infrequent cases in this section. Unlike the Euclidean space that the objects are randomly distributed without correlation, in road networks the objects are connected and organized by edges and paths between them due to the network properties. Thus, the *LARC++* adopts a **path-based dominance updating** approach to discover the paths that construct the dominance region by exploiting the properties of paths that connect objects in road networks.

Algorithm 6: Keyword-aware continuous kNN

Input: $Q = (l_q, w_q, k)$

Output: $R = \{v_1, v_2, \ldots, v_k\}$

```
1  Obtain EP_q and R_m;
2  Compute two IKkNN queries and obtain R_s and R_e;
3  if R_s = R_e then
4      DI = EP_q;
5  else
6      Generate a result array A;
7      if R_s ∩ R_e ≠ φ then
8          We have n = |A| − k + 1 dominance intervals;
9          Compute the {DI} and {R_i};
10     else
11         Compute the {DI} and {R_i} by Lemma 4.3;
12 while Q is on EP_q do
13     Find the dominance interval DI(R_i) that Q locates on;
14     return R_i;
```

4.4.1 Path-based Dominance Updating

Analogous to existing works on CkNN, we adopt the concept of dominance region where no recomputation is demanded on all inside paths, and aim to find such a region as large as possible with low cost.

Concepts and Notations. First, we give a formal definition of dominance region as follows,

Definition 4.5 (**Dominance Region**). *The dominance region $\mathcal{DR}$ w.r.t. vertex v, i.e., $v \prec \mathcal{DR}^k(v)$, is a region where every inside point* p *is dominated by v. Likewise, given current IKkNN results $R = \{v_1, v_2, \ldots, v_k\}$, the dominance region w.r.t. R, i.e., $R \prec \mathcal{DR}^k(R)$, is a region where every inside point* p *is dominated by R. Thus, we have:*

$$\mathcal{DR}^k(R) = \bigcap_{v \in R} \mathcal{DR}^k(v) \tag{4.5}$$

From Definition 4.5, we need to find the $\mathcal{DR}^k(v)$ where v is always one of the $\mathbb{K}k$NN results. However, it is unlikely to compute such a region directly since the $\mathbb{K}k$NN results that contain v may have many different combinations, which makes this problem inapplicable. Fortunately, inspired by the construction of oder-k Voronoi Diagram in Euclidean space, we only need to compute $\mathcal{DR}^1(v)$ where v is the $\mathbb{K}$1NN result, which is similar to the concept of order-1 Voronoi Diagram.

Lemma 4.4. *Given $G = (V, E)$, the current $\mathbb{K}k$NN results $R = \{v_1, \ldots, v_n\}$. The dominance region w.r.t. R is computed as,*

$$\mathcal{DR}^k(R) = \bigcap_{v \in R} \mathcal{DR}^1_{(V \setminus R \bigcup v)}(v) \tag{4.6}$$

Note that $\mathcal{DR}^1_{(V \setminus R \bigcup v)}(v)$ is the dominance region w.r.t. v that constructed in the sub-network $V \setminus R \bigcup v$.

Proof. From [75] in Euclidean space, we know that the order-k Voronoi Diagram w.r.t. R is the intersection of all the order-1 Voronoi Diagrams w.r.t. $v \in R$ that constructed by ignoring the rest results in R. Analogously, $\mathcal{DR}^1_{(V \setminus R \bigcup v)}(v)$ is same as the concept of order-1 Voronoi Diagram while the bisectors are defined by network distance. Obviously, all the points $\mathtt{p} \in \mathcal{DR}^1_{(V \setminus R \bigcup v)}(v)$ are closer to v than $u' \in V \setminus R$ in terms of network distance. Therefore, $\mathcal{DR}^k(R)$ is the intersections of all $\mathcal{DR}^1_{V \setminus R \bigcup v}(v)$. □

In order to resolve the boundary of dominance region, we are obliged to find a set of vertices $u \in V \setminus R$ containing w_q, which are influential in determining whether the current $\mathbb{K}k$NN result R is valid. Therefore, we introduce the concept of potential neighbour as follows,

Definition 4.6 (Potential Neighbour). *Given current $\mathbb{K}k$NN results $R = \{v_1, v_2, \ldots, v_k\}$, and a query keyword w_q. We assume that $u \in V \setminus R$ contains w_q. The vertex u is a potential neighbour w.r.t. v_i, i.e., $u \in \mathcal{PN}(v_i)$, if and only if there does not exist a vertex o on $\mathcal{SP}(u, v_i)$ that also contains w_q.*

After obtaining the potential neighbour $\mathcal{PN}(R)$, we are able to resolve the dominance status of the vertices between $\mathcal{PN}(R)$ and R. Without loss of generality, we define such vertices inbetween as enrolled vertex as follows,

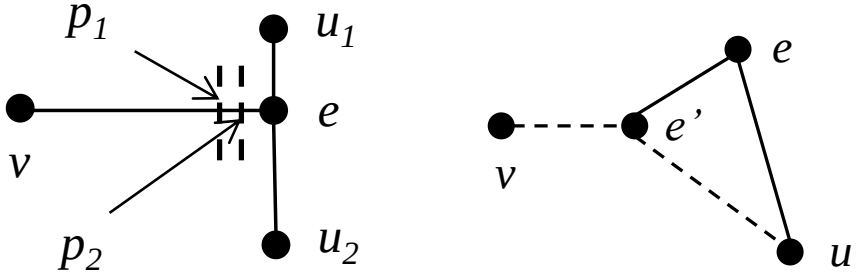
Definition 4.7 (Enrolled Vertex). *Given current $\mathbb{K}k$NN results $R = \{v_1, v_2, \ldots, v_k\}$, and a potential neighbour $u \in \mathcal{PN}(R)$. An intersection vertex e is enrolled, $e \in \mathcal{EN}(v)$, if and only if e is on the path between $v_i \in R$ and u, and $d_G(v_i, e) < \max_{u \in PN(R)} d_G(v_i, u)$.*

Dominance Region Construction. When a $\mathbb{K}$CkNN query is issued, i.e., $Q = (l_q, w_q, k)$, we compute an initial $\mathbb{K}k$NN set R w.r.t. the start point l_q by Algorithm 5. For each $v \in R$, we first determine the potential neighbours $\mathcal{PN}(v)$, and in this process, we obtain a set of enrolled vertices in order that we can further validate the dominance status of the paths they locate on. After these paths are resolved, we merge them to form the final $\mathcal{DR}^k(R)$.

It is worth to notice that the step of determining $\mathcal{PN}(R)$ is critical since it to some extent defines the strength of $\mathcal{DR}^k(R)$. In other words, the more vertices are enrolled in the step, we are more possible to obtain a larger $\mathcal{DR}^k(R)$. However, it is really time consuming and unnecessary to discover all the potential neighbours for each $v \in R$. Instead we only need to find the nearest potential neighbour $u \in \mathcal{PN}(v)$ and $u \notin R$ for each v as tradeoff.

Assume that we obtain a set of enrolled vertices $\mathcal{EN}(R)$, each $e \in \mathcal{EN}(R)$ is sorted by their distances to $v \in R$, in the form of $(v, d_G(e, v))$. Next, we proceed to resolve the dominance status of these enrolled vertices and by which potential neighbour or result vertex they are dominated. Straightforwardly, we can compute the $\mathbb{K}$1NN for all enrolled vertices by Algorithm 5. Thus the dominance of each $e \in \mathcal{EN}(R)$ is easily determined. However, it is quite unoptimized that some $\mathbb{K}$1NN results of $e \in \mathcal{EN}(R)$ are repeatedly computed due to the observation that many enrolled vertices are actually dominated by the same potential neighbour or result vertex. Hence, we propose a *path-based dominance updating* method, i.e., *PathDom* (See Algorithm 7), to resolve the dominance of $e \in \mathcal{EN}(R)$ based on such an observation that if e is dominated by v, then all the other vertices e' on the path $\mathcal{P}(v, \ldots, e)$ are also dominated by v. In other words, we only need to find the furthest enrolled vertex e dominated by v.

We start the dominance status validation process from the enrolled vertex $e \in \mathcal{EN}(R)$ with the maximum value of $d_G(e, v)$, in the sense that more vertices are supposed to be included by a longer path $\mathcal{P}(v, \ldots, e)$. Then Algorithm 5 is applied from e to hit v, in this process, some intermediate enrolled vertices $e' \in \mathcal{EN}(v)$ are traversed with distance $d_G(e, e')$. If $v \in R$ is the first vertex containing w_q reached in this process, $\mathcal{SP}(e, \ldots, v)$ is inserted into $\mathcal{DR}$. Otherwise, if we hit another vertex u that also contains query keyword w_q before we hit v, then we have $u \in \mathcal{PN}(v)$. For all the intermediate vertices e', if $d_G(v, e') + d_G(e, e') = d_G(v, e)$, then e' is on the shortest path $\mathcal{SP}(v, \ldots, e)$, which is regarded as a candidate path for $\mathcal{DR}$. For these $e' \in \mathcal{SP}(v, \ldots, e)$, we store two entries, i.e., $\{(v, d_G(e', v)), (u, d(e', u))\}$ where $d(e', u) = d_G(e, e') + d_G(e, u)$, and remove these e' from $\mathcal{EN}(R)$. Note



(a) Updating condition 1 (b) Updating condition 2

Figure 4.4: Potential neighbour

that $d(e', u)$ may not be the minimum distance between e' and u and requires further updating.

Then we repeat this process and start from $e \in \mathcal{EN}(R)$ with the maximum value of $d_G(e, v)$ in the rest $\mathcal{EN}(R)$. Here we introduce two updating conditions.

Path Updating Condition 1. Consider such a situation, if e hits a potential neighbour u', and encounters an intermediate enrolled vertex e' that is contained by a candidate path $\mathcal{SP}(e_o, \ldots, v)$ and has two entries $\{(v, d_G(e', v)), (u, d(e', u))\}$ w.r.t. v and u. Thus, we split the candidate path $\mathcal{SP}(e_o, \ldots, v)$ into two candidate paths $\mathcal{SP}(e_o, \ldots, e')$ and $\mathcal{SP}(e', \ldots, v)$ for further validation, and keep a new candidate path $\mathcal{SP}(e, \ldots, e')$. Then we compare the two distances $d(e', u)$ and $d(e', u')$.

Lemma 4.5. *Given two potential neighbour $u_1, u_2 \in PN(v)$ w.r.t. v, the paths $\mathcal{P}(v, \ldots, e, \ldots, u_1)$ and $\mathcal{P}(v, \ldots, e, \ldots, u_2)$ share a common path $\mathcal{P}(v, \ldots, e)$. As shown in Fig. 4.4(a), if $d_G(e, u_1) < d_G(e, u_2)$ and $d_G(e, u_1) < d_G(e, v)$, thus u_1 governs u_2 w.r.t. the path $\mathcal{P}(v, \ldots, e)$, denoted as $u_1 \curvearrowleft u_2$ on $\mathcal{P}(v, \ldots, e)$, which means the dominance status of $e' \in \mathcal{SP}(e, \ldots v)$ is only determined by v or u_1.*

Proof. If $u_1 \curvearrowleft u_2$ on $\mathcal{P}(v, \ldots, e)$, we know that $d_G(e, u_1) < d_G(e, u_2)$ and $d_G(e, u_1) < d_G(e, v)$. For the dominance region between v and u_1, the bisect point $\mathtt{p}_1$ locates on $\mathcal{P}(v, \ldots, e)$ since $d_G(v, \mathtt{p}_1) = (d_G(v, e) + d_G(e, u_1))/2 < d_G(v, e)$. For u_2, the distance between bisect point $\mathtt{p}_2$ and v, i.e., $d_G(v, \mathtt{p}_2) = (d_G(v, e) + d_G(e, u_2))/2 > d_G(v, \mathtt{p}_1)$, therefore we know that $\mathtt{p}_1$ is closer to v than $\mathtt{p}_2$. □

From Lemma 4.5, we know that if $d(e', u') < d(e', u)$, then we update the entries into $\{(v, d_G(e', v)), (u', d(e', u'))\}$ since it might be that $u' \curvearrowleft u$ on the path $\mathcal{P}(v, \ldots, e')$. Note that, we also update the entries of all the enrolled vertices on candidate path $\mathcal{SP}(e', \ldots, v)$.

Path Updating Condition 2. Consider another situation, if e hits a potential neighbour u, and

encounters an intermediate enrolled vertex e' that already has two entries $\{(v, d_G(e', v)), (u, d(e', u))\}$ w.r.t. v and u, as shown in Fig. 4.4(b). If $d_G(e, e') + d_G(e, u) < d(e', u)$, we know that the path $\mathcal{P}(e', \ldots, e, \ldots, u)$ is a shorter path than that in the previous iterations. Similarly, we update $d(e', u)$ with $d_G(e, e') + d_G(e, u)$ for all e' on the path $\mathcal{P}(v, \ldots, e')$. In addition, if this enrolled vertex e hits a potential neighbour u, and the first intermediate vertex e' that it encounters has two entries $\{(v, d_G(e', v)), (u, d(e', u))\}$ w.r.t. v and u. If $d_G(e, e') + d_G(e, u) > d(e', u)$, we know that the path $\mathcal{P}(e', \ldots, u)$ has already been visited by a shorter path in the previous iterations, thus we do not update the entries of vertices on the path $\mathcal{P}(v, \ldots, e')$. This process terminates when $\mathcal{EN}(R) = \emptyset$.

Lemma 4.6. *When $\mathcal{EN}(R) = \emptyset$, for each enrolled vertex e, in the entries $\{(v, d_G(e, v)), (u, d(e, u))\}$, $d(e, u)$ is the minimum distance $d_G(e, u)$.*

Proof. According to our algorithm, all the possible paths between e and u have been traversed, and there must exist a path that is the shortest one. Therefore,we have $d(e, u) = d_G(e, u)$. □

Finally, the dominance status of each vertex e on candidate paths can be easily determined by comparing $d_G(e, v)$ and $d_G(e, u)$, as well as the dominance intervals on these candidate paths. Then we insert all the dominance intervals into $\mathcal{DR}$.

Example 4.4. *As shown in the running example Fig. 4.1, given $\mathbb{K}C1NN$ query $Q = (v_1, \text{“}d\text{”}, 1)$, we first search for the $\mathbb{K}1NN$ result of v_1, i.e., v_3. Then we continue to resolve the dominance region $\mathcal{DR}(\{v_3\})$. As the $\mathbb{K}1NN$ result is v_6, thus the enrolled vertex set $\mathcal{EN}(\{v_6\}) = \{v_5, v_1, v_7\}$. Then we use v_5 to hit target vertices, and obtain $d_G(v_5, v_6) = 3$. Note that v_1 is traversed in this process, thus v_1 has two entries $\{(v_3, 3), (v_6, 4)\}$. Next we use v_7 to hit targets, and obtain $\{(v_3, 3), (v_6, 3)\}$. Therefore, the dominance region $\mathcal{DR}(\{v_6\}) = \{\mathcal{P}(\mathrm{p}(v_1, v_5, 0.5), \ldots, v_3), \mathcal{P}(v_7, \ldots, v_3)\}$.*

4.4.2 Combination of *LARC* and *LARC++*

Generally, if the query keywords are densely distributed on road network, *LARC* is able to resolve the short dominance intervals efficiently. By contrast, if the query keywords are sparse in road network, *LARC++* is capable of determining the large dominance regions well. However, if we use *LARC* to deal with low frequency keywords or *LARC++* to cope with high frequency keywords, either redundant communication cost or computation cost will be incurred. As *LARC* and *LARC++* are sensitive to

Algorithm 7: PathDom()

Input: $v, e \in EN(v)$

Output: For each e we have $(v, d_G(e, v)), (u, d_G(e, u))$

```
1  while EN(v) ≠ ∅ do
2      e = EN(v).top;
3      if e hits u ∈ PN(v) before v then
4          for e′ on P(v, ..., e) do
5              if The entry of e′ is ∅ then
6                  d(e′, u) = d_G(e, e′) + d_G(e, u);
7                  Store (v, d_G(e′, v)), (u, d(e′, u));
8              else if The entry of e′ contains u′ and d(e′, u′) < d(e′, u) then
9                  Update into {(v, d_G(e′, v)), (u′, d(e′, u′))};
10             else if The entry of e′ contains u and d_G(e, e′) + d_G(e, u) < d(e′, u) then
11                 Update d(e′, u) = d_G(e, e′) + d_G(e, u);
12             else if The entry of e′ contains u and d_G(e, e′) + d_G(e, u) > d(e′, u) then
13                 Continue;
14     return all entries;
```

different keyword frequencies, we combine these two algorithms to develop a new algorithm *LARC-C* that when the query keywords are of high frequency, we use *LARC*; when the query keywords are of low frequency, then *LARC++* is utilized.

The key point in this combination algorithm is that we designate a threshold that half of the keyword occurrences are handled by *LARC* and the other half by *LARC++*. Motivated by [53], we assume that the keywords of road network are of Zipf's distribution [134]. In the experimental dataset, we use the keyword id *kid* to denote the its rank of frequency. We know that in Zipf's distribution, $freq(w) \propto \frac{1}{kid}$. For $G = (V, E)$, we have $|W|$ keywords and $|freq(W)|$ keyword occurrences. As we know, the sum of occurrences of top n frequent keywords is proportional to the Harmonic number n, i.e., $H_n = \sum_{kid=1}^{n} \frac{1}{kid} = \ln n$. Therefore, we have $H_{|W|} = \ln |W|$ and $\frac{H_{|W|^{1/2}}}{H_{|W|}} = \frac{\ln |W|^{1/2}}{\ln |W|} = 1/2$, which means the top $|W|^{1/2}$ keywords cover half the keyword occurrences. Therefore, in our real dataset of

Beijing, we have $|W|^{1/2} = 298$ that we use *LARC* for top 298 keywords and use *LARC++* for the rest keywords.

