THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

EFFICIENT DISTANCE-BASED QUERY PROCESSING IN SPATIAL NETWORKS

Jiping Wang

Bachelor of Science

A thesis submitted for the degree of Master of Philosophy at

The University of Queensland in 2014

School of Information Technology & Electrical Engineering

Abstract

The prevalence of GPS-enabled mobile devices and wireless communication has given rise to a wide range of location-based services (e.g. Foursquare, Google Maps). These services support various types of queries related to the objects' spatial locations. For example, a user may want to find the five restaurants nearest to his location while visiting an unfamiliar city. Since these services provide real-time response to people's queries, the efficiency of query processing is crucial for them.

In real-world applications, the movements of the objects are often constrained to underlying paths such as spatial networks. In these circumstances, the distance between two locations is measured by the length of the shortest path between them. As shortest path distance is the metric for most query types in spatial networks, their query processing is all network distance based. This makes the query processing in spatial networks quite different from that in the Euclidean space. Thus the traditional algorithms designed for the Euclidean space are not directly applicable. This thesis aims at supporting efficient query processing in spatial networks. Specifically, it focuses on two research problems.

The first work in the thesis studies the efficient object query processing in spatial networks. A novel graph partitioning based index, the Partition Tree, is proposed. It takes account of both the network topologies and the object distribution. Based on the Partition Tree, efficient algorithms are proposed for common types of queries in spatial networks including shortest path query and k-NN query. Furthermore, a cost model is derived to balance the cost of indexing and query efficiency.

The second work in the thesis studies the efficient trajectory query processing in spatial networks. A trajectory is a record of moving history of an object and the trajectory dataset contains rich information of specific moving patterns. To support efficient trajectory query processing, the Partition Tree is modified to index trajectories in this work. Then correspondent algorithms are proposed for trajectory queries in spatial networks.

Comprehensive experiments are conducted to verify the performance of the proposed

approaches. The experimental results demonstrated that the proposed methods in this thesis have superior performance over the existing works.

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 General Award Rules of The University of Queensland, immediately made available for research and study in accordance with the Copyright Act 1968.

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

Conference papers:

- J. Wang, K. Zheng, H. Jeung, H. Wang, B. Zheng and X. Zhou. Cost-efficient Spatial Network Partitioning for Distance-based Query Processing. *MDM*, 2014.

- H. Wang, K. Zheng, H. Su, J. Wang, S. Sadiq and X. Zhou. Efficient Aggregate Farthest Neighbor Query Processing on Road Networks. *ADC*, 2014.

Publications included in this thesis

J. Wang, K. Zheng, H. Jeung, H. Wang, B. Zheng and X. Zhou. Cost-efficient Spatial Network Partitioning for Distance-based Query Processing. *MDM*, 2014. - incorporated as Chapter 3.

Contributor	Statement of contribution
Jiping Wang (Candidate)	Designed algorithms Designed experiments (80%) Wrote the paper (70%)
Kai Zheng	Designed experiments (10%) Wrote the paper (10%) Discussion and analysis of the algorithm design
Hoyoung Jeung	Designed experiments (10%) Wrote the paper (10%) Discussion and analysis of the algorithm design
Xiaofang Zhou	Wrote the paper (10%) Discussion and analysis of the algorithm design
Haozhou Wang	Proof reading for the paper
Bolong Zheng	Proof reading for the paper

Contributions by others to the thesis

For all the research work included in this thesis, Prof. Xiaofang Zhou, as my principle advisor, has provided very helpful insight in the overall ideas. My associate advisor, Kevin Zheng, has also assisted with both the refinement of the idea and the technical details.

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

None.

Acknowledgments

This thesis represents a major portion of the research undertaken during my MPhil candidature at The University of Queensland. During these splendid two years, I have received great support from my family, friends and the fellows at DKE.

First I would like to thank Prof. Xiaofang Zhou, my principal advisor, who offered valuable guidance on research directions throughout my whole candidature. His insightful comments inspired me to pursue better solution in each research problem. I would also like to thank my associate advisor, Dr. Kevin Zheng, for his precious guidance and patient discussion in my research problems. Dr. Hoyoung Jeung also gave me a lot helpful comments and suggestions during the work of my first paper, I really appreciate his help and kindness.

My deepest thanks go to my fellows and friends at DKE as well. They have helped me so much and we shared so many valuable moments.

Last but not least, I own my best gratitude to my family. Their support is always the power driving me forward, and their love is the most precious treasure in my life.

Keywords

spatial networks, query processing, efficiency, indexing, trajectory

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

1.1 Motivation

An important way to picture the real world is to investigate its spatial and temporal attributes. Since spatio-temporal databases were first designed in 1990's, the modeling and query of entities with their spatio-temporal information has always been of great interest to the research community. Meanwhile, wireless communication technology and location-aware devices are becoming indispensable in people's lives. One example is the proliferation of global positioning system and mobile phones. A recent survey suggests that 91 percent of people on earth own a mobile phone till 2013. The development of spatio-temporal database and rapid growth of location-awareness has given rise to a wide range of real-world applications that model and analyze temporal-spatial information associated with objects in the dataset. These applications cover the needs of government, business and academic parties. Representative areas of those applications include:

- **Urban Planning:** Each sensor, device, person, vehicle, building and street in the urban areas is used as a component to sense the city dynamics to enable a city-wide computing to tackle the challenges in urban areas so as to better serve the residents. For example, as shown in Figure 1.1, human mobility history and points of interest

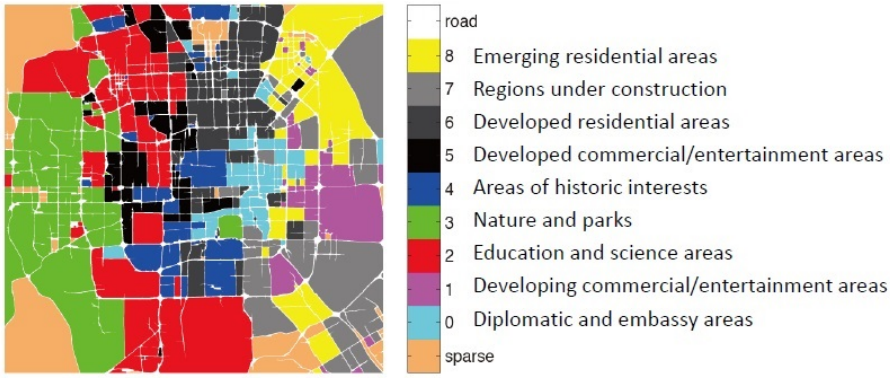


FIGURE 1.1: Discovering Regions of Different Functions

(POIs) can be used to discover regions of different functions [58].

- **Location-based Services :** Location-based services provide value-added information by considering the locations of the mobile users in order to give them customized information [20]. Examples include listing and rating nearby restaurants, and identifying friends' locations on social media applications. Figure 1.2 gives a list of some most widely used location-based services.

- **Smart Route Recommendation Systems:** Smart route recommendation systems provide smart route recommendation by analyzing heterogeneous data (e.g. spatial and textual data). For example, route recommendations are provided to users by considering the current traffic status and local history trajectories at the same time.

- **Group Organization:** Clustering of moving objects identifies groups of moving objects that travel together for a period of time. For example, finding the cars that follow the same routes at the same time may be used for the organization of carpooling.

FIGURE 1.2: Applications with Location-based Services

The basic task of these applications is to address various spatial queries. Spatial queries search the objects in the dataset by a set of spatial attributes. For example, a user may invoke a nearest neighbor query to find the nearest restaurant when traveling in an unfamiliar city. There are various types of spatial queries, of which the most representative ones are:

- **k-NN Query:** Given a query location q and a set of objects O, find the k objects from O that are nearest to q.
- **Range Query:** Given a query location q and a set of objects O, find the spatial objects of which the distances to q are within a specified range r.
- **Spatial Join Query:** Given two sets of spatial objects, find the pairs of objects which satisfy a given predicate.

In many applications, the extent of the objects can be neglected so that each object is seen as a point in the dataset. Therefore, the objects referred to in this thesis are all point objects if not declared specifically. In addition, another important type of spatial

FIGURE 1.3: GeoLife GPS Trajectories

data is trajectory. A trajectory is a series of locations recording the movement history of an object. For example, Figure 1.3 shows the trajectories collected from the movements of 182 users in Beijing during a period of over three years [65][64]. Trajectories convey rich information to reason about the nature of the real world. For example, trajectories generated by taxis of a city can be used to model the road conditions and the preference of local drivers of this city. This makes trajectory query become another important type of spatial queries. A trajectory query often retrieves the trajectories that satisfy a given predicate. Representative types of trajectory queries [66] include:

- **P-Query:** Given a set of query locations Q, find the trajectories that satisfy the specified spatio-temporal relationship to these query locations.
- **R-Query:** Given a spatio-temporal region R, find the trajectories passing by region R.
- **T-Query:** Given a query trajectory T, find the trajectories that their distances to T are within a threshold.

In many real-world applications, a vast amount of data need to be processed to address these spatial queries. This massive amount of data came with the prevailing of GPS, wireless communication enabled mobile devices and location-based services. For example, there are 4 million of check-ins generated per day. However, the online nature of these applications calls for real-time response to the users' queries anytime, and there are often a great number of queries invoked at the same time. Thus the efficiency of spatial query processing is crucial for them.

Traditional approaches for spatial query processing utilize multi-dimensional indexing structures such as *R-tree* [17] and *Grid File* [35]. With these indexing structures, correspondent query algorithms can access the spatial objects and prune search space efficiently. For example, some algorithms [43][19] adopt best-first or depth-first search with the help of R-tree to retrieve the objects and prune the search space.

However these approaches are based on metrics in Euclidean space. In real world applications, the movements of objects are often constrained to pre-defined paths that are specified by underlying spatial networks (e.g. road networks). In this case, the distance between two objects is decided by the shortest path between them, which makes the query processing in spatial network quite different from that in Euclidean space. The shortest path distance needs to be obtained through processing, while the Euclidean distance can be obtained directly from the coordinates. Thus the cost of distance computation in spatial network is much more expensive. For example, the classic solution to shortest path query is Dijkstra's algorithm [14]. It traverses the vertices of the network incrementally from start point until reaching the target point and return the shortest path (distance). The computation complexity of Dijkstra's algorithm is $O(m + n \log n)$, where n is the number of vertices and m is the number of edges. This is obviously not efficient enough for large networks, since the location-based services need to deal with a large number of queries at the same time and all queries need to be responded in real time.

Because of the deficiency of online network expansion approaches like Dijkstra's algorithm, alternative approaches have been proposed to process queries in spatial networks. These approaches typically conduct pre-computation on all pairs of vertices or a selected set of vertices. Then they utilize these pre-computed information in their algorithms to improve the query efficiency. Some of them are quite efficient but they also bring huge cost of pre-processing time and indexing space. For example, the space cost of SILC, one of the most well-known approaches for the shortest path queries, is over 24 GB for a network with 1 million vertices.

Research Problem Motivated by this, this thesis aims at finding an indexing technique for efficient query processing in spatial networks. This indexing should support efficient processing of most common queries, such as shortest path queries , k-NN queries and trajectories queries by locations. Meanwhile, the overhead of this indexing needs to be moderate so that it is practical for large scale networks. Therefore the ultimate goal of the work in the thesis is to find a good balance between indexing cost and query efficiency.

1.2 Key Challenges

In real world applications, there are many different types of spatial queries to be addressed. The most fundamental category of queries is to retrieve objects which are seen as points in the search space. This is based on the assumption that the extent of an object is insignificant compared with the scale of query space. In addition to point objects, another important category of queries is to retrieve trajectories. A trajectory is a series of sample points recording the moving history of an object. Compared with point objects, the metric definition related to trajectories is more complicated. These two categories of queries cover the most important spatial queries, so this thesis will focus on their processing:

- **Object Query Processing:** The entities to be indexed and queried are single objects which are modeled as points in the spatial networks. The distance between two objects is defined as the shortest path distance between them. So the most fundamental type of query is shortest path (distance) query. Another important type of spatial query is k-NN query, which returns the k objects that are nearest the given query location.

- **Trajectory Query Processing:** The entities to be indexed and queried are trajectories. Each trajectory is a series of points recording the movements of an object. Given a set of query locations, location-based trajectory queries return the trajectories that are nearest to the given query locations.

For each type of query mentioned above, efficient query processing algorithm need to be designed. To achieve this goal, the following challenges are identified:

- **Avoid large scale network expansion:** Shortest path (distance) computation is the basis of query processing in spatial networks. The existing works are either based on computing network distances on-line, or utilizing the index structures. On-line distance computation usually adopts Dijkstra's algorithm. It retrieves the objects in ascending order of their distances to the query location. But this performs poorly when the objects are not densely distributed in the network because a large portion of the network will be traversed. The algorithms based on indexing structures can filter out a candidate set first during search. But the distance computation between the query location and candidates still need to traverse the network if no alternative solution is provided. So the first challenge is to avoid large scale network expansion. This will reduce the cost of traversing the vertices caused by distance computation.

- **Prune search space efficiently:** Effective pruning of search space could help avoid

unnecessary computing and reduce the cost of spatial query processing. In Euclidean space, traditional algorithms utilize indexing structures like R-tree to retrieve objects and prune the search space. For example, the best-first algorithm [43] and depth-first algorithm [19] for k-NN query are both based on R-tree. In spatial networks, the metrics for such pruning technique are not valid anymore. Thus our proposed indexing structure should support efficient search space pruning for query processing as well.

- **Control indexing cost:** For real world applications, the efficiency of query processing is crucial for their service quality but a moderate indexing cost is also very important. Compared with on-line network expansion, the approaches based on indexing structures like SILC [46][44] are quite efficient but they have a huge space cost for indexing. Thus a good balance between indexing cost and query efficiency needs to be reached . Specifically, what materialization strategy to take and how to organize the materialized information is important.

1.3 Contributions

This thesis focuses on the processing of two categories of spatial queries, object query processing and trajectory query processing. For each task, the proper indexing structure and efficient algorithms for the query processing are investigated. Then extensive experimental analysis is done and the performance is compared with state-of-the-art approaches to verify the superiority of our proposed approach.

1.3.1 Object Query Processing in Spatial Networks

The first contribution of this thesis is that it developed an indexing and query processing technique for efficient object search in spatial networks. Inspired by the observation that certain vertices are more important for query processing, the vertices of the network are

organized into a hierarchy through a series of graph partitioning processes. For each subgraph, it pre-computes the distances necessary for the query processing rather than all-pairs shortest paths for vertices of this subgraph. Then algorithms utilizing the partitioning topology and pre-computed information are proposed for shortest path queries and k-NN queries.

In real-life applications, the query probabilities are often related to the distribution of objects (points of interest), since queries are often invoked around these objects. So the areas that contain more objects are more worthy to be partitioned. This motivated us to find a partitioning strategy efficient for query processing with a specific object set. Since the efficiency of our query processing is influenced by both partitioning topology and object distribution, a cost model is proposed to estimate these influences. Then a cost-efficient graph partitioning is achieved by incorporating the evaluation of cost model into the index construction.

To sum up, the following contributions are made in this part of work:

- We propose a hierarchical graph partitioning based index, the Partition Tree. It organizes the vertices of a spatial network into a hierarchy through a series of graph partitioning processes and associates pre-computed information with it to facilitate efficient query processing.

- Based on the Partition Tree, a dynamic programming algorithm is proposed for shortest path queries and a best-first algorithm for k-NN queries. These algorithms utilize the Partition Tree to avoid extensive network expansion and process queries efficiently.

- We propose a query-oriented optimization on top of the Partition Tree. This optimization uses a cost model to evaluate the influence of partitioning strategy on query efficiency. Then it incorporates this cost model in index construction to achieve a partitioning efficient for the query processing with a specific object set.

1.3.2 Trajectory Query Processing in Spatial Networks

The second contribution of this thesis is that it extends the Partition Tree to support efficient trajectory query processing in spatial networks. It adopts the framework of the indexing for object queries in the first work and adapt it to index trajectories. The following contributions are made in this part of work:

- We propose an indexing method, the Partition Tree, for efficient trajectory query processing in spatial networks. It organizes the vertices of a spatial network into a hierarchy through a series of graph partitioning processes and associates precomputed distances and trajectory information with it to facilitate efficient query processing.
- Based on Partition Tree, we propose efficient algorithms for trajectory distance computation and k nearest trajectories query with a query location.
- We propose an incremental k nearest trajectory algorithm to retrieve the k nearest trajectories with multiple query locations.

