THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

A STUDY ON TIME-DEPENDENT REACHABILITY AND ROUTE SCHEDULING IN ROAD NETWORK

Lei Li

Master of Computer Science and Engineering

A thesis submitted for the degree of Doctor of Philosophy at

The University of Queensland in 2018

School of Information Technology & Electrical Engineering

Abstract

For thousands of years, humans have been innovating new technologies to plan their journeys: from looking up the starry sky, to depending on the magnetic compass; from referring to precious ancient maps, to interacting with locals for nearby information. However, these approaches are either inaccurate or hard to grasp by ordinary people. Thanks to the recent rapid development of online map services and GPS devices, we are able to identify where we are on earth, find any place we want to go, and retrieve a route to it. Although it is convenient and fast enough for basic uses, it is still far from optimal. For starters, most systems just provide a shortest path without considering the traffic condition. Secondly, some systems consider the current traffic condition to provide an estimated travel time. However, the lack of estimation for future traffic condition cannot help us plan the travel ahead of time. To make the things worse, the computation that takes traffic condition into consideration grows slower as the planning time interval and the distance grow longer. Therefore, we study how to plan a travel that considers traffic information from the following aspects.

The first one is the *reachability problem*. A road network, or a map, is essentially a graph with nodes representing intersections and edges representing roads. For a well-maintained map, the nodes are reachable to each other. However, this is not always the case when we obtain our map data. For example, the nodes along the boundary might not reach the other nodes on the map. We propose a *High-Dimensional Graph Dominance Drawing* approach to answer if one node can reach another quickly on large graphs. In fact, it takes only constant time to answer reachability query in road network. We run our algorithm on various of graph structures with different configurations to fully test its performance. The results help us have a deeper understanding on the reachability problem.

The second one is the *speed profile generation*. A speed profile is a set of functions that return the travel time of any road by providing any departure time. Many existing works just assume such a speed profile exists, or generate one synthetically. Other real-life applications tend to use real-time data from sensors monitoring major roads, which is expensive to deploy and unable to cover a large area. In this work, we use historical trajectories of taxis to generate a speed profile. It involves map-matching, speed data collecting, missing value estimation and compression. By using different speed profile for different types of day, we can provide route scheduling that satisfying user's need. Extensive experiments show that our speed profile is accurate and space efficient.

The third one is the *minimal on-road travel time route scheduling (MORT)*. This is a general form of all the single criteria path problems. All the existing path finding problem does not allow

waiting on some vertices along the route, nor can they benefit from it. We extend this problem by allowing waiting. In this way, the total travel time is made up of two parts: on-road travel time and waiting time. Now we are able to find a route with minimum on-road time. Such query is needed by logistics company and tourists. The challenging part of the routing problem lies in the computational complexity when determining if it is beneficial to wait on specifying the parking places and the corresponding time of waiting to maximize the benefit. To cope with this challenging problem, we propose two efficient algorithms using minimum on-road travel cost function to answer the query. We further introduce several approximation methods to speed up the query answering. Experiments show that our method is more efficient and accurate than baseline approaches extended from the existing path planning algorithms.

The last one the *time-dependent 2-hop labeling*.The time-dependent path algorithms are still slow because the fastest path problem's complexity is $\mathit{\Omega}(T(|V|\log|V| + |E|))$, where T is the number of turning points in the result's function and it is large in real-life query. Therefore, we extend the *2-Hop labeling* index to time-dependent environment. Besides, our approach can also answer time-dependent reachability query. However, even for a static graph, the label size is already at least $\mathit{\Omega}(|V||E|^{\frac{1}{2}})$, so it would be much bigger if we extend it directly. So we propose a partition-based framework to adapt to the real-life road network. By breaking the query into three parallel parts, we are able to speed up any time-dependent query for thousands of times. To further reduce the label size and speed up query answering, we propose an approximation approach with the worst case error bounded. Comparison with the existing on-line search indexes shows that our time-dependent 2-hop can answer queries much faster, with an acceptable index size.

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, financial support 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 higher degree by research candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

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

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis and have sought permission from co-authors for any jointly authored works included in the thesis.

Publications during candidature

- Li, Lei, Wen Hua, and Xiaofang Zhou. *HD-GDD: high dimensional graph dominance drawing approach for reachability query.* World Wide Web 20, no. 4 (2017): 677-696.

- Li, Lei, Xiaofang Zhou, and Kevin Zheng. *Finding Least On-Road Travel Time on Road Network.* In Australasian Database Conference, pp. 137-149. Springer International Publishing, 2016.

- Li, Lei, Wen Hua, Xingzhong Du, and Xiaofang Zhou. *Minimal on-road time route scheduling on time-dependent graphs.* Proceedings of the VLDB Endowment 10, no. 11 (2017): 1274-1285.

- Hosseini, Saeid, and Lei Thor Li. *Point-of-interest recommendation using temporal orientations of users and locations.* In International Conference on Database Systems for Advanced Applications, pp. 330-347. Springer, Cham, 2016.

- Li, Lei, Kai Zheng, Sibo Wang, Wen Hua, and Xiaofang Zhou. *Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory.* The VLDB Journal (2018): 1-25.

- Zhou, Xiaofang, and Lei Li. *Spatio-Temporal data - Trajectories*. In Encyclopedia of Big Data Technologies, 2018.

Publications included in this thesis

Li, Lei, Wen Hua, and Xiaofang Zhou. *HD-GDD: high dimensional graph dominance drawing approach for reachability query.* World Wide Web 20, no. 4 (2017): 677-696. -incorporated as Chapter 3

Contributor	Statement of contribution
Lei Li	Conception and design (80%) Analysis and interpretation(70%) Drafting and production(80%)
Wen Hua	Conception and design (10%) Analysis and interpretation(15%) Drafting and production(10%)
Xiaofang Zhou	Conception and design (10%) Analysis and interpretation(15%) Drafting and production(10%)

Li, Lei, Xiaofang Zhou, and Kevin Zheng. *Finding Least On-Road Travel Time on Road Network.* In Australasian Database Conference, pp. 137-149. Springer International Publishing, 2016. -incorporated as Chapter 5.

Contributor	Statement of contribution
Lei Li	Conception and design (70%) Analysis and interpretation(80%) Drafting and production(80%)
Xiaofang Zhou	Conception and design (20%) Analysis and interpretation(10%) Drafting and production(10%)
Kai Zheng	Conception and design (10%) Analysis and interpretation(10%) Drafting and production(10%)

Li, Lei, Wen Hua, Xingzhong Du, and Xiaofang Zhou. *Minimal on-road time route scheduling on time-dependent graphs.* Proceedings of the VLDB Endowment 10, no. 11 (2017): 1274-1285. -incorporated as Chapter 5.

Contributor	Statement of contribution
Lei Li	Conception and design (70%) Analysis and interpretation(70%) Drafting and production(70%)
Wen Hua	Conception and design (10%) Analysis and interpretation(10%) Drafting and production(10%)
Xingzhong Du	Conception and design (10%) Analysis and interpretation(10%) Drafting and production(10%)
Xiaofang Zhou	Conception and design (10%) Analysis and interpretation(10%) Drafting and production(10%)

Li, Lei, Kai Zheng, Sibo Wang, Wen Hua, and Xiaofang Zhou. *Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory.* The VLDB Journal (2018): 1-25. -incorporated as Chapter 4 and Chapter 5.

Contributor	Statement of contribution
Lei Li	Conception and design (60%) Analysis and interpretation(80%) Drafting and production(80%)
Kai Zheng	Conception and design (10%) Analysis and interpretation(5%) Drafting and production(5%)
Sibo Wang	Conception and design (10%) Analysis and interpretation(5%) Drafting and production(5%)

Wen Hua	Conception and design (10%) Analysis and interpretation(5%) Drafting and production(5%)
Xiaofang Zhou	Conception and design (10%) Analysis and interpretation(5%) Drafting and production(5%)

Manuscripts included in this thesis

Li, Lei, Sibo Wang, and Xiaofang Zhou. *Time-Dependent 2-Hop Labeling* -incorporated as Chapter 6.

Contributor	Statement of contribution
Lei Li	Conception and design (80%) Analysis and interpretation(80%) Drafting and production(80%)
Sibo Wang	Conception and design (15%) Analysis and interpretation(15%) Drafting and production(10%)
Xiaofang Zhou	Conception and design (5%) Analysis and interpretation(5%) Drafting and production(10%)

Contributions by others to the thesis

For all the published research work included in this thesis, Prof. Xiaofang Zhou, as my principal advisor, Prof. Kai Zheng and Dr. Wen Hua, as my associate advisors, have provided very helpful insight into the overall as well as the technical details and research problems; guidance for problem formulation as well as constructive comments and feedback. They also assisted with both the refinement of the idea and the pre-submission edition.

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

None.

Research Involving Human or Animal Subjects

No animal or human participants were involved in this research.

Acknowledgments

I wish to express my sincere appreciation to those who have contributed to this thesis and supported generously me during my PhD journey.

First of all, I am extremely grateful to my principal supervisor, Prof. Xiaofang Zhou, for his generous support, and his valuable and in-depth guidance for my PhD study, research and life. With all those heated brainstorming, many new ideas are created, polished, deeply analyzed and finally well presented. It is by his guidance that I have learned how to do the research and how to be a better man. Those I have learned from him are beneficial for my whole life.

I am also deeply grateful for my associate advisor Prof. Kai Zheng, who helps me to sort out all those messy details and bring the ideas down to ground, especially on my early stage. It would be hard to have all my works done without his persistent and generous help.

I am also sincerely grateful to my second associate advisor Dr. Wen Hua, who has worked with me tightly during this journey. It is an amazing experience to grow together, from the time when we were both tutors to now I am a tutor of her. I have learned a lot from her, especially on the presentation skills and paper writing.

I am also lucky to spend the last three years with all the great people from DKE group. It is my pleasure to specially acknowledge Prof. Shazia Sadiq, Prof. Helen Huang, Prof. Xue Li, Prof. Yufei Tao, Dr. Mohamed Sharaf, and Dr. Hongzhi Yin. In my daily work I have been blessed with a friendly and cheerful group of fellow students. Big thanks to Dr. Han Su, Dr. Haozhou Wang, Dr. Jialong Han, Dr. TieKe He and Dr. Weiqing Wang, you helped me a lot on my early life in Brisbane. Also special thanks to Dr. Xingzhong Du, Dr. Bolong Zheng and Dr. Junhao Gan, we have spent so many great time before the white board, around the lake and around different tables. Life would be colorless without you. Also great thanks to all the members of ‘Common Room Lunch Group’, together we keep our lab full of vigor and happiness.

Last but not least, my deepest gratitude goes to my family. My parents raise me up to who I am and support me to as far as I can achieve, without a complaint of my long leave. My girl friend Zoe gave up her life to company and support me all the way from Perth to Brisbane. It is your direct helps that I could devote myself fully into research.

Financial support

No financial support was provided to fund this research.

Keywords

reachability, trajectory, road network, time-dependent graph, fastest path, index, 2-hop

Australian and New Zealand Standard Research Classifications (ANZSRC)

ANZSRC code: 080604, Database Management, 80%

ANZSRC code: 080201, Analysis of Algorithms and Complexity, 20%

Fields of Research (FoR) Classification

FoR code: 0806, Information Systems, 80%

FoR code: 0802, Computation Theory and Mathematics, 20%

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we give a brief introduction of the research in this thesis, including background, problem statements, contributions, and organization of the thesis.

1.1 Background

Thanks to the fast development of the mobile network, free map services and the widespread using of GPS devices, traveling on earth has never been easier. A variety of navigation applications can direct people to travel by bike, car or public transportation to wherever they want based on their current locations. What behind these navigation applications are the reachability testing, traffic condition prediction, and route scheduling in road network.

There are two types of data used in route query answering: road network and traffic condition. Essentially, a road network, or a map, is a graph. Due to geographic property, or the quality the map data, not all the vertices are reachable to each other. Therefore, the *reachability* query could serve as a preprocessing method, or a testing query to validate the reachability before answering the actual path query. On the other hand, the traffic condition of each road can be represented in a linear piecewise function, with the travel time depending on the departure time. A set of such functions that cover the whole road network is called a speed profile, which could be generated from real-time traffic monitoring systems. However, such an expensive system is impossible to cover the whole road network. Hence, we take advantage of the massive trajectory data to generate a speed profile, which could be used for traffic prediction.

The route scheduling problem is the core of this work. It has two levels: ground truth result computation and fast query answering. For the first one, we propose a *minimum on-road time route scheduling algorithm (MORT)* that not only can reduce the actual traveling time on-road with the help of waiting on some vertices, but also can answer all the existing time-dependent fastest path queries by simply changing the configurations. For the fast query answering, we propose a *time-dependent 2-hop labeling* by extending the *2-hop labeling* approach to the time-dependent environment. It can answer both the fastest travel time query and time-dependent reachability.

In summary, this thesis provides a solution of road network route scheduling from data preprocessing (reachability problem) and data preparation (speed profile generation), to actual route scheduling (*MORT*) and fast time-dependent query answering (*time-dependent 2-hop*).

1.2 Problem Statement

1.2.1 Graph Reachability Problem

A reachability query on a directed graph $G(V, E)$ (V is the vertex set and E is the edge set) tests if there exists a path from one vertex to another. Although this query can be answered by a simple *BFS/DFS*, it is not efficient to traverse on the whole graph each time a query comes, especially when a large number of queries come in a short time. So various index structures are proposed to speed up the reachability query answering. The most straightforward approach pre-computes the reachability relation for all the vertex pairs and stores them in memory such that the query answering time is constant. But it takes $O(|V|^2)$ time to compute and store, which is impossible for a graph even not too big. Therefore, some works try to compress the size of transitive closure by introducing different relations on graph like the post-order number or the topological order (essentially they are the same) [1, 2, 3, 4, 5]. Another similar trend applies *Hop Labeling* technique [6, 7, 8, 9, 10, 11], which attaches an *in-set* and an *out-set* to each vertex and answers the reachability query by testing if the intersection of the two sets is empty or not. However, the space complexity of them is still $\Omega(nm^{1/2})$ and the construction times are non-linear, which make them impossible to work on a large and extensive graph. To further achieve the goal of constructing an index on a large graph in nearly linear time, *Refined Online Search* approaches [12, 13, 14, 15, 16, 17] sacrifice a little efficiency on query answering time by only guaranteeing the unreachable relation using the labels. In this way, the query answering time becomes either $O(1)$ or a little better than *BFS* since the searching space is pruned. Obviously, the more accurate the labels, the better the performance. So our proposal is inspired by the graph drawing approach adopted by *FELINE* [14] and the high dimension strategy applied by *IP* [18]. We try to reduce the searching space by attaching more topological information, which still takes nearly linear time to construct and applicable on large graphs, and the index size is linear to the graph size. As for the reachability query in road network, it always takes $O(1)$ time to answer the reachability query. The extensive evaluations on different graph structures and real-life large graphs demonstrate the effectiveness and improvement of our algorithm.