4.5 Experiments

In this section, we conduct extensive experiments on real road network datasets to study the performance of the proposed index structures and algorithms.

4.5.1 Experimental Settings

All these algorithms were implemented in GNU C++ on Linux and run on an Intel(R) CPU i7-4770@3.4GHz and 16G RAM.

Datasets. We use two real datasets, the road network datasets of Beijing and New York City from the 9th DIMACS Implementation Challenge[1]. Each dataset contains an undirected weighted graph that represents a part of the road network. Each edge in a graph represents the distance between two endpoints of the edge. We obtain the keywords of vertices from the OpenStreetMap[2]. As shown in Table 4.2, for D1 in Beijing, we have 168,535 vertices and 196,307 edges. We also have 88,910 distinct keywords contained by vertices with the total occurrence 1,445,824. For D2 in New York, we have more vertices and edges than D1 in road network with almost twice the size of D1, the set of keywords contained are larger than D1 as well. For each experiment, we generate 50 IKCkNN queries, each of which is a sequence of locations in the form of (u, v, d_s). The query location can be either a vertex or a point locates on an edge.

TABLE 4.2: Statistics of dataset

	Beijing	New York
$\#\|V\|$	168,535	264,346
$\#\|E\|$	196,307	733,846
$\#\|W\|$	88,910	102,450
$\#\|\Phi(V)\|$	1,445,824	3,086,166

[1] http://www.dis.uniroma1.it/challenge9/download.shtml

[2] https://www.openstreetmap.org

Algorithms Evaluated. We introduce two baseline algorithms, Dijk-BF and V*-RN, for comparison. The first baseline algorithm is the brute-force approach (**Dijk-BF**), which computes the kNN results for every location reported by the moving object. If the object locates at vertex v_q, Dijkstra-algorithm is applied that expands from the query location and traverse the other vertices in the best-first way until reaching k vertices with the query keyword w_q. Additionally, if the object locates on edge (u, v), i.e., $l_q = \mathrm{p}(u, v, d_s)$, both the two ends u and v are regarded as the source of Dijkstra search with the initial distances d_s and $l(u, v) - d_s$, respectively. The second baseline algorithm is **V*-RN** that extended from V*-Diagram [74] since V*-Diagram is designed for the Euclidean space only. Generally, V*-RN keeps a safe region to reduce the communication cost. At the first step of retrieving $(k + \delta)$NN results, all the paths accessed by Dijkstra search are saved as the known region. To identify the safe region boundary of each path in the known region, we compute the network distances for both two endpoints of these paths, and finally obtain the safe region. For our algorithms proposed in this chapter, we have exact algorithm *LARC* in Section 4.3, *LARC*++ in Section 4.4 and the combination algorithm *LARC-C*. For the construction of label index, we adopt Pruned Landmark Labelling [6] and Hub-based Labelling [1] to generate the 2-hop label. For these four algorithms, we evaluate the CPU computation cost, and the communication cost between server and client. In the experiments, we vary the size of datasets, the length of the query objects, the number of results k, the speed of query object and the frequency of query keyword, to study the effects of these parameters.

Parameters. To evaluate the algorithms under various settings, we vary the value of some parameters. For the speed of object, we vary the report distance from 20 to 100 meters. For the number of the results, we vary the k from 5 to 50. For the length of query object, we vary the length from 200 to 1000. For the query keyword frequency, we vary the frequency from 10^0 to 10^4. We default choose the speed of object as 40, the k as 10, the length as 400, and the keyword frequency as 10^3. The parameters are summarized in Table 4.3.

4.5.2 Experimental Results

Among all the algorithms discussed in this chapter, we perform a comparative experimental study on Dijk-BF, V*-RN, *LARC*, *LARC*++ and *LARC-C*. The next experiments compare these algorithms using different experimental parameters and study their effects on the performance. Most experiments

TABLE 4.3: Parameter settings

Parameters	**Values**
Speed of object	20, **40**, 60, 80, 100
k number of results	5, **10**, 15, 20, 50
Length of query	200, **400**, 600, 800, 1000
Keyword frequency	10^0, 10^1, 10^2, $\mathbf{10^3}$, 10^4

presented in this subsection are using D1 dataset from Beijing.

Effect of dataset cardinality. In this set of experiments, we vary the datasets to study the effect of data cardinality for Beijing and New York. For the performance study, we compare the CPU time and communication cost of these three algorithms by varying the size of these two datasets. For Beijing dataset, we vary the size of vertices from 40K to 160K. For New York dataset, we vary and the size of vertices from 100K to 250K. As shown in Fig. 4.5, the CPU time of our proposed algorithm *LARC-C* outperforms these two baseline algorithms Dijk-BF and V*-RN, and the CPU time of each algorithm keeps relative stable with varying the size of datasets. In addition, the communication cost of Dijk-BF is constant since it communicates every time when the query location updates. For V*-RN and *LARC-C*, the communication costs are thoroughly reduced, and *LARC-C* incurs even less communication cost than V*-RN. This confirms the superiority of our proposed algorithm.

Effect of query length. In this set of experiments, we use the query lengths of 200, 400, 600, 800, 1000 to study its effect on these three algorithms. The query length is the number of locations the moving object reported. Intuitively, as the number of locations increases, both the computation and communication costs increase. As shown in Figure 4.6, for the computation cost, Dijk-BF and V*-RN increase faster than *LARC-C*, since both of them evolve the nearest neighbour search by Dijkstra algorithm and keyword checking for every encountered vertex. For *LARC-C*, the increasing trend of computation cost is slow because they only need to search KP tree to obtain $\mathbb{K}k$NN results and in the meantime construct a dominance region to avoid recomputation of kNN. For the communication cost, as Dijk-BF does not construct the safe region, the communication cost is linear to the query length. For V*-RN, *LARC-C*, both of them adopt the concept of safe region or dominance region. As a result, we can see that the communication costs are highly reduced, and our proposed algorithm *LARC-C* is slightly better than V*-RN.

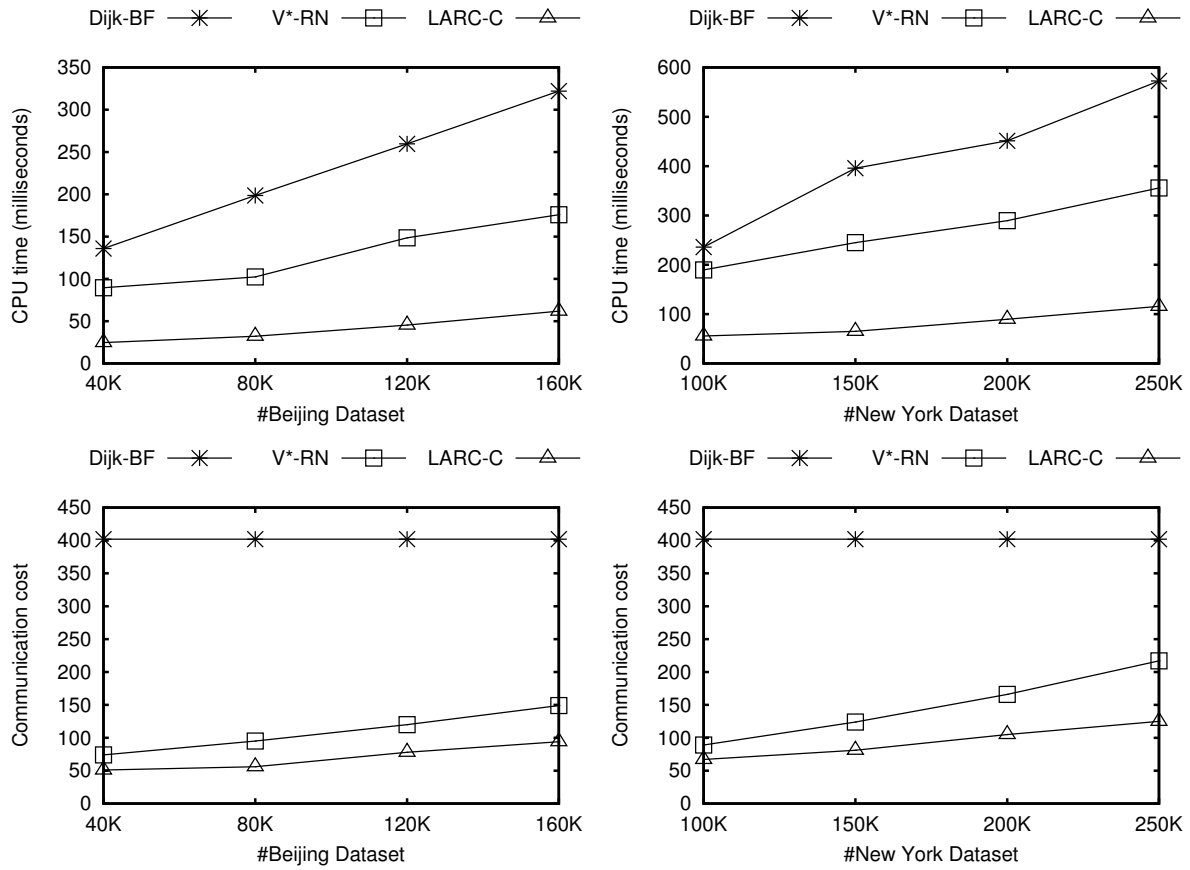


FIGURE 4.5: Effect of dataset cardinality

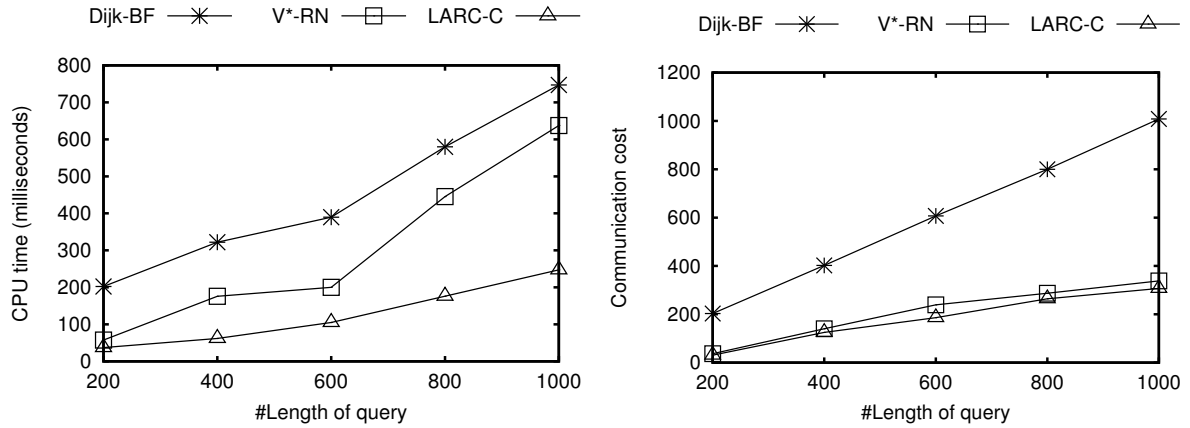


FIGURE 4.6: Effect of query length

Effect of k. In this set of experiments, we vary the value of k by 5, 10, 15, 20 and 50 to study the effect on these three algorithms. As shown in Figure 4.7, we can see that our proposed algorithm

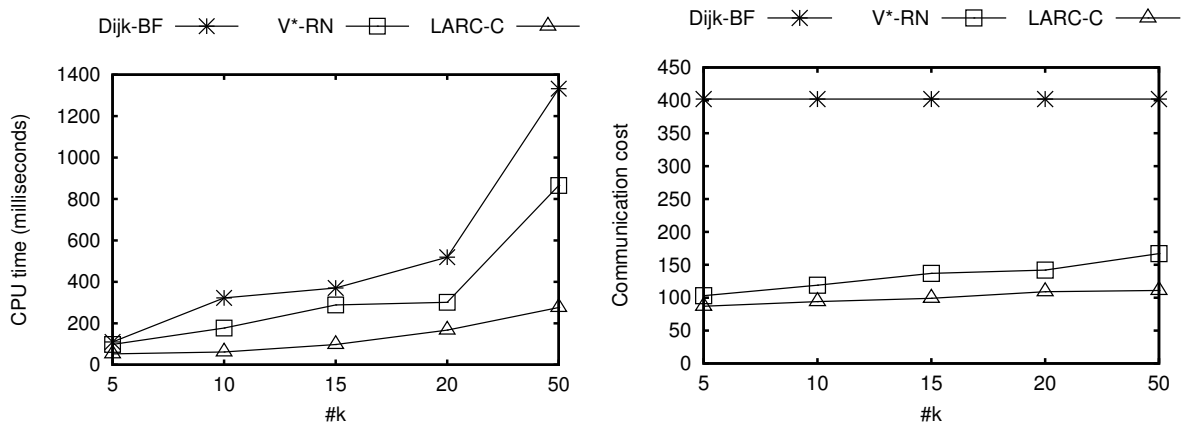


FIGURE 4.7: Effect of k

LARC-C well outperforms the baseline algorithms Dijk-BF and V*-RN on both computation and communication costs. As we know that when we enlarge k, more vertices are included for consideration of keyword checking. When the value of k is small, say 5, we can see that the computation costs of these algorithms are close. However, the computation costs of Dijk-BF and V*-RN are increasing much faster than *LARC-C* due to the massive network traversals with enlarging k. For the communication cost, Dijk-BF gains a constant value since it reports the $\mathbb{K}k$NN results for every location. If the query length does not change, the communication cost is kept the same. For V*-RN, *LARC-C*, we have a similar observation to previous set of experiments that our proposed algorithm *LARC-C* is slightly better than V*-RN. Note that, even the communication cost of V*-RN is low, the computation overhead is incurred in the construction of safe region. This explains that V*-RN has a low communication cost but still has a high computation cost. As a result, this confirms the superiority of our proposed algorithm.

Effect of query object speed. In this set of experiments, we vary the speed of query object by 20, 40, 60, 80, 100 to study the effect on these three algorithms. The speed of query object is determined by the distance between two reported locations of query objects. Therefore, we vary this distance to simulate the speed of moving object. As shown in Figure 4.8, we can see the similar pattern that our proposed algorithm *LARC-C* well outperforms the baseline algorithms Dijk-BF and V*-RN on both computation and communication costs. For the computation cost, Dijk-BF has a steady trend because the speed of query object does not have an obvious effect on it. For V*-RN, *LARC-C*, if the distance is small, the query object is more possible to stay in the safe region or dominance region, thus we do

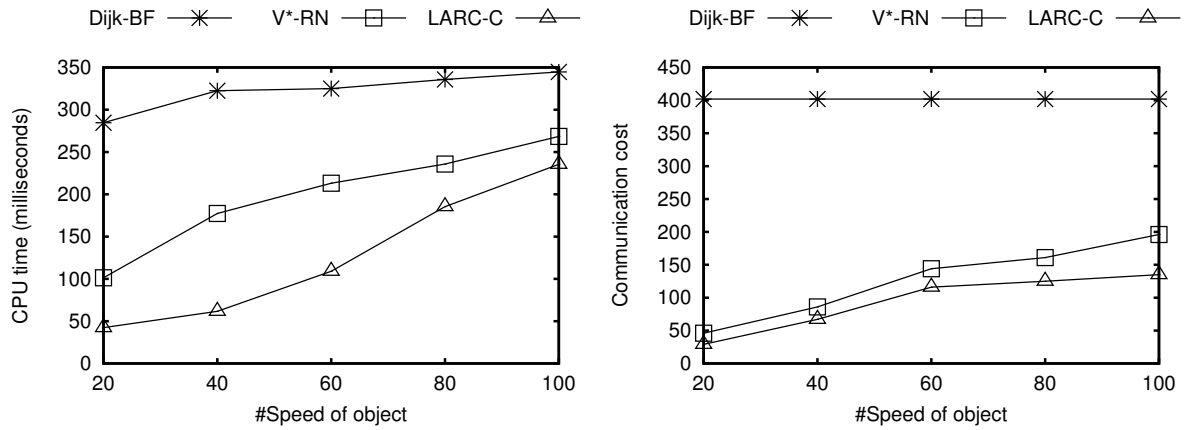


FIGURE 4.8: Effect of speed

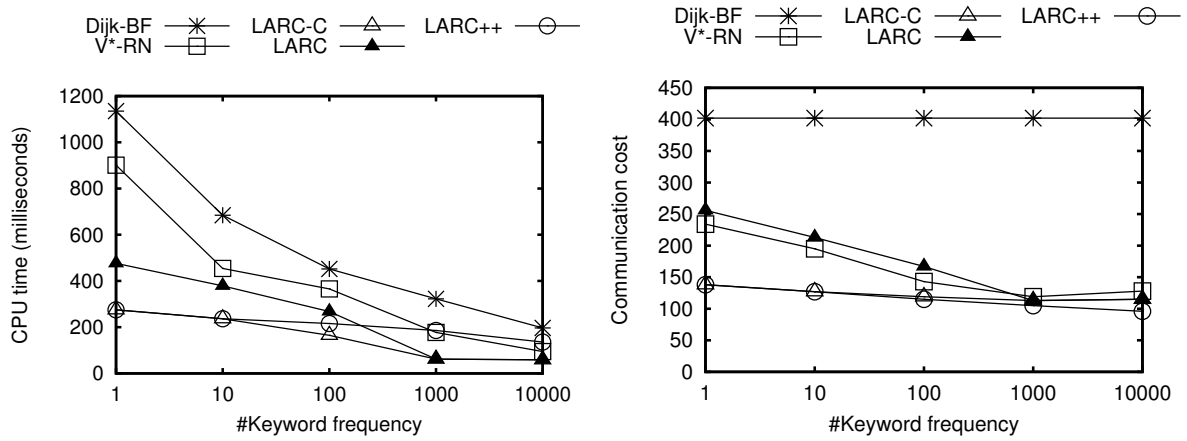


FIGURE 4.9: Effect of keyword frequency

not have more recomputation for I*k*NN and region construction. If the distance is large, the query object is more possible to move out of the safe region or dominance region, thus the recomputation of I*k*NN and region reconstruction are incurred more often. Therefore, the computation costs increase in all these three algorithms when the distance is enlarged. For the communication cost, Dijk-BF keeps the same cost, and our proposed algorithm *LARC-C* are slightly better than V*-RN.