1.4 Thesis organization

The remainder of this thesis is organized as follows.

Chapter 2 introduces state-of-the-art techniques for spatial query processing, including approaches for shortest path queries, object queries and trajectory queries.

Chapter 3 presents our indexing method, the Partition Tree, and query algorithms for object search in spatial networks.

Chapter 4 extends our proposed index to support efficient trajectory query processing.

Chapter 5 gives concluding remarks.

Chapter 2

Literature Review

The prevalence of GPS-enabled mobile devices and wireless communication gave rise to a wide range of location-based services (e.g. Foursquare, Google Maps). These services require efficient processing of spatial queries such as k nearest neighbors queries and range queries. In many applications, the movements of objects are constrained to spatial networks (e.g. road networks). In this case, the distance between two objects is decided by the shortest path distance rather than the Euclidean distance. So the query processing in spatial networks requires different approaches from those in Euclidean space.

The spatial query processing in Euclidean space has been extensively studied [43][60] [57][61]. As a counter part, the query processing in spatial networks has also been investigated in many works before.

The most important and fundamental type of query in spatial networks is shortest path (distance) query. A plethora of techniques [55][18][46][49][24][26][16][6][40] have been proposed to address it in past few decades, and some state-of-the-art approaches have achieved great improvement over the traditional network expansion based algorithm.

Another important type of query is k nearest neighbors query. We will introduce some state-of-the-art approaches [21][38][38][13][36][29][31] [32] [44].

Then we will investigate trajectory queries, especially the queries based on a given set

of query locations [53] [4] [15] [52] [11] [50] [62].

Next we will present the existing works for each research issue, categorizing the main approaches together with important research results in literatures.

2.1 Shortest Path Query Processing

Dijkstra's Algorithm

The most classic solution for shortest path and distance query is Dijkstra's algorithm, it is first proposed by Edsger Dijkstra [14] [37]. It solves single-source shortest path query for a graph with non-negative edge weights. Given a source vertex s, to compute the shortest path distance between any vertex t to it, Dijkstra's algorithm traverses the vertices of G in ascending order of their distances from s until reaching target point t. Then the shortest path from s to t is computed and returned. The traversing can be seen as a process to produce a shortest path tree, in which nodes are added in order of their distances to source vertex s. The computation complexity for Dijkstra's algorithm is $O(m + n \log n)$, where n is the number of vertices in graph, and m is the number of edges. For sparse graph, it can be approximated as $O(n \log n)$.

Dijkstra's algorithm is simple and elegant. It works well for small scale networks but is quite inefficient for large networks. For example, for a large network with a huge number of vertices, Dijkstra's algorithm will traverse a large portion of the network when two points are far apart from each other, which is a considerate computation cost.

A variant of Dijkstra's algorithm is A* algorithm. The difference is that, while choosing the next vertex to visit, it considers not only the vertices' distances to s, but also their expected distances to t, which are estimated by a heuristic function. Therefore it combines the pieces of information that Dijkstras algorithm uses and the information that Greedy Best-First-Search uses. Compared with Dijkstra's algorithm, A* algorithm has a better performance. But it is still not efficient enough.

To overcome the deficiency of Dijkstra's algorithm, various alternative approaches have been proposed. These approaches pre-compute partial or all-pairs shortest paths for vertices of the networks and utilize these pre-computed information to facilitate efficient query processing. We categorize these approaches into three groups.

1. Vertex Importance based Indexing

   The approaches in this group are based on the assumption that certain vertices in the network are more important for shortest path query processing. So rather than pre-computing and maintaining shortest path distances for all pairs of vertices, they conduct pre-processing to a selected set of vertices which are more important in terms of shortest path and distance computation. Representative approaches include *ATL* [16], *TNR* [6] [7], *Highway Hierarchies* [45] and *Contraction Hierarchies* [40] [41].

2. Graph Partitioning based Indexing

   The approaches in this category [5][24][25][26] [51] focus on the topologies of networks by hierarchical graph partitioning and pre-compute distances among certain boundary nodes generated by the partitions. Then they use these pre-computed distances to facilitate efficient shortest path and distance query processing. Representative approaches in this category include *HEPV* [24][25] and *HiTi* [26].

3. Spatial Coherence based Indexing

   The approaches in this group pre-compute shortest paths and distances for all pairs of vertices in the network and utilize their spatial coherent property to compress these paths and distances. Spatial coherent property lies in the observation that if vertices s and $s^{'}$ are close to each other and vertices t and $t^{'}$ are close to each other, then the shortest path from s to t is likely to share vertices with that from $s^{'}$ to $t^{'}$. Representative approaches of this group include *Arc-labels* [18][34] [54], *Spatial*

Induced Linkage Cognizance (SILC) [46][44] and *Path-Coherent Pairs Decomposition (PCPD)* [49][47].

Vertex Importance based Indexing

Goldberg and Harrelson proposed an shortest path algorithm called ALT [16]. It uses A* search in combination with a lower bounding technique based on landmarks and triangle inequality. It first selects a small set of vertices as landmarks, then pre-computes distances between each vertex in the network and each landmark. With the pre-computed distances, it easily derives lower bounds while conducting Dijkstra's algorithm, thus achieving efficient search space pruning. ALT improves the efficiency of shortest path query compared to Dijkstra's algorithm. But its performance is highly dependent on the selection of landmarks. In some cases, it still need to traverse a large number of vertices during query processing.

Transit Node Routing (TNR) [6] [7] is an indexing technique based on pre-processing of a set of access nodes brought by imposing a grid on the spatial networks. It pre-computes the distances from each vertex v to each access node of the cell that contains v and distances between any pair of access nodes. With these pre-computed distances, TNR can efficiently derive the distances between any pair of vertices in networks. The performance of TNR depends highly on the granularity of the grid imposed on the spatial network. A finer grid brings more efficiency of query processing but also generates higher space cost, while sparser grid suffers from poorer efficiency but has a lower space overhead.

Different from ALT and TNR, CH [40] [41] [45] imposes a total order to the vertices in the networks in terms of their importance in query processing by a group of heuristic methods. Then it adds shortcuts to eliminate less important vertices through a process called contraction. Thus a CH graph is constructed by the end of contraction, it has the added shortcuts but preserves the shortest paths of the original graph. To process shortest

path query, CH conducts a variant of the bidirectional version of Dijkstra's search, during which it utilizes pre-computed distances to accelerate query processing. One disadvantage of CH is that its performance highly depends on the orders of the vertices. Thus in the worst case it has a $O(n^2)$ space cost for the shortcuts and $O(n^2 \log n)$ time complexity for shortest path searching.

Graph Partitioning based Indexing

Hierarchical Encoded Path (HEPV) [24][25] is an indexing method designed for shortest path distance query processing based on graph partitioning. It partitions the network into multiple fragments by a method called Spatial Partitioning Clustering (SPC) and pushes up all border nodes to construct a more compact network at a higher level. This fragmentation (partition) process is repeated to construct a network hierarchy. For the network at each level, it pre-computes distances for all pair of vertices for each fragment in this network. To compute shortest path distance, HEPV iteratively checks the network at a higher level if target vertices are located in different fragments then use pre-computed distances to derive the shortest path distance. The disadvantage of HEPV is its high space cost since it pre-computes and stores all pairs of distances for each fragment at each level. And when the levels of the hierarchy is large, the query efficiency of HEPV suffers a lot, even worse than Dijkstra's algorithm.

HiTi [26] is another indexing method based on graph partitioning. It constructs the network hierarchy by iteratively partitioning each subgraph until the fragment is small enough. Then for each subgraph in the hierarchy, it pre-computes shortest path distances between each pair of boundary nodes. With these pre-computed distances, HiTi employs a modified version of Dijkstra's algorithm for efficient query processing. Compared to HEPV, HiTi has a much smaller space cost, but its query processing efficiency is inferior to HEPV.

Spatial Coherence based Indexing

Arc labels [18][34] is an indexing method taking advantage of shortest path information encoding. It first imposes an partitioning structure, such as grid, on the network. In the preprocessing phase, for each edge (arc) a in the network, it tags the grid cells in which their is at least one vertex that has a shortest path to it passes through a. Then given any two vertices, a modified version of Dijkstra's algorithm is adopted to void visiting irrelevant edges by identifying the edges that doesn't tag the cell that the target vertex located in. Möhring [12] did an extension of this technique to multiple levels of partitioning and studied the influence of different partitioning strategy on the performance of this technique.

Spatially Induced Linkage Cognizance (SILC) [46][44] pre-computes distances for all pairs of vertices in the spatial network. Then it stores these paths and distances in a concise format by a quadtree-based encoding technique, which achieves a $O(n^{1.5})$ space cost. With these encoded path and distance information, for any pair of vertices, their shortest path processing has the complexity of $O(k \log n)$, where k is the number of vertices in the shortest path. SILC is also efficient for nearest neighbor queries as shown in [44]. The disadvantage of SILC is that it incurs significant preprocessing time and space consumption for large networks, so it might be not practical to some applications.

Path Coherent Pairs Decomposition (PCPD) [49][47] is similar to SILC. It also pre-computes all pairs shortest path and distances of the network. Different from SILC, it employs a concept of path-coherent pairs and pre-computes a set of path-coherent pairs, such that any pair of vertices can be be covered by a unique path-coherent pair. During query processing, it uses this path-coherent pair set to derive the shortest path. PCPD also has an time complexity of $O(k \log n)$, where k is the number of vertices in the shortest path. Sankaranarayanan and Samet [48] proposed a revised version of PCPD that can handle approximate distance queries efficiently. The space overhead of PCPD is similar with SILC, but the practical performance of PCPD is inferior to SILC in terms of query

efficiency and pre-processing time.

Wu et al.[56] conducted an experimental evaluation on several state-of-the-art approaches for shortest path and distance query processing, including TNR, CH, SILC and PCPD. They used a variety of real world road networks with up to twenty million vertices. Their experimental results show that CH is the most space-economic technique compared TNR, SILC and PCPD. In terms of query efficiency, CH also performs well and is only inferior to SILC. SILC outperforms other techniques in computation time, but it incur significant preprocessing time and space consumption.

2.2 k-NN Queries in Spatial Networks

Jensen et al. [21] first formalize the problem of k-NN search in road networks and propose a system prototype for such queries. They proposed graph representation to model the road networks. Their shortest path distance between a query point and an object is obtained by online calculation based on Dijkstra's algorithm.

INE [38] incorporate Dijkstra's algorithm to process k-NN queries. It incrementally gets the nearest neighbors during network expansion and returns their distances to the query point at the same time. But it suffers from poor performance when the objects are not densely distributed in the network, since it will traverse a large portion of the network to reach all nearest neighbors.

IER [38] integrates network expansion with Euclidean information. It uses Euclidean distance to prune search space and filter out a candidate set, then computes the distance of these candidates to get the k nearest neighbors.

LBC [13] improves INE by avoiding the visit of vertices that cannot lead to k nearest neighbors utilizing an Euclidean heuristic function, and improves IER by avoiding repeated visits to vertices that appear on the shortest paths to these nearest neighbors. Its

drawback is the maintenance cost of multiple instances of heuristic search (wavefront), as it requires k priority queues and maintaining them could be expensive.

SWH [36] improves LBC by utilizing a novel heuristic function and avoids the maintenance of multiple priority queues, thus is more memory-efficient than LBC.

The disadvantage of INE, IER, LBC and SWH is that they are all online network expansion approaches based on Dijkstra's algorithm, so their efficiency suffers for large networks which make them inapplicable to real time applications.

Voronoi based Indexing [29] incorporates the concept of Voronoi regions to the query processing in road networks. It partitions large road network into multiple Voronoi regions, then pre-computes distances both within and across these regions. With these preprocessing information along with Voronoi regions, it can quickly find the nearest neighbors. The drawback of this approach is the high maintenance cost of Voronoi regions.

ROAD [31] [32] recursively partitions a network into sub-networks and pre-computes the distances between the boundary vertices of each sub-network. It also incorporates Dijkstra's algorithm, but is able to skip sub-networks without any object in it during k-NN search. So it improves Dijkstra's search in terms of computation time. But it still performs poorly in large networks.

SILC [44] decouples the process of computing shortest path distances from that of finding the nearest neighbors. It pre-computes and encodes shortest paths and distances between all pair of vertices. It assumes the existence of a search hierarchy T for the object set and use intersection of blocks to do distance computation during k-NN search. The efficiency of this algorithm is superior compared to networks expansion approaches based on Dijkstra's algorithm, with a price of $O(n^{1.5})$ space cost.

2.3 Trajectory Query Processing

The wide adaptation of wireless communication and location-based services led to a great amount of trajectory data recording the movement history of moving objects. This motivates many research efforts processing and analyzing large-scale trajectory data, including trajectory indexing [39] [9] [42] [12] , trajectory query processing [53] [4] [15] [52] [11] [50] [62] and trajectory pattern mining [10] [23] [22] [63] [30].

Here we introduce some representative trajectory query processing methods most related to the work of this thesis.

Čeikutė and Jensen [8] did an evaluation of the quality of routing services by comparing them with local driver behaviors. They concluded that the history trajectories of local drivers hold great potential to significantly increase the quality of existing routing services.

Yuan et al. [59] proposed an approach to mine smart driving directions from the historical GPS trajectories of a large number of taxis. Their idea is to build a time-dependent landmark graph and then perform a two-stage routing algorithm based on this graph to find the practically fastest route.

Chen et al. [11] proposed an efficient approach to address trajectory queries based on a set of locations. They defined a similarity function to evaluate how well a trajectory connects the query locations and proposed k-BCT query based on it. Then an Incremental k-NN based Algorithm(IKNN) is proposed to process this type of queries efficiently. But the query processing is based on the assumption of Euclidean Space.

Shang et al. [50] investigate user oriented trajectory search for trip recommendation. This query considers both textual domain and spatial domain. It assumes that each trajectory has a set of textual attributes to describe its features. Then it proposes a spatial-textual distance function to evaluate how well a trajectory satisfies the user's query. A collaborative searching approach conducts query processing in the spatial and textual domains

alternatively. And a pair of upper and lower bounds are devised to constrain the searching in the two domains.

[62] studied the query of activity trajectory, which associate spatial trajectories with activity information. It proposed a hybrid grid index named GAT to process the queries efficiently.

2.4 Summary

On-the-fly network expansion like Dijkstra's algorithm has no extra space cost, but it is inefficient for large networks. Full materialization approaches are quite efficient, but the huge space cost makes them inapplicable for large networks. Partial materialization approaches have moderate space cost and query efficiency compared with previous two groups of approaches. Their challenges lie in finding a materialization strategy that achieves a balance between the space cost and the query efficiency. This motivated us to develop an indexing technique that supports efficient query processing and has a moderate space cost at the same time, so that it is scalable to large networks

Furthermore, although the approaches mentioned above have achieved great improvement compared to classic network expansion based approaches, they merely considered network topologies but not the data distribution. Actually the distribution of object set often influences the query processing efficiency significantly. That motivated us to investigate an object set oriented indexing method. By combining distribution of object dataset and network topologies into index construction, it provides a better trade-off between space consumption and query efficiency.

Chapter 3

Efficient Object Query Processing in Spatial Networks

3.1 Motivation

As stated in Chapter 1 and Chapter 2, shortest path queries and k-NN queries are the most important queries in spatial networks. But the existing works are either not efficient enough or bring in great space cost for indexing. This motivated us to develop an index that supports efficient query processing and has a moderate space cost at the same time, so that it is scalable to large networks.

3.2 Problem Settings

In this section, we present some important concepts and notations that will be used throughout this paper.

Definition 1. Graph We model a spatial network as a connected planar graph $G = (V, E)$. V is the set of vertices in G, and $E \subseteq V \times V$ is the set of edges. Each edge is assigned with a real-valued weight to represent the traveling cost of this edge.