1.2.2 Speed Profile Generation from Historical Trajectories

Speed profile is used to describe the traffic condition of the roads. Given a departure time from one end of a road, it returns the time cost to travel through that road. We aim to learn a city's traffic condition from the taxi trajectories. There are several problems to transfer the trajectories into speed profile. The first one is how to transfer the trajectories to speeds on roads. Trajectory data is in the format of $< x, y, t >$, where x and y are the GPS coordinates and t is the timestamp. We need to match the GPS data on the map first, then infer the speed on the roads between a consecutive pair of points. Thus, the trajectories will be converted to a set of speed data like $< road, speed, timestamp >$. The second issue is how to organize these speed data. A straightforward approach is to line up the data points directly. However, this will result in a set of zigzag linear functions which is hard to use and cannot represent a road's speed obviously. Another approach is using regression to compute an approximate line. But this regression line is only computed by the time points with speed data, so it is unreasonable to use it to describe the time points that have no data. Thus, the only working method is using a histogram to average the data that fall within each bin. But the problem here is the granularity of the histogram. If the time slot size is big, then it cannot tell the difference between the smaller time intervals. And if the size is small, most of the time slots will not have enough data and become empty. The third issue is the missing value problem. After collecting the speed data by the histogram, there are still many time slots of many roads have no data at all. We have to use the existing data and map data to estimate these missing values. Finally, a speed profile with full data is big for data storage, especially when the number of the time slot is big, so it is necessary to compress a large speed profile to a smaller one.

1.2.3 Minimal On-Road Time Route Scheduling

A road network is organized as a graph $G(V, E)$ in database together with the time dependent traffic condition information $w_{i,j}(t)$ on each edge (v_i, v_j) (from speed profile). A subset of the vertices are called *parking vertices*, which allow waiting on them. A query aiming to find the minimal on road travel time path can be generalized as follows: Given a source vertex v_s with a departing time interval $[t_{s_1}, t_{s_2}]$ and a destination vertex v_d with a latest arrival time t_d, a path p_{v_s,v_d} can travels through a series of road each with cost $w_{i,j}(t)$ at time t and waiting on parking vertices is allowed. Thus, the total on-road travel time is $T_{ORT} = \Sigma_{i=s}^{d} w_{i,i+1}(t)$. Our query is to find a path with the minimal T_{ORT}. We call this problem the *minimal on-road time (MORT) problem*. It is different from the *store and forward* related problems [19], which focus on optimizing the flow rather than individual vehicle. And it is different from all the other path problems due to the different minimizing object and all of them actually have longer on-road travel time than a *MORT* path.

The challenge of solving this problem is that in the *MORT* problem, waiting on the parking vertices is allowed and may lead to different results. We do not know if waiting on the current parking vertex will result in a shorter on road travel time or not. And if it does, how long should we wait? This is the first problem that allows waiting on any vertex. The *shortest path algorithms* [20] cannot solve this problem because their cost functions on roads are static, and waiting on vertex does not change any static cost along the path. The *single starting time shortest path* on a FIFO time-dependent graph is just a variant of the shortest path, which cannot wait on any vertex since waiting can only result in a longer travel time. As for the *interval starting time fastest path*, only waiting on the starting vertex is allowed, and waiting on other vertices is insufficient as proved in [21]. In fact, their goal is to find the optimal departure time within the starting time interval that will result in the shortest total travel time. The time interval that they take into consideration is just the starting time interval on the source vertex, while in a *MORT* problem, we have to compute on the whole time interval between the earliest starting time t_{s_1} and the latest arrival time t_d, which is much longer than the starting time interval. Thus, all the above algorithms cannot solve the *MORT* problem.

1.2.4 Time-Dependent 2-Hop Labeling

Although the *MORT* algorithm can answer the time-dependent path queries correctly, it is slow to use in practice due to its inherit complexity of $\Omega(T(|V| \log |V| + |E|)$, where T is the number of turning points in the linear piecewise functions and could be really big. To answer the query faster, we propose a *time-dependent 2-hop labeling approach.* For each vertex $v \in V$, we pre-compute two sets of labels: out-labels $L_{out}(v_i) = \{(v_j, f_{v_i,v_j}(t))\}$ and in-labels $L_{in}(v_i) = \{(v_j, f_{v_j,v_i}(t)\}$, where v_j is a hop vertex and $f_{v_i,v_j}(t)$ returns the minimal cost from v_i to $_j$ at different departure time t. We use $L = \{(L_{out}(v_i), L_{in}(v_i)) | \forall v_i \in V\}$ to denote the set of all the labels. If L can answer all the queries on G, then we say L is a time-dependent 2-hop cover. To answer a time-dependent path query, we first find all the hop vertices. If v_i exists in both v_s's out-label set and v_d's in-label set, v_i is a hop vertex of this query. By visiting all the hop vertices, we are able to construct the final result, without traversing the graph. The result can answer any fastest travel time query and time-dependent reachability query.

However, because the *2-hop label* on static graph is already too big, it would be much larger on the time-dependent environment. Therefore, we partition the big road network into subgraphs, and build the *time-dependent 2-hop* within each subgraph and between the boundary vertices. In this way, we bring the index size down to a tolerable level. Furthermore, we provide an approximation method to compress the index, with a guaranteed error bound.

1.3 Contributions

1.3.1 Graph Reachability Problem

In this study [22], we propose a High Dimensional Graph Dominance Drawing (*HD-GDD*) approach for fast index construction and fast reachability queries. To have deeper understanding of the problem, we analyze the cause of false positive suffered by all refined online search approaches. Then, to further improve the query performance, we propose two refinement approaches, namely *false positive cache* and *false positive removal*. Finally, we empirically analyze the behavior of our approach on several types of synthetic and real world graphs. The experimental results verify our analysis of the cause of the false positive problem, and demonstrate that our proposal outperforms state-of-the-art methods.

1.3.2 Speed Profile Generation

In this study [23], we provide a solution to convert the trajectory data to speed profile in road network. We first extensively test the performance of speed profiles under different granularity and choose an appropriate one. Then we propose two missing value estimation methods and compare them with a widely used method. Finally, we introduce the *Piecewise linear approximation (PLA)* into the speed profile compression field and have conducted tests on three different *PLA* algorithms. The generated speed profile is used in Chapter 5 and Chapter 6.

1.3.3 Minimal On-Road Time Route Scheduling

In this study [24, 25, 23], we identify a general form of time-dependent route scheduling problem, called *MORT*, to make use of parking facilities in a road network to minimize the on-road travel time, instead of the total travel time. In fact, it is a general form of time-dependent path problem that can cover all the existing ones. Then we propose a *minimum cost function* and two novel algorithms to solve the *MORT* route scheduling problem efficiently and accurately, and an approximation approach for faster query answering. The *Basic MORT Algorithm* performs the *MORT* search for a vertex after each iteration, until the destination is reached. We show that its time complexity is $O(T|V|\log|V| + T^2|E|)$. The *Incremental MORT Algorithm* visits the vertices starting from a small subinterval to fill the full time interval incrementally, and its time complexity is $O(L(|V|\log|V| +|E|))$. Both algorithms require $O(T(|V| + |E|))$ space. T is the average number of turning points in minimum cost functions, and $L > T$ is the average number of subintervals during computation. The α-*MORT* approach can return an approximate result faster than the exact algorithms, with the worst error bounded. Finally, we evaluate the effectiveness and efficiency of our *MORT* algorithms with extensive experiments in road network and small world graphs, measuring both the reduction of the minimal on-road time and the algorithm running time.

1.3.4 Time-Dependent 2-Hop Labeling

We propose a time-dependent 2-hop labeling approach to speed up the fastest path query answering on small graph by hundreds of times. It is the first time-dependent path index that does not use online search. In order to scale to large road network, we propose a partition-based time-dependent 2-hop labeling to the answer fastest path query. After that, we apply a piecewise linear approximation approach on the label set to reduce the index size and further speed up the query answering. Finally, We thoroughly evaluate our approach with extensive experiments on the real-life road network and linear-piecewise-function-based speed profile. Results show that our approach outperforms the linear piecewise version of time-dependent variations of *CH* and *SHARC*.

1.4 Thesis Organization

The rest of this thesis is organized as follows: In Chapter 2, we review the existing works on solving graph reachability problems, speed profile generation, path finding problems and path query indexes. In Chapter 3, we present our *high-dimensional graph dominance drawing* approach that can answer reachability query faster even on large graph. We also analyze the influence of graph structures on the query performance. In Chapter 4, we present our solution to transform a large amount of trajectory into speed profile, which servers as the time-dependent functions in the later chapters. In Chapter 5, we introduce a new type of time-dependent route scheduling problem: *minimal on-road time route scheduling*, and present two accurate algorithms and a fast approximate algorithm. We further show that our algorithms can solve all the existing single criteria path problems. In Chapter 6, we further speed up the time-dependent travel time query and time-dependent reachability answering by adopting *2-hop* labeling approaches to time-dependent environment. Finally, the conclusions and the future research directions suggested by the thesis are given in Chapter 7.

Chapter 2

Literature Review

In this chapter, we first provide a brief overview of the fields we talked about in the report, including the reachability problem and path problems on graph, which will be presented in Section 2.1 and Section 2.3 respectively. In section 2.2, we provide a brief description of the existing techniques related to the speed profile construction from trajectory.

2.1 Reachability Problem

In the early years when graphs were small, basic algorithms like *Breadth-First Search / Depth-First Search (BFS/DFS)* were efficient enough to solve the reachability problem. However, with the rapid development of Internet, e-commerce services, social networks, road networks and many other applications, the sizes of graphs have increased dramatically nowadays, making it indispensable to answer reachability queries efficiently on extensive graphs. Tremendous efforts have been devoted to designing novel indexing or querying strategies for reachability checking. Most of the existing approaches can be classified into three categories, as depicted in Table 2.1.

2.1.1 Extreme Approaches

The first category is the *Extreme*, which contains two approaches, namely *BFS/DFS* and *Transitive Closure*. *BFS/DFS* traverses the entire graph starting from the source vertex to find the destination vertex without any pre-computation or pruning. Obviously, it requires the smallest space complexity but has the largest query time complexity of $O(|V| + |E|)$. *Transitive Closure* pre-calculates and

TABLE 2.1: Categories of Reachability Algorithms

Category	Type	Index Construction Time	Index Size	Query Time
Extreme	BFS/DFS	0	0	$O(\lvert V\rvert + \lvert E\rvert)$
	Transitive	$O(\lvert V\rvert\lvert E\rvert)$	$\lvert V\rvert^2$	$O(1)$
Set Cover	Transitive Closure Compression	non-linear	non-linear	sub-linear
	Hop	non-linear	$\Omega(nm^{1/2})$	sub-linear
Topological	Refined Online Search	$O(\lvert V\rvert + \lvert E\rvert)$	$O(\lvert V\rvert)$	$O(\lvert V\rvert + \lvert E\rvert)$ or $O(1)$

stores all the reachable vertex pairs in a $|V| \times |V|$ reachability matrix. Thus, it can answer reachability queries within constant time by directly retrieving the results from the matrix. However, its index size and construction time are both $O(|V|^2)$, making it infeasible even for small graphs. All the approaches in the other two categories either try to reduce the storage cost of *Transitive Closure* or to reduce the searching space of *BFS/DFS*.

2.1.2 Set Cover Approaches

Set Cover consists of two streams: *Transitive Closure Compression (TCC)* and *Hop Labeling*. Both of them store an *in-neighbor set* and an *out-neighbor set* for each vertex, and answer reachability query by intersecting corresponding neighbor sets. If the intersection result is not empty, then two vertices are reachable.

Transitive Closure Compression

TCC tries to compress the large transitive closure by considering intermediate paths or topological orders so that both the index construction complexity and the space complexity could be decreased. *Range Compression* [1] first finds a spanning tree of the graph. Obviously, the reachability information on a tree can be recorded by a post-order traverse. Thus, each vertex has an interval $[r_a, r_b]$, where r_b is its post-order number and r_a is the smallest post-order number on its subtree. For any two vertices v_r and v_s on the same spanning tree, if $[r_a, r_b] \subset [s_a, s_b]$, then $v_r \to v_s$. This is because the post-order number is actually the reverse depth-first topological order, and the smallest subtree number stores from which no-out-neighbor vertex that this reverse topological ordering starts. The parent vertex always has a larger post-order number than its children, and has the smallest subtree number from all its children, so its interval can contain all its children's interval. At the same time, two vertices that cannot reach each other are not in each other's subtrees. Since it uses post-order, one subtree has to be traversed thoroughly before another subtree can be traversed. So the intervals on different subtrees cannot contain each other. However, one set of tree cover is not enough to cover all the information on the original graph. If an edge in the original graph is not contained in a spanning tree, then this reachable information is not covered by this tree. For example, there is an edge (v_r, v_s) in G but not in the spanning tree. To add the reachable information $r \rightarrow s$ back to the tree, *Range Compression* will add the interval of v_r to v_s. So the interval of v_s is $\{[s_a, s_b], [r_a, r_b]\}$ now. If one interval is covered by another interval in the interval set of a vertex, then these two intervals can combine to a single interval (This is the compression part). Apparently, the total number of the intervals depends on how the spanning tree is built. To reduce the interval size, the algorithm first computes the predecessor set of each vertex. The more predecessors indicates it has more reachable information. Then the spanning tree is generated in topological order, with each vertex only keeps the in-edge from the in-neighbor that has the largest predecessor set. However, the worst case of this approach is still $O(|V|^2)$ in the case of the bipartite graph, and the index construction time is $O(|V||E|)$ while the query time is $O(log|V|)$.

Chain Cover [3] follows the idea proposed by [26] that decomposes a *DAG (directed acyclic graph)* into disjoint chains such that on each chain, if vertex v appears above vertex u, there is a path from v to u in G. Then, each vertex v is assigned an index (i, j), where i is a chain number, on which v appears, and j indicates v's position on the chain. In addition to this, v is associated with an index sequence $(1, j_1), ..., (i-1, j_{i-1}), (i+1, j_{i+1}), ..., (k, j_k)$ such that for any vertex u with index (x, y) if $x = i$ and $y > j$ or $x \neq i$ but $y \geqslant j_x$, it is a descendant of v, where k is the number of the disjoint chains. For this method, the space overhead and the query time are $O(k|V|)$ and $O(\log k)$, respectively. However, finding a minimized set of chains for a graph, [26] needs $O(|V|^3)$ time. With a similar thought, [3] can decompose a graph into a minimized set of disjoint chains in $O(|V|^2 + b|V|\sqrt{b})$ time, where b is G's width, defined to be the size of a largest vertex subset V of G such that for every pair of vertices $u, v \in V$, there does not exist a path from u to v or from v to u. This enables them to generate a compressed transitive closure in $O(b|E|)$ time, improving the existing methods for the problems of practical size by one order of magnitude or more. The space overhead and the query time are bounded by $O(b|V|)$ and $O(\log b)$, respectively.