Effect of keyword frequency. In this set of experiments, we vary the query keyword frequency by 10^0, 10^1, 10^2, 10^3, 10^4 to study the effect on these five algorithms. For each keyword frequency, we compute an average value by selecting some keywords with close frequencies to it. As we can see in Figure 4.9, when the keyword frequency is low, Dijk-BF and V*-RN have a bad computation performance, since they have to traverse large portion of the road network to obtain I*k*NN results,

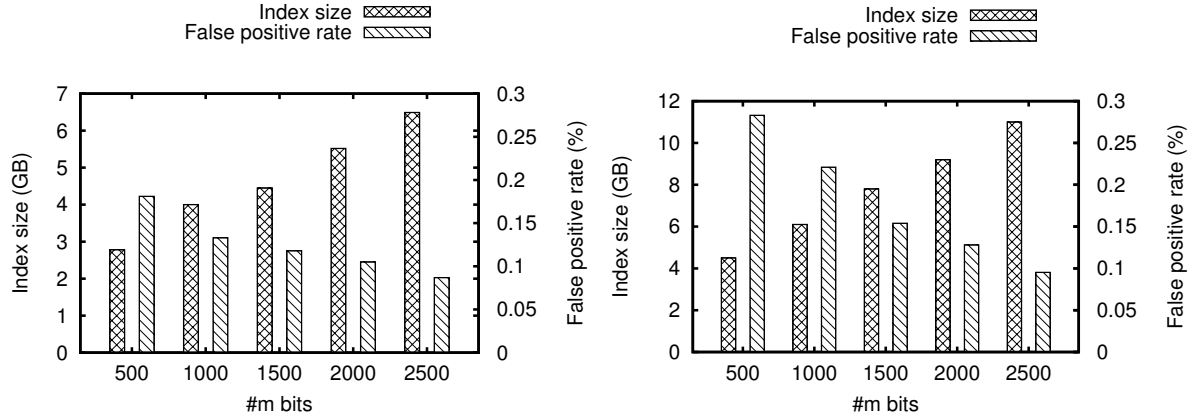


FIGURE 4.10: Effect of m

and this process incurs large computation overheads on network traversal and keyword checking. We can see that that our proposed algorithms *LARC-C* well outperforms the baseline algorithms Dijk-BF and V*-RN because it adopts the $FindNext$ method to search the KP tree. For the communication cost, Dijk-BF gains a constant value just as the same to previous experiments. For V*-RN, *LARC-C*, both of them have a decrease in the communication cost when we enlarge the keyword frequency. We can also see that our proposed algorithms are slightly better than V*-RN. For *LARC*, *LARC++* and *LARC-C*, *LARC* outperforms *LARC++* when the query keywords are of high frequency in terms of CPU cost. But *LARC++* is slightly better than *LARC* in terms of communication cost, because the dominance region determined by *LARC++* is always larger than *LARC*.

Effect of m bits. In this set of experiments, we vary the number of m bits in hash code by 500,1000,1500,2000,2500 to study the effect on index size and false positive rate. As we can see in Figure 4.10, the index size increases and the false positive rate decreases when we enlarge the value of m. This is because when the value of m is large, more space will be used to construct the index structure especially the KP tree, and keywords are less possible to share a same hash code.

Effect of multiple keywords. In this set of experiments, we vary the number of query keywords by 1, 2, 3, 4, 5 to study the effect on these three algorithms. As we can see in Figure 4.11, the experiment results have a similar pattern that our proposed algorithm *LARC-C* well outperform these two baseline algorithms in terms of both the CPU cost and communication cost. Because when the number of query keywords increases, the number of target vertices decreases. Therefore, *LARC-C* has a better performance than Dijk-BF and V*-RN.

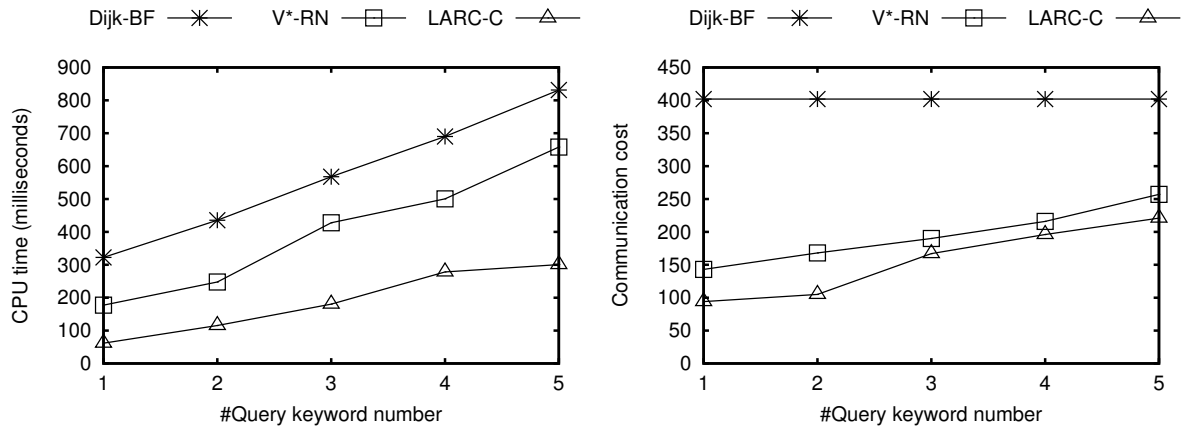


FIGURE 4.11: Effect of multiple keywords

4.6 Summary

In this chapter, we study the problem of efficiently processing $\mathbb{K}$CkNN query on road networks with low computation and communication costs. By utilizing the 2-hop label technique on road networks, we modify the original index structure and *LARC++* construct a keyword-based label index. Based on such index, we first introduce the $\mathbb{K}k$NN query processing, and then propose two efficient algorithms *LARC* and for processing the $\mathbb{K}$CkNN on road networks. For *LARC*, we introduce a *window sliding approach* to build a dominance interval to deal with low frequency keywords. For *LARC++*, we propose a *path-based dominance updating approach* to resolve a dominance region for high frequency keywords. Our experimental evaluation demonstrates the effectiveness and efficiency of our solution for processing the $\mathbb{K}$CkNN queries on large real-world datasets, which outperforms the state-of-the-art method with almost 50% decrease on computation cost and almost 20% decrease on communication cost.

Chapter 5

Efficient Clue-based Route Search on Road Networks

With the advances in geo-positioning technologies and location-based services, it is nowadays quite common for road networks to have textual contents on the vertices. Previous works on identifying an optimal route that covers a sequence of query keywords have been studied in recent years. However, in many practical scenarios, an optimal route might not always be desirable. For example, a personalized route query is issued by providing some clues that describe the spatial context between PoIs along the route, where the result can be far from the optimal one. Therefore, in this chapter, we investigate the problem of clue-based route search (CRS), which allows a user to provide clues on keywords and spatial relationships. First, we propose a greedy algorithm and a dynamic programming algorithm as baselines. To improve efficiency, we develop a branch-and-bound algorithm that prunes unnecessary vertices in query processing. In order to quickly locate candidate, we propose an AB-tree that stores both the distance and keyword information in tree structure. To further reduce the index size, we construct a PB-tree by utilizing the virtue of 2-hop label index to pinpoint the candidate. Extensive experiments are conducted and verify the superiority of our algorithms and index structures.

The remainder of this chapter is organized as follows. We give an introduction in Section 5.1 and define the necessary concepts, formulate the problem of clue-based route search CRS and introduce some preliminary knowledge in Section 5.2. Then we propose a greedy algorithm GCS in Section 5.3 to answer CRS approximately. Section 5.4 presents a clue-based dynamic programming algorithm

CDP to return exact answer to CRS query. Efficient branch-and-bound algorithm BAB is introduced in Section 5.5, as well as two index structures AB-tree and PB-tree. Section 5.6 presents a semi-dynamic mechanism for proposed index structure. Section 5.7 reports the experimental observations. Finally, Section 5.8 concludes the work.

5.1 Introduction

With the rapid development of location-based services and geo-positioning technologies, there is a clear trend that an increasing amount of spatial-textual objects are available in many applications. For example, the location information as well as concise textual descriptions of some businesses (e.g., restaurants, hotels) can be easily found in online local search services (e.g., yellow pages). Another example is the GPS navigation system, where a PoI (Point-of-Interest) is a specific point location that someone may find useful or interesting, and is usually annotated with textual information (e.g., descriptions and users' reviews). By marking a PoI as destination on the map, users are able to plan a trip with suggestions. Moreover, in many social network services (e.g., Facebook, Flickr), a huge number of photographs are accumulated everyday that are geo-tagged by users. These uploaded photographs are usually associated with multiple text labels. To provide better user experience, various keyword related spatial query models and techniques have emerged such that the spatial-textual objects can be efficiently retrieved. A basic spatial keyword query takes a geo-location and a set of keywords as arguments and returns relevant top-k objects [23]. In more sophisticated case, keyword queries are also combined with travel planning in commercial applications such as GPS navigation systems or online map services. The existing solutions (e.g., [16, 63, 93]) for trip planning or route search are dealing with the scenarios when a user wants to visit a sequence of PoIs, each of which contains a user specified keyword. Different optimization constraints were proposed in these works, and the goal was to find an optimal route with minimum cost. In general, the cost can be of various different types, such as travel distance, time or budget.

However, in many practical scenarios, an optimal route might not always be desirable. It is not uncommon that a user aims to plan a trip in a region but can only provide partial and approximate spatial context around the PoIs within the trip. For example, a user wants to find an Italy restaurant in a city visited many years ago. She cannot remember the exact name and address but she still recalls

that "*On the way driving to the restaurant from her home, she passed a cafe at about 1km away, and drove about another 2km to reach the restaurant*". The information given above usually cannot precisely locate a PoI, but intuitively it provides clues to identify the most likely PoIs along the route. It is obvious that the optimal route definitely does not satisfy the user's search intention since it could be far from the best route. Consider another scenario, "*A user wants to find a buffet restaurant and a nearby cinema only in walking distance, say 3km, thus he can watch a movie after dinner. Therefore, after having delicious food, he can walk to the cinema in order to maintain a healthy lifestyle*". These personalized requirements make the route search become distance-sensitive such that the distance between PoIs along the route must be as close as possible to the user specified distance. Another example is that when a user wants to know the direction for a specific place and asks others for help, she may still not be able to exactly figure out the route after obtaining the answers from them. Therefore, a novel type of route search which interprets the clues contained in such answers becomes necessary.

Motivated by these observations, in this work, we investigate the problem of clue-based route search (CRS), which allows a user to provide clues on textual and spatial context within the route. Formally, a CRS query is defined over a road network G, and the input to the query consists of a source vertex v_q and a sequence of clues $C = \{\mu_i\}$, where each clue μ_i contains a query keyword w_i and a user expected network distance d_i. The query returns a path $\mathcal{P}$ in G starting at v_q, such that $\mathcal{P}$ passes through vertices (PoIs) that contain all the query keywords and comply the same keyword order as in C. In the meantime, it has the minimum matching distance, which is defined as the degree of satisfaction of the user to $\mathcal{P}$. To the best of our knowledge, none of the existing solutions (e.g., [16, 63, 93]) on trip planning or route search can be applicable for solving CRS queries.

In order to process the CRS query efficiently, we need to overcome several challenges. The first challenge is concerned with the large amount of possible routes for validation. Basically, the CRS requires candidate vertices that contain query keywords in the route to comply a specific order defined in query. As a feasible path is supposed to cover all the query keywords, the number of feasible paths increases exponentially with the amount of clues. Therefore, a greedy approach to solve our query is proposed, which continuously finds the next candidate vertex with minimum matching distance. Unfortunately, the optimal result can be substantially different from what the greedy algorithm suggests. Then, we propose a dynamic programming algorithm to answer CRS query exactly, but it requires

quadratic time and is not scalable especially for more frequent keywords. To avoid unnecessary route search, we develop a branch-and-bound algorithm which adopts filter-and-refine paradigm, thus much fewer feasible paths are considered. The second challenge is how to quickly locate candidate vertices in road networks. Given a query vertex u, the matching distance between u and its next candidate v is supposed to be smaller or equal to a threshold. The network expansion approach can be applied here, but it is inefficient due to excessive network traversals. Therefore, we propose a novel index structure, called AB-tree, which stores both keyword and distance information in each node. On top of it, the candidate w.r.t. a query clue can be quickly retrieved. The third challenge is how to reduce the index construction time and space. As AB-tree involves an all-pair matrix computation and has a space cost of $O(|V|^2)$, we propose a PB-tree to further improve the performance. Inspired by the 2-hop label [2,6], which answers distance queries with a small label index, we modify the structure of original label index to construct a binary tree on each pivot. In addition, we propose a semi-dynamic mechanism for PB-tree to support the index updating.

The principal contributions of this work can be summarized as follows.

- We propose a greedy clue search algorithm (GCS) to answer the CRS query approximately with no index involved. In GCS, we adopt the network expansion approach to greedily select the current best candidate at each step to construct feasible paths.

- We also develop a clue-based dynamic programming algorithm (CDP) that attempts to enumerate all feasible paths and finally returns the optimal result. In CDP, distance oracle is used to compute the network distance between candidates.

- We further propose a branch-and-bound algorithm (BAB) by applying filter-and-refine paradigm such that only a small portion of vertices are visited, hence improves the search efficiency. In order to quickly locate the candidate vertices, we develop AB-tree and PB-tree structures to speed up the tree traversal, as well as a semi-dynamic index updating mechanism to keep the index maintainable when growing bigger.

- Our experimental evaluation demonstrates the efficiency of our algorithms and index structures for processing the CRS queries on real-world datasets. We show the superiority of our algorithms in answering CRS when compared with the baseline algorithms.

5.2 Problem Statement

We model a road network as a weighted undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each edge $(u, v) \in E$ has a positive weight, i.e., length or travelling time on the edge, denoted as $e(u, v)$. Each vertex $v \in V$ contains a set of keywords, denoted as $\Phi(v)$. Given a path between vertices u and v, denoted as $\mathcal{P}(u, \ldots, v)$, the length is the sum of weights of all edges along the path. For any two vertices u and v, the network distance between u and v on G, denoted as $d_G(u, v)$, is the length of the shortest path between u and v, which is denoted as $\mathcal{SP}(u, v)$. The notations used in this work is summarized in Table 5.1.

Table 5.1: Summary of notations in CRS

Notation	**Definition**
$G = (V, E)$	Road network with vertex V and edge E
$\mathcal{P}(u, \ldots, v)$	A path from u to v
$\mathcal{FP}(u, \ldots, v)$	A feasible path from u to v
$d_G(u, v)$	Network distance between u and v in G
$\mu(w, d)$	A clue with query keyword w and expected network distance d
$\sigma(u \rightarrow v)$	A match from u to v
$d_m(\mu, \sigma)$	Matching distance between clue μ and vertex pair σ
$d_m(C, \mathcal{FP})$	Matching distance between query C and feasible path $\mathcal{FP}$
$L(v)$	2-hop label of v
$BT(v)$	Binary tree of v with keyword and distance information
$PR(o)$	Pivot-based reserve label of vertex o
$PB(o)$	Binary tree of pivot o

5.2.1 Problem Definition

Definition 5.1 (Clue)**.** *A clue is defined as $\mu(w, d)$, where w is a query keyword and d is a user defined distance. Given a source vertex u, the clue implies that we can find a vertex v that contains w, and the network distance $d_G(u, v)$ is as close as possible to d, in order that the user's search intention is satisfied.*

Definition 5.2 (Matching Distance)**.** *Given a source u and a clue $\mu(w, d)$, we say that the vertex pair $\sigma(u \to v)$ is a match w.r.t. clue μ, if $w \in \Phi(v)$. The matching distance between a clue μ and its match $\sigma(u \to v)$ in G, denoted as $d_m(\mu, \sigma)$, is computed by d and the network distance $d_G(u, v)$, such that*

$$d_m(\mu, \sigma) = \begin{cases} 1, & \text{if } d_G(u, v) \geq 2d; \\ \frac{|d - d_G(u,v)|}{d}, & \text{otherwise} \end{cases} \tag{5.1}$$

Definition 5.3 (Feasible Path)**.** *We define a query $Q = (v_q, C)$ where C is a sequence of clues denoted as $C = \{\mu_1(w_1, d_1), \ldots, \mu_k(w_k, d_k)\}$. Given a query Q, if a path $\mathcal{P}(v_q, v_1, \ldots, v_k)$ that starts from v_q and matches all keywords in C in the same order, i.e., $w_1 \in \Phi(v_1), \ldots, w_k \in \Phi(v_k)$, we call such path as a feasible path, denoted as $\mathcal{FP}(v_q, v_1, \ldots, v_k)$. Moreover, each segment $\overline{v_i v_{i+1}} \in \mathcal{FP}$ is the shortest path $\mathcal{SP}(v_i, v_{i+1})$ in G. Note that, if v_i is the matching vertex corresponds to the clue μ_i, then it is taken as the source to find next matching vertex v_{i+1} that corresponds to μ_{i+1} and constructs $\sigma(v_i \to v_{i+1})$. The matching distance between C and its feasible path $\mathcal{FP}$ is defined as the maximum matching distance between all clues $\mu \in C$ and their corresponding matches $\sigma \in \mathcal{FP}$, that is*

$$d_m(C, \mathcal{FP}) = \max_{\mu_i \in C, \sigma_i \in \mathcal{FP}} d_m(\mu_i, \sigma_i) \tag{5.2}$$

Definition 5.4 (Clue-based Route Search)**.** *A clue-based route search (CRS) contains a query $Q = (v_q, C)$, and it finds a feasible path $\mathcal{FP}(v_q, v_1, \ldots, v_k)$, such that $d_m(C, \mathcal{FP})$ is minimized.*

It is worth to note that the CRS query can be easily extended to have a destination by assuming that the query keyword contained in destination is unique within G, or have no source involved. In addition, for simplicity, we only discuss the optimal feasible path in this chapter, but the algorithms introduced can be easily extended to find top-k feasible paths.