Definition 2. Shortest Path Query Given a graph $G = (V, E)$, for a query pair $s, t \in V$, the shortest path between them is a series of edges $SP(s,t) = \{(v_1, v_2), (v_2, v_3), ..., (v_{x-1}, v_x)\}$ connecting them that has the minimum total length, where $v_1 = s$, $v_x = t$. The shortest path distance between s and t is $dist(s,t) = \sum_{i=1}^{x-1} w(v_i, v_{i+1})$, where $w(v_i, v_{i+1})$ is the weight of edge (v_i, v_{i+1}).

Definition 3. k-NN Query Given a spatial network $G = (V, E)$, an object set O, a query point q and a positive integer k, a k-NN query returns k objects from O that have the smallest distances to query point q. Each object in object set O represents an object (e.g. a specific type of facility) we are interested in. For simplicity, we assume that all objects are located at vertices of the network.

A network can be divided into several parts through a process called graph partitioning:

Definition 4. Graph Partitioning Given a graph $G = (V, E)$, a d-way partitioning of the graph is to divide it into d subgraphs $G_1, G_2, ..., G_d$, such that (1) $G_i = (V_i, E_i)$, (2) $\cup_{1 \leq i \leq d} V_i = V$, (3) For any two subgraphs G_i and G_j, $i \neq j, V_i \cap V_j = \emptyset$, (3) For $\forall u, v \in V_i$, if $(u, v) \in E$, then $(u, v) \in E_i$.

Definition 5. Borders Assume that graph G is partitioned to subgraphs $G_1, G_2, ..., G_k$. If there exists an edge (u, v), $u \in G_i$ and $v \in G_j, i \neq j$, then u is a border of G_i and v is a border of G_j. All borders of G_i form a border set $B(G_i)$.

For example, in Figure 3.1, the original graph G is partitioned into two subgraphs G_1 and G_2. Since there are edges (v_5, v_{10}), (v_5, v_{14}), (v_8, v_{15}) connecting vertices of G_1 and G_2, so G_1 has borders $\{v_5, v_8\}$ and G_2 has borders $\{v_{10}, v_{14}, v_{15}\}$.

3.3 Proposed Indexing Structure

In this section, we present our proposed index. From the definition of graph partitioning, we have the following observation: Once a network is partitioned into a set of subgraphs,

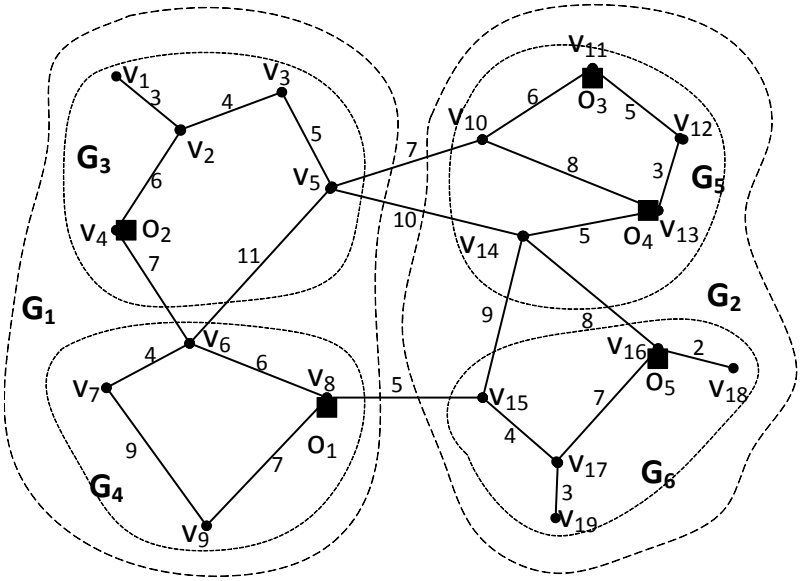


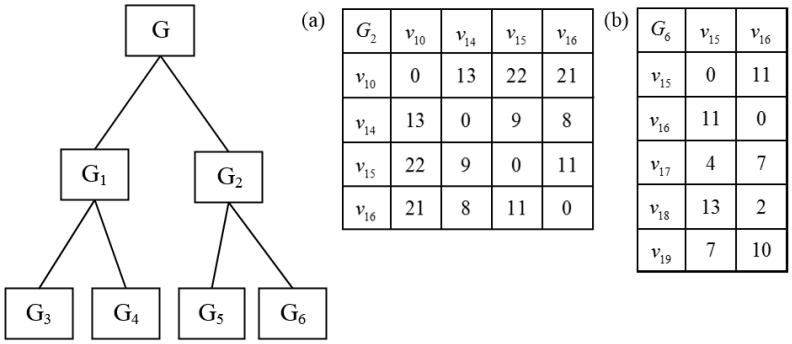
FIGURE 3.1: Hierarchical Graph Partitioning

the shortest path between vertices of two different subgraphs will pass by at least one border of each subgraph. For example, in Figure 3.1, the shortest path between v_7 and v_{12} will pass by one border of subgraph G_4 and one border of G_5. In this case, the borders brought by partitioning become the important points in shortest path query processing. This observation motivated us to develop a graph partitioning based index. It organizes the vertices of a spatial network into a hierarchy and associates pre-computed information with it to facilitate efficient spatial query processing.

3.3.1 Index Overview

Given a spatial network modeled as a graph $G = (V, E)$, we build its Partition Tree through following steps:

1. Conduct d-way partitioning to the graph in a hierarchical manner, until termination condition is fulfilled (e.g. the number of vertices for each leaf subgraph is under a threshold).



G_2	v_{10}	v_{14}	v_{15}	v_{16}
v_{10}	0	13	22	21
v_{14}	13	0	9	8
v_{15}	22	9	0	11
v_{16}	21	8	11	0

G_6	v_{15}	v_{16}
v_{15}	0	11
v_{16}	11	0
v_{17}	4	7
v_{18}	13	2
v_{19}	7	10

FIGURE 3.2: Partition Tree

2. Use a tree structure to represent the partitioning hierarchy, such that each node in the tree represents a subgraph in the hierarchy. Each internal (non-leaf) node represents a subgraph that is partitioned into several subgraphs. And each leaf node represents a subgraph that is not further partitioned.

3. For each tree node, we maintain its border set and a distance matrix recording pre-computed distances related to the subgraph it represents.

   - For each internal node, its distance matrix contains the distances between all pairs of borders that belong to its child nodes.
   - For each leaf node, its distance matrix contains the distances between each pair of vertex and border that belongs to this subgraph.

For example, given the graph in Figure 3.1, we conduct hierarchical partitioning to it until each leaf subgraph contains no more than 5 vertices. Original graph G is firstly partitioned to two subgraphs G_1 and G_2. Then G_1 is further partitioned to G_3 and G_4 while G_2 is partitioned to G_5 and G_6. We use the Partition Tree in Figure 3.2 to represent this partitioning hierarchy. Internal node G_2 has two children G_5 and G_6. G_5 has two borders v_{10} and v_{14} while G_6 has two borders v_{15} and v_{16}. So the distance matrix of G_2 contains distances between each pair of these borders, as shown in Figure 3.2(a). Leaf node G_6

contains vertices v_{15}, v_{16}, v_{17}, v_{18}, v_{19} and two borders v_{15} and v_{16}, so its distance matrix contains distances between each vertex and these borders, as shown in Figure 3.2(b).

The tree structure and border sets record the partitioning topology of the spatial network. Distance matrices record the pre-computed information for efficient query processing. Therefore, with the Partition Tree, we encode partitioning topology and pre-computed information of the graph at the same time. For the consideration of space cost, we do not pre-compute and store shortest path information for all pairs of vertices. In following discussion, we will show that our indexing method is efficient enough for query processing in spatial networks.

Note that, although the previous introduced HEPV [24][25] and HiTi[26] are also graph partitioning based methods, the graph partitioning conducted here is totally different. HEPV partitions the network into multiple fragments and pushes up all border nodes to construct a higher level graph. This process is repeated until the graph at the highest level is compact enough. On the other hand, HiTi pushes up the cut edges and shortcut edges between borders of each subgraph to construct a higher level graph. Different from them, our partitioning process adopts a top-down strategy. Each subgraph is partitioned into multiple subgraphs at a lower level. Furthermore, because of the difference of the partitioning strategies, different pre-computing methods are conducted here. In the following discussion, we can see that the space cost of our index is superior to HEPV and HiTi.

3.3.2 Index Construction

For the consideration of query efficiency and space cost, the goal of our partitioning is to generate equal-sized subgraphs and minimize the number of borders at the same time. One classic solution for graph partitioning is Kernighan-Lin algorithm [28], it firstly partitions the graph into two parts randomly and then refines the partitioning by greedy swaps.

Kernighan-Lin algorithm is straightforward but the $O(n^3)$ complexity makes it inapplicable for large networks. In this work, we adopt a heuristic graph partitioning algorithm with $O(n)$ time complexity named multilevel graph partitioning [27]. It firstly converts the original graph into a much smaller graph through a series of processes called coarsening. Then partitioning is conducted on this coarsened graph with just a few hundred vertices, so approach like Kernighan-Lin algorithm [28] is efficient enough. Finally the partitioning is projected back to original graph and refined.

Given a graph G, we first partition G into d subgraphs. And then each subgraph is further partitioned in the same way. We repeat this process until the termination condition (e.g. each leaf subgraph contains no more than τ vertices) is fulfilled. Then we use Dijkstra's algorithm to conduct computation for distance matrix of each subgraph.

Now we take a look at the space cost of the Partition Tree. Given a graph with n vertices, we conduct d-way hierarchical partitioning to generate a balanced tree structure. Thus each subgraph at level i contains n/d^i vertices. According to the $\sqrt{n}$-Separator Theorem [33], the partitioning of a subgraph at level i generates $O(\sqrt{n/d^i})$ borders, which are shared by its children at level $i+1$. So the size of distance matrix for internal nodes at level i is $O(n/d^i)$. Since there are d^i nodes at level i, the total size of distances matrices for nodes at level i is $O(n)$. The height of the partition tree h is very small compared to n, so the total size of distance matrices for non-leaf nodes is $O(n)$. Assuming each leaf subgraph contains τ vertices, its number of borders is $O(\sqrt{\tau})$. So the size of the distance matrix for a leaf node is $O(\tau\sqrt{\tau})$. Since there are n/τ leaf nodes, then the total size of distance matrices for all leaf nodes is $O(\sqrt{\tau}n)$. When τ is much smaller than n, the total space cost for all leaf nodes is $O(n)$.

3.4 Query Processing

In this section, we present our query processing algorithms based on Partition Tree. For shortest path queries, we propose a dynamic programming algorithm utilizing precomputed distances to avoid large-scale network expansion. For k-NN queries, we propose a best-first algorithm which utilizes dynamic programming algorithm to compute distance bounds efficiently.

3.4.1 Shortest Path Query Processing

As mentioned in section IV, once a network is partitioned into a set of subgraphs, the shortest path between vertices of two different subgraphs has the following property:

Lemma 1: For any two vertices s and t of different subgraphs G_s and G_t, the shortest path between s and t contains at least one border of G_s(resp. G_t).

For example, in Figure 3.1, vertices v_7 and v_{12} belong to different subgraphs G_4 and G_5 respectively. The shortest path from v_7 to v_{12} is $(v_7, v_6, v_8, v_{15}, v_{14})$. v_8 is a border of G_4 and v_{14} is a border of G_5. Lemma 1 is prominent, so we skip the proof of it.

This property can be generalized to hierarchical partitioning. As shown in Figure 3.3, the graph G is hierarchically partitioned, such that vertices s and t reside in different leaf subgraphs G_s and G_t. From Lemma 1 we know that the shortest path between s and t contains at least one border of G_t, so their distance is decided by:

$$dist(s,t) = \min_{b_i \in B(G_t)} (dist(s,b_i) + dist(b_i,t)) \tag{3.1}$$

Since G_t belongs to the higher level subgraph G_{t-1} in the partitioning hierarchy, if s is not in G_{t-1} , the shortest path will also contain at least one border of G_{t-1}. So the distance between s and b_i is decided by:

$$dist(s,b_i) = \min_{b_{i-1} \in B(G_{t-1})} (dist(s,b_{i-1}) + dist(b_{i-1},b_i)) \tag{3.2}$$

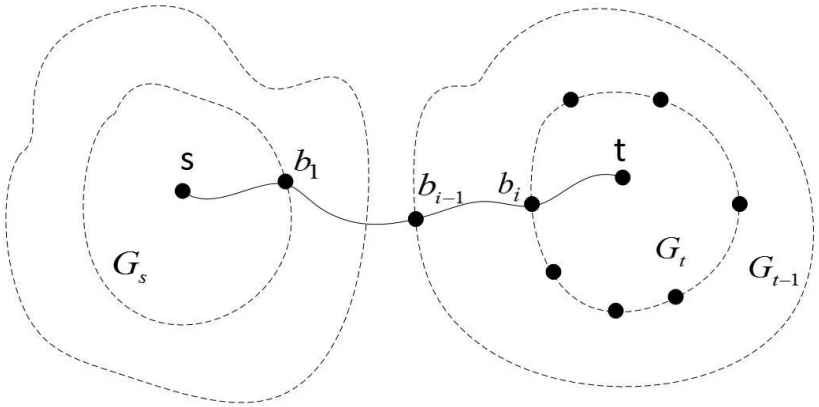


FIGURE 3.3: Shortest Path Decomposition

This indicates that we can use a dynamic programming algorithm to compute the distance between vertices of different subgraphs. Suppose s and t are in leaf subgraphs G_s^x and G_t^y respectively and their lowest common ancestor is G_s^0 (G_t^0), the parent hierarchy from G_s^x to G_s^0 is G_s^x, G_s^{x-1}, ..., G_s^1, G_s^0 and the parent hierarchy from G_t^y to G_t^0 is G_t^y, G_t^{y-1}, ..., G_t^1, G_t^0. Then $G_s^x, ..., G_s^1, G_t^1, ..., G_t^y$ form a search hierarchy of shortest path between s and t, and the shortest path will pass by at least one border of each subgraph in this hierarchy. We can decompose the distance computation according to this search hierarchy and get the total distance by iteratively processing calculation according to equations (3.1) and (3.2).

For example in Figure 3.1, v_7 and v_{12} are in different leaf subgraphs G_4 and G_5. And they have the lowest common ancestor G_0 in the Partition Tree. So the search hierarchy of shortest path between v_7 and v_{12} is G_4, G_1, G_2, G_5. Since G_5 has two borders v_{10} and v_{14}, $dist(v_7, v_{12})$ is decided by the minimum value of $dist(v_7, v_{10}) + dist(v_{10}, v_{12})$ and $dist(v_7, v_{14}) + dist(v_{14}, v_{12})$. Likewise, $dist(v_7, v_{10})$ and $dist(v_7, v_{14})$ can be decided by distances from v_7 to the borders of the higher level subgraph G_2. We keep processing in this way until the distance to v_{12} is obtained, as shown in Figure 3.4.

Note that during dynamic programming, the decomposed distances that are directly used can be classified into three categories.

In the first category, the distance is between a vertex and a border of its resident leaf subgraph. Since each leaf-node maintains a distance matrix recording distances between each pair of vertex and border. So distances in this category can be obtained directly from the matrix of a leaf subgraph. For example in Figure 3.4, the distance between v_7 and v_6 can be obtained from distance the matrix of G_4.

In the second category, the distance is between a border of a subgraph and a border of its parent subgraph. Each non-leaf node maintains a distance matrix recording the distances between all pairs of borders belonging to its child nodes. So the distances in this category can be obtained from the distance matrix of an internal node. For example in Figure 3.4, the distance between v_6 and v_8 can be obtained from the distance matrix of G_1.

In the third category, the distance is between two borders belonging to two sibling subgraphs respectively. The distance between them can also be obtained from the distance matrix of the parent node of these two subgraphs. For example in Figure 3.4, the distance between v_5 and v_{14} can be obtained from distance matrix of G_0.

So all the distances directly used in shortest path decomposition can be obtained from distance matrices associated with Partition Tree.

So far the dynamic programming algorithm returns the distance between s and t and also a subset of the shortest path between them, $SP^{'}(s,t) = \{v_1, v_2, ..., v_i, v_{i+1}, ..., v_m\}$, such that $v_1 = s$, $v_m = t$ and other vertices are all borders of subgraphs in the search hierarchy from s to t. Then we can get the whole shortest path by inserting necessary vertices between each pair of adjacent vertices of the subpath.