Path Tree Cover [4] proposes a graph structure called *path-tree* to cover the *DAG*, where each vertex in the tree represents a path in the original graph. It is suitable for the sparse graph that the number of edges is no more than two times of the number of vertices. Such a structure can be constructed in $O(|E| + |V| \log |V|)$ time. Then, the labels are attached to each vertex with the connection information of the paths by traversing the path tree in a post-order manner. Therefore, their labels can be viewed as the combination of tree cover intervals and a path number. They prove that the *path-tree cover* can always perform the compression of transitive closure better than or equal to the *optimal tree cover* approaches and *chain decomposition* approaches.

TF-Label [5]) proposes a data structure called *Topological Folding* and uses it to attach their reachability labels. First of all, the vertices in a graph is assigned a *topological level* such that a vertex can never reach the vertices on smaller or equal level due to the property of topological order. Then the graph is folded by 2 each time through deriving the reachability of the even levels from the odd levels. So the original graph is converted into a set of smaller *TF-graphs* and each vertex is associated with one of these *TF-graphs*. Again, like the previous algorithms, the labels are assigned to vertices based on the transitive closure. So in fact, *TF-Label* aims to use the *TF number* to compress the transitive closure. However, their index size and query time are not bounded and the index construction time depends on the characteristic of the graph.

Hop Labeling

Hop Labeling algorithms originate from Cohen's work [6]. This branch of methods tries to find a subset of vertices as landmarks and answers reachability queries between vertices by reachability between landmarks. The most famous work is the *2-Hop* [6]. For each vertex u, let $l_{in}(u)$ denotes the set of vertices that can reach u, and $l_{out}(u)$ denotes the set of vertices that can be reached by u. The key observation of this approach is that every vertex in $l_{in}(u)$ can reach every vertex in $l_{out}(u)$. However, finding the optimal label sets is actually a *NP-H* problem, which is proved by reducing it to the *3-SAT problem* [6]. Therefore, many approaches have been proposed to reduce the label size and label construction time by introducing various heuristic rules.

Hierarchical Labeling of Sub-Structures (HLSS) [7] is a hierarchical approach. Since a graph often contains different types of substructures whose reachability is easier to encode with different labeling techniques, they extract such substructures and apply efficient techniques suitable to each of them.

Their labeling algorithm has two phases and each focusing on exploiting different characteristics of the input graph G: The first phase, *tree-reachability reduction*, begins with a preprocessing step that identifies each strongly connected component, collapses the component into one representative vertex, and uses this vertex to label others in the component. Then, they identify tree structures in G and assign interval labels to vertices based on these tree structures. Containment of interval labels implies reachability through tree paths. This phase also computes a remainder graph G_r that captures any remaining reachability information not encoded by interval labels. Specifically, a vertex can reach another vertex through portals in G_r. They label vertices by their portals to facilitate reachability checking. The second phase, *remainder graph-reachability encoding*, aims at compressing the reachability information in the remainder graph G_r produced by the first phase. They do so by assigning additional labels to portals so that reachability among them can be checked efficiently by comparing their labels. They use several techniques for assigning such labels, including an enhanced version of the *2-hop* approach as well as techniques inspired by data mining, linear algebra, and graph algorithms. To sum up, their algorithm produces a four-level hierarchy label for each vertex: The first one is a *strongly connected component label* $l_s(u)$, which is a representative vertex in the strongly connected component containing u, if any. It is assigned by the tree-reachability reduction phase. The second one is a pair of numeric interval labels, $l_i^{\vdash}(u)$ and $l_i^{\dashv}(u)$, which form the interval $[l_i^{\vdash}(u), l_i^{\dashv}(u)]$. They are assigned by the tree-reachability reduction phase. The third one is a pair of *portal labels*, $l_{in}p(u)$ and $l_{out}p(u)$, which are two portals of u in G_r if they exist. They are also assigned by the tree-reachability reduction phase. The last one is a pair of remainder labels, $L_{in}r(u)$ and $L_{out}r(u)$, each of which consists of a set of symbols in general. They are assigned by the remainder graph-reachability encoding phase. So this approach is just a combination of the ten-years-ago-state-of-art techniques.

Dual Labeling [27] is another hybrid index for sparse graphs. Indicated by the name, it has two schemes: *Dual-I* and *Dual-II*. The *Dual-I* labeling scheme has constant query time, and for sparse graphs, the labeling complexities of both *Dual-I* and *Dual-II* are almost linear. The *Dual-II* scheme has higher query complexity but uses less space in practice. They view a graph as two components: a tree (spanning tree) plus a set of t non-tree edges. For sparse, tree-like graphs, they assume $t \ll n$. The two components together contain the complete reachability information of the original graph. The dual labeling scheme seamlessly integrates: i) interval-based labels, which encode reachability in the spanning tree, and ii) non-tree labels, which encode additional reachability in the rest of the graph. At query time, it first consults the interval-based labels to see if two vertices are connected by tree edges, if not, it consults non-tree labels, and check if they are connected by paths that involve non-tree edges. For *Dual-I*, both operations have constant time complexity. For *Dual-II,* the second operation takes $O(\log t)$ time. Since $t \ll n$ for sparse graphs, $O(\log t)$ is often negligible. Furthermore, the two set of labels can be assigned by a depth-first traversal of the graph, which is of linear complexity. The preprocessing step may take $O(t^3)$ time in the worst case and this cost is almost negligible for sparse graphs. To check reachability encoded by non-tree labels, the *Dual-I* approach relies on an additional data structure of size t^2. Since the spanning tree of a connected graph has $|V| - 1$ edges, the number of non-tree edges t is at most $|E||V| + 1$.

3-Hop [10] aims to the large and dense problem faced by the *2-hop*. The basic idea in *3-hop* index is to utilize a simple graph structure *line chain*, rather than a sole vertex, as an intermediate hop to describe the reachability between source vertices and destination vertices. Such a chain structure is analogous to the highway system of the transportation network. To reach a destination from a starting point, you simply need to get on an appropriate highway and get off at the right exit to the destination. This index scheme does not need to compute the entire transitive closure. Instead, it only needs to compute and record a number of so-called *contour* vertex pairs, which can be orders of magnitude smaller than the size of the transitive closure. Indeed, it is even much smaller than the compressed transitive closure of the chain cover. The connectivity of any pair of vertices in the DAG can be answered by those contour vertex pairs. Further, they *factorize* these contour vertex pairs by recording a list of *entry points* and *exit points* on some intermediate chains. Since each chain has a direction, each vertex u records a list of *entry points* (the smallest vertices) it can reach on some chains. It also records a list of *exit points* (the largest vertices) which can reach it in some chains. Here, the order of vertices in the chain is with respect to their topological order in that chain, i.e., a vertex with a smaller number can reach a vertex with a larger number. Given this, the three hops are: 1) the first hop from the starting vertex to the entry point of some chain, 2) the second hop from the entry point in the chain to the exit point of the chain, and finally 3) the third hop from the exit point of the chain to the destination vertex. The goal of 3-hop index is to assign all vertices with a minimal total number of entry and exit points so that they can maximally compress the transitive closure. So they propose an efficient algorithm to generate an index which approximates the minimal *3-hop* index by a logarithmic factor. Theoretically, it is shown that *3-hop* labeling always has a better minimal compression ratio than *2-hop* labeling, and its construction time is much faster than that of *2-hop*. However, its construction time complexity is still $O(k|V|^2|Con(G)|)$, where $|Con(G)|$ is transitive closure contour number. Thus, although is has better performance on dense graphs, it is still not applicable on large graphs.

Hierarchical Labeling (HL) and *Distributed Labeling (DL)* [11] are two labeling approaches for different graph structures. They assume a *DAG* can be represented in a hierarchical (multi-level) structure, such that the lower-level reachability needs to go through upper-level (but not vice versa), then they can somehow recursively broadcast the upper-level labels to lower-level labels. In other words, the labels of lower-level vertices (L_{in} and L_{out}) can directly utilize the already computed labels in the upper-level. Thus, on one side, by using the hierarchical structure, the completeness of labeling can be automatically guaranteed. On the other side, it provides an importance score (the level) of every hop, and each vertex only records those hops whose levels are higher than or equal to its own level. The *HL* structure is produced by a recursive reachability backbone approach, i.e., finding a reachability backbone G^* from the original graph G and then applying the backbone extraction algorithm on G^*. Recall that the reachability backbone is introduced by the latest *SCARAB* framework [28] which aims to scale the existing reachability computation approaches. Here they apply it recursively to provide a hierarchical DAG decomposition. Given this, a fast labeling algorithm is designed to quickly compute L_{in} and L_{out} one vertex by one vertex in a level-wise fashion. In *Distribution Labeling* the sophisticated reachability backbone hierarchy is replaced with the simplest hierarchy a total order, i.e., each vertex is assigned a unique level in the hierarchy structure. Given this, instead of computing L_{in} and L_{out} one vertex at a time, the labeling algorithm will distribute the hop one by one (from higher order to lower order) to L_{in} and L_{out} of other vertices. The worst case computation complexity of this labeling algorithm is $O(|V|(|V| + |E|))$ (of the same order as transitive closure computation), so they are still not useful on large graphs.

So all the above algorithms cannot work on large and dense graphs due to the large construction time and label size.

2.1.3 Refined Online Search Approaches

Our work falls into the third category of *Refined Online Search*, which is also called *Topological Cover* approach. By attaching topological information to each vertex as a label within linear time, *Topological Cover* can easily scale to large graphs although it is sometimes slower in query answering than *Set Cover* approaches. Note that all the existing *Topological Cover* approaches suffer from the *false positive* problem, where the positive result of label comparison cannot guarantee that two vertices are reachable. In other words, the unreachable relationship can be retrieved directly from the labels within constant time, while the reachable relationship detection decays into a guided or pruned *BFS/DFS*. So if two vertices are unreachable, the query time is $O(1)$. If they are reachable, the worse case could be $O(|V|+|E|)$ when the *false positive* exists, and $O(1)$ when the *false positive* is eliminated.

Grail (Graph Reachability Indexing via RAndomized Interval Labeling) [12] is the first work that can work fast on large graph. Similar to the tree-cover approaches, it also traverses the vertices on graph in a post-order manner and assign intervals. The difference between it and the tree covers is that the first number of the interval is not the smallest post-order on a spanning tree, but the smallest post-order it can reach on graph. So the interval it has actually contained more information than needed that even if one vertex's interval is contained by another vertex's, the reachability between these two vertices cannot be guaranteed. In fact, only the unreachability is guaranteed. And the post orders of the vertices that have the same father vertex are assigned randomly, it just ignores the information that could help reduce the false positive rate. Thus, although it can scale to large graph and answer the reachability query in a short average time, its performance is bounded by the 1-dimensional label.

FERRARI (Flexible and Efficient Reachability Range Assignment for Graph Indexing) [13] starts its labeling from the *Range Compression* [1]'s result. Unlike the approaches in the *transitive closure compression* category which aim to compress without loss, it relaxes the requirement that some degrees of loss is acceptable. Such compression is achieved by combining the consecutive intervals using a greedy algorithm, and the compression rate is specified by the predefined parameter. Thus, the label size decreases and the false positive appears. Since it still applies the 1-dimensional label, its performance is also bounded.

FELINE (Fast rEfined onLINE search) [14] is the state-of-the-art *Topological Cover* algorithms. Its origin can be traced back to the *graph drawing*. [15] is the first work that tries to use graph drawing to answer reachability query. Their goal is to label the vertices by vectors in such a way that u can reach v if and only if $C_u < C_v$, where C_u is a d-dimensional vector that is assigned to vertex u. Although their goal is promising, they only focus on the graphs that are planar. That is to say, all the vertices with no in-neighbors are on the boundary of the same face while all the ones with no out-neighbors are on the other face. Then they use a simple *left-DFS* followed by a *right-DFS* algorithm to compute a 2-dimensional vector for each vertex. However, this planar condition is so strong that it does not allow the existence of false positive. In fact, it is the feature of non-planar that causes the false positive. So their algorithm is limited to a small type of graphs. Twenty years later, [16] expands the condition from planar graph to bipartite graph, which is made up of a set of source vertices and a set of sink vertices, and all edges are directed from a source to a sink. They first call the vector assignment approach the *dominance drawing* and prove the *dominance drawing dimension* d is no bigger than $min(|source|, |sink|)$. Then they give an algorithm to compute a 2-dimensional dominance drawing in the case of such drawing exists, which is also a strong assumption. So they just avoid the false positive problem. Kornaropoulos revisited this problem in [17] [29] and proposed *weak dominance drawing* to expand the above algorithm to any graph type, which can only guarantee the unreachability relation between two vertices if the source is not smaller than the destination. They give the upper bound of false positive pairs, which is $inc(G) - (dim(G) - 2)$, where $inc(G) = \frac{|V|(|V|-1)}{2} - |E^*|$ is the number of incomparable pairs of a graph, $dim(G)$ is the smallest graph dominance drawing dimension, and $|E^*|$ is the number of reachable pairs. With the weak dominance drawing, they attached a two-dimensional coordinate to each vertex regardless it is planar or not. Thus, this was the first time that the graph drawing approach was using on any graph type. And they proved that finding the optimal weak dominance drawing is *NP-C*. Although they did lots of theoretical work, no experiment result was released. *FELINE* adopts the weak dominance drawing concept from Kornaropoulos's work and treats it as a *Refined Online Search* method. All they do is to use the ancient 2-dimensional vector generating algorithm [15] to give each vertex a coordinate and use it to rule out the unreachable vertex pairs. However, their approach is merely an implementation of the 1975 work, which is two-dimensional and has no theoretical progress.

2.1.4 Summary

As explained above, all the *set cover* approaches aim to answer the reachability query only by labels. Due the natural bounds of their schemes, most of them can only work on small graph. Although some of them can work on large graph, they still require the graph to be sparse, which is a strong condition. To extend the index on the general large graph, several *refined online search* algorithms are proposed. They sacrifice the query time a little for the ability to work on large graph. Due to this purpose, false positive is the main issue that affects the query performance. The lower the false positive rate, the faster the average query time. However, none of the existing approaches has an effective solution to reduce the false positive rate. Thus, we aim to reduce the false positive rate by analyzing the graph structure and adding more dimensions of labels. Such an approach can only increase the construction time complexity by a constant factor, but will decrease the false positive pair number and boost the average query time.

2.2 Organization of Speed Profiles

Speed profile is used to organize the speed of each road at different time of a day in a city. It can be used to predict the traffic condition of a city and serves as the input of the path queries. In this section, we breifly review the existing works on a series aspects of the speed profile generation.

2.2.1 Speed Profile Collection

Theoretically, the speed profile is organized as a continuous function. However, in reality, such a function does not even exist. The most accurate method is to use sensor, which collects the traffic data on road in a time slot of 1 minute and still is a kind of histogram-based approach.

[30] uses sensors to build up their speed profile with a granularity of 1 minute. However, such an approach is expensive and impractical in most places around the world apart from some experimental systems. So many works try to learn the traffic condition road from the trajectories of the cars. Firstly, the trajectory has to be matched to the roads [31, 32]. Then, with the path length and the time spent between the two points, we can get the average speed spent on these roads. By repeating this procedure from the first GPS point to the last, we can get all the roads' speeds on this trajectory, together with their corresponding time intervals.