Example 5.1. *Figure 5.1 shows an example of G, where $\sigma(v_4 \to v_8)$ is a match of clue $\mu(w_2, 4)$ with $d_m(\mu, \sigma) = 0.25$. Given query $Q = (v_1, \{(w_2, 5), (w_1, 4), (w_3, 5)\})$, it is easy to see $\mathcal{P}(v_1, v_3, v_6, v_7)$ is*

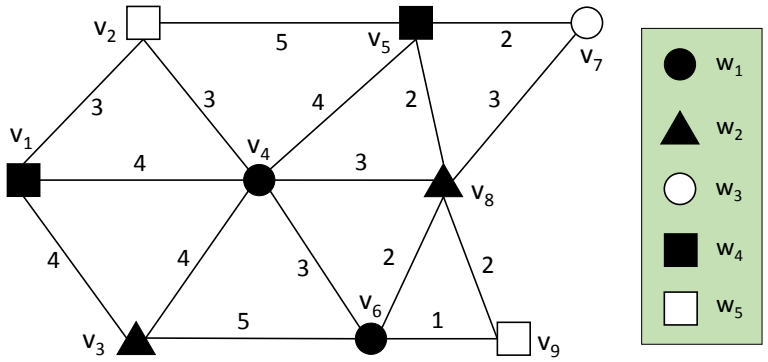
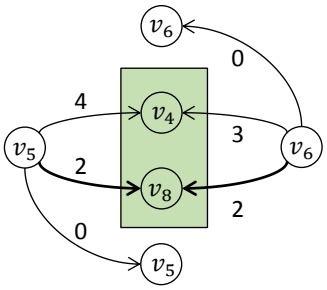


Figure 5.1: Running example of G

a feasible path with $d_m(C, \mathcal{P}) = 0.25$. *Note that,* $\mathcal{P}(v_1, v_3, v_4, v_7)$ *is the optimal feasible path with minimum matching distance* $d_m(C, \mathcal{P}) = 0.2$.

5.2.2 Preliminary: Distance Oracle

We adopt the idea of distance oracle $\mathcal{DO}$ to calculate the network distance between two input vertices. Given a source-target pair of vertices, $\mathcal{DO}$ returns the shortest network distance between them. As we know, the algorithms and data structures on $\mathcal{DO}$ have been extensively studied by existing works, which can be roughly summarized into two categories, expansion-based methods and lookup-based methods. The most famous expansion-based method for $\mathcal{DO}$ is Dijkstra's algorithm [34], which, given a s-t pair in road network G, traverses the vertices in G from s to t. However, the problem of using Dijkstra's algorithm is that it must visit every vertex that is closer to s, and the number of such unneeded vertices can be enormous when s and t are far apart, which incurs redundant network traversal. Besides, the lookup-based methods usually have to store some precomputed results. For example, all-pair method is space inefficient that we have to precompute and store a distance matrix, which requires $O(n^2)$ space for a road network G with n vertices. To the best of our knowledge, one of the most notable recent developments is the emergence of practical *2-hop labeling* methods [2,5,6,54] for $\mathcal{DO}$ on large networks. It constructs labels for vertices such that a distance query for any vertex pair u and v can be answered by only looking up the common labels of u and v. For each vertex v, we precompute a label, denoted as $L(v)$, which is a set of label entries and each label entry is a



$L(v_1)$	$\{(v_1,0),(v_2,3),(v_3,4),(v_4,4)\}$
$L(v_2)$	$\{(v_2,0),(v_4,3),(v_5,5)\}$
$L(v_3)$	$\{(v_3,0),(v_4,4),(v_6,5)\}$
$L(v_4)$	$\{(v_4,0)\}$
$L(v_5)$	$\{(v_4,4),(v_5,0),(v_8,2)\}$
$L(v_6)$	$\{(v_4,3),(v_6,0),(v_8,2)\}$
$L(v_7)$	$\{(v_4,6),(v_5,2),(v_7,0),(v_8,3)\}$
$L(v_8)$	$\{(v_4,3),(v_8,0)\}$
$L(v_9)$	$\{(v_4,4),(v_6,1),(v_8,2),(v_9,0)\}$

FIGURE 5.2: 2-hop label index of G

pair $(o, \eta_{v,o})$, where $o \in V$ and $\eta_{v,o} = d_G(v,o)$ is the distance between v and o. We say that o is a **pivot** in label entry if $(o, \eta_{v,o}) \in L(v)$. Given two vertices u and v, we can find a common pivot o that $(o, \eta_{u,o}) \in L(u)$ and $(o, \eta_{v,o}) \in L(v)$:

$$d_G(u,v) = \min\{\eta_{u,o} + \eta_{v,o}\} \tag{5.3}$$

We say that the pair (u,v) is covered by o and the distance query $d_G(u,v)$ is answered by o with smallest $\eta_{u,o} + \eta_{v,o}$. Therefore, we can compute $d_G(u,v)$ in $O(|L(u)| + |L(v)|)$ time by using a merge-join like algorithm. As shown in right side of Figure 5.2, we can find the generated label index. For the distance query between v_5 and v_6, we first find their common pivots v_4 and v_8, then $d_G(v_5,v_6) = 4$ is returned since $\eta_{v_5,v_8} + \eta_{v_6,v_8} = 4 < \eta_{v_5,v_4} + \eta_{v_6,v_4} = 7$.

5.3 Greedy Clue Search Algorithm

We develop a greedy algorithm as a baseline for answering the CRS query, which is called Greedy Clue Search (GCS) algorithm. Given a query $Q = (v_q, C)$, first we simply add v_q into a candidate path. Then we use the Procedure *findNextMin()* to determine the next candidate v_1 that contains w_1 and the matching distance between μ_1 and $\sigma_1(v_q \rightarrow v_1)$, i.e., $d_m(\mu_1, \sigma_1)$, is minimized. Afterwards, we insert v_1 into the candidate path, and continue to find its contagious candidate by *findNextMin()*. This process is repeated until all the matching vertices are determined, thus the candidate path forms

a feasible path, denoted as $\mathcal{FP}_{v_q}$.

It is worth to note that, although $\mathcal{FP}_{v_q}$ can be quickly retrieved, its matching distance can be substantially different from the optimal result. In order to improve the accuracy, we further select the vertices that contain the most infrequent keyword w_τ in C as our initial candidates, which can be easily determined based on the frequencies of occurrences in road network G. Then, for each vertex u that contains w_τ, we fetch it into a candidate path and use Procedure *findNextMin()* to determine its contagious candidate v that contains $w_{\tau+1}$ (or $w_{\tau-1}$). The rest of algorithm is the same as processing v_q. After all vertices u that contain w_τ are processed, we obtain $|V_{w_\tau}| + 1$ feasible paths in total, where V_{w_τ} is the set of vertices that contain w_τ. Finally, we select the feasible path with the smallest matching distance, i.e., $d_m(C, \mathcal{FP})$, to answer the CRS approximately, and the result is denoted as $\mathcal{FP}_{gcs}$.

In Procedure *findNextMin()*, we utilize the network expansion algorithm [51] to find the nearby vertices that contain the query keywords. The algorithm details are shown in Algorithm 8. Given the source u, query keyword w and user expected distance d, we aim to find v that the difference between $d_G(u, v)$ and d is minimized. In the network traversal starting from u, we check every visited vertex to see if it contains w. If v is the first visited vertex containing w and $d_G(u, v) > d$, then we stop and return v since the difference incurred by the remaining unvisited vertices cannot be less than $d_G(u, v) - d$. Otherwise, we continue to find the next vertex v' that contains w. If v' is found and $d_G(u, v') < d$, we update v by v' since v' renders a smaller difference than v. Otherwise, we compare $d - d_G(u, v)$ with $d_G(u, v') - d$ and return the smaller one as the result.

Basically, we use the most infrequent keyword w_τ because this reduces the number of operations to find all feasible paths $\mathcal{FP}_u$. As we know, in GCS, we have $|V_{w_\tau}| + 1$ feasible paths as candidates. For each feasible path, we execute k times of *findNextMin()* to determine candidates. If we assume Procedure *findNextMin()* costs time f, then the time complexity of GCS is $O(|V_{w_\tau}| \cdot k \cdot f)$.

Example 5.2. *In running example Figure 5.1, we are given query* $Q = (v_1, \{(w_2, 5), (w_1, 4), (w_3, 5)\})$. *First, we fetch* v_1 *into feasible path, and call findNextMin*$(v_1, w_2, 5)$ *and return* v_3 *with* $d_m = 0.25$. *Therefore, we repeat the process and finally obtain* $\mathcal{FP}_{v_1} = (v_1, v_3, v_4, v_7)$ *with* $d_m(C, \mathcal{FP}_{v_1}) = 0.2$. *Then we take* w_3 *as* w_τ *since* w_3*' frequency is only 1. Hence, we fetch* v_7 *into feasible path and repeat the process. Finally, we obtain feasible path* $\mathcal{FP}_{v_7} = (v_1, v_3, v_6, v_7)$ *and* $d_m(C, \mathcal{FP}_{v_7}) = 0.25$. *Therefore, we have* $\mathcal{FP}_{gcs} = \mathcal{FP}_{v_1}$ *and* $d_m(C, \mathcal{FP}_{gcs}) = 0.2$.

Algorithm 8: Greedy Clue Search GCS

Input: $Q = (v_q, C = \{(w_1, d_1), \ldots, (w_k, d_k)\})$

Output: $\mathcal{FP}_{gcs}$ with minimum $d_m(C, \mathcal{FP})$

```
1  Find w_τ ∈ C and u ∈ V_{w_τ};
2  for each u ∈ V_{w_τ} do
3      for w_i ← w_τ ∈ {w_τ, ..., w_k} and {w_τ, ..., w_1} do
4          u_{i+1} ← findNextMin(u_i, w_{i+1}, d_{i+1});
5          or u_{i-1} ← findNextMin(u_i, w_{i-1}, d_i);
6      Obtain d_m(C, FP_u);
7      if d_min > d_m(C, FP_u) then
8          d_min ← d_m(C, FP_u);
9  Compute FP_{v_q};
10 return FP_gcs and d_m(C, FP_gcs) ← d_min;
```

Procedure *findNextMin*(u, w, d)

```
1  From u, find v contains w, thus obtain d_G;
2  while true do
3      Find next v' contains w, thus obtain d'_G;
4      if d_G < d and d'_G > d then
5          break;
6      else
7          v ← v' and d_G ← d'_G;
8  return min{d_m(μ, σ)} and v;
```

5.4 Clue-based Dynamic Programming Algorithm

As we know, even though GCS has a short response time, the accuracy of the answer cannot be guaranteed. To achieve better accuracy, we propose an exact algorithm, called Clue-based Dynamic Programming (CDP), to answer the CRS query. Generally, it is challenging to develop an efficient exact algorithm for CRS queries, since we cannot avoid exhaustive search for PoIs in road networks.

For instance, the number of vertices that contain $w_i \in C$ is denoted as $|V_{w_i}|$, thus the time complexity of the brute-force approach, which attempts all possible combinations, is $O(\prod_{w_i \in C} |V_{w_i}|)$.

In CDP, we construct a keyword posting list for each keyword w, which is a list of vertices that contain w. When a CRS query is issued, we sort the posting lists according to the keyword order of $w_i \in C$. Note that the order of the vertices within each posting list does not matter and can be arbitrary, hence are sorted by vertex id for simplicity. It is easy to see that these posting lists actually construct a *k-bipartite* graph G', which in fact shows all feasible paths for a given C. The weight of each edge in G' is computed as the matching distance. Specifically, for each $u \in V_{w_i}$, we define $D(w_i, u)$ to denote the minimum matching distance one can achieve with a walk that passes the keywords from w_1 to w_i consistent with the order in C and stops at u. In other words, the weight of vertex $u \in G'$ is computed by $D(w_i, u)$, which is the minimum matching distance of all partial feasible paths end at u. Then we compute $D(w_i, u)$ by the following recursive formula:

$$D(w_i, u) = \begin{cases} \min\limits_{v \in V_{w_{i-1}}} \{\max\{D(w_{i-1}, v), d_m(\mu_i, \sigma(v \to u))\}\}, & i > 1 \\ d_m(\mu_i(w_i, d_i), \sigma(v_q \to u)), & i = 1 \end{cases} \tag{5.4}$$

For each iteration, we have $|V_{w_{i-1}}| \cdot |V_{w_i}|$ combinations, thus the time required in Equation 5.4 is $O(\sum_{i=2}^{k} |V_{w_{i-1}}| \cdot |V_{w_i}|)$. The details of CDP is shown in Algorithm 9. In order to compute $D(w_i, u)$, we have to access the posting list of w_{i-1}. For each vertex v in this list, we compute $d_m(\mu_i, \sigma(v \to u))$. Then we compare it with $D(w_{i-1}, v)$, and keep the greater one as intermediate value. Finally, we find the minimum one as $D(w_i, u)$ from these $|V_{w_{i-1}}|$ intermediate values. After we recursively process all the keywords, we finally find the minimum $D(w_k, u)$ and backtrace the corresponding vertices that construct $\mathcal{FP}_{cdp}$.

In each iteration, we have a clue $\mu_i(w_i, d_i)$, therefore we have to compute $d_G(u, v)$ between each $u \in V_{w_i}$ and its precedents $v \in V_{w_{i-1}}$ as prerequisites for determining $d_m(\mu_i, \sigma(v \to u))$. Here we adopt the distance oracle introduced in Section 5.2.2 to compute $d_G(u, v)$.

Example 5.3. *As shown in Figure 5.3, given query* $Q = (v_7, \{(w_1, 6), (w_2, 4), (w_4, 5)\})$. *To compute* $D(w_4, v_1)$, *we first compare* $D(w_2, v_3) = 0$ *with* $d_m(\mu_2, \sigma(v_3 \to v_1)) = 0.2$, *and obtain intermediate value 0.2. Then we also compute the other intermediate value 0.4, therefore* $D(w_4, v_1) = 0.2$. *Likewise, we have* $D(w_4, v_5) = 0.6$. *Therefore, CDP returns* $\mathcal{FP}_{cdp} = (v_7, v_4, v_3, v_1)$ *with* $d_m(C, \mathcal{FP}_{cdp}) = 0.2$.

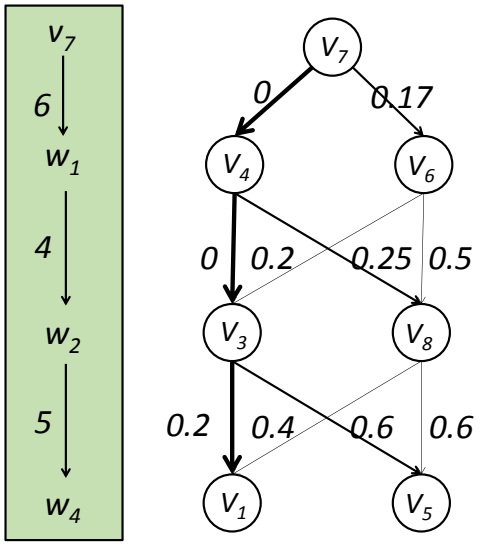


FIGURE 5.3: Matching distances of CDP

5.5 Branch and Bound Algorithm

Although CDP provides an exact solution, the search efficiency cannot be maintained. For instance, consider the worst case, we assume that all vertices contain query keywords, then the time is $O(k \cdot |V|^2)$. To propose a more efficient algorithm, we assume there is an artificial directed graph G', which is similar to the *k-bipartite* graph in CDP that formed by all candidate vertices containing keywords in C, where the edge of G' is a match of one clue and in the meantime its direction complies the keyword order of the clue. Note that, G' is organised into k levels, and each level i corresponds to each keyword w_i. Based on G', we develop a Branch-and-Bound (BAB) algorithm to search G' in a depth-first manner by applying the filter-and-refine paradigm, which only visits a small portion of vertices in G'. We can use the result of GCS to speed up the search process since it can serve as an initial upper bound. We start the searching from level 1 to k to obtain a feasible path $\mathcal{FP}$, if the matching distance $d_m(C, \mathcal{FP})$ is greater than the current upper bound, we continue to search for the next candidate feasible path, otherwise we update the upper bound. It is worth to note that it is not necessary to go through every candidate feasible path. If the matching distance at intermediate level already exceeds the upper bound, it can be removed. This process terminates when the matching distance next to be processed at level 1 can be filtered, since it is impossible to find a feasible path with smaller match distance.