For each pair of adjacent vertices v_i and v_{i+1} in the subpath already got, if there is no edge between them, we will find the vertex between them in the shortest path and insert it as follows.

1. If v_i and v_{i+1} are the borders of different leaf subgraphs, we check the distance matrix of their lowest common ancestor to find if there exists a border v_x, such that

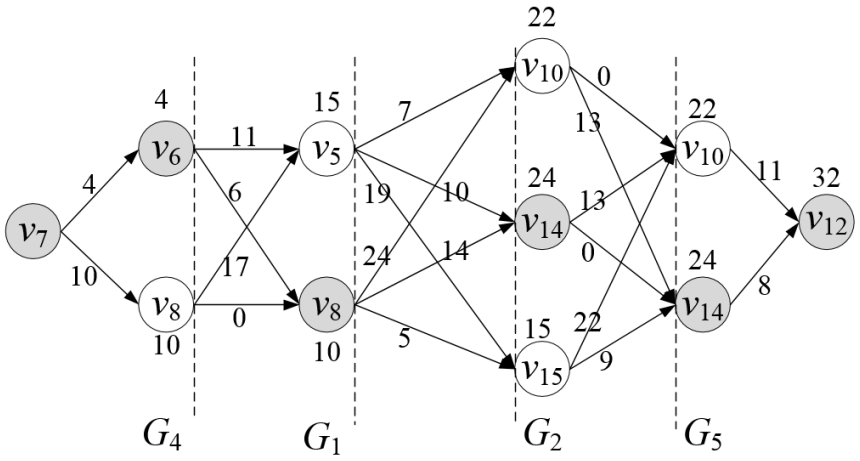


FIGURE 3.4: Dynamic Programming Algorithm

$dist(v_i, v_{i+1}) = dist(v_i, v_x) + dist(v_x, v_{i+1})$. If there is, we insert this border to the subpath between v_i and v_{i+1}. Otherwise, we check the ancestors of the lowest common ancestor until finding such a border to insert.

2. If v_i and v_{i+1} are the borders of the same leaf subgraph, the shortest path between them may include non-border vertices. First we will check the distance matrix of this leaf subgraph to find if there exists one non-border vertex v_x, such that $dist(v_i, v_{i+1}) = dist(v_i, v_x) + dist(v_x, v_{i+1})$. If there is, we insert this vertex to the subpath between v_i and v_{i+1}. Otherwise, we check the distance matrices of the ancestors of this leaf node as talked before until finding a vertex to insert.

3. If one of v_i and v_{i+1}, e.g. v_i, is non-border, they will be in the same leaf subgraph. We will check the distance matrix of the leaf subgraph to find if there exists a border v_x such that $dist(v_i, v_{i+1}) = dist(v_i, v_x) + dist(v_x, v_{i+1})$. If there is, we insert this border to the subpath between v_i and v_{i+1}. Otherwise, this means the shortest path between v_i and v_{i+1} only contains vertices inside the leaf subgraph. Then we will check the neighbors of v_i to find a neighbor v_x satisfying $dist(v_i, v_{i+1}) = dist(v_i, v_x) + dist(v_x, v_{i+1})$ and put it between v_i and v_{i+1} in the subpath.

For each adjacent pair of vertices in the subset of shortest path between s and t, we will process the insertion as discussed above until the whole shortest path is generated.

To sum up, given a graph and its Partition Tree, to get the shortest path between s and t, we take different approaches according to their locations.

Local Search

If s and t are in the same leaf subgraph, we use Dijkstra's algorithm to get the shortest path. Since the size of leaf subgraphs is small, the efficiency is guaranteed.

Global Search

If s and t are in different leaf subgraphs, we decompose the shortest path between them into several parts divided by a series of borders according to the partition hierarchy and then use dynamic programming algorithm to get the shortest path .

For local search, the time complexity of Dijkstra's algorithm is $O(\tau \log(\tau))$, where τ is the number of vertices in leaf subgraphs. For global search, the cost of dynamic programming algorithm is decided by the number of borders. If the search hierarchy from s to t is $\{G_s^x, ..., G_s^1, G_t^1, ..., G_t^y\}$, the border sets of these subgraphs are noted as B_{h_1}, B_{h_2}, ..., $B_{h_{x+y}}$, where $|B_{h_i}| = \rho_i$, then the computation cost can be denoted as $\sum_{i=1}^{x+y-1} \rho_i \cdot \rho_{i+1}$. To return the shortest path between s and t, the complexity is $O(l + \sum_{i=1}^{x+y-1} \rho_i \cdot \rho_{i+1})$ where l is the number of vertices in the shortest path.

3.4.2 k-NN Query Processing

Given a spatial network $G = (V, E)$, an object set O, a query point q and a positive integer k, a k-NN query returns k objects from O that have the smallest distances to the query point q. An advantage of the Partition Tree is that it also supports efficient distance

computation between a vertex and a subgraph. Based on this we propose a best-first algorithm for k-NN query.

First we give the definition of distance between a vertex v and a subgraph G_x:

1. If v is in the subgraph G_x, $dist(v, G_x)$ = 0

2. If v is not in the subgraph G_x, then

$$dist(v, G_x) = \min_{b_i \in B(G_x)} dist(v, b_i)$$

If v is not in subgraph G_x, we use the dynamic programming algorithm similarly as discussed above to compute their distance. The only difference here is that the dynamic processing stops at the borders of target subgraph rather than a particular vertex inside it. For example, the distance between v_7 and G_5 is decided by minimum value of $dist(v_7, v_{10})$ and $dist(v_7, v_{14})$, as shown in Figure 3.4.

Next we will show how to use this point-subgraph distance to prune search space efficiently during k-NN search. Our k-NN query algorithm is based on the following properties:

Lemma 2: Given a subgraph G_x of graph G and a vertex $v \notin G_x$, for any vertex u in G_x, the inequation $dist(v, u) \geq dist(v, G_x)$ holds true.

Lemma 3: Given a subgraph G_x of graph G and a vertex $v \notin G_x$, for any child subgraph G_y of G_x, the inequation $dist(v, G_y) \geq dist(v, G_x)$ holds true.

Lemma 2 and Lemma 3 indicate that if the distance between a subgraph and the query point is larger than those of the candidates, there is no need to search the points inside this subgraph. Thus we can use this point-subgraph distance to prune search space in k-NN search. Algorithm 1 gives the illustration of our best-first k-NN algorithm. It uses a priority queue Q to maintain the distances of the subgraphs or objects that have been computed. Then it always selects the element with the smallest distance from Q to process next.

Algorithm 1: best-first k-NN search

```
1:  Input:
    query point q,
    required number of neighbors k,
    object set O,
    network G and its Partition Tree T
2:  Output: result set R
3:  priority queue Q = φ
4:  R = φ
5:  Q = {(T.root, 0)} ;
6:  while |R| < k and Q is not empty do
7:      e = Q.dequeue();
8:      if e is an object then
9:          insert e into R;
10:     else
11:         // e is a subgraph
12:         for each c in e.children() do
13:             put ⟨c, dist(q, c)⟩ into Q;
```

For example in Figure 3.1, given the network G, its Partition Tree T, an object set $O = \{o_1, o_2, o_3, o_4, o_5\}$ and a query point v_7, to process 2-NN query, best-first algorithm works in following way:

(1) Initialize the priority queue Q with the root of Partition Tree, $Q = \{(G_0, 0)\}$.

(2) G_0 is dequeued, its children G_1 and G_2 are put into priority queue, $Q = \{(G_1, 0), (G_2, 15)\}$.

(3) G_1 is dequeued, its children G_3 and G_4 are put into priority queue, $Q = \{(G_4, 0), (G_3, 11), (G_2, 15)\}$.

(4) Leaf subgraph G_4 is dequeued, its contained object o_1 is put into priority queue, $Q = \{(o_1, 10), (G_3, 11), (G_2, 15)\}$.

(5) o_1 is dequeued, $Q = \{(G_3, 11), (G_2, 15)\}$, $R = \{(o_1, 10)\}$.

(6) Leaf subgraph G_3 is dequeued, its contained object o_2 is put into priority queue, $Q = \{(o_2, 11), (G_2, 15)\}$.

(7) o_2 is dequeued, $Q = \{(G_2, 15)\}$, $R = \{(o_1, 10), (o_2, 11)\}$.

Then o_1 and o_2 are returned for 2-NN query at point v_7.

3.5 Query-oriented Optimization

In previous discussion, we only consider the Partition Tree under the assumption of balanced partitioning. The complexity analysis in section V suggests that the performance of the Partition Tree is influenced by the partitioning strategy taken during construction. Meanwhile, we have the observation that there is no need of further partitioning for areas where few queries happen, since it won't influence the performance that much. But for areas queries happen frequently, it is worthy to do further partitioning. And the query probabilities are often related to the distribution of objects, since queries are often invoked around objects of interest. This motivated us to find a partitioning efficient for query processing by taking advantage of network topology and object distribution. To achieve this, we develop a cost model to estimate influence of partitioning and object distribution on query efficiency.

3.5.1 Cost Model

Given a leaf subgraph P in the partitioning hierarchy, we consider the cost of distance computation since it is the most fundamental operation in our query processing. For any point s in P and an arbitrary point t, the distance computation between them takes different approaches according to where t locates. If t is in P, we see it as a local search

and use $LC(P)$ to represent its cost. If t is not in P, we see it as a global search and use $GC(P)$ to represent its cost. Assuming $Pr(P)$ and $Pr(\overline{P})$ are the probabilities of these two situations, the expected cost of distance computation between s and t is:

$$EC(P) = Pr(P) \cdot LC(P) + Pr(\overline{P}) \cdot GC(P)$$

For local search, it takes Dijkstra's algorithm. So cost $LC(P)$ can be estimated as $O(\tau \log(\tau))$, τ is the number of vertices in P.

For global search, it takes dynamic programming algorithm. If the search hierarchy from s to t is $\{G_s^x, ..., G_s^1, G_t^1, ..., G_t^y\}$, the border sets of these subgraphs are denoted as B_{h_1}, B_{h_2}, ..., $B_{h_{x+y}}$, where $|B_{h_i}| = \rho_i$, then cost $GC(P)$ can be estimated as $\sum_{i=1}^{x+y-1} \rho_i \cdot \rho_{i+1}$.

Now we consider how the expected cost changes if P is further partitioned. Assuming P is partitioned to subgraphs P_1 and P_2, both the cost of local search and that of global search are changed.

For local search, if s and t are both in subgraph P_1, it uses Dijkstra's algorithm and the cost is $O(\tau_1 \log(\tau_1))$, where τ_1 is the number of vertices in P_1. Similarly, when s and t are in P_2, the cost can be estimated as $O(\tau_2 \log(\tau_2))$. When s and t are in P_1 and P_2 respectively, it uses dynamic programming algorithm and the cost is $\rho_1 \cdot \rho_2$, where ρ_1 and ρ_2 are the number of borders for P_1 and P_2. We use $Pr(P_1|P)$ (resp. $Pr(P_2|P)$) to denote the probability of a vertex resides in P_1 (resp. P_2) in the context of P. The probabilities for the three cases of distance computation discussed above are $Pr^2(P_1|P)$, $Pr^2(P_2|P)$ and $2Pr(P_1|P)Pr(P_2|P)$. So the change of values for $LC(P)$ after partitioning is:

$$\begin{aligned} \Delta LC(P) &= Pr^2(P_1|P) \cdot \tau_1 \log(\tau_1) \\ &+ Pr^2(P_2|P) \cdot \tau_2 \log(\tau_2) \\ &+ 2Pr(P_1|P)Pr(P_2|P) \cdot \rho_1 \cdot \rho_2 \\ &- \tau \log(\tau) \end{aligned} \tag{3.3}$$

For global search, it still takes dynamic programming algorithm to do distance computation. But the search hierarchy is one layer higher since it contains a subgraph between s and P. If s locates in P_1, the increased value of cost $GC(P)$ can be estimated as $\rho_1 \cdot \rho$, where ρ_1 is the number of borders for P_1 and ρ is the number of borders for P. Similarly, if s locates in P_2, the increased value of cost $GC(P)$ can be estimated as $\rho_2 \cdot \rho$. The probabilities of these two cases are $Pr(P_1|P)$ and $Pr(P_2|P)$ respectively. So the change of values for $GC(P)$ is:

$$\Delta GC(P) = Pr(P_1|P) \cdot \rho_1\rho + Pr(P_2|P) \cdot \rho_2\rho \tag{3.4}$$

Therefore we get the total change of expected cost for distance computation when a leaf subgraph P is partitioned:

$$\Delta EC = Pr(P) \cdot \Delta LC(P) + Pr(\overline{P}) \cdot \Delta GC(P) \tag{3.5}$$

The probabilities used here are decided by the distribution of objects in different subgraphs. Therefore this cost model takes account of both partitioning topology and object distribution. Note that here we only use bisection in discussion, but it can be easily adapted to k-way partitioning.

3.5.2 Cost-efficient Graph Partitioning

The value of the cost function for subgraph P estimates the expected cost of distance computation for any point in it. The changed value of cost function when P is further partitioned indicates the influence of further partitioning to the performance of the Partition Tree. Thus we can use it as an evaluation condition in the graph partitioning process.

During the construction of Partition Tree for graph G, for each leaf subgraph P, we evaluate $\Delta EC(P)$ when P is further partitioned. To better evaluate the change, we use the percentage growth of $\Delta EC(P)$, namely $\Delta EC(P)/\Delta EC(P_f)$, where P_f is father of P. We set up a threshold value ξ for evaluation.

If $\Delta EC(P) > 0$, it means the expected cost of distance computation is increasing with further partitioning. The partitioning should stops at current level.

If $\Delta EC(P) < 0$, but $\Delta EC(P)/\Delta EC(P_f) < \xi$, it means the decreasing rate of the expected cost is not prominent. The partitioning should stops at current level.

If $\Delta EC(P) < 0$, $\Delta EC(P)/\Delta EC(P_f) > \xi$, then partitioning is continued.

By conducting this evaluation, we achieve a partitioning efficient for query processing with a given objects distribution.

3.6 Experiments

In this section, we present the experimental evaluation of our proposed index and algorithms in terms of query efficiency and indexing cost.

3.6.1 Experiments setup

Environment and Competitors

For shortest path queries, we compared our algorithm with Dijkstra's algorithm [14] and CH [40][41], since Dijkstra's algorithm is the most classic solution for shortest path queries and CH is one well-known state-of-the-art approach that has great overall performance while incurring minimal cost of space and pre-computation. For k-NN queries, we compared our algorithm with IER [38] and INE [38]. All approaches were implemented with C++, and we adopted implementation of CH from [1]. All experiments were run on a 64-bit windows machine with Intel 3.40GHz CPU and a 16GB RAM.

Datasets and Query Sets

We used four real-world networks obtained from [3], each of which is a part of the road network of US. Table 3.1 lists the characteristics of these four networks. And Figure 3.5

TABLE 3.1: Dataset Characteristics

Name	Region	Number of Vertices	Number of Edges
NY	New York	264,346	733,846
COL	Colorado	435,666	1,057,066
FLA	Florida	1,070,376	2,712,798
CAL	California	1,890,815	4,657,742

shows the distribution of these networks. In addition, we used a real world POI set of California obtained from [2] with 105,725 points of interest as the object set.

For shortest path queries, we generated 10 query sets Q_1, Q_2, ..., Q_{10} for each network, of which each query is a pair of vertices naming the start point s and the end point t. Each query set contains 100 queries randomly selected from vertices of the network. For the ith query set Q_i, the length (i.e. number of vertices) of shortest path of each query is between $[(i-1)l, il]$, where l is obtained by dividing the maximum length for all shortest paths in the network by 10. So the length of shortest path for each query pair in Q_i is larger than that in Q_{i-1}.

The POI set is used in evaluation of k-NN queries and query-oriented optimization. To evaluate the efficiency of k-NN queries, we randomly selected 1000 points from vertices as query points. To evaluate the performance of query-oriented optimization for distance and shortest path queries, we generate 10 query sets Q_1, Q_2, ..., Q_{10} similarly as mentioned above except that the queries are generated from POI set.