After gaining the speed data on each road at different time, we have to use histogram to collect them. By averaging the speed values within each bin, the outlier speed value's defect can be eliminated. However, choosing the granularity of a speed profile is a problem. If the granularity is too big, then it cannot reflect the traffic condition on road correctly. If the granularity is too small, sometimes we just do not have enough data for each bin. Thus, most of the time slots have no data. This choice actually depends on the data set. Different data may have different suitable granularity. For example, [33, 34] apply a granularity of 15 minutes. [35, 36] apply 10 minutes. [37, 38] use 5 minutes as the size of their time slot.

2.2.2 Missing Value Estimation

It is not surprising that even if we have collected a large amount of trajectory, there are still many time slots on many roads that still have no data at all. To estimate the missing values based on the existing ones, different approaches are proposed. Since most of the speed profiles are organized into histogram, which can be viewed as a vector of each road, or a matrix of the whole road network, most of the research are working on filling the missing values of vector or matrix. However, as an early work, [39] uses regression to organize the speed profile. Thus, its missing values are filled only by the regression function itself. But this only works on special cases, like the two-lane rural highways studied in their paper. So it fails on other road network types that do not share the same property with the rural highway.

Markov model [40] is a widely used model to learn the traffic state changing patterns in both temporal and spatial dimensions. [33] uses spatio-temporal hidden Markov models *(STHMM)* to model correlations among different traffic time series. Their algorithms are able to learn the parameters of an *STHMM* while contending with the sparsity, spatio-temporal correlation, and heterogeneity of the time series. In the offline learning stage, a state formulation and parameter learning module takes as input a collection of time series that is obtained from historical GPS trajectories. It outputs an *STHMM*. As real-time GPS records stream in, the online inference component infers near-future travel costs by using the learned *STHMM*. The time-dependent edge weights in road network can be updated based on the inferred travel costs.

[41] uses *Matrix-Factorization based Collaborative Filtering (MF-CF)* [42, 43] to estimate the missing values in their gas consumption prediction work. The rows of the matrix are the roads and the columns are the time slots. The elements are the speeds of each road at different time slot. To improve the accuracy, they add a side information matrix to make their approach context aware. The first context is the road information, including the length, direction, level, tortuosity, speed limit and lane number. The road static information is organized into a matrix with rows stand for roads and columns stand for feature. The second feature is related to POI (Point of Interest). The POIs that are within a distance of 200m are counted as belonged to a road. They set 10 categories of POI like school, bank, restaurant and so on. Thus, this POI feature is a vector with 10 elements for each road, with each number stands for the number of POI of each category that this road has. The third context is the global position feature. They divide the city into 4×4 grids, and label the neighboring grid with 1, others with 0. Thus, the global position of each road that it resides in can be organized into a

vector of 16 elements. These three context features are placed into one big matrix. The general idea is that road segments with similar road feature, POI feature and global position feature could share a similar traffic condition.

[35] takes advantage of the above methods. They build a multi-view road speed prediction framework. In the first view, temporal patterns are models by a layered hidden Markov model and in the second view, spatial patterns are modeled by a collective matrix factorization model. The two models are learned and inferred simultaneously in a co-regularized manner.

2.2.3 Compression

The raw histogram speed profile takes a large space to store, especially when the time slot size is small, its growth is linear to the number of time slots. A large speed profile does not only waste storage and memory space, but also takes a longer time to load it into memory. As observed in reality, lots of the values in the speed profile are the same, especially during night time when there is nearly no car on the road, the road speed tend to be same. So compression on speed profile is meaningful and practicable.

Histogram-based speed profile is essentially a time series data [44], which is a general data type that has an ordered sequence of n real valued variables. The segmentation task on time series data aims at creating an accurate approximation of time series, by reducing its dimensionality while retaining its essential features. The objective of segmentation is to minimize the reconstruction error between a reduced representation and the original time series. *Piecewise Linear Approximate(PLA)* [45, 46] is the main approach to achieve this. The main idea behind PLA is to split the series into most representative segments, and then fit a polynomial model for each segment. There are three basic approaches: *sliding window* [45], *top-down* [47, 48] and *bottom-up* [49].

The *sliding window* algorithm is a fast online algorithm whose time complexity is $O(n)$, where n is the time series length. It keeps expanding the approximate line from the left starting point to the right until the error surpasses the threshold ε. Then it uses the end point of the latest generated segment as the next starting point and repeats until all the points are visited. The *top-down* algorithm finds the best splitting position each time (i.e. the two resulting segments have the smallest combined error). If any of the two resulting segments' errors are larger than threshold ε, the algorithm repeats recursively. The algorithm ends when all the segments' errors are smaller than ε. It breaks the search space into 2 pieces each time and calls itself recursively at most 2 times. At the same time, the $Error$

function calculates the difference between the approximate line and the original line, which takes $O(n)$ times. So the overall time complexity of the top-down algorithm is $O(n \log n)$. As the threshold ε grows, less recursion is needed, and the overall running time decreases.The *bottom-up algorithm* is reverse to the *top-down* algorithm. In the initial step, it connects the consecutive points, whose errors are all 0. Then it merges consecutive lines with the smallest error iteratively until the smallest error exceeds the threshold ε. The worst case is to erase all the intermediate points, which runs $\frac{n(n-1)}{2}$ times. Thus, the time complexity of bottom-up algorithm is $O(n^2)$.

2.2.4 Summary

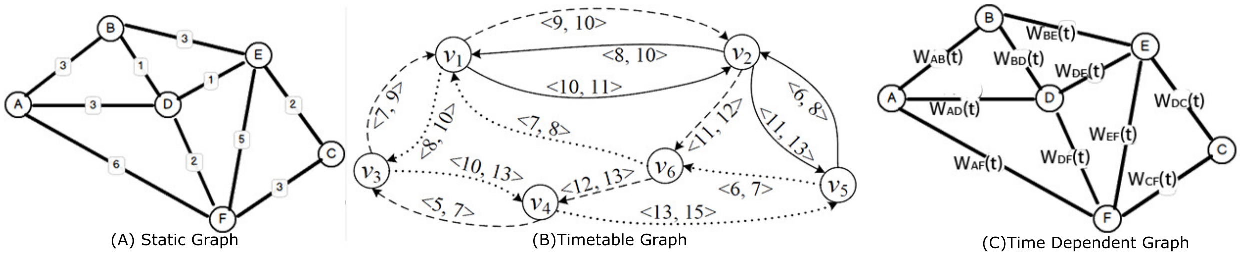
No previous work has provided a thorough solution from trajectory data to compressed speed profile. Depend on the data set, different granularities are used. There is no pervasive rule for it, so we have to test on our own dataset and find the appropriate granularity for it. As for the missing value estimation, the matrix factorization based collaborative filtering is adopted by many works, so we choose it as a comparison method. No work on speed profile compression has been reported. Since the speed profile is essentially a set of consecutive lines, we introduce the compression methods in the time series data field and test their performance.

2.3 Path Problems

The road network is naturally a graph, with vertices denote the intersections of roads or the turning points between road segments and the edges denote the roads or road segments. Depending on the type of information that is attached on the edges, the graphs that are used on the road network can be categorized into the following three types:

1. ***Static Graph*** $G_s(V, E)$ as shown in Figure 2.1(a), where $V = v_i$ is the set of vertices, $E \subseteq V \times V$ is the set of edges. $\forall(v_i, v_j) \in E$, there is a value $weight(v_i, v_j)$. This value can be used to denote the distance or any other static weights of the roads.

2. ***Timetable Graph*** $G_t(V, E)$ as shown in Figure 2.1(b), where $V = v_i$ is the set of vertices, E is the edge set. An edge $e \in E$ is a $quadruple(v_i, v_j, t, \lambda)$, where $v_i, v_j \in V$, t is the starting time and the λ is the traversal time to go from v_i to v_j at time t. $t + \lambda$ is the ending time. Such graph is used in problems related to public transportation systems like train, bus and airplane [50].



(A) Static Graph

(B)Timetable Graph

(C)Time Dependent Graph

FIGURE 2.1: Three Types of Graph to Represent Road Network

3. ***Time Dependent Graph*** $G_T(V, E, W)$ as shown in Figure 2.1(c) , where $V = \{v_i\}$ is the vertex set, $E \subseteq V \times V$ is the directed edge set and W is a set of functions. For each edge $(v_i, v_j) \in E$, there is a function $w(v_i, v_j, t) \in W$, where t is the time in time domain $\hat{T}$, that tells how much time it costs to travel from v_i to v_j at time t. Such graph is used in describing traffic condition in a city.

We will talk about the path algorithms on each type of the graph in the following sub-sections.

2.3.1 Path Problems on Static Graph

Since the weight W on each edge is static, the path problem on static graph, mostly known as the *shortest path problem*, aims to minimize the total W on the path. So Given a query $Q(v_s, v_d)$, the objective is to find a path from v_s to v_d that has the minimum $\Sigma_{i=s}^{d} weight(v_i, v_{i+1})$.

The research on this field can be categorized into two streams: finding the shortest path and answering a shortest distance query quickly. The first stream is the basis of the second one.

Shortest Path Computation

Depending on the algorithm can find the shortest path from one source vertex or all the vertices on the graph, the shortest path algorithms can be categorized into *Single Source Shortest Path* algorithm and *All Pair Shortest Path* algorithm.

The most widely used single source shortest path algorithm is the *Dijkstra Algorithm* [20], which acts as the base technique for almost all the other path algorithms. It can find the shortest path from a source vertex to all the other vertex in the graph whose weights on edges are all non-negative. It keeps a priority queue Q to store the vertices based on their distance to the source vertex v_s. Each time we retrieve the top vertex v_i, which has the minimum value in Q. Thus, it has founded its shortest distance from the source vertex v_s, and it can relax the distance of its out-neighbors. The reason is that all the other vertices in Q have larger distance than v_i, then the paths via these vertices will result in a longer distance. So the top vertex in Q has found its shortest distance. For example, the current top vertex is v_i and its distance is d_i. It has a out-neighbor v_j whose distance is d_j. The weight from v_i to v_j is $weight(v_i, v_j)$. If $d_i + weight(v_i, v_j) < d_j$, then we set $d_j = d_i + weight(v_i, v_j)$. If the destination vertex is on the top of Q, then we can draw the conclusion that the shortest distance between v_s and v_d has been found. The complexity of *Dijkstra Algorithm* depends on the implementation of the priority queue. If the priority queue is implemented by a binary heap, the time complexity is $O((|V| + |E|)log|V|)$. If the priority queue is implemented as a Fibonacci Heap [51], the time complexity can achieve $O(|V|log|V| + |E|)$. If all the costs are integers within a range $[0, C]$, the time complexity can be further reduced to $O(|E| + |V|\sqrt{C})$, with the help of multi-level bucket [52].

There are several variations of *Dijkstra Algorithm*. The first one the *Bidirectional Search* [53], which runs two simultaneous searches: one forward from the source vertex v_s, and the other one backward from the destination vertex v_d, stopping when the two meet in the middle. The reason for this approach is that in many cases it is faster: for instance, in a simplified model of search problem in which both searches expand a tree with branching factor or out-degree b, and the distance from start to goal is d, each of the two searches has complexity $O(b^{d/2})$, and the sum of these two search times is much less than $O(b^d)$ which would result from a single search from v_s. For road networks, bidirectional search visits roughly half as many vertices as the unidirectional approach. Another famous variation is A^* Algorithm[54]. It uses a new function $f(n) = g(n) + h(n)$ as the minimization goal rather than the pure distance. $g(n)$ here is the path distance while $h(n)$ is the heuristic function that calculates the geographical distance of the current vertex to the destination vertex. $f(n)$ is the sum of the current shortest distance from v_s and the estimate distance to the v_d. So a vertex has a smaller geographic distance to the destination would have a smaller $f(n)$, thus, it would appear earlier on the top of the queue, which could help speed up the shortest path finding. Since these variations all use heuristic functions, there are no guaranteed upper bounds although they run faster than the original *Dijkstra Algorithm*.

Bellman-Ford Algorithm [55] can compute the single source shortest path on a more general case where the weight on edge can be negative. If there is a negative loop on graph, then it can tell that such path does not exist. It visits each edge $|V| - 1$ times to relax its distance to the source. The relaxation process is the same with the *Dijkstra*'s. Thus, its time complexity is $O(|V||E|)$. Although its worst case is worse than the *Dijkstra*, it is often much faster in average, making it competitive with *Dijkstra* in some scenarios.

As for the all pair shortest path problem, the naive approach is to call *Dijkstra Algorithm* $|V|^2$ times to compute all the paths, whose best time complexity is $O(|V|^2 log|V| + |V||E|)$. However, it can only work on the graphs with no negative edge weight. If we call *Bellman-Ford Algorithm* $|V|^2$ times to work on general graphs, then the time complexity would be $O(|V|^3|E|)$. *Floyd-Warshall Algorithm* [56] is a classic algorithm that can solve the problem on the graph with no negative loop in $\Theta(|V|^3)$ time. It considers the intermediate vertices along a path. Suppose the vertices in graph is numbered from 1 to N and a function $F(i, j, k)$ returns a possible shortest path from v_i to v_j using the vertices in set $\{v_1, v_2, .., v_k\}$ as the intermediate vertices along the path. Thus, for any vertex pair (v_i, v_j), the shortest path from v_i to v_j is either only using vertices in set $\{v_1, v_2, .., v_k\}$ or a combination of path from v_i to v_k and from v_{k+1} to v_j. Since the shortest path from v_i to v_j through v_k is $F(i, j, k)$, it is obvious that if there is a better path from v_i to v_j through v_{k+1}, then the length of this path should be the concatenation of the shortest path from v_i to v_{k+1} and the shortest path from v_{k+1} to v_j, which can be formulated as $F(i, j, k + 1) = min(F(i, j, k), F(i, k + 1, k) + F(k + 1, j, k)$. This recursive formula takes $O(|V|^3)$ time to return the results of all the vertex pairs. [57] compares the weights on edges for $O(|V|^{5/2})$, which would result in a better time complexity of $O(|V|^3(loglog|V/LogV|)^{1/3})$.

Johnson Algorithm [58] solves the all pair shortest path problem on sparse graph. It works by using *Bellman-Ford Algorithm* first to compute a transformation of the input graph that removes all negative weights. It is done by adding a dummy vertex v_q to the original graph, and the edge weights from v_q to all the other vertices are set to 0. Then the *Bellman-Ford Algorithm* is applied to test if there is a negative loop. If there is a negative loop, then the algorithm terminates. Otherwise, the edges of the original graph are re-weighted using the values computed by the *Bellman-Ford Algorithm*: an edge from v_i to v_j, having length $W(v_i, v_j)$, is given the new length $w(v_i, v_i) + h(u_i)h(v_j)$, where $h(v_i)$ is the distance from v_q to v_i. Then the *Dijkstra Algorithm* could be applied to the transformed graph. Since it is applying *Dijkstra* in essence, its time complexity is $O(|V|^2 log|V| + |V||E|)$.