Algorithm 9: Clue-based Dynamic Programming CDP

Input: $Q = (v_q, C = \{(w_1, d_1), \ldots, (w_k, d_k)\})$

Output: $\mathcal{FP}_{cdp}$ with $d_m(C, \mathcal{FP}_{cdp})$

```
1  for each u ∈ V_{w_1} do
2      Initial D(w_1, u);
3  for 1 < i ≤ k do
4      for each u ∈ V_{w_i} do
5          Initial intermediate vector iv(u);
6          for each v ∈ V_{w_{i-1}} do
7              if d_m(μ_i, σ(v → u)) < D(w_{i-1}, v) then
8                  iv(u) insert D(w_{i-1}, v);
9              else
10                 iv(u) insert d_m(μ_i, σ(v → u));
11         D(w_i, u) ← min{iv(u)}
12 Find min{D(w_k, u)};
13 return FP_cdp and d_m(C, FP_cdp) ← min{D(w_k, u)};
```

Initially, we keep a stack to store the partial candidate path, which contains a sequence of vertices and corresponding matching distances. First, we fetch a vertex v_q into the stack, then we continue to find next candidate at level 1. Basically, the key component of this algorithm is to quickly locate the next best vertex, and the details of Procedure *findNext()* will be introduced later. Given a partial candidate path $\mathcal{P}(v_q, v_1, \ldots, v_i)$ obtained at level i, we apply *findNext()* to find the next candidate v_{i+1} at level $i + 1$. Once v_{i+1} is found, we compute $d_m^{i+1}(\mu_{i+1}, \sigma(v_i, v_{i+1}))$ and compare it with current UB. For simplicity, we use $d_m^{i+1}(v_{i+1})$ to denote the matching distance at level $i + 1$ resulted by v_{i+1}, i.e., $d_m^{i+1}(\mu_{i+1}, \sigma(v_i, v_{i+1}))$. Note that, v_{i+1} is accepted as a candidate and inserted into the stack if and only if its matching distance $d_m^{i+1}(v_{i+1})$ is smaller than UB. Otherwise, v_i is removed from the stack as well as $d_m^i(v_i)$. In other words, v_i is not valid that the path $\mathcal{P}(v_q, v_1, \ldots, v_{i-1})$ cannot survive by passing v_i, then we have to find an alternative v_i'. As we know v_i is the current best candidate at level i, therefore we have to relax the matching distance by finding v_i' where $d_m^i(v_i) \leq d_m^i(v_i')$ and $d_m^i(v_i')$ is minimum among

all the rest vertices untouched at level i. Afterwards, if v'_i is valid, we continue to apply *findNext()* on it.

Specifically, after we obtain $\mathcal{P}(v_q, v_1, \ldots, v_{k-1})$ at level $k-1$, if v_k is returned by *findNext()*, then we check if $d^k_m(v_k)$ exceeds UB. If v_k is not valid, we prune v_k and simply repeat the above process. Otherwise, we insert v_k into the stack, and a complete feasible path is determined. Hence, $\mathcal{P}(v_q, v_1, \ldots, v_k)$ is regarded as a temporary result, and UB is updated by the minimum matching distance among all $d^i_m(v_i)$s. It is easy to see that, we cannot find a better feasible path by alternating v_k with v'_k at level k, since no further level is available to make up the relaxation caused by v'_k. Therefore, in addition to remove v_k, we continue to remove v_{k-1} from the stack and repeat the above process.

In general, the pruning happens from the lower levels to the higher levels, i.e., from level k to level 1. In the end, at level 1, if the matching distance induced by the next candidate vertex is greater than UB, it is impossible to find another feasible path, thus the stack becomes empty after the last vertex v_q is removed, and this process terminates.

Example 5.4. *In the running example, given query* $Q = (v_7, \{(w_1, 6), (w_2, 4), (w_4, 5)\})$. *First we fetch* v_7 *into the stack, and findNext() returns* v_4 *with* $d^1_m(v_4) = 0$. *Then we insert* v_4 *into stack and continue to find next candidate vertex, and* v_3 *is obtained with* $d^2_m(v_3) = 0$. *The process continues and then we have* v_1 *with* $d^3_m(v_1) = 0.2$. *As the size of stack is same as the number of query keywords, a feasible path* $\mathcal{FP} = (v_7, v_4, v_3, v_1)$ *with* $d_m(C, \mathcal{FP}) = 0.2$ *is obtained, and UB is updated by 0.2. Next, we remove* v_1 *and* v_3 *from the stack, and continue to find next candidate of* v_4. *As* $d^2_m(v_3) = 0$, *we relax the matching distance and call findNext() which returns* v_8 *with* $d^2_m(v_8) = 0.25$. *Then we have to remove* v_4 *from the stack since* $d^2_m(v_8)$ *already exceeds current upper bound UB. Now we move on to apply findNext() on* v_7 *and returns* v_6 *with* $d^1_m(v_6) = 0.17$. *However, the next candidate* v_5 *has* $d^2_m(v_3) = 0.25$ *greater than UB, thus we remove* v_6 *and* v_7 *from stack. Therefore, the algorithm terminates since no other feasible path exists. We have* $\mathcal{FP}_{bab} = (v_7, v_4, v_3, v_1)$ *with* $d_m(C, \mathcal{FP}_{bab}) = 0.2$.

5.5.1 All-Pair Distance Approach

In BAB, the Procedure *findNext()* is applied on v_{i-1} to find the next candidate vertex v_i. We can simply use Procedure *findNextMin()* in GCS to locate the next candidate, but it is inefficient due to redundant network traversal when $d_i \in \mu_i$ is large. Moreover, when we prune v_i and attempt to find alternative v'_i,

Algorithm 10: Branch and Bound BAB

Input: $Q = (v_q, C = \{(w_1, d_1), \ldots, (w_k, d_k)\})$

Output: $\mathcal{FP}_{bab}$ with $d_m(C, \mathcal{FP}_{bab})$

```
1  Initial stackV and stackD;
2  Initial search threshold θ;
3  Push v_q into stackV;
4  while stackV is not empty do
5      i ← stackV.size();
6      if findNext(v_{i-1}, d_i, w_i, θ) = true then
7          Obtain v_i and d_m^i(v_i);
8          θ ← 0.0;
9          Push v_i into stackV;
10         Push d_m^i(v_i) into stackD;
11         if i equals to k then
12             if max{stackD} <= UB then
13                 Update UB by max{stackD};
14                 Update FP_bab by stackV;
15             Pop twice stackV;
16             Pop stackD;
17             Update θ by top of stackD;
18             Pop stackD;
19     else
20         Pop stackV;
21         Update θ by top of stackD;
22         Pop stackD;
23 return FP_bab and d_m(C, FP_bab) ← UB;
```

it is easy to see *findNextMin()* cannot be directly applied. Therefore, we propose an All-pair Binary tree (AB-tree) index to improve the search efficiency.

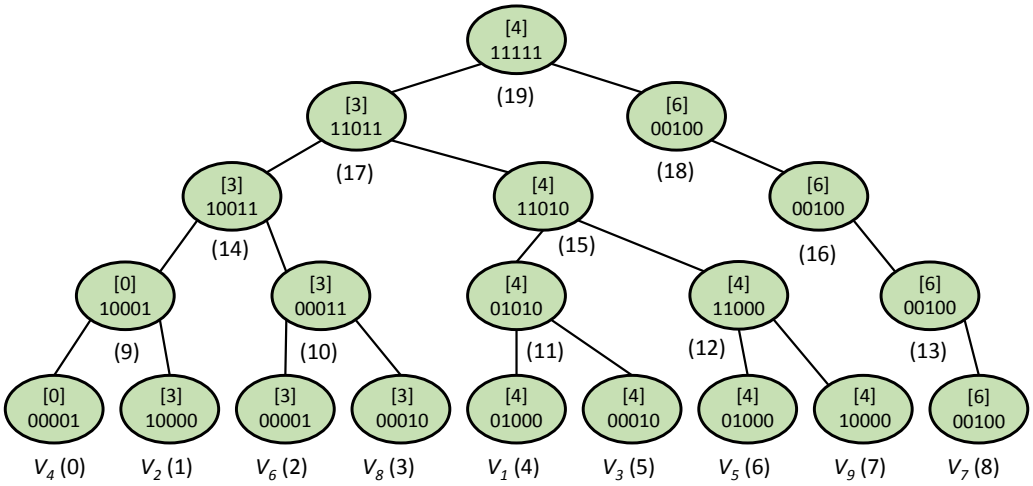
All-Pair Binary Tree

Given a vertex u, we aim to find a vertex v containing keyword w such that the matching distance between $\sigma(u \rightarrow v)$ and query clue μ is slightly greater than and closest to a threshold θ among all vertices containing w in G. Note that, the threshold θ is settled by previous filtered candidate at the same level with v, and it is 0 at initial stage. In other words, we are supposed to find the vertex v that the difference between $d_G(u, v)$ and $d \in \mu$ is close to $\theta \cdot d$. To this end, we construct AB-tree as follows.

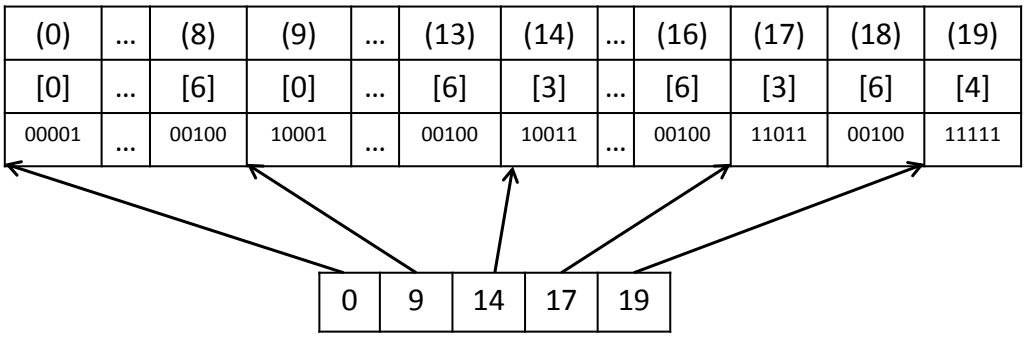
For each $v \in V$, we construct a binary tree $BT(v)$ that contains the information of network distances and keywords. After the all-pair distance matrix is obtained, for each v, we have a list of vertices sorted in ascending order of network distance to v. By utilizing the tree structure, the vertices in the list are divided into fragments that the network distances w.r.t. v of the vertices in the same fragment are close to each other, which speeds up the looking up for vertices by network distance. In addition, the keyword information within each fragment is also stored in $BT(v)$ such that the vertices containing query keyword in a fragment can be efficiently retrieved.

We utilize a hash function H that maps keywords and vertices to a binary code with h bits. For each keyword w, one of its h bits in $H(w)$ is set to 1. Hence, the binary code of a vertex v is the superimposition of $H(w)$ for all w it contains, i.e., $H(v) = \vee_{w \in \Phi(v)} H(w)$. Likewise, for a set of vertices S, $H(S)$ is the superimposition of $H(v)$ that $v \in S$. It is worth to note that a non-zero value of $H(w) \wedge H(S)$ indicates that there may exist a vertex $v \in S$ containing w, and $H(w) \wedge H(S) = 0$ means w is definitely not contained by any $v \in S$. $BT(v)$ is actually a B^+-tree with fanout $f = 2$. Each leaf node contains the information of a vertex u with both the network distance $d_G(u, v)$ and binary code $H(u)$ stored. For non-leaf node, it also keeps a routing element, which equals the maximum network distance of its left subtree. Therefore, $BT(v)$ is constructed recursively in bottom-up manner as shown in Figure 5.4(a).

Storing $BT(v)$ in an array. As we know, storing the tree structure as an array enables a better performance than storing pointers. Therefore, we propose a scheme to sequentially store all nodes of $BT(v)$ in an array from nodes on height 0 to the root, as shown in Figure 5.4(b). In addition to this



(a) Overview of $BT(v_4)$



(b) Storing $BT(v_4)$ in array

Figure 5.4: Overview of all-pair binary tree

array, we also keep an auxiliary array that indicates the number of nodes in each level of $BT(v)$, by which we can quickly determine the indices of the subnodes of a non-leaf node, or the index of its parent node, in the array. For example, if we want to find the left and right subnodes of node 16 in $BT(v_4)$, we know there are two nodes on its left side by $16-14 = 2$ where 14 is the start index of nodes at height 2, so the index of its left subnode is $9+2*2 = 13$ and the right is $9+2*2+1 = 14$. However, we notice 14 is actually at height 2, then we figure out node 16 does not have a right subnode.

Predecessor and Successor Queries on AB-tree

After the construction of AB-tree, we discuss how to use it so that the next vertex in candidate path can be quickly located. Initially, if there is no previous vertices accessed at the next level of v_{i-1}, the

network distance $d_G(v_{i-1}, v_i)$ between v_{i-1} and next candidate v_i is supposed to be smaller or equal to $lD = d_i$, or greater or equal to $rD = d_i$, where $d_i \in \mu_i$. Additionally, consider the aforementioned scenario, we have $\mathcal{P}(v_q, v_1, \ldots, v_i)$, but v_{i+1} returned at level $i+1$ exceeds UB. Then we have to remove v_i from the stack and turn to find v'_i as alternative, where $d^i_m(v_i) \leq d^i_m(v'_i)$. It is easy to see the difference between $d_G(v_{i-1}, v'_i)$ and d_i must be greater or equal to $d^i_m(v_i) \cdot d_i$. In other words, the network distance $d_G(v_{i-1}, v'_i)$ is smaller or equal to $lD = d_i - d^i_m(v_i) \cdot d_i$ or greater or equal to $rD = d_i + d^i_m(v_i) \cdot d_i$. Therefore, the predecessor and successor queries can be issued on $BT(v_{i-1})$ to retrieve next candidate with two boundary network distances lD and rD, respectively.

Predecessor query. Given $BT(u)$, a query keyword w and network distance lD, we aim to find vertex v that contains w and $d_G(u, v)$ is smaller or equal to and closest to lD. First, we compute binary code $H(w)$ for query keyword w. Then we start the process of searching $BT(u)$ recursively from top to bottom. For non-leaf node o, if $H(w) \wedge H(o)$ is non-zero, we continue to search its subtrees. If lD is smaller than the routing element of o, only its left subtree needs to be considered. Otherwise, we first check if we could find v in its right subtree (if exists), if not, we turn to search its left subtree. For leaf node v, we directly check if v contains w and $d_G(u, v)$ is smaller or equal to lD, therefore, false positives can be avoid. Finally, v is obtained. For example, a predecessor query on $BT(v_4)$ with keyword w_2 and $lD = 4$. First, we have $H(w_2) = 00010$. The search starts from root, and as lD equals to the routing element 4, thus we first search its right subtree. After checking $H(w_2)$ with binary code of Node 18, we find it does not contain w_2 and we turn to search Node 17. As the routing element of Node 17 is smaller than lD, we move to search Node 15. Then we check $H(w_2)$ with the binary code in Node 12, and find Node 12 does not contain w_2. After checking with Node 11, we have v_3 as result of the predecessor query.

Successor query. Likewise, we have $BT(u)$, a query keyword w and network distance rD, the goal is to find vertex v that contains w and $d_G(u, v)$ is greater or equal to and closest to rD. For non-leaf node o, if $H(w) \wedge H(o)$ is non-zero, the subtrees of o need to be considered. If rD is smaller or equal to the routing element of o, we search the left subtree to see if v could be found, if not, we turn to search the right subtree (if exists). Otherwise, we simply search the right subtree (if exists) to locate v. For leaf node v, if v contains w and $d_G(u, v)$ is greater or equal to rD, v is reported as result. For example, a successor query on $BT(v_4)$ with keyword w_2 and $rD = 4$. We first check the root with $H(w_2) = 00010$, and rD equals to the routing element, which means we first search Node 17 to see if

it contains w_2, then search Node 18. As the right routing element of Node 17 is smaller than rD, we only need to check Node 15. Then, since the routing element of Node 15 is same as rD, we turn to search Node 11. Finally, we obtain v_3 as result of the successor query.

As mentioned before, we process a predecessor and a successor queries on $BT(v_{i-1})$ with lD and rD respectively to locate candidate at level i. If both predecessor and successor queries find candidate vertices, we compare their matching distance and report the smaller one as result. If only one of them finds candidate vertex, we directly report it. Otherwise, no candidate is found. Note that, in the process to replace v_i with v'_i, we must skip v_i in the tree traversal to avoid infinite loop caused by the special case $d^i_m(v_i) = d^i_m(v'_i)$.