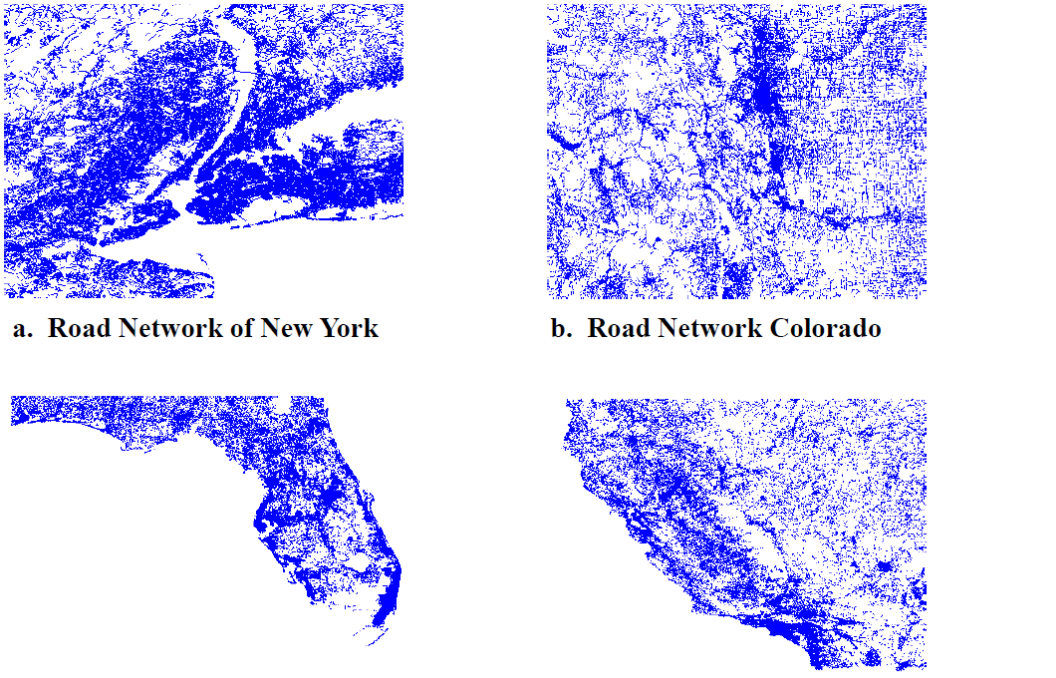


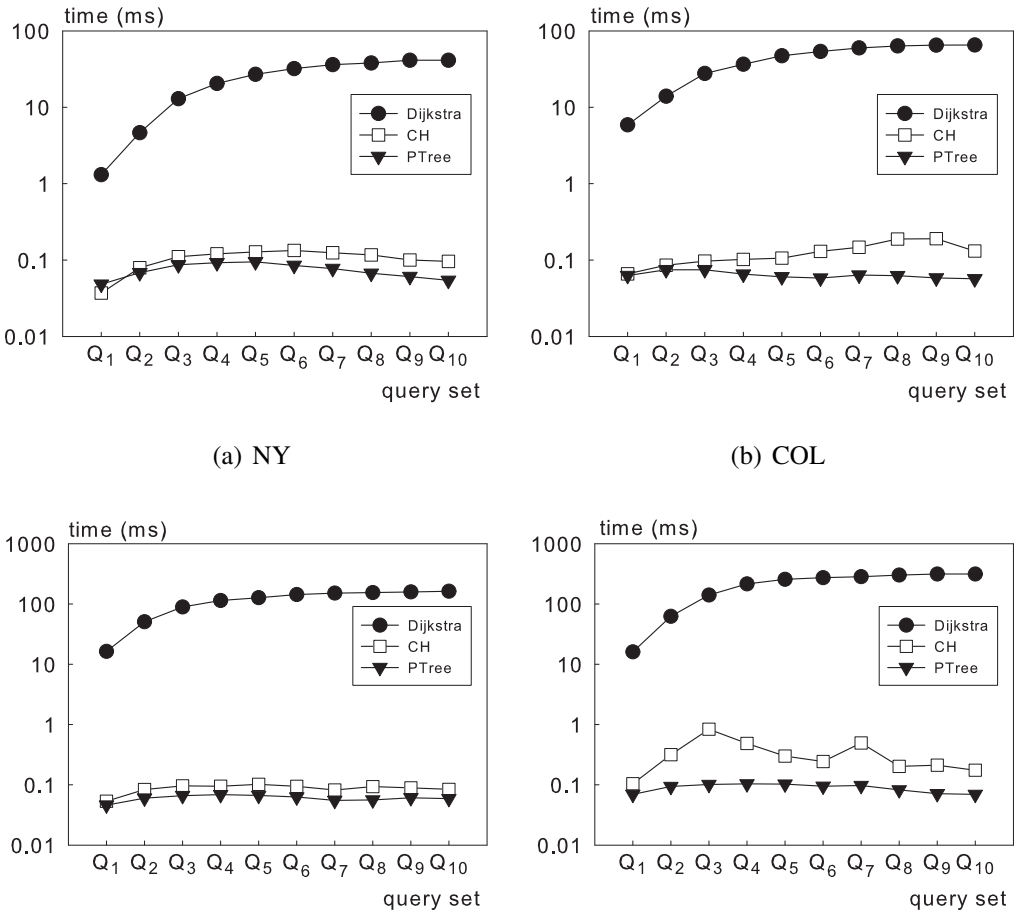
FIGURE 3.5: Network Distribution

3.6.2 Evaluation Results

Shortest Path Queries

We tested our shortest path algorithm on four networks mentioned above and compared it with two competitor techniques, Dijkstra's algorithm and CH. For each network, we run the queries in ten query sets and get the average running time of each query set. Figures 3.6 show the efficiency comparison between these three approaches. We can see that our algorithm (PTree) outperforms Dijkstra's algorithm and CH in experiments on every network. This corresponds with the fact that our dynamic programming algorithm avoids large-scale network expansion incurred in Dijkstra'a algorithm and CH.

From Figure 3.6(a) to 3.6(d), the number of vertices in each network is increasing. We can see that the efficiency of our dynamic programming algorithm doesn't suffer with



(a) NY (b) COL

(c) FLA (d) CAL

FIGURE 3.6: Efficiency of Shortest Path Queries

this increasing trend. This demonstrates that our algorithm is scalable to large networks. For each network, from Q_1 to Q_{10}, the number of vertices in the shortest path between queries in it is increasing. The average time cost of Dijkstra's algorithm is increasing at a high rate from Q_1 to Q_{10}, since the vertices to be traversed during network expansion is increasing. CH also has a increasing trend for processing time from Q_1 to Q_{10}, but the increasing rate is much smaller than Dijkstra's algorithm. Different from these two techniques, the efficiency of our algorithm is not necessarily increasing with the length of shortest path, since it is related to the number of borders and layers of subgraphs to be

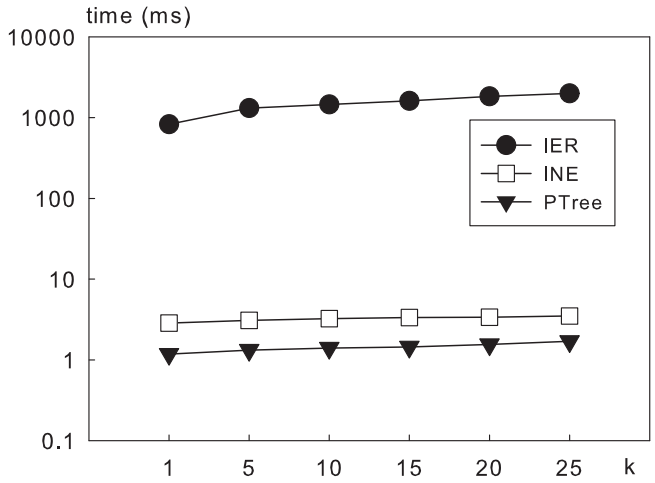


FIGURE 3.7: Efficiency of k-NN Queries

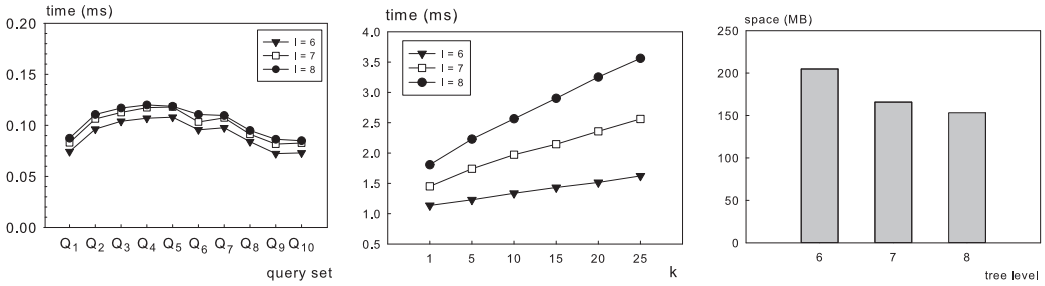
processed during dynamic programming.

k-NN Queries

We tested our best-first algorithm on network of California and compared it with IER and INE, varying the query parameter k from 1 to 25 at the same time. Figure 3.7 shows the efficiency comparison of these three approaches. We can see that our best-first algorithm (PTree) outperforms IER and INE, since it prunes the search space effectively to avoid unnecessary network expansion.

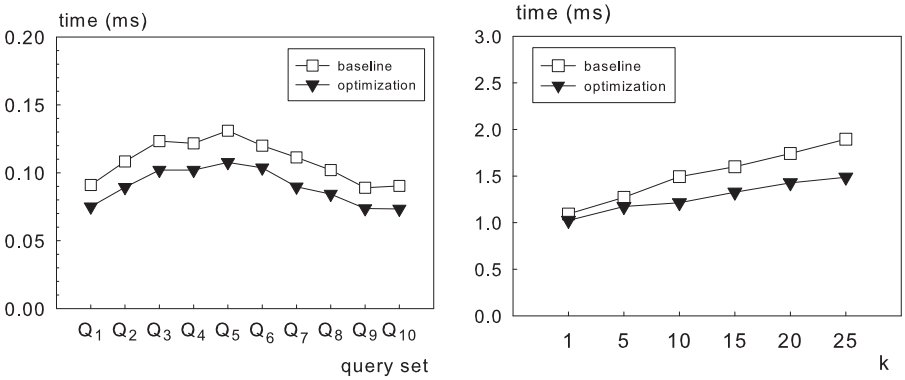
Effect of Tree Height

This set of experiments evaluate the performance of baseline index (vertex-balanced Partition Tree) in terms of level of the partitioning hierarchy. We tested the performance of the Partition Tree with tree height varying from 6 to 8. Figure 3.8 shows the performance comparison for shortest path queries and k-NN queries. We can see that the performances of the Partition Tree for shortest path queries and k-NN queries both suffer when tree height is increasing from 6 to 8. This is because the dynamic programming algorithm



(a) Shortest Path Queries (b) k-NN Queries (c) Space Cost

FIGURE 3.8: Effect of Tree Height



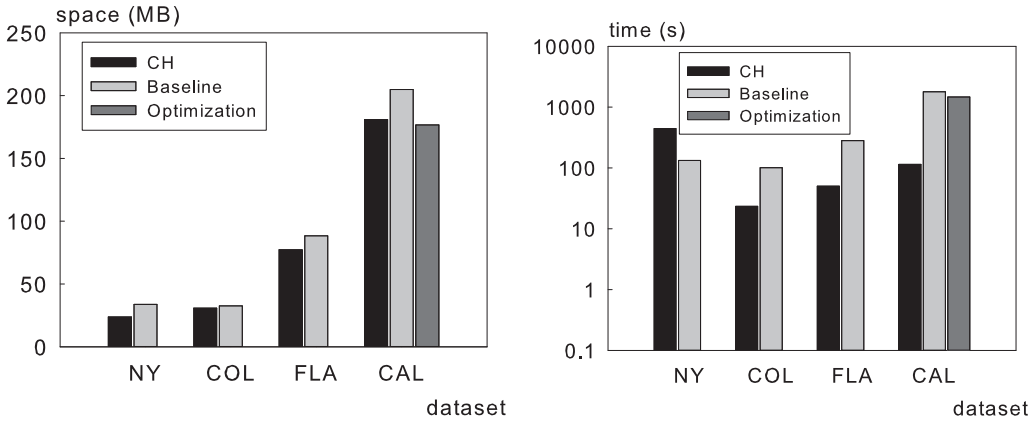
(a) Shortest Path Queries (b) k-NN Queries

FIGURE 3.9: Query-oriented Optimization

involves more borders in computation when the height is increasing. Figure 3.8(c) shows that the space cost of indexing is decreasing when tree height is increasing from 6 to 8. This is because the number of vertices in leaf nodes is decreasing so the space cost for distance matrices is reduced.

Query-oriented Optimization

We tested the performance of our query-oriented optimization on road network and POI set of California. Figure 3.9 shows the performance comparison of the baseline index and



(a) Space (b) Time

FIGURE 3.10: Evaluation of Indexing Cost

the query-oriented optimization. For the baseline index, we set the tree height to be 6, which is proved to have the best performance in the previous set of experiments. We can see that the query-oriented optimization improves the efficiency of shortest path queries and k-NN queries. Meanwhile this optimization also reduces the space cost of index, as shown in Figure 3.9(a).

Cost of Indexing

We tested the indexing cost of the Partition Tree on four networks and compared it with that of CH. Figure 3.10(a) shows the space cost of the Partition Tree (baseline) and CH on various networks. From the results we can see that the Partition Tree's space cost is comparable to CH, which has minimal space cost among all existing works. Meanwhile from the results on network of California, we can see that the query-oriented optimization reduces the space cost of the baseline index. Figure 3.10(b) shows the time cost for indexing construction of the Partition Tree and CH. The time cost of Partition Tree is higher CH. This is because the preprocessing of CH is mostly local shortest path computation while the Partition Tree involves a lot of global shortest path computation as well.

3.7 Summary

In this chapter, we propose a hierarchical graph partitioning based index named Partition Tree. It organizes the vertices of the network into a hierarchy through a series of graph partitioning processes and maintains a matrix recording pre-computed distances for each node in the tree structure. For shortest path queries, we propose a dynamic programming algorithm utilizing the partitioning topology and pre-computed distances, thus to keep network expansion in a small scale. For k-NN queries, we propose a best-first search algorithm. It utilizes the Partition Tree to compute lower bounds efficiently during search space pruning. To achieve a partitioning efficient for query processing with a given object distribution, we propose a query-oriented optimization on top of the Partition Tree. In this optimization, we develop a cost model to evaluate the influence of partitioning on query efficiency and use it to find the cost-efficient partitioning. We tested our index and query algorithms on four datasets and compared the performance with the state-of-the-art approaches. The experimental results show that the efficiency of the Partition Tree outperforms CH and Dijkstra's algorithm for shortest path queries, and outperforms IER and INE for k-NN queries. The space cost of the Partition Tree is comparable to that of CH, which has smallest space cost among all previous works. These results demonstrate that the Partition Tree is scalable to large-scale networks. Further experiments also show that our query-oriented optimization improves the performance of the baseline Partition Tree in terms of the query efficiency and the space cost for indexing.

Chapter 4

Efficient Trajectory Query Processing in Spatial Networks

4.1 Motivation

The ubiquity of GPS-embedded mobile devices and wireless communication has made it convenient for people to record their geographical positions with time stamps anytime anywhere. For instance, GPS devices equipped with the vehicles are able to record their detailed moving history. Another example is that people can log in and update their locations and activities on location-based services such as Foursquare and Twitter. One typical form of these spatio-temporal data is trajectory, which represents the history movements of a moving object. Such data carry rich information of the moving patterns of the objects in the real world, which makes trajectory analysis is a valuable issue in the area of spatio-temporal database. Representative research problems include efficient trajectory query processing[53] [4] [15] [52] [11] [50] [62] and trajectory pattern mining [10] [23] [22] [63] [30].