All of the above algorithms are the basis of the path algorithms on time-dependent graphs. Although they cannot solve the various time-dependent path directly, they provide core techniques that are shared by all the following algorithms.

Shortest Distance Query Answering

When it comes to the shortest distance query answering, there are two extreme approaches. The first one has no pre-computation process and runs the above shortest path algorithms each time when a query comes. Although the above algorithms are fast enough on small graphs, they all perform not well on real-life large graphs. So it is not applicable to the query intensive applications that require a high degree of responsiveness. Another extreme approach is to pre-compute all the distances between any two vertices, such that the query answering time is $O(1)$. Since it only looks up the precomputed structure once, it is also known as *1-Hop* approach, corresponding to the *2-Hop* labeling below. However, its space complexity $O(|V|^2)$ is not acceptable even for graphs that are not too large. So people further propose different precomputed indexes on the static graph to speed up the shortest distance query.

The first technique is the *labeling* [59]. One of the most important early work of the labeling approach is *2-Hop* [6]. It assigns each vertex a distance label of the network such that the distance between two vertices can be computed only using the labels of these two vertices. For each vertex $v \in V$, it creates 2 labels $\mathscr{L}_{in}(v)$ and $\mathscr{L}_{out}(v)$ so that if $dist_G(s,t) \neq \infty$, then it can find a pivot u such that $(u, d_1) \in \mathscr{L}_{out}(s), (u, d_2) \in \mathscr{L}_{in}(t)$ and $d_1 + d_2 = dist_G(s,t)$, and there does not exist any u' such that $(u', d'_1) \in \mathscr{L}_{out}(s), (u', d'_2) \in \mathscr{L}_{in}(t)$ and $d'_1 + d'_2 < dist_G(s,t)$. Then it is guaranteed that the vertex pair (s,t) is covered by u. They proved that finding the minimum *2-hop cover* is NP-H and the lower bound of the label size is $\Omega(|V||E|^{1/2})$. [59, 60, 61] are the previous works considering labeling, but they only considered undirected graphs and the worst case results. *Hop-Doubling Label* [62] narrows the *2-Hop* approach on the scale-free graph which is directed and unweighted, i.e. the weight on edges are all one. Thus, they can use the degree of a vertex to approximate its betweenness centrality, which reflects the importance of a vertex in a graph by considering how many shortest paths go through this vertex. Their strategy is to rank all vertices uniquely according to non-increasing degrees, with the highest rank given to the highest degree vertex. This is due to the intuition that the higher ranked vertices are likely to hit more shortest paths. Then they generate label entries to cover shortest paths with increasing number of hops. These labels are generated based on a set of rules they proposed, and will be further pruned to reduce the label size.

Transit Node Routing (TNR) [63] is an approximation of *2-Hop* in road network. For a given road network, it computes a small set of transit nodes with the property that every shortest path between that covers a certain not too small euclidean distance passes through at least one of these transit nodes.

Then for each vertex in road network, it computes a set of closest transit nodes, which is as small as 10 on average on their US road network. Thus, it is not time-consuming to compute the distance between each vertex and these transit nodes on a local query. As for a non-local shortest distance query from v_s to v_d, they first retrieve the closest transit node sets T_s and T_d of v_s and v_d, respectively. After that, they have to compute all the shortest distance combinations $d(v_s, t_s) + d(t_s, t_d) + d(t_d, v_d)$, where $t_s \in T_s, t_d \in T_d$ and $d(v_s, t_s)$ is the distance between v_s and v_t. Among all the possible results, it uses the minimum one as the shortest distance.

Pruned Highway Labeling (PHL) [64] is a combination of labeling and transit node. Instead of maintaining a set of transit nodes, it decomposes the graph into disjoint shortest paths and computes a label for each vertex that contains the distance from it to the vertices in a small subset of such shortest paths. Thus, the shortest path from u to v can be expressed as $u \rightarrow p \rightarrow q \rightarrow v$, where $p \rightarrow q$ is a sub-path that appears in u and v's labels. It can take advantages of *pruned labeling* technique [65] to improve the efficiency.

Another stream of taking advantages of the hierarchical natural of road networks such that the long shortest paths can converge to a smaller arterial network of important roads like highways. *Contraction Hierarchies (CH)* [66] is one of these indexing techniques that uses the vertices' importance to impose a total order. The distances among the vertices are computed based on this total order. Obviously, the efficiency is determined by the total order. An inferior ordering can lead to $O(|V|^2)$ shortcuts, which in turn results in an $O(|V|^2 log|V|)$ time complexity for queries. The *CH* exploits the hierarchical nature of road network by contracting the vertices in a precomputed total order. A vertex v is contracted by removing it from the graph in such a way that the shortest paths in the remaining graphs are preserved. Such a property can be achieved by replacing paths of $< u, v, w >$ by a shortcut edge $< u, w >$. It should be noted that the shortcut $< u, w >$ is needed only if $< u, v, w >$ is the only shortest path from u to w. Since the optimal total order is a difficult problem, they applied a simple local heuristics approach which keeps the vertices in a priority queue sorted by some estimate of how attractive it is to contract a vertex. They use the *edge difference*, which is the number of shortcuts introduced when contracting v minus the number of edges incident to v, to do this estimation. The intuition behind this is that the contracted graph should have as few edges as possible.

[67] proposes a *hierarchical hub labellings*, which is designed for road networks. It is a natural special case where the relationship vertex v is in the label of vertex w defines a partial order on the vertices. They show that for every total order there is a minimum hierarchical labeling. They use

this theory to develop efficient algorithms for computing the minimum labeling from an ordering, and for computing orderings which yield small labellings.They also show that *CH* and *hierarchical labellings* are closely related and obtain new top-down *CH* preprocessing algorithms that lead to faster *CH* queries.

[68] proposes a *highway centric* for answering distance queries in a large sparse graph. Their scheme provides better labeling size than 2-hop both theoretically and empirically. Intuitively, it utilizes a tree structured highway to serve as the intermediaries to link the start vertex and the end vertex, a generalization of 2-hop which only utilizes a single vertex as the intermediary. Though highway structure is widely used in shortest path computation in road networks, it is primarily used for speeding up the online search [69, 70]. The heart of the highway-centric labeling consists of a fast greedy algorithm for a general bipartite set cover problem, which by itself is very interesting and useful as it significantly generalizes the classic set cover problem. Furthermore, as a side product, we are able to speed up 2-hop without sacrificing its labeling size (empirically offering even better results). This scheme also offers both exact and approximate distance with bounded accuracy. *Spatially Induced Linkage Cognizance* [71] aims to compress the all pair distance to $O(N^{1.5})$. *Path-Coherent Pairs Decomposition* [71, 72] are two approaches aiming to compress the all pair distance.

Landmark [73, 74] is another major approach which can return approximate result fast. The basic idea of these methods is to select a subset L of vertices as landmarks and pre-compute the distance $d_G(l, u)$ between each landmark $l \in L$ and all the vertices $u \in V$. When the distance between two vertices u and v is queried, they answer the minimum $d_G(u, l) + d_G(l, v)$ over landmarks $l \in L$ as an estimate. Generally, the precision for each query depends on whether actual shortest paths pass nearby the landmarks. Therefore, by selecting central vertices as landmarks, the accuracy of estimation becomes much better than selecting landmarks randomly [75, 76]. However, for close pairs, the precision is still much worse than the average, since lengths of shortest paths between them are small and they are unlikely to pass nearby the landmarks [77]. [78] presents two improvements to existing landmark-based shortest path estimation methods. The first improvement relates to the use of *shortest-path trees (SPTs)*. Together with appropriate short-cutting heuristics, the use of *SPT*s allows achieving higher accuracy with acceptable time and memory overhead. Furthermore, *SPT*s can be maintained incrementally under edge insertions and deletions, which allows for a fully-dynamic algorithm. The second improvement is a new landmark selection strategy that seeks to maximize the coverage of all shortest paths by the selected landmarks. [79] proposes a query-dependent local landmark scheme, which identifies a local landmark close to both query nodes and provides more accurate distance estimation than the traditional global landmark approach. They also propose efficient local landmark indexing and retrieval techniques, which achieve low offline indexing complexity and online query complexity. Two optimization techniques on graph compression and graph online search are also proposed, with the goal of further reducing index size and improving query accuracy. Furthermore, the challenge of immense graphs whose index may not fit in the memory leads us to store the embedding in the relational database, so that a query of the local landmark scheme can be expressed with relational operators. Effective indexing and query optimization mechanisms are designed in this context. [80] proposes an *ALT (A^*,Landmark & Triangle inequality)* algorithm, which is also a preprocessing-based technique for computing distance bounds. It is the first exact shortest path algorithm with preprocessing that can be applied to arbitrary graphs. It carefully chooses a small number of landmarks, then computes and stores the shortest distances between all vertices and each of these landmarks. Lower bounds are computed in constant time using these distances in combination with the triangle inequality.

[81, 77] propose approaches based on *tree decomposition* methodology. The graph is first decomposed into a tree in which the node (a.k.a. bag) contains more than one vertex from the graph. The shortest paths are stored in such bags and these local paths together with the tree are the components of the index of the graph. Based on this index, a bottom-up operation can be executed to find the shortest path for any given source and target vertices.

As indexing on the static graph is already complex and hard to create enough, little work has been expanded on the time-dependent graphs. Some of them will be discussed in the following sub-section.

2.3.2 Path Problems on Timetable Graph

Such a graph can be used to demonstrate the following networks:

- Transportation Network: Like train, plain and bus that have a time table. Each vertex represents a location, and an edge (u, v, t, λ) is a transportation from u to v departing at time t and the travel duration is λ.

- Social networks: Each vertex represents a person or an organization, and an edge is an interaction between u and v at time t which takes λ time.

- SMS, email or Phone call network: Each vertex models a person or a device and an edge indicates that u calls or sends a message to v at time t.

The path problems on timetable graph can be categorized into the following types:

1. **Earliest Arrival Path** is the path p that arrives v_d earliest, whose

   $p.Arrival(v_d) = min\{p'.Arrival(v_d), p' \in P\}$.

2. **Latest Departure Path** is a path p that departs v_s latest but still can reach v_d, whose

   $p.Depart(v_s) = max\{p'.Depart(v_s).p' \in P\}$

3. **Shortest Path** is a path p that has the shortest traveling distance $min\{\Sigma_{i=s}^{d} weight(v_i, v_{i+1})\}$.

4. **Fastest Path** is a path p that has the minimum total travel time

   $min\{p'.Arrival(v_d) - p'.Departure(v_s)\}$.

Path Computation

[82] applies two approaches computing these paths. The first one is using the edge stream. The edge stream is to present a graph only by edges. By sorting the edges in stream according to the start time, they are able to answer the above four queries in one-pass scan of all the edges. The earliest arrival path and the latest departure path can be answered in linear time, while the latter two is nearly the same as Dijkstra, except for the difference on maintaining the data structures. Although edge stream is not a new way to present graph, it is the first time to be used for solving this problem and achieves a promising performance. The second method is a traditional way, which views the timetable graph as a graph with multiple edges between each vertex pair. By viewing each edge in the edge stream as a single edge, the timetable graph can be derived to a static graph. The vertex with multiple out-edges can split into several copies of itself to meet the restriction that there should be only one edge between any vertex pair, as shown in Figure 2.2. Thus, the above four paths can be solved using the Dijkstra algorithm [50]. However, this transformation may result in a graph several times larger than the original one. If the number of the edges between the pair vertices are big, the size growth is exponential [83]. This approach is also applied by many earlier works that tried to simulate the dynamic network [84, 85, 86, 87, 80]. Another similar approach is *Connection Scan Algorithm* [88], which views the timetable graph as two sequences of edges, such that the first sequence sorts edges

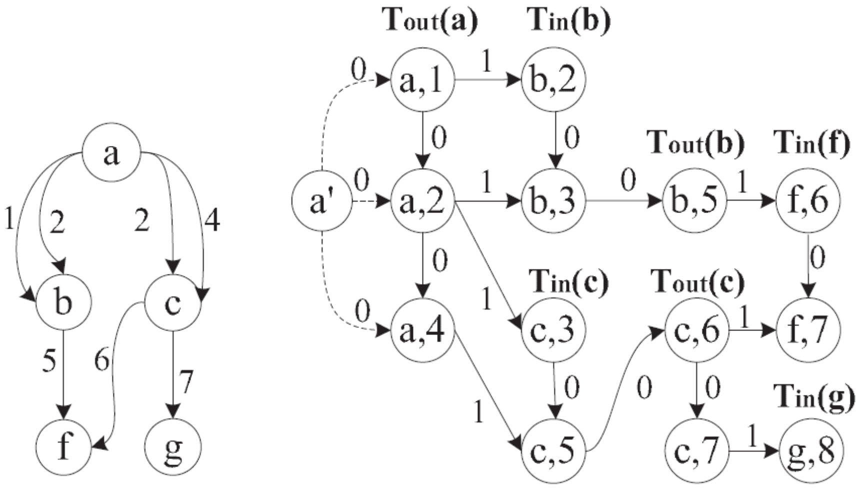


FIGURE 2.2: From Timetable Graph to Transformed Graph

in ascending order of their departure timestamps, while the second sequence in descending order of their arrival timestamps. For any path query, it derives the query answer using one linear scan of one of the edge sequences. This technique incurs very small preprocessing overheads, and is shown to outperform the temporal Dijkstra algorithm in terms of query time.

Although the algorithm discussed in this sub-section is different from the algorithm in time-dependent graph, some notions brought by them is quite useful. For example, *Earliest arrival time* and *latest departure time* will be used in our algorithm to solve the *least on road travel time* problem.

Path Query Answering

Like the works on the static graph, many works aim to build up precomputed index to boost the efficiency of query. [89] applies pre-computation based on node contraction: gradually removing nodes from the graph and adding shortcuts to preserve shortest paths.

[90] applies labeling on timetable graph to improve the query time. The basic idea of this approach is to associate each node v_i with a set of labels, each of which records the shortest travel time from v_i to some other node v_j given a certain departure time from u_j. Such labels would then be used during query processing to improve efficiency. [89] extends *CH* to work on the timetable graph. It pre-processes the graph by constructing shortcuts among the vertices, such that each shortcut captures a fastest route between the two vertices that it connects. During query processing, it employs a bidirectional Dijkstra-like search from the source and destination nodes simultaneously, and it utilizes the pre-computed shortcuts to reduce the number of nodes that need to be traversed. [91] proposes

T-Patterns to pre-compute a set of fastest paths and record them in a set S. When it comes to query processing, it utilizes the pre-computed paths to construct the major parts of the query results, which improves query performance. However, it does not guarantee exact query results, that is to say, its answer for an earliest arrival time query might not be an actual earliest arrival time. So the *T-Pattern* is to trade query accuracy for efficiency.