Example 5.5. *For* $Q = (v_7, \{(w_1, 6), (w_2, 4), (w_4, 5)\})$*, assume we already have stack* (v_7, v_4, v_3)*. At level 2, we intend to remove* v_3 *and find an alternative. Given* $d^2_m(v_3) = 0$*, we apply a predecessor and successor queries on* $BT(v_4)$*. For the predecessor query, we take* w_2*,* 4 *and* 0.0 *as input. As* v_3 *is previous result, we skip it and return* v_8*. For the successor query, no vertex is found. Therefore, we report* v_8 *as our next candidate with* $d^2_m(v_8) = 0.25$*.*

Lemma 5.1. *Given* $G = (V, E)$*, the space cost of AB-tree is* $O(|V|^2 \cdot h)$*.*

Proof. For each $v \in V$, we have $|V|$ elements in distance matrix, thus each $BT(v)$ has an index size $O(|V| \cdot h)$. It is easy to see the size of AB-tree is $O(|V|^2 \cdot h)$. □

Algorithm 11: Procedure *findNext()* with AB-tree

Input: Query vertex v_{i-1}, clue w_i and d_i, threshold θ

Output: Next candidate v_i with $d^i_m(v_i)$

```
1  Obtain BT(v_{i-1});
2  lD ← d_i − d_i · θ;
3  rD ← d_i + d_i · θ;
4  v_p and d_p ← BT(v_{i-1}).predecessor(lD, w_i) ;
5  v_s and d_s ← BT(v_{i-1}).successor(rD, w_i) ;
6  if d_i − d_p ≤ d_s − d_i then
7      return v_p with d_m(v_p);
8  else
9      return v_s with d_m(v_s);
```

Procedure *Predecessor*(*lD*, *w*, *Node*)

```
1      if Node is a leaf node then
2          Obtain v_p and d_p of current node;
3          if v_p contains w and d_p ≤ lD then
4              return v_p and d_p;
5          else
6              return false;
7      else
8          Generate H(w);
9          if H(w) ∧ H(Node) = 0 then
10             return false;
11         if lD < Node.routing then
12             lNode ← index of its left subnode;
13             return Predecessor(lD, w, lNode);
14         else
15             rNode ← index of its right subnode;
16             lNode ← index of its left subnode;
17             if rNode exists then
18                 if Predecessor(lD, w, rNode);
19                  then
20                     return v_p and d_p
21                 else
22                     return Predecessor(lD, w, lNode);
23             return Predecessor(lD, w, lNode);
```

5.5.2 Keyword-based Label Approach

Even though AB-tree is able to answer *findNext()* query fast, the index space cost is still high and could only be stored in disk, which results in undesired I/O consumption. In this section, we introduce a main memory based index structure, namely Pivot reverse Binary tree (PB-tree), to deal with *findNext()* query.

Pivot Reverse Binary Tree

As introduced in Section 5.2.2, we know 2-hop label possesses the nature to process distance queries between any two vertices in network with fast response time, whilst keeping the size of the generated label index as small as possible. The problem of reducing label size is orthogonal to our work, thus we fully utilize the state-of-the-art results to build a small index in this work. As we know, in 2-hop label, the distance between any vertex pair (u, v) can be computed correctly through a common pivot o, in other words, each vertex u can reach any other vertex v in network through a pivot o. Therefore, based on this intuition, we modify the structure of original 2-hop label to construct a *pivot reverse index*, i.e., *PR* index [123] which stores all label entries $(o, \eta_{v,o}) \in \bigcup_{v \in V} L(v)$ regarding vertex o as pivot into the *PR* label of vertex o, i.e., $(v, \eta_{v,o}) \in PR(o)$. In $PR(o)$, we assume that all the label entries $(v, \eta_{v,o})$ are sorted in ascending order of distance. For example, we have $(v_3, 0) \in L(v_3)$ and $(v_3, 4) \in L(v_1)$. Through the transformation, we have $PR(v_3) = \{(v_3, 0), (v_1, 4)\}$.

In order to find vertex by keyword and distance information, each $PR(o)$ is organized as same as the binary tree mentioned before, thus forms $PB(o)$. The structure is shown in Figure 5.5, it is worth to note that any network distance $d_G(u, v)$ is divided into two parts, the first part $d_G(u, o)$ between u and its pivot o can be found in $L(u)$, and the other part $d_G(o, v)$ between pivot o and target v can be found in $PB(o)$. Therefore, combined with original label index whose label entries are also sorted in ascending order by network distance, PB-tree could be used to answer predecessor and successor queries more efficiently than AB-tree with a much smaller size.

Predecessor and Successor Queries on PB-tree

With the construction of PB-tree, we discuss the predecessor and successor queries on top of it. Given $PB(v_{i-1})$, we aim to find candidate v_i that contains w_i and $d_G(v_{i-1}, v_i)$ is smaller or equal to lD, or

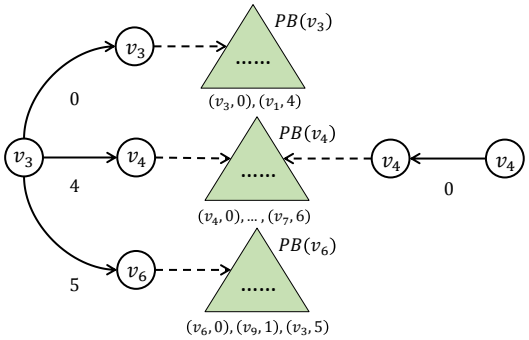


FIGURE 5.5: Overview of pivot reverse binary tree

greater or equal to rD. As we know, $d_G(v_{i-1}, v_i)$ can be divided into two parts $d_G(v_{i-1}, o)$ and $d_G(o, v_i)$. Therefore, straightforwardly, we can apply predecessor and successor queries on $PB(o)$ for each pivot $o \in L(v_{i-1})$ with two bound network distances $lD_o = lD - d_G(v_{i-1}, o)$ and $rD_o = rD - d_G(v_{i-1}, o)$, respectively. For each $PB(o)$, we are supposed to obtain a temporary candidate. Through comparison, we can finally find the next candidate vertex v_i.

Fortunately, it is worth to note that we are not necessarily to access all $PB(o)$s to process predecessor and successor queries. Basically, we know $d_m^i(v_i)$ must not exceed upper bound matching distance, therefore current UB can be utilized to prune the search space. That is to say, v_i could only be found if $d_G(v_{i-1}, v_i)$ is greater or equal to $lB = d_i - d_i * UB$, or is smaller or equal to $rB = d_i + d_i * UB$. Particularly, for each $PB(o)$, the bound distances can be computed as $lB_o = lB - d_G(v_{i-1}, o)$ and $rB_o = rB - d_G(v_{i-1}, o)$. Therefore, the search space can be narrowed down into $[lB_o, lD_o]$ and $[rD_o, rB_o]$. For current pivot o being processed, if we have $rB < d_G(v_{i-1}, o)$, we are impossible to find a candidate in $PB(o)$ since rB_o is negative. In other words, the network distance between v_{i-1} and any vertex in $PB(o)$ is definitely greater than rB thus is not qualified. As we know, the pivots in $L(v_{i-1})$ are sorted in ascending order of network distance, the rest pivots o' after o do not need to be considered since they have even greater network distances to v_{i-1} than o. Therefore, the process terminates.

Predecessor and successor queries. Given $PB(o)$, a query keyword w and two network distance bound ranges $[lB_o, lD_o]$ and $[rD_o, rB_o]$, we aim to find a temporary candidate vertex in $PB(o)$. In particular, the difference between AB-tree and PB-tree is that, given query vertex u, any target v only

shows up once in $AB(u)$, but it might appear in multiple $PB(o)$s. Moreover, if we find a candidate v in $PB(o)$, $d_G(u,o) + d_G(o,v)$ is not necessarily equal to $d_G(u,v)$ since the network distance can only be calculated by the pivot with minimum distance summation. Therefore, we use original label index to check if $\mathcal{P}(u,\ldots,o,\ldots,v)$ is the shortest path $\mathcal{SP}(u,v)$. As mentioned before, if $rB \geq d_G(v_{i-1},o)$, we first apply a successor query on $PB(o)$. After we obtain a temporary vertex v_{tmp} locates in $[rD_o, rB_o]$, we check if o is on the shortest path $\mathcal{SP}(v_{i-1},v_{tmp})$ by comparing $d_G(v_{i-1},v_{tmp})$ with $d_G(v_{i-1},o) + d_G(o,v_{tmp})$. If so, v_{tmp} is reported as a temporary successor result on $PB(o)$. Otherwise, we update rD_o by $d_G(o,v_{tmp})$ and continue to apply a new successor query. This process is repeated until we find a result. After successor query, we compare $d_G(v_{i-1},o)$ with lD to determine if we need to apply a predecessor query on $PB(o)$. Based on the same intuition, if $lD \geq d_G(v_{i-1},o)$, the predecessor query is applied in a similar approach as successor query. Finally, we compare the results of predecessor and successor queries, and obtain the temporary candidate found in $PB(o)$. It is worth to note that we can further narrow down the search space by updating lB and rB. That is, after processing pivot o, if we find a temporary candidate v_{tmp}, lB can be updated by $d_G(v_{i-1},v_{tmp})$ and rB by $2 * d_i - lB$, which benefits the processing of rest o'.

Example 5.6. *For $Q = (v_7, \{(w_1,6),(w_2,4),(w_4,5)\})$, assume we already have stack (v_7,v_4,v_3). At level 2, we intend to find the next candidate. Initially, θ is set as 0.0, therefore we have $lD = rD = 5$. As current $UB = 0.2$, we have $lB = 4$ and $rB = 6$. As shown in Figure 5.5, we first check $PB(v_3)$ with $d_G(v_3,v_3) = 0$. Then we have $lD_{v_3} = rD_{v_3} = 5$, $lB_{v_3} = 4$ and $rB_{v_3} = 6$. A successor query is applied and no vertex is found, and a predecessor query returns v_1. As $d_G(v_3,v_1) = 4$ does not exceed lB_{v_3}, v_1 is taken as the temporary result for pivot v_3. Then we continue to search $PB(v_4)$ with $lD_{v_4} = rD_{v_4} = 1$, $lB_{v_4} = 0$ and $rB_{v_4} = 2$ but no vertex is found, neither in $PB(v_6)$. Finally, we report v_1 with $d_m^3(v_1) = 0.2$.*

Lemma 5.2. *Given $G = (V,E)$ and label index $L(v)$ for all $v \in V$, the space cost of PB-tree is $O(|L| \cdot h)$.*

Proof. For each $v \in V$, we have $|L(v)|$ label entries, thus each $PB(v)$ has an index size $O(|L(v)| \cdot h)$. It is easy to see the size of PB-tree is $O(|L| \cdot h)$ where $|L|$ is the size of label index. □

Algorithm 12: Procedure *findNext()* with PB-tree

Input: Query vertex v_{i-1}, clue w_i and d_i, threshold θ

Output: Next candidate v_i with $d^i_m(v_i)$

```
1  lD ← d_i − d_i · θ;
2  rD ← d_i + d_i · θ;
3  lB ← d_i − d_i · UB;
4  rB ← d_i + d_i · UB;
5  for each pivot o ∈ L(v_{i−1}) do
6      Obtain PB(v_{i−1}), lD_o, rD_o, lB_o and rB_o;
7      if d_G(v_{i−1}, o) > rB then
8          break;
9      else
10         rD_o ← rD − d_G(v_{i−1}, o);
11         while PB(v_{i−1}).successor(rD_o, w_i) and d_G(o, v_tmp_r) ≤ rB_o do
12             Obtain v_tmp_r;
13             if d_G(v_{i−1}, v_tmp_r) ≠ d_G(v_{i−1}, o) + d_G(o, v_tmp_r) then
14                 rD_o ← d_G(o, v_tmp_r);
15             else
16                 Obtain temp suc result on PB(o);
17                 break;
18         if d_G(v_{i−1}) < lD then
19             lD_o ← lD − d_G(v_{i−1}, o);
20             while PB(v_{i−1}).predecessor(lD_o, w_i) and d_G(o, v_tmp_l) ≥ lB_o do
21                 Obtain v_tmp_l;
22                 if d_G(v_{i−1}, v_tmp_l) ≠ d_G(v_{i−1}, o) + d_G(o, v_tmp_l) then
23                     lD_o ← d_G(o, v_tmp_l);
24                 else
25                     Obtain temp pre result on PB(o);
26                     break;
27         if d_i − d_G(v_{i−1}, v_tmp_l) ≤ d_G(v_{i−1}, v_tmp_r) − d_i then
28             lB ← d_G(v_{i−1}, v_tmp_l);
29             rB ← 2 ∗ d_i − lB;
30             v_i ← v_tmp_l;
31         else
32             rB ← d_G(v_{i−1}, v_tmp_r);
33             lB ← 2 ∗ d_i − rB;
34             v_i ← v_tmp_r;
35 return v_i with d^i_m(v_i);
```

5.6 Dynamic Maintenance

In this section, we discuss how to maintain the PB-tree for road network updating. To avoid recomputing the index structure from scratch, we propose a semi-dynamic mechanism to adjust the PB-tree with a low overhead. As we know, PB-tree is built based on label index, thus the updating is divided into two phases, the updating of label index and the updating of PB-tree. Instead of recomputing a new label index, [7] introduces a dynamic label index scheme for distance queries on time-evolving graphs, and we adopt the algorithm for the first phase label index updating.

5.6.1 Semi-Dynamic Index Structure

Basically, we have 4 operations to update the network: insert a new vertex with an edge connecting to an existing vertex, delete a vertex with only one edge, insert an edge and delete an edge. As the deletion operation is much harder than insertion, and it seems impossible to find an efficient approach to support deletion in label generation. Moreover, it is rare to see deletion happens in road networks, thus we only take insertion into consideration. As the newly updated vertex is isolated, its label can be viewed as an empty set. Inserting a new vertex can be easily done by inserting an edge connecting to it, thus we only need to focus on edge insertion. As keyword updating is easy to implement, thus we omit it here.

Label index updating. Assume we insert an edge (a, b) into G, some shortest paths in old network may change by passing (a, b). Based on the label generation algorithm, we do not have to remove outdated distances in label but resume BFSs of affected vertices and add new label entries into index. It is worth to note that only the pivots in $L(a)$ and $L(b)$ are affected by network updating, and it suffices to conduct resumed BFSs originally rooted at pivot v_k if $v_k \in L(a) \cup L(b)$. Different with previous pruning method, a prefixal pruning method is proposed to apply in BFS with a new parameter k, where k is the vertex ordering of v_k. The prefixal method is to answer the distance query between v_k and u from the pivots in $L(v_k) \cap L(u)$ whose vertex orderings are at most k. Interested readers can refer to [7] for algorithm details.

Pivot-based forest. To propose a semi-dynamic index structure, we present a general framework to convert PB-tree into pivot-based forest (PF), which is inspired by the *logarithmic method* [12]. Given $PB(o)$ with m label entries, we divide it into $l = \lfloor \log m \rfloor + 1$ partitions $P_0, \ldots P_{l-1}$. Each partition

P_i either has 2^i label entries or is empty. We first compute a l-bit binary value of m. Interestingly, whether P_i is empty or not is determined by the ith bit, if ith bit is 0 then P_i is empty. For non-empty P_i, we follow the method introduced in Section 5.5.1 to construct a binary tree $PF(o)_i$ on these 2^i label entries. Finally, all these binary trees together form the pivot-based forest structure.

PF index updating. After label index updating, we add new label entries or rewrite distances of existing label entries. Assume we add a new label entry $(v, d_G(o, v))$ into $PB(o)$, we first find the smallest i such that $PF(o)_i$ is empty. If i equals to 0, we simply build $PF(o)_0$ with only one label entry $(v, d_G(o, v))$. Otherwise, we union all label entries of $PF(o)_o, \ldots, PF(o)_{i-1}$, together with $(v, d_G(o, v))$, into $PF(o)_i$. It is worth to note that $PF(o)_i$ now has 2^i elements and $PF(o)_o, \ldots, PF(o)_{i-1}$ become empty. As we know, the label entries in original $PB(o)$ are sorted in ascending order of distance. In $PF(o)$, we do not consider the global distance order but instead consider a local order in each $PF(o)_i$ when we rebuild the index. To rewrite distances of existing label entries, we only need to update the $PF(o)_i$ they belong to.

Query processing on PF index. Given query vertex v_{i-1} and a clue $\mu(w_i, d_i)$, we introduce how to answer *findNext()* on PF index. As we know, both the predecessor and successor queries are decomposable. Therefore, we simply apply the predecessor and successor queries on all non-empty $PF(o)_i$. Fortunately, it is not necessary to process queries on all $PF(o)_i$s. If the query distance is smaller than the minimum network distance stored in $PF(o)_i$, the predecessor query is not required, where the similar case holds for successor query. Finally, we merge these intermediate results to obtain the result.

5.7 Experiments

In this section, we conduct extensive experiments on real road network datasets to study the performance of the proposed index structures and algorithms.

5.7.1 Experimental Settings

All these algorithms introduced in this work were implemented in GNU C++ on Linux and run on an Intel(R) CPU i7-4770@3.4GHz and 32G RAM.

Datasets. We use two real datasets, the road network datasets of Beijing and New York City from the 9th DIMACS Implementation Challenge[1]. Each dataset contains an undirected weighted graph that represents a part of the road network. The weight of each edge in a graph represents the distance between two endpoints of the edge. We obtain the keywords of vertices from the OpenStreetMap[2]. As shown in Table 5.2, for D1 in Beijing, we have 168,535 vertices and 196,307 edges. We also have 88,910 distinct keywords contained by vertices with the total occurrence 1,445,824. For D2 in New York, we have more vertices and edges than D1 in road network with almost twice the size of D1, and the number of keywords contained is larger than D1 as well.