An important type of trajectory queries is searching trajectories by locations, aiming to find the trajectories ”nearest” to all query locations. Here we define these queries as

(aggregate) k nearest trajectories query. Given a set of query locations, k nearest trajectories query asks for the k trajectories from the dataset that have least aggregate distances to the query locations. This type of query are useful in many real world application scenarios. Imagine you are a tourist traveling in an unfamiliar city and you have several places of interest to visit, how can you decide the best route to visit these places. One solution is to find the shortest path connecting these places. But in real scenarios shortest path may not be favorable due to the practical factors such as road conditions or traffic flows. Some works [8] have shown that the routing quality can be improved by using history trajectories. Therefore, a more practical and favorable solution is to find the most popular path using the trajectories of past travelers. Since there are considerable large quantity of trajectories and they are considered to be good choices tested by people before and naturally preferred than unfamiliar paths. In this case, k nearest trajectories query becomes one of the most important query types for smart routing service providers like Foursquare.

Meanwhile, to address trajectory queries, the service providers often need to process massive quantity of data since trajectory data are accumulating constantly at a high rate due to the large number of moving objects and the widespread of location-based services. With this great volume of trajectory data, the efficiency of query processing is important, as most applications need to respond to users' queries in real time.

Most existing works are based on the assumption of Euclidean metrics [11] [62]. However in real application scenarios, the movements of objects are often constrained to pre-defined underlying networks. In this case, the indexing and query algorithms proposed for Euclidean space cannot be directly used. So in this work we explore the efficient trajectory query processing in spatial networks. Here we define the distance between a trajectory and a query location set as the sum distance of the trajectory to each query location. This is based on the assumption that if all the locations to visit are well separated and the deviation cost from each location is acceptable, people always tend to go back to the original path. And the assumption of well separated query locations can be easily

guaranteed by combining close query locations to one point in preprocessing.

In spatial networks, the distance between two points is decided by the shortest path between them. Therefore, the metric for trajectory query processing is bases on shortest path distance computation. Traditional algorithm for shortest path (distance) computation is Dijkstra's algorithm. It traverses the vertices of the network incrementally until the shortest path (distance) has been found. But the network expansion is expensive when the network is large and especially when the trajectory points are not densely distributed. So alternative approach need to be proposed to enhance the query efficiency.

To support efficient processing of location-based trajectory queries, we identify the following challenges to address:

1. Quickly filter out a set of promising candidate trajectories while avoiding large scale network expansion.

2. Compute distance between trajectories and query locations efficiently and avoid repeated computation.

3. Schedule the searches from different query locations and prune search space efficiently.

To address these challenges, we modified the indexing structure proposed above, the Partition Tree, to index the trajectories in spatial networks. It organizes the vertices in the network into a hierarchy through a series of graph partitioning process. Then information of trajectories are associated with this tree structure to facilitate efficient query processing. Based on the Partition Tree, efficient algorithms are proposed for k nearest trajectory query with a single query location and trajectory distance computation. As for k nearest trajectory search with multiple query locations, an incremental algorithm is proposed to utilize these two fundamental algorithms and prune search space effectively.

4.2 Problem Settings

4.2.1 Spatial Networks

In this work, road networks are modeled as connected and undirected planar graphs G(V, E). V is the set of all vertices, and E is the set of all edges. Each edge is assigned with a weight to represent the length of the edge. Given a path P connecting several vertices $(v_1, v_2, ..., v_n)$, the length of path P is the sum of edge weights between each pair of adjacent vertices. The path with the least length connecting two vertices is called the shortest path. Given two locations on road networks, the network distance is the length the shortest path between them. Here, we assume all locations of interest, including the query locations, are vertices on road networks for sake of simplicity.

4.2.2 Trajectory

A trajectory is a serious of sample points picturing the movements of an object. Here, we assume the sample points have all been aligned to the vertices of the graph. There are some map-matching algorithms to do this preprocessing, but it is not the focus of our work. So given the previous assumption, a trajectory τ can be seen as a path connecting a series of vertices $(p_1, p_2, ..., p_n)$ on the graph.

4.2.3 Query Definition

On road networks N, the distance between a point q and a point p is defined as the shortest path distance between them:

$$Dist(q, p) = SPDist(q, p)$$

Distance between a query location q and a trajectory $\tau = (p_1, p_2, ..., p_n)$:

$$Dist(q, \tau) = \min_{p_i \in \tau} Dist(q, p_i)$$

Distance between a set of query locations $Q = \{q_1, q_2, ..., q_m\}$ and a trajectory $\tau = (p_1, p_2, ..., p_n)$:

$$Dist(Q, \tau) = \sum_{q_i \in Q} Dist(q_i, \tau)$$

Note that this distance definition is based on the assumption that the shortest path from the trajectory to each query location and the reverse shortest path is the same. For simplicity, we only considered undirected graph in our work. Therefore, this distance assumption holds true.

Based on these distance definitions, we propose the Aggregate k Nearest Trajectories Query:

Definition 1. Aggregate k Nearest Trajectories Query Given a set of trajectories $T = \{\tau_1, \tau_2, ..., \tau_n\}$, and a set of query locations $Q = \{q_1, q_2, ..., q_m\}$, the k Nearest Trajectories Query asks for the trajectory subset $T^{'}$ containing k trajectories that have least distances from the query set Q:

$$Dist(Q, \tau_i)_{\tau_i \in T'} \leq Dist(Q, \tau_j)_{\tau_j \in T - T'}$$

For example in Figure 4.1, in the spatial network, given the query set $Q = \{v_2, v_{17}\}$, and the trajectory set $T = \{\tau_1, \tau_2, \tau_3\}$, the result of nearest trajectory query ($k = 1$) is τ_1, because $Dist(Q, \tau_1) = 11$ is smaller than that of τ_2 and τ_3.

4.3 Baseline Algorithm

First, we propose the baseline algorithm for k Nearest Trajectories Queries based on the well-known Dijkstra's algorithm. Assume no additional information is provided, the nearest neighboring trajectories can only be obtained by traversing the vertices of the network

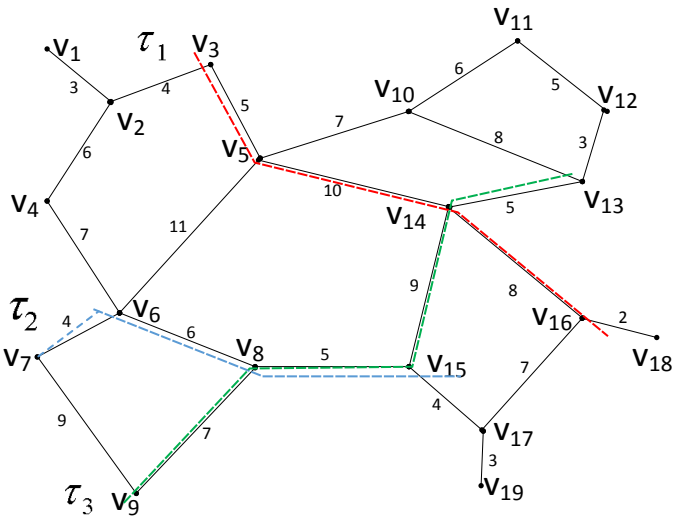


FIGURE 4.1: Trajectory Query in Spatial Networks

incrementally. We made a modification on Dijkstra's algorithms, during expansion for each vertex, we check the trajectories passing by it to update the candidate trajectory set and check the lower and upper bounds to terminate the searching. The baseline algorithm is demonstrated in Algorithm 2.

In the Incremental Network Expansion based algorithm:

1. From each query location $q_i \in Q$, browsing wavefronts are expanded in turn. Similar to Dijkstra's algorithm, for expansion starting at q_i, it select the vertex, denoted as v, with the minimum distance from q_i to visit (line 4, 5) .

2. Then we check each trajectory τ passing by current vertex v:

(1) If trajectory τ has not been scanned by wavefront from q_i before, label it as scanned by q_i (line 8).

(2) If trajectory τ is scanned by all query locations in Q by now, the distance $Dist(Q, \tau)$

can be obtained. Then τ is put into the candidate set $C_{full-scanned}$ (line 10). Otherwise, put τ in candidate set $C_{part-scanned}$ if it has not been placed in $C_{part-scanned}$ (line 12). So $C_{full-scanned}$ contains the trajectories that has been fully scanned by all query locations. And $C_{part-scanned}$ contains the trajectories that are only scanned by some query locations in Q.

3. Then global upper bound and lower bound are updated according to the current expansion status and candidate trajectory sets (line 14):

Let $C^{'}$ be the subset of $C_{full-scanned}$ that contains the k trajectories with the minimum distances, thus a global upper bound for all fully scanned trajectories can be obtained by:

$$UB = \max_{\tau_i \in C^{'}} Dist(Q, \tau_i)$$

For each trajectory τ in $C_{part-scanned}$, it is scanned by some query locations but not all location in Q, a lower bound of the distance between τ and Q is obtained by:

$$LB(Dist(Q, \tau)) = \sum_{q_i \in Q_s} Dist(q_i, \tau) + \sum_{q_i \in Q_n} d(q_i)$$

Here, Q_s is the set of query locations from which the wavefronts have scanned trajectory τ. And Q_n represents the set of query locations from which the wavefronts have not scanned R. For each q_i in Q_n, $d(q_i)$ is the shortest path length of the current wavefront from it.

Algorithm 2: Incremental Network Expansion based k Nearest Trajectory Search

Input:
Q - Query location set,
k - Required number of trajectories,
Output: R - result set
$LB = \infty, UB = \infty$
while true **do**
  for each $q_i \in Q$ **do**
    $v \leftarrow expand(q_i)$;
    for each trajectory τ passing by v **do**
      if $\tau.scan(q_i) = false$ **then**
        $\tau.scan(q_i) = true$;
      if $\tau.scan(q) = true, \forall q \in Q$ **then**
        Get $dist(Q, \tau)$, and put τ in the candidate set $C_{full-scanned}$
      else
        Put τ in the candidate set $C_{part-scanned}$
    Update global lower bound LB and upper bound UB
    if $LB > UB$ **then**
      $R \leftarrow k$ minimum values in $C_{full-scanned}$
      return result set R

Thus the global lower bound for all partly scanned and non-scanned trajectories is:

$$LB = \min_{\tau_i \in C_{part-scanned}} LB(Dist(Q, \tau_i))$$

4. Compare LB and UB:

(1) If $LB > UB$, it means the distances for all partly-scanned and non-scanned trajectories are larger than the trajectories already found, the search can be terminated. Return C' as result set R (line 17).

(2) If $LB < UB$, keep on network expansion.

The disadvantage of this baseline algorithm is that the cost of network expansion is going to be huge if the trajectories are not densely distributed in the network, since it will traverse a large portion of the network. Therefore alternative approach need to be proposed to better address the trajectory query processing in spatial networks.

4.4 Proposed Indexing Structure

Form previous discussion, we can see that the simple network expansion based algorithm is not efficient enough. Therefore we want to utilize pre-computation and indexing strategy to facilitate efficient trajectory query processing.

In Chapter 3, we studied efficient object query processing and proposed an indexing structure called the Partition Tree, which supports efficient shortest path (distance) computation and k nearest neighbor search. These algorithms address query processing with point data, namely each object to be considered is a single point in the network. Since

each trajectory is seen as a series of vertexes in the network, one solution to process trajectory query processing is to index all the trajectory points, vertexes that has trajectories passing by, and then conduct search to these trajectory points. Once a nearest neighbor point is obtained, the trajectories passing by it are labeled as candidates.

The disadvantage of this approach is that the point search may not find trajectories effectively. For example, the k nearest neighbor points could belong to the same trajectory, in which case only one trajectory is returned. Furthermore, this approach does not consider the global trajectory information with respect to multiple query locations. When a trajectory is already scanned by most of the query locations, it has more potential to be one of the target trajectories. For these trajectories, they should be evaluated with high priorities, such as efficient lower bound computation.

Motivated by this, we modified the Partition Tree to index trajectories. The minimum element to be indexed and processed is a trajectory segment. Similar as before, the Partition Tree organizes the vertices of the network into a tree structure through a series of graph partitioning process and maintains pre-computed information in distance matrices. Meanwhile, global trajectory information and segmentation information are maintained in a collection of auxiliary structures associated with the Partition Tree .

First, we present some important terms and definitions used here:

Definition 2. Graph Partitioning Given a graph $G = (V, E)$, a d-way partitioning of the graph is to divide it into d subgraphs $G_1, G_2, ..., G_d$, such that (1) $G_i = (V_i, E_i)$, (2) $\cup_{1 \leq i \leq k} V_i = V$, (3) For any two subgraphs G_i and G_j, $i \neq j, V_i \cap V_j = \emptyset$, (3) For $\forall u, v \in V_i$, if $(u, v) \in E$, then $(u, v) \in E_i$.

Definition 3. Borders Assume graph G is partitioned to subgraphs $G_1, G_2, ..., G_d$, if there exists an edge (u, v), $u \in G_i$ and $v \in G_j, i \neq j$, then u is a border of G_i and v is a border of G_j. All borders of G_i form a border set $B(G_i)$.

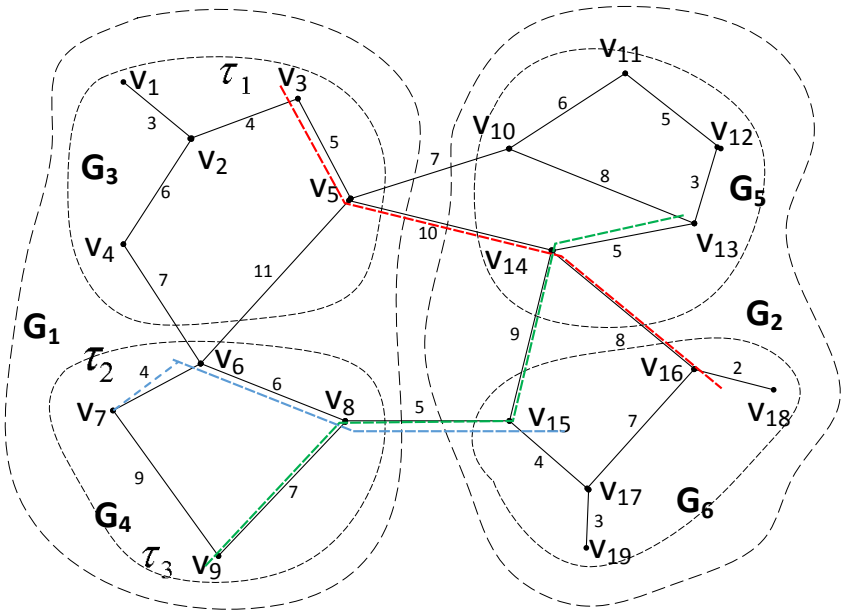


FIGURE 4.2: Hierarchical Graph Partitioning

Definition 4. Trajectory Segment Assume graph G is partitioned to d subgraphs $G_1, G_2, ..., G_d$, then a trajectory τ in the graph is divided into r segments if its vertices reside in r different subgraphs, and all vertices that belong to the same subgraph form a trajectory segment.

Next, we will give a detailed introduction of our proposed indexing technique, the Partition Tree, including the following components:

Partition Tree

To construct the Partition Tree, we conduct graph partitioning to the spatial network hierarchically until the size of the leaf subgraph is small enough. Then we use a tree structure to represent the partition hierarchy, such that each node in the tree represent a subgraph.