2.3.3 Path Problems on Time-Dependent Graph

The time-dependent graph on a road network first has to comply the *FIFO (First-In-First-Out)* property, which claims that a car drives into a road earlier than another car, then the time it leaves this road cannot be later than the later car. Such a property further requires a constraint on the general time-dependent graph, which requires the cost functions gradient has to be no smaller than -1 [21].

Given a time-dependent graph $G_T(V, E, W)$, a general path $p = ((v_1, v_2), (v_2, v_3)..., (v_{k-1}, v_k))$ which travels from v_1 to v_k can be decomposed into a series of travel segments as shown below:

$$p_{1,2} : \begin{cases} Arrival(v_1) = ts_1, \\ Depart(v_1) = Arrival(v_1) + waiting_1, \\ Cost_{1,2} = w(v_1, v_2, Depart(v_1)) \end{cases}$$

$$p_{2,3} : \begin{cases} Arrival(v_2) = Depart(v_1) + Cost_{1,2}, \\ Depart(v_2) = Arrival(v_2) + waiting_2, \\ Cost_{2,3} = w(v_2, v_3, Depart(v_2)) \end{cases}$$

$$\ldots$$

$$p_{k-1,k} : Arrival(v_k) = Depart(v_{k-1}) + Cost_{k-1,k}$$

The fast path algorithm can be categorized into two types based on if waiting on the departure vertex is allowed. If it is not, then it is a *Single Starting Time Fastest Path*, which can be solved by applying the Dijkstra Algorithm directly. If waiting on the departure vertex is allowed, then the problem is changed into finding the optimal departure time within this time interval that could result in a fastest path. It objective is to minimize $Arrival(v_k) - Departure(v_1)$.

Fastest Path Computation

Dreyfus [92] first shows the time-dependent fastest path problem is solvable in polynomial time if the graph is restricted to have FIFO property in 1969. Other early theoretical works are Halpern [93] in 1997 and Orda [94] in 1991. However, these algorithms are so complicated to implement that no works have reported any computational evaluation of them.

DOI [95] uses a discrete approach, which splits the starting time interval into k segments and computes the fastest paths of at boundary time points. Although its result is not optimal in theory, it is the most practical one since the speed profile is always histogram rather than linear function in reality. Thus, even if this approach is not as complex as the other theoretical methods, it is the most practical one. [86] extends the *Bellman-Ford* Algorithm [55]. It keeps refining the *earliest arrival time* function of each vertex to find the optimal starting time. The time complexity of it is $O(|V||E|\alpha(T))$, where $\alpha(T))$ is the time required in a function operation in interval T. Obviously, this high time complexity makes it infeasible to work on large or dense time-dependent graphs.

[96] uses an extension of A^* algorithm to find the fastest path. It maintains the corresponding *earliest arrival time* and *total traveling time* functions of all possible paths. It also applies a heuristic function which estimates the arriving time by the geographical distance divided by the current speed. It maintains a priority queue of all the paths to be expanded, sorted by the minimum heuristic value of each vertex's expected arrival time. When the path is expanded, it adds more than one edge to the queue, since they are different newly generated path. When a new path is generated, it also generates a new pair of functions. This algorithm is efficient only when the heuristic estimation can assist pruning the search space effectively, and v_s and v_e are closed to each other in graph. However, It is difficult to find such an estimation in general graphs, and the travel time estimation is still not good enough for the road network.

[21] proposes a *Dijkstra*-based algorithm to solve the problem faster than the previous methods. Unlike the [96] that maintaining the priority queue of paths, it maintains a priority queue for each vertex, sorted by its earliest arrival time by its current expanded upper bound of the starting time interval (smaller than the original upper bound). And by increasing the starting time step by step, it refines the earliest arrival time function of each vertex step by step. When the starting time reaches its upper bound, the earliest arrival time function of each vertex is set, and the fastest traveling time can be retrieved. It achieves a much faster query time than the [96]. Its time complexity is $O(\alpha(T)(|V|\log|V| + |E|))$, where $O(|V|\log|V| + |E|)$ is due to the *Dijkstra*-based algorithm

structure, and the $\alpha(T)$ is time to maintain the earliest arrival time functions.

Another stream of finding the fastest path on road is by mining the trajectories. [97] proposes a mining-based algorithm *PATE* to predict the estimated travel time. This paper uses a travel time evaluation table to find the shortest path within a user-specified travel time constraint. In addition, a prefix-tree-based structure, called *NPST*, is proposed to efficiently find the shortest navigation path. However, all of the researches were focused on the path finding problem with single destination. [98] proposes navigation pattern mining from web navigation database to find a series of consecutive patterns which satisfy the user-specified minimal support threshold. The algorithm is divided into three parts: (1) maximal forward references, (2) large reference sequences, and (3) maximal reference sequences. The navigation pattern is defined as the consecutive and acyclic pattern. [99] proposes techniques for popular path mining from navigation log to discover the consecutive patterns which satisfy a minimal frequency threshold. [100] developes a framework, called *Trajectory-based Path Finding (TPF)*, which is built upon a novel algorithm named *Mining-based Algorithm for Travel time Evaluation (MATE)* for evaluating the travel time of a navigation path and a novel index structure named *Efficient Navigation Path Search Tree (ENS-Tree)* for efficiently retrieving the fastest path. With *MATE* and *ENS-tree*, an efficient fastest path finding algorithm for single destination is derived. To find the path for multiple destinations, they propose a strategy named *Cluster-Based Approximation Strategy (CBAS)*, to determine the fastest visiting order from specified multiple destinations.

As discussed above, none of the existing algorithms can compute the *MORT*. They all aim to find the shortest total traveling time, i.e. find the optimal departure time that can result in the shortest travel time. And applying to compute the approximate *MORT* is both slow and not close the optimal result. So we need to propose a new algorithm to solve this new path problem. However, the techniques used by these algorithms are inspiring and will be expanded to solve our problem.

Fastest Path Query Answering

We classify and compare the existing approaches, in terms of index construction time, query answering time and index size, as illustrated in Figure 2.3. Such a classification also applies for reachability query [14, 22] and distance query [62, 65] on static graph. On one extreme, we can compute the fastest path between each vertex pair by calling the existing fastest path algorithms [96, 21, 25, 95, 101, 24] directly. It consumes no extra space because it has no index at all. However, even the fastest one of them has to take several seconds to return the result on a graph not too big, which is too slow to serve

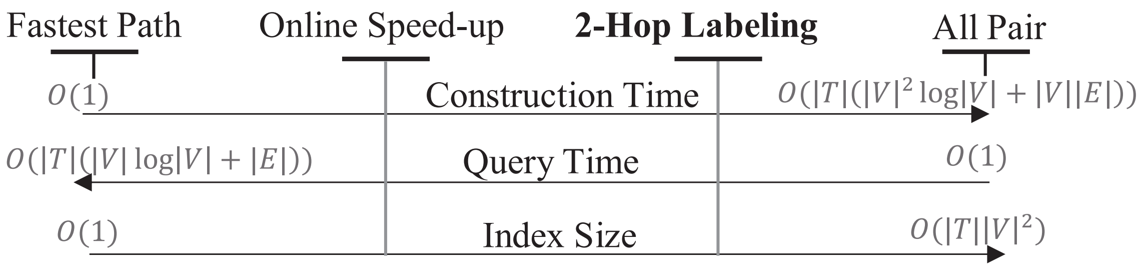


FIGURE 2.3: Comparison between Time-Dependent Indexes

as the foundation for other applications. On the other extreme, we could answer this query in $O(1)$ time by retrieving the result from the precomputed all-pair fastest paths between all vertices during the whole time span. But such an approach is not applicable even on static graph due its unrealistically long preprocessing time ($O(|V|^2 \log |V| + |V||E|)$)) and quadratic large index size ($O(|V|^2)$), not to mention multiplying a large T on them. Therefore, some speed-up techniques are proposed for answering fastest path query in an online search way, extended from their original static versions. For example, *Bidirectional A* Search* [102] augments *A* Search Using Landmarks (ALT)* [103, 80, 104], *Time-dependent Contraction Hierarchies* [105, 106] expands the original *CH* [66], *Time-dependent SHARC* [107] extends *Arc-Flag* [108, 109] and *Time-dependent CH*, and *Core-ALT* [110, 111] combines *Landmark, bidirectional search* and *Contraction*. However, since these speed-up approaches still have to traverse the graph, they remain time-consuming. And that is why although they claim they can achieve fastest path query answering time in *ms* level, they all conduct their experiments on the histogram-based speed profile which constrains T to a small fixed number and much easier and faster to compute, rather than using the linear piecewise functions which is widely used in original fastest path algorithms. Nevertheless, the benefits of them are the shorter construction time and smaller space consumption.

Chapter 3

Reachability on Graph

Efficiently answering reachability queries, which checks whether one vertex can reach another in a directed graph, has been studied extensively during recent years. However, the size of the graph that people are facing and generating nowadays is growing so rapidly that simple algorithms, such as *BFS* and *DFS*, are no longer feasible. Although *Refined Online Search* algorithms can scale to large graphs, they all suffer from the *false positive* problem, which deteriorates their performance. In this chapter, we analyze the cause of *false positive* and propose an efficient high dimensional coordinate generating method based on Graph Dominance Drawing to answer reachability queries in linear or even constant time. We conduct experiments on different graph structures and different graph sizes to fully evaluate the performance and behavior of our proposal. Empirical results demonstrate that our method outperforms state-of-the-art algorithms and can handle extensive graphs. As for the reachability query in road network, our approach can always answer it in constant time.

3.1 Introduction

A reachability query $Q(G, s, t)$ checks whether the source vertex s can reach the destination vertex t in a directed graph $G(V, E)$, where V is the vertex set and E is the edge set. The reachability operation is an essential step for map data processing, which could help prune out those standalone vertices. It can also prevent a useless but time-consuming search from the beginning if one vertex cannot reach another. Apart from the applications on the road network, it is also a fundamental step for various higher-level applications. For example, reachability queries can be used to conduct the ancestor-descendant search in XML databases [4, 112], to estimate a given molecule's influence on the expression of genes in biological networks[113], or to determine whether two users are related in social networks [114, 115]. Other applications include website analysis [116, 117], web usage mining [118, 119], and so forth.

In the early years when the graphs were small, basic algorithms like *BFS/DFS* (Breath-First Search / Depth-First Search) were efficient enough. However, with the rapid development of the Internet, e-commerce services, social networks, road networks and many other applications, the sizes of graphs have increased dramatically in recent years, making it indispensable to answer reachability queries efficiently on extensive graphs. Tremendous efforts have been devoted to designing novel indexing or querying strategies for reachability checking. The detailed reachability problem related work is presented in Chapter 2.

Our work falls into the third category of *Refined Online Search*, which is also called *Topological Cover* approach. By attaching topological information to each vertex as a label within linear time, *Topological Cover* can easily scale to large graphs, although it is slower in query time than *Set Cover* approaches. Note that all the existing *Topological Cover* approaches suffer from the *false positive* problem, where the positive result of label comparison cannot guarantee that two vertices are reachable. In other words, the unreachable relationship can be retrieved directly from the labels within constant time, while the reachable relationship detection decays into a guided or pruned *BFS/DFS*. *Grail* [12], *FERRARI* [13] and *FELINE* [14] are the state-of-the-art Topological Cover algorithms.

Rather than using the time-consuming pruned *BFS/DFS* to cope with the false positive issue, a better alternative to guarantee the accuracy of reachability queries is to reduce the false positive ratio in underlying index. This in turn calls for a new topological labeling approach. Our study is inspired by the graph drawing approach adopted by *FELINE* [14] and the high dimension strategy applied by *IP* [18]. We analyze the cause of false positive problem and observe that false positive can be reduced by increasing the dimension of labels. We propose a *High Dimensional Graph Dominance Drawing* (*HD-GDD*) approach for exact reachability queries. However, it is nontrivial to find the optimal labels which minimize both false positive ratio and index size at the same time. To this end, we introduce several heuristic rules to guide index construction, and propose two refinement approaches (i.e., *false positive cache* and *false positive removal*) to further speed up reachability checking with the sacrifice of index size or index construction time. We examine the behavior of our algorithm on different graph structures, and conduct extensive experiments on real large graphs to compare the performance of our approach and existing state-of-the-art *Refined Online Search* methods. We summarize the main contributions of our work as follows.

- We analyze the cause of false positive suffered by all refined online search approaches;
- We propose a High Dimensional Graph Dominance Drawing (*HD-GDD*) approach for fast index construction and fast reachability queries;
- We propose two refinement approaches, namely *false positive cache* and *false positive removal*, to further improve the query efficiency;
- We empirically analyze the behavior of our approach on several types of synthetic and real world graphs. The experimental results verify our analysis of the cause of the false positive problem, and demonstrate that our proposal outperforms state-of-the-art *Refined Online Search* algorithms on extensive graphs.

The rest of the chapter is organized as follows. In Section 3.2, we introduce some preliminaries and analyze the false positive problem. In Section 3.3, we summarize the related works of graph drawing. Section 3.4 describes our index construction algorithm and reachability query algorithm in detail, followed by complexity analysis and two refinement approaches. Section 3.5 demonstrates the experimental analysis of *HD-GDD* and its performance on real graphs. Finally, Section 3.6 make a conclusion of this chapter.

3.2 Problem Statement

3.2.1 Reachability, SCC and DAG

Definition 3.1. *(Reachability) Given a directed graph $G = (V, E)$ where V is the vertex set and E is the edge set, a reachability query $Q(G, s, t)$ asks whether there exists a directed path from source vertex s to destination vertex t in G.*

We use $s \to t$ to represent s can reach t in G and $s \nrightarrow t$ otherwise. In other words, the reachability query $Q(G, s, t)$ returns *True* if $s \to t$ and *False* otherwise. If both $s \to t$ and $t \to s$ hold at the same time, we say vertices s and t are strongly connected.

Definition 3.2. *(Strongly Connected Component, SCC) Given a directed graph $G = (V, E)$ where V is the vertex set and E is the edge set, an SCC of G is defined as a maximal subset of V where any two vertices are strongly connected.*

TABLE 3.1: Reachability Important Notations

Notation	Description
C_u	Coordinate of vertex u $(C_{u,1}, C_{u,2}, ..., C_{u,k})$
T	Topological ordering $(t(v_1), t(v_2), ..., t(v_{\|V\|})$
T	Set of all topological orderings $\{T^1, T^2, .., T^i, ...\}$
$l(v)$	Topological level number of vertex v
$\{L_0, L_1, .., L_t\}$	Topological level set, $L_i = v\|l(v) = i$

Definition 3.3. *(Directed Acyclic Graph, DAG) Given a directed graph $G = (V, E)$ where V is the vertex set and E is the edge set, we can condense G into a DAG $G' = (V', E')$ where*

- *V' is the set of SCCs of the original graph G;*
- *E' is aggregated from the edge set E of the original graph G. Specifically, if $e = (u, v) \in E$, then $e' = (SCC_u, SCC_v) \in E'$ where SCC_u denotes the SCC u belongs to in G.*

After condensing the original graph G into a DAG G', a reachability query $Q(G, s, t)$ from source vertex s to destination vertex t in G can be answered by a transformed reachability query $Q(G', SCC_s, SCC_t)$ between the SCCs s and t belong to. In particular, $s \to t$ in G iff 1) s and t are in the same *SCC* or 2) $SCC_s \to SCC_t$ in the corresponding DAG G'. The condensing process from the original graph to a DAG can be conducted by Tarjan's algorithm [120]. In this chapter, we assume that the input graph of our algorithm has already been converted into a DAG.