TABLE 5.2: Statistics of dataset

	Beijing	New York
$\#\|V\|$	168,535	264,346
$\#\|E\|$	196,307	733,846
$\#\|W\|$	88,910	102,450
$\#\|\Phi(V)\|$	1,445,824	3,086,166

Algorithms. We evaluate the performance of three algorithms, greedy clue search algorithm (GCS), clue-based dynamic programming algorithm (CDP) and branch and-bound-algorithm (BAB). In CDP, we use two different distance oracles $\mathcal{DO}$ to compute network distance, i.e., all-pair and 2-hop label. In BAB, we evaluate the performances of three index structures, i.e., AB-tree, PB-tree and PF.

TABLE 5.3: Parameter settings

Parameters	**Values**
Dataset cardinality	4K, 8K, 12K, **16K**
The number of clues	2, 3, **4**, 5, 6, 7, 8
Keyword frequency	10, 50, 100, 500, **1000**, 5000, 10000
Average distance (km)	2, 4, 6, 8, **10**, 12, 14, 16, 18, 20
h bits hash code	**64**, 128, 256, 512

Parameter settings. To evaluate the algorithms under various settings, we vary the value of some

[1]http://www.dis.uniroma1.it/challenge9/download.shtml

[2]https://www.openstreetmap.org

parameters to study the performance, as shown in Table 5.3. For default settings, we choose 16K for dataset cardinality (the number of vertices), 4 for the number of clues in query, 1000 for keyword frequency, 10km for average expected distance and 64 for hash code length.

5.7.2 Performance Evaluation

TABLE 5.4: Performance of proposed algorithms and index structures

Algorithm		QT (s)		IS (GB)		IT (min)	
		BJ	NY	BJ	NY	BJ	NY
GCS		154.55	213.54	-	-	-	-
CDP	Allpair	24.89	32.67	106.1	260.6	-	-
	Label	80.74	91.17	0.51	0.78	-	-
BAB	AB-tree	2.84	3.59	856	2104	1045	2569
	PB-tree	**1.02**	**1.57**	**2.1**	**3.21**	**2**	**3.1**
	PF	1.36	2.15	2.2	3.36	2	3.4

Table 5.4 shows the performance comparison of proposed algorithms and index structures on query time, index size and index construction time. As GCS uses a small size keyword posting list, we omit the evaluation of its size and construction time. The construction time of all-pair and 2-hop label, which have been studies by existing works, are also excluded in our performance comparison. For the query time evaluation, it is easy to see that BAB well outperforms GCS and CDP. Besides, applying all-pair in CDP has a shorter response time but a larger space cost than 2-hop label, and using PB-tree in BAB has a better performance than using AB-tree and PF. For index size and construction time, label based approaches have a much smaller size and less time than all-pair based approaches. As NY has a larger size than BJ, more time and space costs are required. For the rest experiments, we only demonstrate the performance on BJ due to the space limit, where the performance on NY is similar to that on BJ.

Effect of the keyword hash code length h. In this set of experiments, we study the effect of keyword hash code length h on performance of AB-tree, PB-tree and PF index structures. As shown in Figure 5.6, the pivot-based indices well outperform AB-tree on index construction time, index size and query time. The space of AB-tree is $O(|V|^2 \cdot h)$ and PB-tree is $O(|L| \cdot h)$. When we enlarge h, both the index

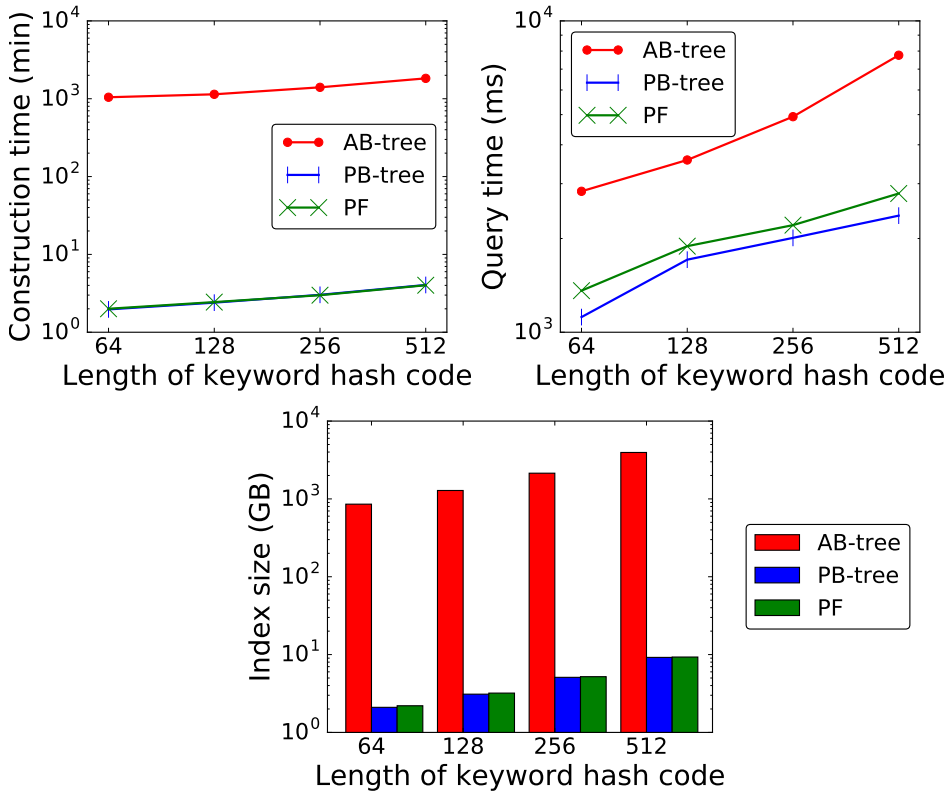


FIGURE 5.6: Effect of the keyword hash code length h

size and construction time linearly increase. For query time, there are more false positives in tree traversal when h is 64, we still have less query time since the bit operation costs less time than larger h.

Effect of the dataset cardinality. In this set of experiments, we vary the size of datasets to study the performance of proposed algorithms and index structures, as shown in Figure 5.7. Obviously, the index size and construction time increase when we enlarge the size of datasets. It is worth to note that the size of AB-tree increases exponentially with the number of vertices, and the sizes of PB-tree and PF increase gently especially when the size is enlarged from 120K to 160K due to the property of 2-hop label. For the query time, the BAB algorithm outperforms the GCS and CDP by a large margin.

Effect of the number of clues. In this set of experiments, Figure 5.8 shows the performance of algorithms by increasing the number of clues in CRS query. Not surprisingly, the response time increases when we enlarge the number of clues of all proposed algorithms. For GCS, the response time increases gently since only more rounds of network expansion are induced. For CDP, when we

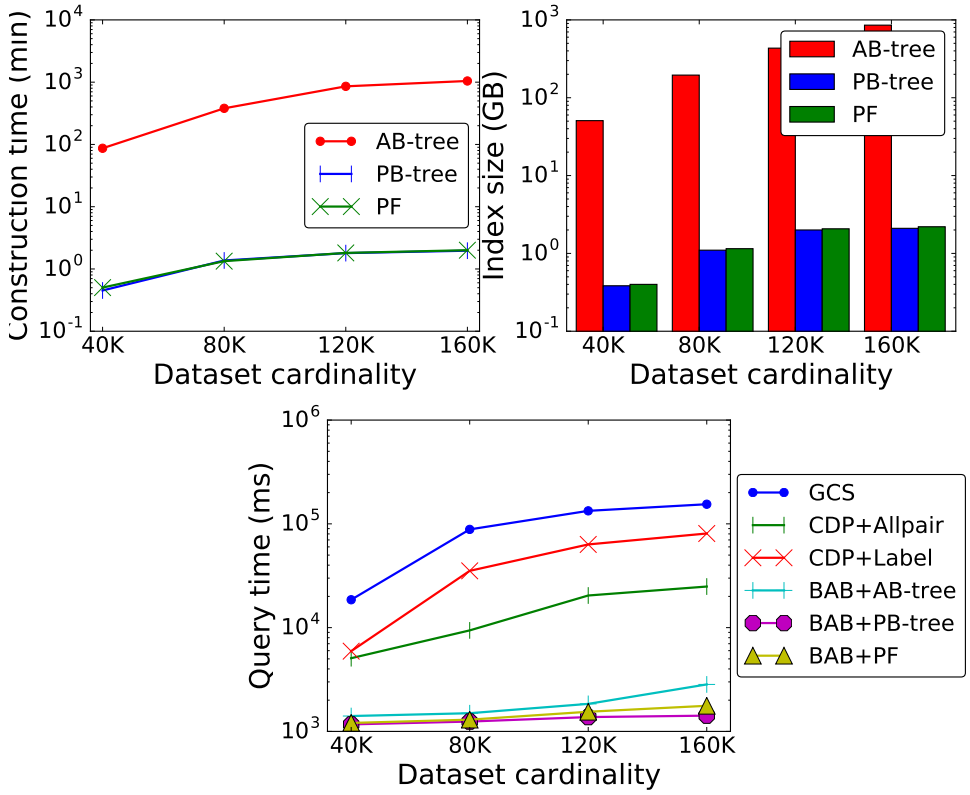


FIGURE 5.7: Effect of the dataset cardinality

enlarge the number of clues, more iterations are triggered for the computation. For BAB, the number of candidate vertices and feasible paths increase thus takes more computation time.

Effect of the average frequency of keywords. In this set of experiments, we study the performance of algorithms by varying the frequency of query keywords, as shown in Figure 5.9. It suffices to say that for low frequency keywords, say the frequency less than 500, it is more efficient if we adopt CDP with all-pair, and for high frequency keywords, BAB with PB-tree has a much better performance on both response time and index size. This is because, for CDP, there are not too many combinations to consider if the frequency is low, but when we enlarge the frequency, the response time increases exponentially to the frequency. For BAB, there are lots of false positives if the frequency is low, and when we enlarge the frequency, the performance becomes much better since we can quickly locate the candidate by using PB-tree.

Effect of the average expected distance. In this set of experiments, we study the effect of average expected distance on the performance of proposed algorithms, as shown in Figure 5.10. As we know,

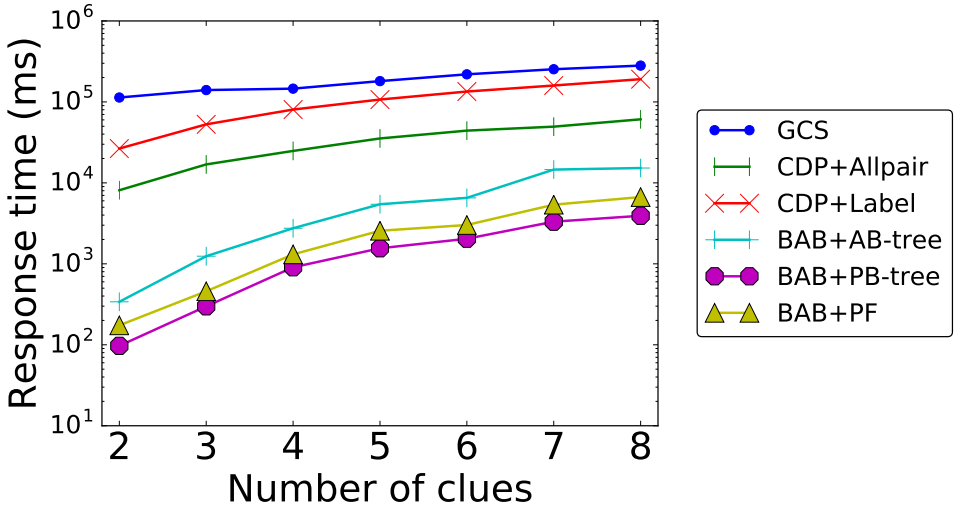


FIGURE 5.8: Effect of the number of clues

we apply the network expansion algorithm in GCS, which makes it sensitive to the expected distance. When the distance increases, more vertices are involved that results in more computation cost. For CDP with all-pair or label index, they both have a small dependency on the query distance. Therefore, the computation time of CDP keeps almost steady as the distance increases. For BAB, the effect is still not obvious but if the distance is small, we are supposed to find the next candidate more quickly since there are only a small portion of vertices after filtered by distance.

TABLE 5.5: Evaluation of index updating

Dataset	Update time	Updated pivots
Beijing	78 ms	3.6
NY	127 ms	5.7

Evaluation of index updating. Here we evaluate the cost of index updating. It is easy to observe that the average update time cost is much smaller than reconstruction the index from scratch. The cost comes from two parts, the updating of label index and updating of PF. For each update, we only have to update a very small number of pivot forest structures, that is, the semi-dynamic update is done locally.

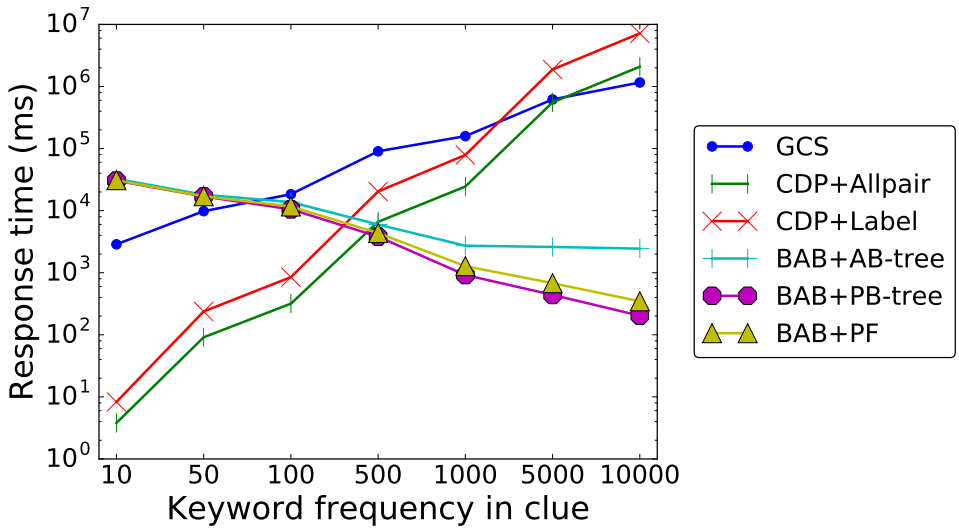


FIGURE 5.9: Effect of the average frequency of keywords

5.8 Conclusion

In this chapter, we study the problem of clue-based route search, denoted as CRS, on road networks, which aims to find an optimal route such that it covers a set of query keywords in a given specific order, and the matching distance is minimized. To answer the CRS query, we first propose a greedy clue-based algorithm GCS with no index where the network expansion approach is adopted to greedily select the current best candidates to construct feasible paths. Then, we devise an exact algorithm, namely clue-based dynamic programming CDP, to answer the query that enumerates all feasible paths and finally returns the optimal result. To further reduce the computational overhead, we propose a branch-and-bound algorithm BAB by applying filter-and-refine paradigm such that only a small portion of vertices are visited, thus improves the search efficiency. In order to quickly locate the candidate vertices, we develop AB-tree and PB-tree structures to speed up the tree traversal, as well as a semi-dynamic index updating mechanism. Results of empirical studies show that all the proposed algorithms are capable of answering CRS query efficiently, while the BAB algorithm runs much faster, and the index size of PB-tree is much smaller than AB-tree.

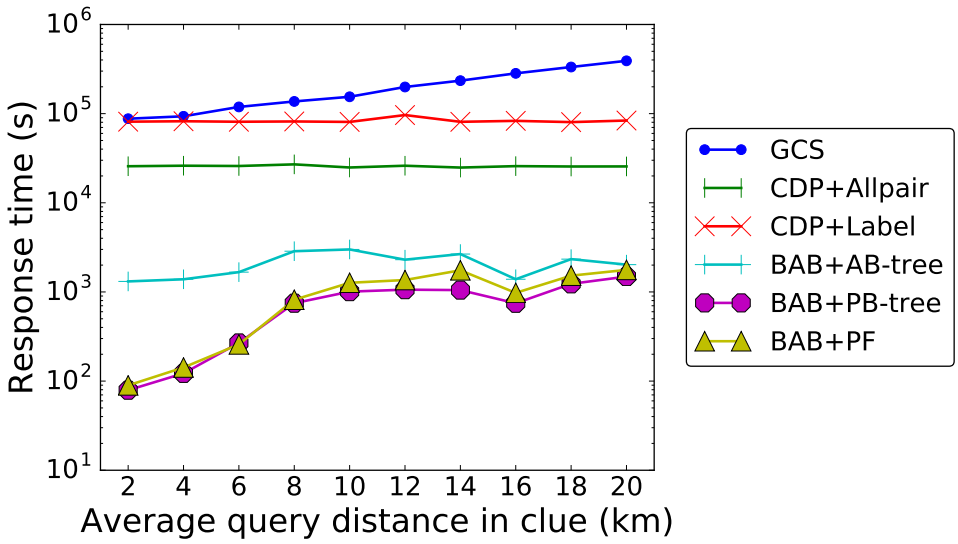


Figure 5.10: Effect of the average expected distance

Chapter 6

Final Remarks

6.1 Conclusions

In this thesis, I present several novel spatial keyword queries and efficient algorithms to manage a variety of applications on geo-textual data. Chapter 3 presents our research on keyword-oriented query on activity trajectories. I discuss our approach to process the keyword-aware continuous k nearest neighbor queries on road networks in Chapter 4. Chapter 5 describes the techniques for clue-based route search on road networks.