For example, given the graph in Figure 4.2, we conduct hierarchical partitioning to

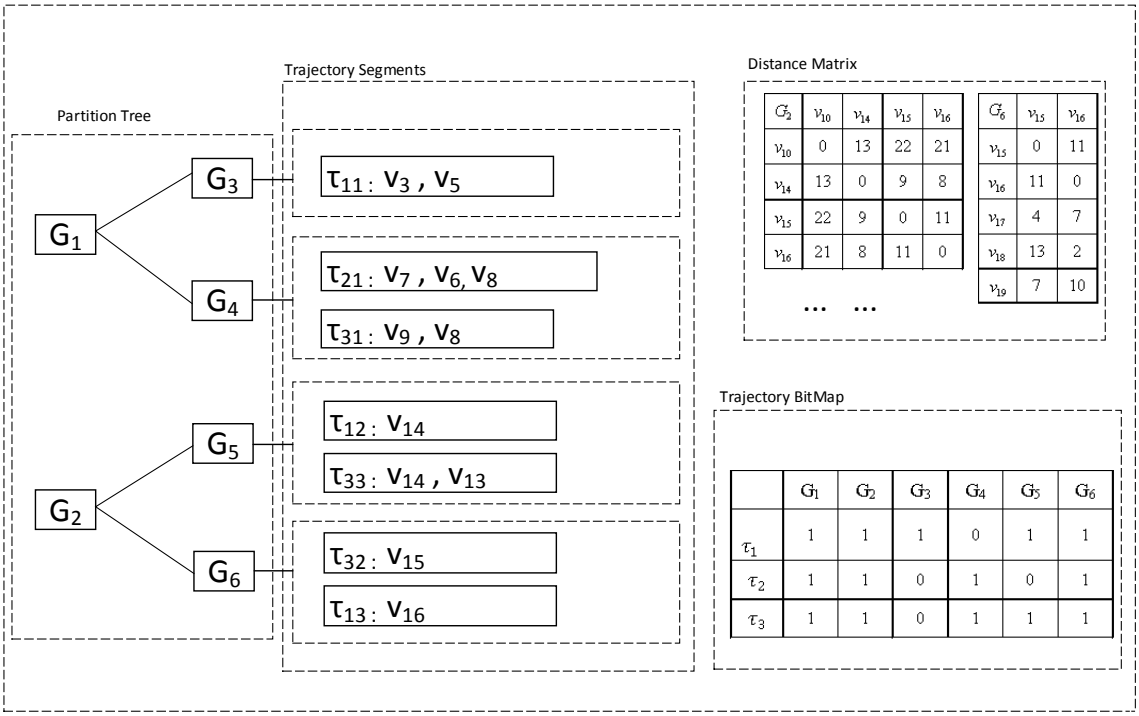


FIGURE 4.3: The Indexing Structures

it until each leaf subgraph contains no more than 5 vertices. First the original graph G is partitioned to two subgraphs G_1 and G_2. Then G_1 is further partitioned to G_3 and G_4 while G_2 is partitioned to G_5 and G_6. We use the Partition Tree in Figure 4.3 to represent this partitioning hierarchy.

Distance Matrix

For each tree node, we maintain its border set and a distance matrix recording pre-computed distances related to these borders:

- For each internal node, its distance matrix contains the distances between all pairs of borders that belong to its child nodes.

- For each leaf node, its distance matrix contains the distances between each pair of vertex and border that belongs to this subgraph.

For example, internal node G_2 has two children G_5 and G_6. G_5 has two borders v_{10} and v_{14} while G_6 has two borders v_{15} and v_{16}. So the distance matrix of G_2 contains distances between each pair of these borders, as shown in Figure 4.3. Leaf node G_6 contains vertices v_{15}, v_{16}, v_{17}, v_{18}, v_{19} and two borders v_{15} and v_{16}, so its distance matrix contains distances between each vertex and these borders, as shown in Figure 4.3.

Trajectory Segment List

For each leaf node, we maintain the trajectories passing by it in a trajectory segment list. For example, in the Parition Tree shown in Figure 4.2, leaf subgraph G_4 has two trajectories passing by it, τ_2 and τ_3. So the trajectory segment list of it consists of two segments $\tau_{21} = \{v_7, v_6, v_8\}$ and $\tau_{31} = \{v_9, v_8\}$. Trajectory τ_3 pass by three leaf subgraphs and is divided into three segments $\tau_{31} = \{v_9, v_8\}$, $\tau_{32} = \{v_{15}\}$, $\tau_{33} = \{v_{14}, v_{13}\}$.

Trajectory Bitmap

Trajectory Bitmap maintains the global information for trajectories in the Partition Tree. The bitmap is a m by d matrix of bool values, here m is the number of trajectories and d is the number of leaf subgraphs. For a trajectory τ_x, if it passes by leaf subgraph G_y, the value of the $[x, y]$-th element in the bitmap is true, otherwise is false. For example, for the partitioned graph in Figure 4.2, trajectory τ_1 passes by subgraph G_3, so the value referred by them in Trajectory Bitmap is 1 (true) as shown in Figure 4.3.

4.5 Query Processing

In this section, we will present how to address trajectories query processing efficiently, specifically k nearest trajectories query with multiple query locations. To start with, we will introduce two fundamental algorithms for trajectory distance computation and trajectory query with a single query location. Then based on these algorithms, we will elaborate how to process k nearest trajectories query with multiple query locations.

4.5.1 Trajectory Distance Computation

Distance computation is the fundamental operation for spatial query processing. In spatial networks, the distance between two point is decided by the shortest path between them. Traditional algorithm like Dijkstra's algorithm is based on network expansion and not efficient enough. To support efficient distance computation, we propose a Partition Tree based algorithm as shown Algorithm 3.

It uses a priority queue to maintain the elements(subgraphs) and their distances to q. And it always choose the element with the minimum distance to process next. Note that the definition of the distance between a subgraph and a vertex is the same as that in Chapter 3.

Meanwhile, the distance computation between a trajectory segment τ_s and query location q is defined as:

$$Dist(q, \tau_s) = \min_{p_i \in \tau_s} Dist(q, p_i)$$

Algorithm 3: Distance Computation between query location q and trajectory τ

Input:
q - query location,
τ - trajectory,
P - the Partition Tree
Output: $Dist(q, \tau)$
$LB = \infty, UB = \infty$
Priority queue $Q = \phi$
Put (P.root, 0) into priority queue Q ;
while Q is not empty **do**
    e = $Q.dequeue()$;
    LB = $Q.mindist()$;
    if e is leaf node **then**
        $\tau_s \leftarrow$ segment of τ in leafnode e
        if $UB > Dist(q, \tau_s)$ **then**
            $UB = Dist(q, \tau_s)$;
    else
        for each child node c of node e **do**
            if c contains any segments of τ **then**
                put $(c, Dist(q, c))$ into Q;
    if $UB < LB$ **then**
        return $Dist(q, \tau) = UB$;

Note that the distance computation, including distance between a subgraph and the query location and distance between a trajectory segment and the query location, is conducted by the dynamic programming algorithm described in section 3.4 of Chapter 3.

First the algorithm puts the root of the Partition Tree into the priority queue (line 5). Then it will dequeue the first element in the priority queue and update the lower bound LB (line 7, 8). If it is a leaf node, then the distance between the trajectory segment τ_s of τ and q is computed. If this distance is lower than the upper bound UB, UB will be updated (line 12). Now it will check the lower bound and upper bound, if $UB < LB$, the search is terminated(line 18). Otherwise, it will dequeue next element in the priority queue and keep processing.

4.5.2 k Nearest Trajectories Query

The simplest case of aggregate k nearest trajectories query is when there is only one single query location, namely k nearest trajectories to query location q. Based on the distance computation algorithm proposed above and our indexing structure, we propose an best-first algorithm as shown Algorithm 4.

It also uses a priority queue to maintain the elements(subgraphs) and choose the element with the minimum distance to process next during trajectory search. Meanwhile it uses a global lower bound and upper bound to decide whether to terminate the searching. The global lower bound represents the lower bound of distances between all un-scanned trajectories and q, which is the distance of the minimum element in Q. The global upper bound represent the upper bound of distances of the scanned trajectories, which can be obtained by the k-th minimum value of all distances between candidate trajectories to q.

Algorithm 4: k nearest trajectories to query location q

1: Input:
q - query location,
k - required number of trajectories,
T - trajectory set,
P - Partition Tree
2: Output: R - result set
3: Result set $R = \phi$, candidate set $C = \phi$
4: $LB = \infty, UB = \infty$
5: Q = {(T.root, 0)} ;
6: **while** $|R| < k$ and Q is not empty **do**
7: e = $Q.dequeue()$;
8: LB = $Q.mindist()$;
9: **if** e is leaf node **then**
10: **for** each trajectory segment τ_s in leafnode e **do**
11: put $(\tau_s, Dist(\tau_s, q))$ in candidate set C;
12: $UB = C.kmindist()$;
13: **if** $UB < LB$ **then**
14: return $R \leftarrow$ the k trajectories in C with minimum values;
15: **else**
16: **for** each child node c of node e **do**
17: **if** c contains any trajectory segments **then**
18: put $(c, dist(q, c))$ into Q;

First, the root of the Partition Tree is put into the priority queue Q. Then the algorithm dequeues the minimum element e in Q (line 7) and update the global lower bound(line 8). If e represents a leaf subgraph, then for trajectory segment τ_s in it, the algorithm calculate

the distance $Dist(q, \tau_s)$ and put $(\tau_s, Dist(\tau_s, q))$ into candidate set C. Then it update the global upper bound UB(line 12). If $UB < LB$ by now, the search is terminated (line 14). If e represents a non-leaf subgraph, then for each child node c, if it contains any trajectory segments, the algorithm will put $(c, dist(q, c))$ into Q.

4.5.3 Aggregate k Nearest Trajectories Query

Given a set of trajectories $T = \{\tau_1, \tau_2, ..., \tau_n\}$, and a set of query locations $Q = \{q_1, q_2, ..., q_m\}$, to find the k nearest trajectories to Q.

1. For each query location q_i, retrieve the λ nearest trajectories and put them into candidate set C_i:

$$C_1 = \{\tau_1^1, \tau_1^2, ..., \tau_1^\lambda\}$$
$$C_2 = \{\tau_2^1, \tau_2^2, ..., \tau_2^\lambda\}$$
$$...$$
$$C_m = \{\tau_m^1, \tau_m^2, ..., \tau_m^\lambda\}$$

Then the trajectories scanned by all query locations form a candidate set C_s:

$$C_s = C_1 \cap C_2 \cap ... \cap C_m$$

For each trajectory τ_x in C_s, its distance to each query location q_i is known. Thus we can get $Dist(Q, \tau_x)$. If C_s contains no less than k trajectories, let C' be the set containing the k trajectories with the minimum distances, we can have an upper bound:

$$UB = \max_{\tau_i \in C'} Dist(Q, \tau_i)$$

And all partly scanned trajectories form a candidate set C_p:

$$C_p = C_1 \cup C_2 \cup ... \cup C_m - C_s$$

For each trajectory τ_x inside it, the lower bound of its distances to Q is:

$$LB(Dist(Q, \tau_x)) = \sum_{\tau_x \in C_i} Dist(q_i, \tau_x) + \sum_{\tau_x \notin C_i} Dist(q_i, \tau_i^{\lambda})$$

Then the lower bound for all partly scanned trajectories and non-scanned trajectories is:

$$LB = \min_{\tau_i \in C_p} LB(Dist(Q, \tau_i))$$

This is because the lower bound for all non-scanned trajectories is:

$$LB_n = \sum_{q_i \in Q} Dist(q_i, \tau_i^{\lambda})$$

It is surely larger than LB of the partly scanned trajectories.

Algorithm 5: k nearest trajectories to query location q

1: Input:
   Q - query location set,
   k - required number of trajectories,
   T - trajectory set
2: Output: R - result set
3: Result set $R = \phi$
4: Candidate set $C_s = \phi$
5: Candidate set $C_p = \phi$
6: $LB = \infty, UB = \infty$
7: $\lambda = k, \Delta = k$
8: **while** true **do**
9: **for** each query location q_i in Q **do**
10: $C_i \leftarrow \lambda$-$NN(q_i)$;
11: $C_s \leftarrow C_1 \cap C_2 \cap ... \cap C_m$;
12: **if** $|C| > k$ **then**
13: compute UB for all fully scanned trajectories;
14: compute LB_n for all non-scanned trajectories;
15: **if** $UB < LB$ **then**
16: break;
17: $\lambda = \lambda + \Delta$;
18: return R;

2. Compare LB_n and UB: (1) If $LB > UB$, it means the nearest trajectories are contained in C_s, the search can be terminated. (2) If $LB < UB$, increase λ by $\Delta\lambda$.

Note that λ and the incremental element Δ are both initiated as k (line 7). Other

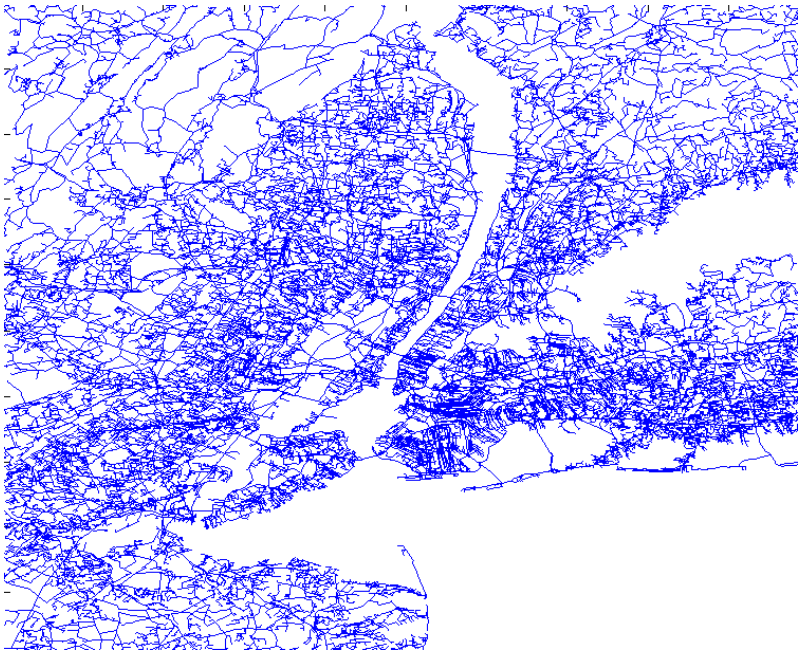


FIGURE 4.4: Trajectory Distribution

values can be assigned to them. In the experiments, the assigned value k results in the performance as good as we expected.

4.6 Experiments

4.6.1 Experiments Setup

We conduct our experiments on the road network of New York obtained from [3], which contains 264,346 vertices and 733,846 edges. since there is no real life trajectory dataset available for New York, synthetic trajectory data were used. Figure 4.4 shows the distribution of the synthetic trajectory dataset. All approaches were implemented with C++ and all experiments were run on a 64-bit windows machine with Intel 3.40GHz CPU and a 16GB RAM.

TABLE 4.1: Parameter Settings

parameter	values
k - required number of trajectories	5, **10**, 15, 20, 25
$\|Q\|$ - number of query locations	2, 3, 4, **5**, 6, 7, 8
$\|T\|$ - trajectory dataset scale	6k, 7k, **8k**, 9k, 10k

Each trajectory τ in the experiments is represented as a sequence of vertexes $\tau = \{v_1, ..., v_m\}$, where m is the length of the trajectory. In addition, for each vertex v_i in the road network, we maintain a trajectory list $L_i = \{\tau_1, ..., \tau_m\}$ such that each trajectory in L_i passes by v_i.

To evaluate the performance of the baseline algorithm and PTree based algorithm, we randomly generated 1000 query sets and get the average processing time. The parameter settings and default values are summarized in Table 4.1. The main metric we adopt for measuring the performance is the query processing time since it represents how efficient each query is addressed. Meanwhile, we evaluate the space cost of indexing as it is the major concern for indexing here.

4.6.2 Evaluation Results

Different Number of Query Locations

In the first set of experiments, we investigate the effect of number of query locations ($|Q|$) on the query efficiency. In real life application scenarios, the number of query locations is relatively small as users seldom put in tens of locations in a query at the same time, so it is practical to assume that $|Q|$ is smaller than 10. In this set of experiments, we set $|Q|$

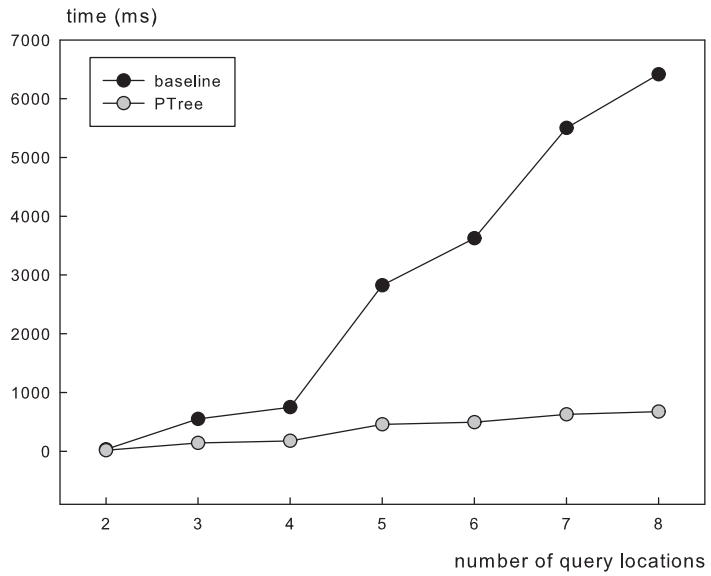


FIGURE 4.5: Effect of $|Q|$

to range from 2 to 8. Meanwhile we fix the required number of trajectories (k) to 10 and number of trajectories in the dataset ($|T|$) to 8,000. For each value of $|Q|$, we evaluate 1000 queries randomly generated and get the average processing time and the results are shown in Figure 4.5. We can see that the processing time of our proposed algorithm based on Partion Tree (PTree) is superior to the baseline algorithm. When the number of query locations is increasing from 2 to 8, the efficiency of the baseline algorithm suffers dramatically while the Partition Tree based algorithm preserves its efficiency.