3.2.2 Graph Drawing and False Positive

Graph drawing is a graph research area of mathematics and computer science, which tries to depict a graph into different visualization categories to meet various needs. There is a branch of graph drawing methods that aims to map the vertices of a graph into a k-dimensional coordinate system such that each vertex is assigned a k-dimensional coordinate, where $k \geqslant 2$. In this way, some properties of the original graph can be derived by computations on the corresponding coordinates. For example, *dominance drawing* can be used to answer reachability queries.

Definition 3.4. *(Dominance Drawing) Given a DAG G, we assign each vertex u in G with a coordinate $C_u = (C_{u,1}, C_{u,2}, ..., C_{u,k})$ such that $s \to t$ iff $\forall i \in [1, k], C_{s,i} < C_{t,i}$. k is called the lowest satisfying dimension number.*

Based on the dominance drawing of a DAG G, we can answer the reachability query $Q(G, s, t)$ from vertex s to t by simply comparing their coordinates. More specifically, 1) if $\forall i \in [1, k], C_{s,i} < C_{t,i}$, then $s \rightarrow t$; 2) if $\exists i \in [1, k], C_{s,i} \geq C_{t,i}$, then $s \nrightarrow t$. However, there is no theory or algorithm that can calculate the exact dominance drawing dimension k of a given graph. k can be extremely large in real graphs. Since no theory can give an upper bound of k, the original dominance drawing method cannot be applied directly. Therefore, a weaker version is defined as below:

Definition 3.5. *(Weak Dominance Drawing) Given a DAG G, we assign each vertex u in G with a coordinate $C_u = (C_{u,1}, C_{u,2}, ..., C_{u,d})$ such that $s \rightarrow t$ only if $\forall i \in [1, k], C_{s,i} < C_{t,i}$. In other words, if $\exists i \in [1, d], C_{s,i} \geq C_{t,i}$, then $s \nrightarrow t$.*

Weak dominance drawing dimension d is a number much smaller than the dominance drawing dimension k. Weak dominance drawing sacrifices the accuracy of reachability queries for less storage cost of coordinate system. According to Definition 3.5, weak dominance drawing can only guarantee unreachability rather than reachability. More specifically, 1) if $\exists i \in [1, d], C_{s,i} \geq C_{t,i}$, then $s \nrightarrow t$; but 2) even if $\forall i \in [1, d], C_{s,i} < C_{t,i}$, t is not necessarily reachable from s. We call such a phenomenon *false positive*, which means although the coordinate comparison result is positive, the reachability relation is false.

Definition 3.6. *(False Positive) Given the weak dominance drawing of a DAG G, if $\forall i \in [1, d], C_{s,i} < C_{t,i}$ but $s \nrightarrow t$, then C_s and C_t is a false positive pair.*

Consider the example in Figure 3.1. The 2-dimensional coordinate assignment is a weak dominance drawing of the graph. Vertex $A(1, 4)$ is not entirely smaller than vertex $B(2, 2)$, and thus $A \nrightarrow B$. Such unreachability information can be derived safely by comparing the 2-dimensional coordinates. As for reachability, although $E(5, 5)$ is entirely larger than $B(2, 2)$, B cannot reach E (as shown by a dotted arrow). Therefore, the vertex assignment pair $B(2, 2)$ and $E(5, 5)$ is a false positive pair which cannot be resolved by the 2-dimensional weak dominance drawing in Figure 3.1.

Definition 3.7. *(Topological Ordering) The topological ordering of a DAG $G = (V, E)$ is a linear ordering of its vertices denoted as $T = (t(v_1), t(v_2)..., t(v_{|V|}))$ such that for every directed edge $e = (u, v) \in E$ from vertex u to v, $t(u) < t(v)$.*

Theorem 3.1. *If $t(u) > t(v)$, then $u \nrightarrow v$.*

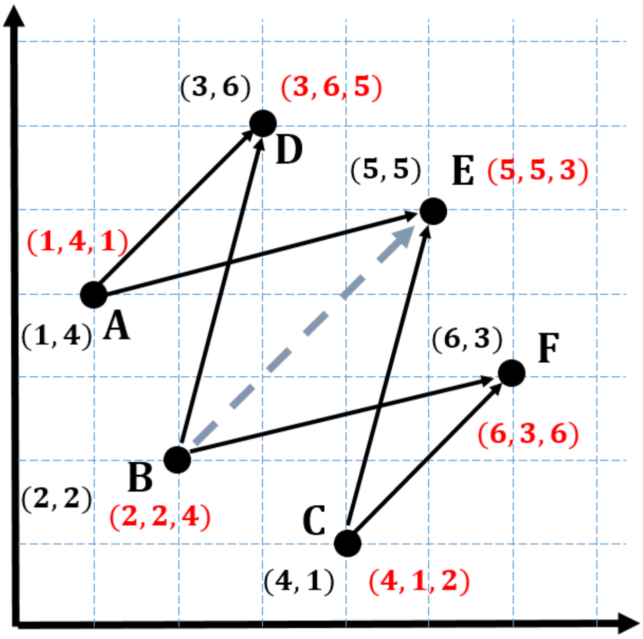


FIGURE 3.1: False Positive Example

Proof. Assume to the contrary that $u \to v$, then there exists a directed path form u to v, denoted as $(u, v_{p,1}), (v_{p,1}, v_{p,2}), ..., (v_{p,j}, v)$. According to Definition 3.7, $t(u) < t(v_{p,1}) < t(v_{p,2}) < ... < t(v_{p,j}) < t(v)$, namely $t(u) < t(v)$. This causes a contradiction. □

Theorem 3.2. *If $u \nrightarrow v$, then $\exists\ T^i \in \{T\}$, where $\{T\}$ is the set of all the possible topological orderings of a graph, $t^i(u) > t^i(v)$.*

Proof. We can safely deduce Theorem 3.1 to "If $\exists\ T^i \in \{T\}$ that $t^i(u) > t^i(v)$, then $u \nrightarrow v$". Now let's assume the contrary of Theorem 3.2: If $u \nrightarrow v$, then $\forall\ T^i \in \{T\}$, $t^i(u) < t^i(v)$, and its contrapositive statement is "If $\exists\ T^i \in \{T\}$ that $t^i(u) > t^i(v)$, then $u \to v$", which is contrary to the deduced version of Theorem 3.1. □

Based on Theorem 3.1 and 3.2, topological ordering can be used to derive the unreachable relationship. In real applications, weak dominance drawing is usually implemented by running d rounds of topological orderings. At the same time, we can reduce the number of false positive pairs in a weak dominance drawing by increasing the dimension number d.

Theorem 3.3. *The false positive between a vertex pair $C_u = (C_{u,1}, C_{u,2}, ..., C_{u,d})$ and $C_v = (C_{v,1}, C_{v,2}, ..., C_{v,d})$ in a d-dimensional weak dominance drawing can be eliminated by adding a new dimension to the coordinates.*

Proof. Since $u \nrightarrow v$, there exists a topological ordering where $t(u) > t(v)$ according to Theorem 3.2. By adding this topological ordering as a new dimension of the coordinates, we can achieve $C_{u,d+1} > C_{v,d+1}$. This breaks the false positive pair between C_u and C_v. □

For example, using the 3-dimensional weak dominance drawing in Figure 3.1, we can safely draw the conclusion that $B \nrightarrow E$ since $B(2, 2, 4)$ is not entirely smaller than $E(5, 5, 3)$.

To further understand the cause of false positive and the difference between *Set Cover* and *Topological Cover*, we need the following definitions:

Definition 3.8. *(Topological Level Number) Given a DAG $G = (V, E)$, the topological level number $l(v)$ of a vertex v is defined as below:*

- *if v has no in-neighbor, then $l(v) = 0$;*
- *otherwise, $l(v) = \max\{l(u) + 1$: u is v's in-neighbor$\}$.*

Definition 3.9. *(Topological Level) Given a DAG $G = (V, E)$, it contains t topological levels denoted as $L_0, L_1, ..., L_t$, where $L_i = \{v|l(v) = i\}$ and t is the largest topological level number in G.*

Theorem 3.4. *If $l(u) = l(v)$, then $u \nrightarrow v$ and $v \nrightarrow u$*

Proof. Assume to the contrary that $u \rightarrow v$, then there exists a directed path from vertex u to v, denoted as $(u, v_{p,1}), (v_{p,1}, v_{p,2}), ..., (v_{p,j}, v)$. According to Definition 3.8, $l(v) \geq l(v_{p,j}) + 1 \geq ... \geq l(v_{p,1}) + j \geq l(u) + j + 1$, namely $l(v) > l(u)$. This causes a contradiction. Therefore, if $l(u) = l(v)$, then $u \nrightarrow v$. Similarly, we can prove that $l(u) = l(v)$, then $v \nrightarrow u$. □

From Theorem 3.4, we can easily observe two essential differences between *Set Cover* approach and *Topological Cover* approach described in Table 2.1. First, in *Set Cover*, vertices in the in-neighbor set consistently have a smaller topological level number than those in the out-neighbor set, and vertices of the same topological level will never appear in each other's neighbor sets. However, in *Topological Cover*, vertices of the same topological level can have either a bigger or smaller topological order than each other during different rounds of topological ordering, which is one reason for the *false positive* problem (another reason is the unreachable pairs from different topological levels). Second, *Set Cover* uses the information positively to answer reachability queries, while *Topological Cover* uses the information negatively. For example, in *Set Cover*, if u and v can find each other in corresponding neighbor sets, their reachability can be answered directly. Whereas in *Topological*

Cover, the unreachability can be answered directly if u has a larger topological order than v in some topological ordering.

3.3 Related Works

[15] is the first work that tries to use graph drawing to answer reachability query. Their goal is to label the vertices by vectors in such a way that u can reach v if and only if $C_u < C_v$, where C_u is a d-dimensional vector that is assigned to vertex u. Although their goal is promising, they only focus on the graphs that are planar. That is to say, all the vertices with no in-neighbors are on the boundary of the same face while all the ones with no out-neighbors are on the other face. Then they use a simple left-*DFS* followed by a right-*DFS* algorithm to compute a 2-dimensional vector for each vertex. However, this planar condition is so strong that it does not allow the existence of false positive. In fact, it is the feature of non-planar that causes the false positive. So their algorithm is limited to a small type of graphs.

Twenty years later, [16] expands the condition from planar graph to bipartite graph, which is made up of a set of source vertices and a set of sink vertices, and all edges are directed from a source to a sink. They first call the vector assignment approach the *dominance drawing* and prove the *dominance drawing dimension* d is no bigger than $min(|source|, |sink|)$. Then they give an algorithm to compute a 2-dimensional dominance drawing in the case of such drawing exists, which is also a strong assumption. So they just avoid the false positive problem.

[17][29] revisited this problem and proposed *weak dominance drawing* to expand the above algorithm to any graph type, which can only guarantee the unreachability relation between two vertices if the source is not smaller than the destination. They give the upper bound of false positive pairs, which is $inc(G) - (dim(G) - 2)$, where $inc(G) = \frac{|V|(|V|-1)}{2} - |E^*|$ is the number of incomparable pairs of a graph, $dim(G)$ is the smallest graph dominance drawing dimension, and $|E^*|$ is the number of reachable pairs. With the weak dominance drawing, they attached a two dimensional coordinate to each vertex regardless it is planar or not. Thus, this was the first time that the graph drawing approach was using on any graph type. And they proved that finding the optimal weak dominance drawing is *NP-C*. Although they did a lot theoretical work, no experiment result was released.

FELINE [14] adopts the weak dominance drawing concept from Kornaropoulos's work and treats it as a *Refined Online Search* method. All they do is to use the ancient 2-dimensional vector generating algorithm[15] to give each vertex a coordinate and use it to rule out the unreachable vertex pairs.

As discussed above, all of the works on graph dominance drawing algorithms are actually based on the 1975's 2-dimensional graph drawing algorithm, and no work has been done to extend it to a higher dimension, which will definitely reduce the false positive pairs and boost the efficiency of the reachability query while still has a linear index size and fast index construction time.

There is another similar research trend called *Graph Embedding* [121, 122] that also could assign a low dimensional vector to each vertex. The outputs are similar as the vectors serve as coordinates. However, the purpose of these methods is to reduce the cost of the graph analytics tasks, which involves node classification, recommendation and prediction. Therefore, the vectors are assigned with different similarity functions in order to preserve the structure of the graph maximumly. While the *Graph Drawing* simply asks for visualizing a graph, and the *Node Embedding* in *Graph Embedding* is a branch of it. In this work, we use topological information rather than similarity to solve the reachability problem, so the existing *Graph Embedding* methods cannot help.

3.4 High Dimensional Graph Dominance Drawing Algorithm

3.4.1 Overview

In this work, we propose a high dimensional weak dominance drawing approach to answer reachability queries. The workflow consists of two parts:

- ***Index Construction***. Given a DAG $G = (V, E)$, we assign a d-dimensional coordinate to each vertex in G using topology ordering. After the first dimensional coordinates have been assigned, we propose several heuristic rules to assign higher dimensional coordinates;
- ***Reachability Query***. Given a pair of vertex s and t, we first compare their coordinates. If $C_s \not< C_t$ namely $\exists i \in [1, d], C_{s,i} \geq C_{t,i}$, then we can conclude that $s \nrightarrow t$; Otherwise, we check whether the descendants of s can reach t iteratively until the search reaches t or all the vertices on the search path are labeled with coordinates not smaller than C_t.

3.4.2 Index Construction

During index construction, we adopt topological ordering to assign coordinates. Before jumping into details, we first discuss a possible solution.

Definition 3.10. *(Valid Coordinate)* $\forall d < k$, *where* d *is the current dimension of the coordinate and* k *is the minimum coordinate dimension required to eliminate all false positive pairs, the coordinate of dimension* $d + 1$ *is a valid coordinate if it eliminates at least one false positive pair of the previous coordinate.*

In order to build a feasible coordinate index for a given DAG $G = (V, E)$, the coordinate for each dimension should be valid. However, it takes $O(|V|^2)$ time to test whether a new dimension is valid or not. A naive solution for index construction is to apply greedy algorithm. Specifically, before adding a new dimensional coordinate, we iterate all vertex pairs and sort vertices based on the time they act as the source of *false positive*. Thus, by dealing with the vertex which acts more times as the source of false positive, the elimination of false positive can be guaranteed. During the next topological ordering, we eliminate false positives according to Theorem 3.3. Although the naive approach can guarantee that the new coordinate is valid, its time complexity is still $O(|V|^2)$, which is too time-consuming for large graphs.