In Chapter 3, I study the problem of searching activity trajectory database given multiple query keywords without location. To support efficient query processing, a novel index structure, called GiKi, is developed that includes two components, i.e., AG-Tree and K-Ref, to index the activity trajectory database. Based on such index structure, an efficient algorithm are proposed to compute the minimum value of spatio-temporal ranking function. In the query processing, the pruning and refinement paradigm is applied to answer the query. Specifically, a dynamic programming algorithm is proposed to compute the lower bound for each candidate trajectory. In addition, a trajectory segmentation algorithm is developed to partition trajectories by leveraging multiple features. Then an enhanced search algorithm is proposed with such segmentation method to answer the KOAT query more efficiently. Extensive experimental results demonstrate that the proposed methods outperform baseline algorithms significantly and achieve good scalability.

In Chapter 4, I study the problem of efficiently processing $\mathbb{K}$CkNN query on road networks with

low computation and communication costs. By utilizing the 2-hop label technique on road networks, I modify the original index structure and *LARC++* construct a keyword-based label index. Based on such index, I first introduce the $\mathbb{K}k$NN query processing, and then propose two efficient algorithms *LARC* and for processing the $\mathbb{K}\mathbb{C}k$NN on road networks. For *LARC*, I introduce a *window sliding approach* to build a dominance interval to deal with low frequency keywords. For *LARC++*, I propose a *path-based dominance updating approach* to resolve a dominance region for high frequency keywords. The experimental evaluation demonstrates the effectiveness and efficiency of the solution for processing the $\mathbb{K}\mathbb{C}k$NN queries on large real-world datasets, which outperforms the state-of-the-art method with almost 50% decrease on computation cost and almost 20% decrease on communication cost.

In Chapter 5, I study the problem of clue-based route search, denoted as CRS, on road networks, which aims to find an optimal route such that it covers a set of query keywords in a given specific order, and the matching distance is minimized. To answer the CRS query, I first propose a greedy clue-based algorithm GCS with no index where the network expansion approach is adopted to greedily select the current best candidates to construct feasible paths. Then, I devise an exact algorithm, namely clue-based dynamic programming CDP, to answer the query that enumerates all feasible paths and finally returns the optimal result. To further reduce the computational overhead, I propose a branch-and-bound algorithm BAB by applying filter-and-refine paradigm such that only a small portion of vertices are visited, thus improves the search efficiency. In order to quickly locate the candidate vertices, I develop AB-tree and PB-tree structures to speed up the tree traversal, as well as a semi-dynamic index updating mechanism. Results of empirical studies show that all the proposed algorithms are capable of answering CRS query efficiently, while the BAB algorithm runs much faster, and the index size of PB-tree is much smaller than AB-tree.

6.2 Directions for Future Work

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

6.2.1 Answering Why-not Spatial Keyword Queries on Road Networks

In Chapter 4, I present efficient techniques for keyword-aware queries on road networks. Under such setting, it may happen that the desirable geo-textual objects are unexpectedly missing from the result. Then users may wonder why they are missing, and it is difficult for users to ensure that the parameters of the score function are properly configured. Therefore, answering such why-not queries is of high interest, and it is able to provide explanations on why the desired geo-textual objects are not included in the result as well as the suggestions on how to revise the query so that these desired objects can be re-included in the result. The main challenge of this future work is that we need to avoid accessing the whole database and computing the scores at runtime, since it is very time-consuming.

6.2.2 An In-memory Implementation of Spatial Keyword Queries

Large volume of geo-textual data may fill up the main-memory of a single machine in a short time, and it is also not desirable to store all the geo-textual data on hard disk. Moreover, when processing the batch spatial keyword queries, the degree of parallelism of a single machine is constrained by its limited memory and concurrency in particular. Therefore, it is necessary to implement a distributed in-memory version of spatial keyword query processing algorithms. Apache Spark is a Hadopp-like distributed framework with in-memory computing technology, it is a good potential platform for us to implement the spatial keyword queries. There are two main challenges we have to consider in this future work. The first is how to design an algorithm for processing batch spatial keyword queries to load-balance the computation of nodes in the cluster. The second is to minimize the the communication costs between nodes, since the data transmission through network is far slower than in-memory.

6.2.3 Spatial Keyword Search by Incorporating Social Influence

The spatial keyword search is a fundamental problem in location based services, which combines both the spatial and textual information to rank the objects. Along with the popular usage of location based services, the social influence from the user's friends may also highly affect the user to make decision. Therefore, if a user issues a spatial keyword query, it is not a surprise that he considers not only the results returned by the query, but also his friends' suggestions. Based on such motivation,

it is interesting to model an influence social network together with geo-textual objects and propose a novel query predicate to study this topic.

References

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, pages 230–241. Springer, 2011.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35. Springer, 2012.

[3] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 782–793. Society for Industrial and Applied Mathematics, 2010.

[4] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *International Conference on Foundations of Data Organization and Algorithms*, pages 69–84. Springer, 1993.

[5] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *ALENEX*, pages 147–154. SIAM, 2014.

[6] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360. ACM, 2013.

[7] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*, pages 237–248. ACM, 2014.

[8] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, pages 918–929, 2006.

[9] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *arXiv preprint arXiv:1504.05140*, 2015.

[10] H. Bast, S. Funke, and D. Matijević. Transit: ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge—Shortest Path*, 2006.

[11] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 46–59. Society for Industrial and Applied Mathematics, 2007.

[12] J. L. Bentley and J. B. Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[13] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander. PIST: an efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica*, 12(2):143–168, 2008.

[14] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *SIGMOD*, 2004.

[15] X. Cao, L. Chen, G. Cong, C. S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M. L. Yiu. Spatial keyword querying. In *International Conference on Conceptual Modeling*, pages 16–29. Springer, 2012.

[16] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware optimal route search. *PVLDB*, 5(11):1136–1147, 2012.

[17] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 2010.

[18] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, 2011.

[19] V. P. Chakka, A. C. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with SETI. In *CIDR*, 2003.

[20] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–5, 2006.

[21] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA*, pages 30–39. SIAM, 2004.

[22] H. Chen, W.-S. Ku, M.-T. Sun, and R. Zimmermann. The multi-rule partial sequenced route query. In *SIGSPATIAL*, page 10. ACM, 2008.

[23] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. *PVLDB*, 2013.

[24] L. Chen and R. Ng. On the marriage of lp-norms and edit distance. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 792–803. VLDB Endowment, 2004.

[25] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM, 2005.

[26] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 591–602. ACM, 2009.

[27] K. L. Cheung and A. W.-C. Fu. Enhanced nearest neighbour search on the r-tree. *ACM SIGMOD Record*, 27(3):16–21, 1998.

[28] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to cnn queries in a road network. In *VLDB*, pages 865–876. VLDB Endowment, 2005.

[29] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document, 1976.

[30] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2009.

[31] G. Cong, H. Lu, B. C. Ooi, D. Zhang, and M. Zhang. Efficient spatial keyword search in trajectory databases. *arXiv preprint arXiv:1205.2880*, 2012.

[32] P. Cudre-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.

[33] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.

[34] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[35] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614 – 656, 2003.

[36] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. *Fast subsequence matching in time-series databases*, volume 23. ACM, 1994.

[37] Y. Gao, X. Qin, B. Zheng, and G. Chen. Efficient reverse top-k boolean spatial keyword queries on road networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(5):1205–1218, 2015.

[38] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[39] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[40] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, et al. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.

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

[42] T. Guo, X. Cao, and G. Cong. Efficient algorithms for answering the m-closest keywords query. In *SIGMOD*, pages 405–418. ACM, 2015.

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

[44] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.

[45] M. Hadjieleftheriou and D. Srivastava. Weighted set-based string similarity. *IEEE Data Eng. Bull.*, 33(1):25–36, 2010.

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

[47] H. Hu, D. L. Lee, and J. Xu. Fast nearest neighbor search on road networks. In *International Conference on Extending Database Technology*, pages 186–203. Springer, 2006.

[48] W. Huang, G. Li, K.-L. Tan, and J. Feng. Efficient safe-region construction for moving top-k spatial keyword queries. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 932–941. ACM, 2012.

[49] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

[50] H. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM TODS*, 30(2):364–397, 2005.

[51] C. S. Jensen, J. Kolářvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *GIS*, pages 1–8. ACM, 2003.

[52] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *VLDBJ*, 2008.

[53] M. Jiang, A. W.-C. Fu, and R. C.-W. Wong. Exact top-k nearest keyword search in large networks. In *SIGMOD*, pages 393–404. ACM, 2015.

[54] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB*, 7(12):1203–1214, 2014.

[55] Y. Kanza, R. Levin, E. Safra, and Y. Sagiv. Interactive route search in the presence of order constraints. *PVLDB*, 3(1-2):117–128, 2010.

[56] Y. Kanza, E. Safra, Y. Sagiv, and Y. Doytsher. Heuristic algorithms for route-search queries over geographical data. In *SIGSPATIAL*, page 11. ACM, 2008.

[57] R. M. Karp. *Reducibility among combinatorial problems*. Springer, 1972.

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

[59] K. C. Lee, W.-C. Lee, and B. Zheng. Fast object search on road networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 1018–1029. ACM, 2009.

[60] K. C. Lee, W.-C. Lee, B. Zheng, and Y. Tian. Road: A new spatial object search framework for road networks. *IEEE transactions on knowledge and data engineering*, 24(3):547–560, 2012.

[61] T. Lee, J.-w. Park, S. Lee, S.-w. Hwang, S. Elnikety, and Y. He. Processing and optimizing main memory spatial-keyword queries. *Proceedings of the VLDB Endowment*, 9(3):132–143, 2015.

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

[63] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290. Springer, 2005.

[64] G. Li, J. Feng, and J. Xu. Desks: Direction-aware spatial keyword search. In *ICDE*, 2012.

[65] J. Li, Y. D. Yang, and N. Mamoulis. Optimal route queries with arbitrary order constraints. *TKDE*, 25(5):1097–1110, 2013.

[66] Y. Li, G. Li, L. Shu, Q. Huang, and H. Jiang. Continuous monitoring of top-k spatial keyword queries in road networks. *J. Inf. Sci. Eng.*, 31(6):1831–1848, 2015.

[67] H. Ling and K. Okada. An efficient earth mover's distance algorithm for robust histogram comparison. *IEEE Transactions on PAMI*, 29(5):840–853, 2007.

[68] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine learning*, 2(4):285–318, 1988.

[69] S. Luo, Y. Luo, S. Zhou, G. Cong, J. Guan, and Z. Yong. Distributed spatial keyword querying on road networks. In *EDBT*, pages 235–246. Citeseer, 2014.

[70] S. Ma, K. Feng, J. Li, H. Wang, G. Cong, and J. Huai. Proxies for shortest path and distance queries. *IEEE Transactions on Knowledge and Data Engineering*, 28(7):1835–1850, 2016.

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

[72] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *PVLDB*, pages 43–54. VLDB Endowment, 2006.

[73] G. Navarro. A guided tour to approximate string matching. *CSUR*, 2001.

[74] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB*, 1(1):1095–1106, 2008.

[75] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial tessellations: concepts and applications of Voronoi diagrams*, volume 501. John Wiley & Sons, 2009.

[76] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 802–813. VLDB Endowment, 2003.

[77] O. Pele and M. Werman. A linear time histogram metric for improved sift matching. In *ECCV*, pages 495–508. 2008.

[78] D. Pfoser, C. S. Jensen, Y. Theodoridis, et al. Novel approaches to the indexing of moving object trajectories. In *VLDB*, 2000.

[79] I. Pohl. *Bi-directional and heuristic search in path problems*. PhD thesis, Dept. of Computer Science, Stanford University., 1969.

[80] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k nearest keyword search on large graphs. *PVLDB*, 6(10):901–912, 2013.

[81] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento. A trajectory splitting model for efficient spatio-temporal indexing. In *Proceedings of VLDB*, pages 934–945, 2005.

[82] M. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *Proceedings of the VLDB Endowment*, 4(2):69–80, 2010.

[83] J. B. Rocha-Junior and K. Nørvåg. Top-k spatial keyword queries on road networks. In *Proceedings of the 15th international conference on extending database technology*, pages 168–179. ACM, 2012.

[84] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *ACM sigmod record*, volume 24, pages 71–79. ACM, 1995.

[85] Y. Rubner, C. Tomasi, and L. Guibas. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

[86] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *Sixth International Conference on Computer Vision, 1998.*, pages 59–66, 1998.

[87] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 43–54. ACM, 2008.

[88] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms*, pages 568–579. Springer, 2005.

[89] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *Proceedings of the 13th annual ACM international workshop on Geographic information systems*, pages 200–209. ACM, 2005.

[90] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *IEEE Transactions on Knowledge and Data Engineering*, 22(8):1158–1175, 2010.

[91] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

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

[93] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDBJ*, 17(4):765–787, 2008.

[94] S. Shekhar and J. S. Yoo. Processing in-route nearest neighbor queries: a comparison of alternative approaches. In *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, pages 9–16. ACM, 2003.

[95] C. S. J. Simonas Saltenis, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.

[96] A. Singhal. Modern information retrieval: a brief overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24, 2001.

[97] M. A. Soliman, I. F. Ilyas, and K. Chen-Chuan Chang. Top-k query processing in uncertain databases. In *ICDE*, pages 896–905, 2007.

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

[99] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In *PVLDB*, pages 790–801, 2003.

[100] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 673–684. IEEE, 2002.

[101] J. Wang, K. Zheng, H. Jeung, H. Wang, B. Zheng, and X. Zhou. Cost-efficient spatial network partitioning for distance-based query processing. In *2014 IEEE 15th International Conference on Mobile Data Management*, volume 1, pages 13–22. IEEE, 2014.

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

[103] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *Proceedings of the VLDB Endowment*, 5(5):406–417, 2012.

[104] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *Proc. VLDB Endow.*, pages 406–417, 2012.

[105] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[106] B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou. Approximate string search in spatial databases. In *ICDE*, 2010.

[107] B. Yao, M. Tang, and F. Li. Multi-approximate-keyword routing in gis data. In *SIGSPATIAL*, pages 201–210. ACM, 2011.

[108] B.-K. Yi, H. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 201–208. IEEE, 1998.

[109] M. L. Yiu, N. Mamoulis, and D. Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):820–833, 2005.

[110] C. Yu, B. C. Ooi, K.-L. Tan, and H. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, volume 1, pages 421–430, 2001.

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

[112] J. Yuan, Y. Zheng, X. Xie, and G. Sun. T-drive: Enhancing driving directions with taxi drivers' intelligence. *IEEE Transactions on Knowledge and Data Engineering*, 25(1):220–232, 2013.

[113] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*, pages 99–108. ACM, 2010.

[114] Y. Zeng, X. Chen, X. Cao, S. Qin, M. Cavazza, and Y. Xiang. Optimal route search with the coverage of users preferences. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 2118–2124, 2015.

[115] C. Zhang, H. Liang, K. Wang, and J. Sun. Personalized trip recommendation with poi availability and uncertain traveling time. In *CIKM*, pages 911–920. ACM, 2015.

[116] C. Zhang, Y. Zhang, W. Zhang, and X. Lin. Inverted linear quadtree: Efficient top k spatial keyword search. *IEEE Transactions on Knowledge and Data Engineering*, 28(7):1706–1721, 2016.

[117] C. Zhang, Y. Zhang, W. Zhang, X. Lin, M. A. Cheema, and X. Wang. Diversified spatial keyword search on road networks. In *EDBT*, pages 367–378, 2014.

[118] D. Zhang, Y. M. Chee, A. Mondal, A. K. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699. IEEE, 2009.

[119] D. Zhang, B. C. Ooi, and A. K. Tung. Locating mapped resources in web 2.0. In *ICDE*, pages 521–532. IEEE, 2010.

[120] H. J. Zhao, M. L. Yiu, Y. Li, Z. Gong, et al. Towards online shortest path computation. *IEEE Transactions on Knowledge and Data Engineering*, 26(4):1012–1025, 2014.

[121] B. Zheng, N. J. Yuan, K. Zheng, X. Xie, S. Sadiq, and X. Zhou. Approximate keyword search in semantic trajectory database. In *ICDE*, pages 975–986. IEEE, 2015.

[122] B. Zheng, K. Zheng, M. Sharaf, X. Zhou, and S. Sadiq. Efficient retrieval of top-k most similar users from travel smart card data. In *MDM*, 2014.

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

[124] K. Zheng, S. Shang, N. J. Yuan, and Y. Yang. Towards efficient search for activity trajectories. In *ICDE*, 2013.

[125] K. Zheng, H. Su, B. Zheng, S. Shang, J. Xu, J. Liu, and X. Zhou. Interactive top-k spatial keyword queries. In *ICDE*, pages 423–434. IEEE, 2015.

[126] K. Zheng, G. Trajcevski, X. Zhou, and P. Scheuermann. Probabilistic range queries for uncertain trajectories on road networks. In *EDBT*, 2011.

[127] K. Zheng, B. Zheng, J. Xu, G. Liu, A. Liu, and Z. Li. Popularity-aware spatial keyword search on activity trajectories. *World Wide Web*, pages 1–25, 2016.

[128] K. Zheng, Y. Zheng, X. Xie, and X. Zhou. Reducing uncertainty of low-sampling-rate trajectories. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1144–1155. IEEE, 2012.

[129] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang. On discovery of gathering patterns from trajectories. In *ICDE*, 2013.

[130] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proceedings of the 18th international conference on World wide web*, pages 791–800. ACM, 2009.

[131] R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An efficient index for knn search on road networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 39–48. ACM, 2013.

[132] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(8):2175–2189, 2015.

[133] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 857–868. ACM, 2013.

[134] G. K. Zipf. Human behavior and the principle of least effort. 1949.