Different Number of Required Trajectories

In the second set of experiments, we evaluate the effect of required number of trajectories (k) on the query efficiency. We set k to range from 5 to 25. The number of query locations is fixed at 5 and the trajectory dataset size at 8,000. Similarly as before for each value of k, we evaluate 1000 queries and get the average processing time and the result is

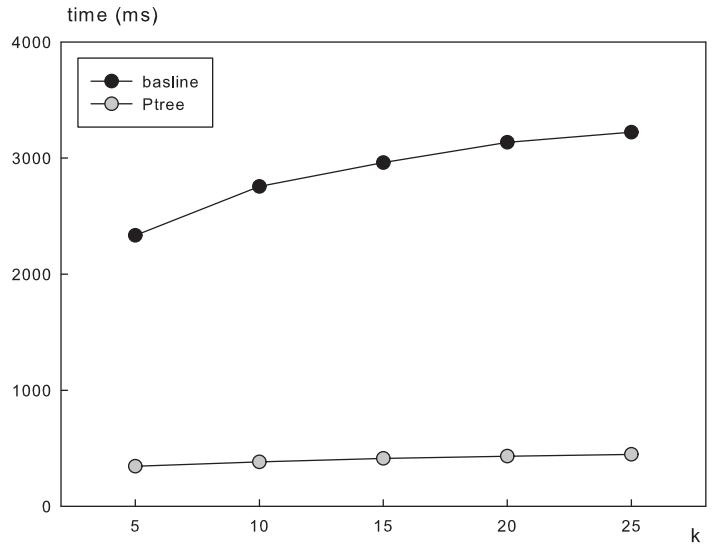


FIGURE 4.6: Effect of k

shown in Figure 4.6. When k is increasing from 5 to 25, the performance of the baseline algorithm suffers greatly. On the other hand, our Partition Tree based algorithm preserves its efficiency for different k.

Different Scale of Trajectory Dataset

In this set of experiments, we investigate the scalability of our proposed indexing and algorithms on different trajectory dataset. We evaluate the performance of our algorithms with the trajectory number ranging from 6,000 to 10,000. The number of query locations ($|Q|$) is fixed at 4 and the required number of trajectories (k) is fixed at 10. The experiments result is shown in Figure 4.7. The result demonstrates that our algorithm is scalable when the number of trajectories is increasing. Note that since the New York road network is relatively small since it only has 264,346 vertexes, 10,000 is a large number for trajectories. Form the result it is safe to conclude that our algorithm is scalable to larger road networks and trajectory dataset.

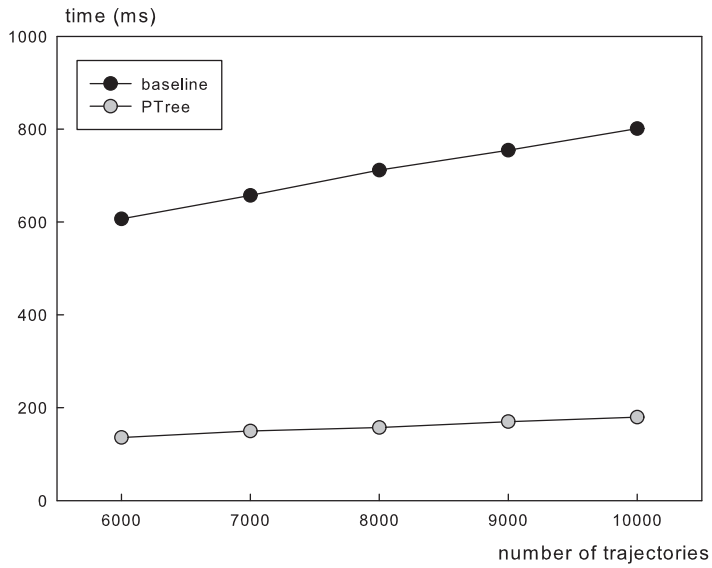


FIGURE 4.7: Effect of $|T|$

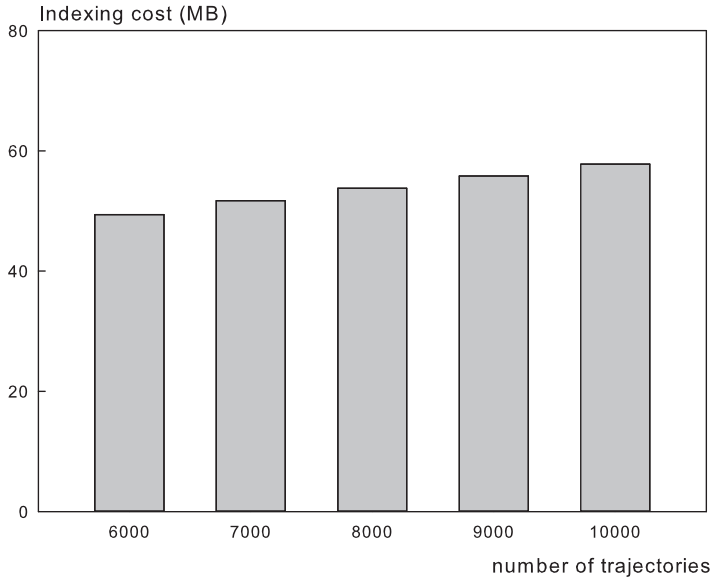


FIGURE 4.8: Space Cost of Indexing

Indexing Cost

In addition of processing time, we also evaluate the space cost of indexing. Figure 4.8 shows the indexing cost with the number of trajectories ranging from 6,000 to 10,000. The space cost increases at a low rate when the number of trajectories is increasing. But for dataset with up to 10,000 trajectories, the space cost is still under 60 MB. This means that the Partition Tree is applicable to large scale networks, and much more space efficient compared with other indexing technique such like SILC [46][44].

4.7 Summary

In this chapter, we investigate efficient trajectory query processing in spatial networks. We proposed an indexing technique, the Partition Tree, for trajectories in spatial networks. It organizes the vertexes of the network into a hierarchy through a series of graph partitioning process. Then pre-computed distances and global trajectory information are associated with this hierarchy to facilitate efficient query processing. For the most challenging type of trajectory queries, k nearest trajectories query, we propose an incremental k nearest trajectory algorithm that avoids large scale network expansion and prunes the search space efficiently. We conducted extensive experiments on real world dataset. And the experimental results demonstrate that our proposed algorithm has superior performance over the baseline algorithm and is scalable to large trajectory dataset. Meanwhile, the indexing cost is low so that it is applicable for large networks and trajectory dataset.

Chapter 5

Conclusions

As a frontier study, this thesis investigated the efficient query processing in spatial networks. Specifically, it focuses on two major categories of spatial queries: the object queries and the trajectory queries. As mentioned in Chapter 1, three key challenges are identified for efficient query processing in spatial networks:

- **Avoid large scale network expansion:** Shortest path distance (network distance) computation is the basis of query processing in spatial networks. Most existing works are based on either computing network distance between the query location and an object on-line, or utilizing the index structures. On-line distance computation usually adopts Dijkstra's algorithm, which retrieves the objects in ascending order of their distances to the query location. But this performs poorly when the objects are not densely distributed in the network because a large portion of the network will be traversed. The algorithms based on indexing structures can filter out a candidate set first during search. But the distance computation between the query location and candidates still need to traverse the network if no alternative solution is provided. So large scale network expansion should be avoided to assure the query

efficiency for spatial networks.

- **Prune search space efficiently:** In Euclidean space, traditional algorithms utilize indexing structure like R-tree to retrieve the objects and prune the search space. For example, the depth-first algorithm [43] and best-first algorithm [19] for k-NN query are both based on R-tree. They take advantage of metrics based on R-tree to order and prune the search tree. In spatial networks, the metrics for such pruning power should be carefully designed. The indexing structure we propose should support efficient search space pruning and distance computation at the same time.

- **Control indexing cost:** For the real-world applications, the efficiency of query processing is crucial for their service quality but a moderate indexing cost is also very important. Compared with non-line network expansion, approaches based on indexing structures like SILC [46][44] are quite efficient but they have a huge space cost for indexing. There is a trade-off between indexing cost and query efficiency. Thus, what materialization strategy to take and how to organize the materialized information is important.

The approach of this thesis towards these challenges is to propose an indexing technique that takes advantage of the network topologies and utilizes the pre-computation efficiently. The proposed indexing structure, the Partition Tree, organizes the vertexes of the network into a hierarchy through s series of graph partitioning. Meanwhile, pre-computed distances and global object (trajectory) information are associated with this hierarchy to facilitate efficient query processing. Based on this indexing technique, the challenges are addressed as follows:

- To avoid large scale network expansion, an efficient dynamic programming algorithm is proposed for shortest path and distance query processing. This algorithm takes advantage of the pre-computed distances associated with each node in the tree

structure. It only processes the important vertexes with respect to the searching hierarchy and all the distances needed are already pre-computed, thus it doesn't need to conduct network expansion to get the shortest path distance.

- To prune the search space efficiently, the search process is organized by utilizing the indexing structure. Best first algorithms are proposed for the nearest object search and trajectory search. Upper and lower bounds are carefully designed to prune the search space effectively and efficiently.

- To control the space cost of indexing, the network is organized into a hierarchy and the pre-computation is only conducted to vertexes important for the query processing. The complexity analysis shows that this cost is much smaller than all-pairs pre-computation. Meanwhile, the experimental results also suggest that the proposed indexing technique has small space consumption and is scalable to large networks and dataset.

For each work, extensive experiments are conducted to demonstrate the performance of our proposed approach. These experiments show that the proposed indexing method and algorithms have superior performance over state-of-the-art approaches and are scalable to large scale networks.

References

[1] http://algo2.iti.kit.edu/routeplanning.php.

[2] http://www.cs.utah.edu/ lifeifei/SpatialDataset.html.

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

[4] L. C. 0002, M. T. zsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD Conference*, pages 491–502, 2005.

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

[6] H. Bast, S. Funke, and D. Matijevic. Transit: ultrafast shortest path queries with linear time preprocessing. *9th DIMACS Implementation Challenge [1]*, 2006.

[7] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *ALENEX*, 2007.

[8] V. Ceikute and C. S. Jensen. Routing service quality - local driver behavior versus routing services. In *MDM (1)*, pages 97–106, 2013.

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

[10] Z. Chen, H. T. Shen, and X. Zhou. Discovering popular routes from trajectories. In *ICDE*, pages 900–911, 2011.

[11] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng, and X. Xie. Searching trajectories by locations: an efficiency study. In *SIGMOD Conference*, pages 255–266, 2010.

[12] P. Cudr-Mauroux, E. W. 0002, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.

[13] K. Deng, X. Zhou, H. T. Shen, S. Sadiq, and X. Li. Instance optimal query processing in spatial networks. *The VLDB Journal*, 18(3):675–693, 2009.

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

[15] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Nearest neighbor search on moving object trajectories. In *SSTD*, pages 328–345, 2005.

[16] A. V. Goldberg and C. Harrelson. Computing the ahortest path: A* meets graph theory. In *SODA*, pages 156–165, 2005.

[17] A. Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[18] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:41–72, 2009.

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

[20] S. Ilarri, E. Mena, and A. Illarramendi. Location-dependent query processing: Where we are and where we are heading. *ACM Computing Surveys (CSUR)*, 42(3):12, 2010.

[21] C. S. Jensen, J. Kolářvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, pages 1–8. ACM, 2003.

[22] H. Jeung, H. T. Shen, and X. Zhou. Convoy queries in spatio-temporal databases. In *ICDE*, pages 1457–1459, 2008.

[23] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. 2010.

[24] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical optimization of optimal path finding for transportation applications. In *Proceedings of the fifth international conference on Information and knowledge management*, pages 261–268. ACM, 1996.

[25] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *Knowledge and Data Engineering, IEEE Transactions on*, 10(3):409–432, 1998.

[26] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *Knowledge and Data Engineering, IEEE Transactions on*, 14(5):1029–1046, 2002.

[27] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[28] B. W. Kemighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49:291307, 1970.

[29] 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.

[30] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD Conference*, pages 593–604, 2007.

[31] 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.

[32] K. C. Lee, W.-C. Lee, B. Zheng, and Y. Tian. Road: a new spatial object search framework for road networks. *Knowledge and Data Engineering, IEEE Transactions on*, 24(3):547–560, 2012.

[33] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[34] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up dijkstras algorithm. In *Experimental and Efficient Algorithms*, pages 189–202. Springer, 2005.

[35] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)*, 9(1):38–71, 1984.

[36] S. Nutanong and H. Samet. Memory-efficient algorithms for spatial network queries. pages 649–660, 2013.

[37] N. J. N. P. E. Hart and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[38] 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.

[39] D. Pfoser, C. S. Jensen, Y. Theodoridis, et al. Novel approaches to the indexing of moving object trajectories. In *Proceedings of VLDB*, pages 395–406, 2000.

[40] D. S. R. Geisberger, P. Sanders and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Workshop on Experimental Algorithms*, pages 319–333, 2008.

[41] D. S. R. Geisberger, P. Sanders and C. Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012.

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

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

[44] 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.

[45] P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *14th European Symposium on Algorithms (ESA)*, pages 804–816, 2006.

[46] 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.

[47] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 652–663. IEEE, 2009.

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

[49] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *Proceedings of the VLDB Endowment*, 2(1):1210–1221, 2009.

[50] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis. User oriented trajectory search for trip recommendation. In *EDBT*, pages 156–167, 2012.

[51] S. Shekhar, A. Fetterer, and B. Goyal. Materialization trade-offs in hierarchical shortest path algorithms. In *Advances in Spatial Databases*, pages 94–111. Springer, 1997.

[52] M. R. Vieira, P. Bakalov, and V. J. Tsotras. Querying trajectories using flexible patterns. In *EDBT*, pages 406–417, 2010.

[53] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

[54] D. Wagner and T. Willhalm. *Geometric speed-up techniques for finding shortest paths in large sparse graphs*. Springer, 2003.

[55] D. Wagner and T. Willhalm. Speed-up techniques for shortest-path computations. In *STACS 2007*, pages 23–36. Springer, 2007.

[56] 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.

[57] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 631–642. IEEE, 2005.

[58] J. Yuan, Y. Zheng, and X. Xie. Discovering regions of different functions in a city using human mobility and pois. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 186–194. ACM, 2012.

[59] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *GIS*, pages 99–108, 2010.

[60] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 443–454. ACM, 2003.

[61] B. Zheng, J. Xu, W.-C. Lee, and L. Lee. Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services. *The VLDB JournalThe International Journal on Very Large Data Bases*, 15(1):21–39, 2006.

[62] K. Zheng, S. Shang, N. J. Yuan, and Y. Yang. Towards efficient search for activity trajectories. In *ICDE*, pages 230–241, 2013.

[63] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang. On discovery of gathering patterns from trajectories. In *ICDE*, pages 242–253, 2013.

[64] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma. Understanding mobility based on gps data. In *Proceedings of the 10th international conference on Ubiquitous computing*, pages 312–321. ACM, 2008.

[65] 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.

[66] Y. Zheng and X. Zhou. Computing with spatial trajectories. 2011.