As discussed in Section 3.2.2, weak dominance drawing is a trade-off between the storage cost of coordinate index and the accuracy of reachability queries. Therefore, the optimal coordinate index of a given graph is the one that minimizes the dimension number k and the false positive pairs at the same time. However, finding the optimal dimension number k of a general graph is still an open problem and is out of the scope of our current work. In fact, the only exact k people know is for *Bipartite Graph*'s [16]. Moreover, minimizing false positive pairs for a given dimension number d is a NP-H problem [17, 29]. Therefore, we propose several heuristic rules to construct the coordinate index effectively and efficiently.

In order to eliminate false positive pairs, we need to assign different topological orders in different rounds of topological ordering to the vertices that can not reach each other. Suppose at a certain stage of topological ordering to calculate the $(d + 1)^{th}$ dimensional coordinate, we maintain a set of candidate vertices which satisfies the following conditions:

- Each vertex has a d dimensional coordinate;

- Each vertex has no in-neighbors (it has no in-neighbors or all its in-neighbors have been visited in previous topological search);

- Cannot reach each other.

The vertices in the candidate set are sorted based on some heuristic rules which we will describe later. We choose vertex u with the largest order from the candidate set to be the next visited vertex and assign $C_{u,d+1}$ an increasing topological order. In fact, the newly assigned topological order is the smallest one in the candidate set, which is intended to eliminate *false positive* from this vertex to the remaining vertices in the candidate set. After deleting it from the set, we update the candidate set by adding the vertices that were connected to the erased vertex and the in-degrees are 0 currently. For example in Figure 3.1, assume the first round of topological sorting is finished and the first dimensional coordinates are generated. In the beginning of the second round of topological sorting, the candidate set is $\{A(1), B(2), C(4)\}$. Suppose the ordering here is opposite to the first coordinate. Then $C(4)$ is visited first and its coordinate is $C(4, 1)$ now. No new vertex is added to the candidate set $\{A(1), B(2)\}$ and $B(2)$ should be visited next and its coordinate is $B(2, 2)$. After B is removed, F's in-degree becomes 0, so $F(6)$ is inserted into the candidate set, which becomes $\{A(1), F(6)\}$ now. Since $F(6)$ is bigger, so its coordinate becomes $F(6, 3)$. This approach runs on till the second round of topological sorting is finished. Thus, we need to define an ordering of the candidate vertices' current coordinates. With an appropriate ordering, we can select the vertex with the largest order each time from the candidate set and assign it a coordinate smaller than all the other vertices in the set. The candidate set can be further indexed by a tree or heap. The followings are some heuristic conditions that could determine the order in candidate set:

- ***Coordinate comparison***. If $\forall i \in [1, d], C_{u,i} < C_{v,i}$, then we make $C_{v,d+1}$ bigger than $C_{u,d+1}$ in this iteration so that v can be topologically visited earlier than u and can have a smaller order than in the previous ones, which can break a false positive pair in the candidate set directly. However, not all the coordinates can be compared to each other, especially when the dimension is higher. So other conditions are needed to further determine the ordering.

- ***Sum of the coordinate***. As shown in Figure 3.1, vertex $A(1, 4)$, $C(4, 1)$ and $E(5, 5)$ should be visited earlier than $(2, 2)$ so that E can have a smaller topological order than B this time. Obviously, they all have a bigger sum of the coordinates ($\sum_{j=1}^{d} C_{i,j}$). So if C_u has a bigger sum than C_v, C_u is bigger in the candidate set.

- ***Deviation of the coordinate***. Also as indicated in Figure 3.1, if we want to set vertex v free so that it can be inserted into the candidate set, we have to visit (1,4) and (4,1) earlier than B. There exists cases that when applying the condition 2, the sum of the coordinates are the same.

Thus we have to further compare the deviation of coordinates. It is obvious that the vertices on the *outer* side of the graph have a bigger deviation, which satisfies our need to set E free. So the tie of the coordinate sum can be broken by a higher deviation.

- ***Minimum of the previous coordinate***. When all the previous conditions are all tied, which is possible, we have to settle the order by examining the minimum topological order a vertex has in the previous topological sorting.

With all the theorems and heuristic conditions discussed above, we can describe Algorithm 1 in detail. The index construction algorithm is made up of 2 steps:

1. *First coordinate computation*. The algorithm applies the topological sort in a DFS manner and sets each vertex a first coordinate $C_{i,1}$ (Line 2).

2. *Higher coordinate computation*. In order to visit the graph topologically later on, we have to collect the vertices' in-degree information first (Line 4-7). Then we compute the higher dimensional coordinates iteratively. Suppose we have a coordinate of d^{th} dimension for each vertex. Now we want to compute the $(d+1)^{th}$ dimension for each vertex. First, the algorithm inserts all the vertices that have no in-neighbors into a tree with the order defined by the above heuristic conditions (Line 12-14). Then, we select the largest vertex u from the tree and assign it the current coordinate and erase it from the tree. After that, we scan u's neighbors to reduce their in-degrees (Line 21). If any of the neighbor's in-degree becomes 0, we insert it into the tree (Line 22-23). And this operation keeps running until the tree becomes empty. When the dimension is finished, a $Sample()$ procedure is called to randomly test the performance of the index.

Algorithm 1: Index Construction

```
Input: A directed acyclic graph $G(V, E)$, the expected dimension number $k$.
Output: The $d$-dimension coordinate $(C_{i,1}, C_{i,2}..., C_{i,k})$ of each vertex $i$.
begin
    $C_{i,1} \longleftarrow$ TopologicalOrdering($G$) //Collect the in-degree of each vertex
    $IN \longleftarrow \{0, 0, ..., 0\}$
    forall $u \in V$ do
        forall $(u, v) \in$ u's edge list do
            $IN_v \longleftarrow IN_v + 1$
    for $d \leftarrow 2$ to $k$ do
        $IN2 \longleftarrow IN$, $tree.init()$, $coor \longleftarrow 1$
        forall $u \in V$ do
            if $IN2_u = 0$ then
                $tree.insert(v)$
        while tree is not empty do
            $u \longleftarrow tree.rightmost$, $C_{u,k} \longleftarrow coor$, $coor \longleftarrow coor + 1$
            $tree.erase(tree.rightmost)$
            forall $(u, v) \in u's\ edge\ list$ do
                $IN2_v \leftarrow IN2_v - 1$
                if $IN2_v = 0$ then
                    $tree.insert(v)$
```

3.4.3 Reachability Query

Algorithm 2 demonstrates the pseudo-code for the reachability query. If C_u is not entirely smaller than C_v, we can safely draw the conclusion that u cannot reach v. Otherwise, the algorithm has to traverse descendants of u to check whether they can reach v. Only the descendants that have not been visited and have a smaller coordinate than v need to be inserted into the queue for further testing (Line 9). In fact, such a condition prunes out most of the search space. If none of the descendants satisfies this condition, we can draw the conclusion that u cannot reach v. If the search process reaches v, the algorithm returns *True*.

Correctness

During the online search process, if $C_u \nprec C_v$ then $u \nrightarrow v$ according to Theorem 3.1; otherwise, the algorithm iteratively checks the descendants of u, denoted as u', to see whether they can reach v. Similarly, if $C_{u'} \nprec C_v$, u' and all its descendants cannot reach v. In other words, u cannot reach v through its descendant u'. When the queue becomes empty, it means none of u's descendants can reach v, which in turn guarantees u cannot reach v. If the search process reaches v, a directed path from u to v has already been detected, namely u can reach v. Therefore, Algorithm 2 returns correct answers for the reachability queries.

```
Algorithm 2: Reachability Query
Input: A directed acyclic graph $G(V, E)$, a source vertex $u$ and a destination vertex $v$, the
       $d$-dimensional coordinates $C_u$ and $C_v$
Output: True if $u$ can reach $v$, False if $u$ cannot reach $v$
begin
    if $C_u < C_v$ then
        queue.push(u)
        while queue is not empty do
            $u' \longleftarrow$ queue.pop()
            forall $(u', v') \in u'$ edge list do
                if $v' = v$ then
                    return True
                else if $v'$ has not been visited AND $C_{v'} < C_v$ then
                    queue.insert($v'$)
    return False
```

3.4.4 Complexity

During the index construction, both the first coordinate computation and the collection of in-degree information require $O(|V| + |E|)$ time, and the higher coordinate computation runs for $k - 1$ times. Within the higher coordinate computation, it spends $O(|V|)$ time on tree initialization, and runs tree insertion for $|V|$ times with each insertion costing $O(k * log|V|)$ time. Therefore, the total time complexity of the index construction algorithm is $O(k * (|V| + |E| + |V| * k * log|V|)))$. The space consumption during index construction is the storage of the graph $O(|V|+|E|)$ and the tree $O(k*|V|)$. Thus the total space complexity of the index construction algorithm is $O(k*|V|+|E|)$. In other words, both the time complexity and space complexity are nearly linear to $O(|V| + |E|)$.

The online query algorithm takes $O(1)$ time when the unreachability information can be obtained directly through coordinate comparison, and $O(|V| + |E|)$ time in other cases. Although the worst case complexity is the same as that of *BFS/DFS*, our query algorithm is still much faster since it uses coordinates to prune out most of the search space. Therefore, the query time complexity is between $O(1)$ and $O(|V| + |E|)$, and it depends on the graph structure.

3.4.5 Refinement

The *HD-GDD* approach works well when the dimension number is not too high. However, it can be observed from the experiment that the number of the eliminated false positive pairs is decreasing as the dimension grows higher. In some cases, no false positive pair can be eliminated by adding a series of coordinate dimensions. Although all the false positive pairs will be eliminated eventually, the cost between adding a new dimension and eliminating a few or even none positive pair is high when the dimension number is high. To further improve the accuracy of the labels, we propose two refinement approaches: *false positive cache* and *false positive removal*.

False Positive Cache

It aims to reduce the index complexity when the coordinate dimension number is high and the remaining false positive number is low. It just saves the false positive pairs the algorithm gets from the online search. When a reachability query comes, it first checks the false positive cache of the source vertex to check if it is a false positive pair. As the size of false positive cache grows, the average time of query answering becomes lower. Since to find the total false positive pairs takes $O(|V|^2)$, it is impossible to test the whole graph ahead when the graph is large. So the cache procedure works when the index is online. *GRAIL* [12] mentions their false positive cache approach. However, it is not recommended in their original paper since the size of their false positive cache could be huge. This is because the dimension number of their interval is low and generated randomly, the number of false positive pairs is large in their algorithm. Unlike *GRAIL*, *HD-GDD* has fewer false positive pair, so the cache size is smaller than that of *GRAIL*'s.

Our false positive cache is made up of two parts: a bitmap index to store the visited pairs and a false positive cache store with each vertex. Whenever a pair of vertices is visited, the bitmap of that pair is set. When all the bits in the bitmap are set, we can guarantee that all the false positive pairs are detected and the online search is no longer needed, the reachability query can be answered only by the coordinate and the false positive cache, which takes only $O(1)$ time.

False Positive Removal

This refinement approach aims to remove the false positive pairs that are hard to be removed by the *HD-GDD* itself. Since the graph structure may be very complex, those four heuristic conditions have to run many iterations before this false positive pair can be removed. Instead of running the basic *HD-GDD* repeatedly, we design an algorithm that guarantee to eliminate at least a *false positive destination vertex* at a time.

Definition 3.11. *(False Positive Source and Destination Vertex). Given a pair of false positive vertices (u, v), u is called the false positive source vertex and v is called the positive destination vertex.*

It is hard to remove a false positive pair from a false positive source vertex. But it is easier to start from a false positive destination vertex. When the dimension number is high and the *HD-GDD* is not efficient, we call *Sample* procedure to randomly collect the false positive destination vertices and sort by the times they act as the false positive destination. Then the algorithm traverses the graph reversely from the false positive destination vertex to the vertices that have no in-neighbors. By ordering the vertices into a sorted list with the topological level, we can assign a coordinate to each vertex. Since the false positive source vertices cannot reach false positive destination vertex, the source vertex is not in the list. So the algorithm can guarantee the false positive source vertices have larger coordinates than this selected false positive destination vertex. For example, in Figure 3.1, (B, E) is a false positive pair, while B is source and E is destination. Thus, we traverse the sub-graph $(A \to E, C \to E)$ first. Therefore, E will have a smaller coordinate than B, which means this false positive pair is broken.

Although this algorithm can prune the destination false positive vertex accurately, it is time consuming due to the sample procedure and only works well when most of the false positive pairs are eliminated by the basic *HD-GDD* algorithm.

3.4.6 Reachability on Road Network

In this section, we discuss the scenario in road network. First of all, unlike general random graphs, road network is near-planar. Because most of the roads are paved on the surface of earth, most parts of a road network is planar by nature. Only some parts that have tunnels and overpasses can cause a little degree of non-planar. Secondly, recall that for a planar graph, it takes only two dimensions to eliminate all the false positive. Therefore, on a road network, it only takes a very small number of dimensions to eliminate its false positives. When all the false positives are eliminated, the reachability query answer time drops to constant. The query time dropping pattern is similar to those sudden dropping lines in Figure 3.5 of the experiments.

3.5 Experiments

The experiment contains two parts: analysis on different graph structures and comparison with state-of-the-art *Refined Online Search* algorithms on real graphs.

3.5.1 Graph Structure Analysis

Experiment Setup

To have a deep understanding of the performance on different graph structures, we use *NetworkX* [123] to generate three different kinds of graphs. Because calculating the indicators needs to compare all the vertex pairs, which is $|V|^2$ and too time consuming when graph is large, we set the graph size to be 10000 in this experiment.

1. ***Erdős–Rényi model.*** It is a random graph generation model often used for theoretical evaluation of algorithm complexity. It was proposed by Erdős and Rényi in 1959 [124]. It generates a random graph based on the given model $G(n, p)$, where n is the vertices number and p is the independent probability of an edge's existence. It has a low clustering coefficient and a low average node-to-node distance. The graph generated by this model is totally random with no structure. In this experiment, we set $d = 0.05, 0.1, 0.15, 0.2$ and 0.25 to simulate the random graphs from sparse to dense.

2. ***Watts and Strogatz small world.*** A small world is a kind of graph in which most vertices are not neighbors to each other but can reach each other by a small number of hops or steps. The distance between two vertices grows proportionally to the logarithm of the number of vertices N in the network: $L \propto logN$. In the *Watts and Strogatz* model [125], the generated graph has a low average node-to-node distance and a high local clustering coefficient. The graph generated by this model has many clusters which can be viewed as "small worlds". The generation parameter is a triplet $\langle N, K, \beta \rangle$, where $N(10000)$ indicates the number of vertices, $K(10, 20, 30, 50)$ indicates the average neighbors and $\beta(0.5, 0.7, 0.9, 1)$ indicates the possibility to rewire an existing edge to another ending vertex(randomness).

3. ***Barabási–Albert model.*** It is used to generate a scale-free graph, which is a graph that exists in many man-made and natural systems like web, social networks, protein-protein interaction networks, airline networks etc. The degree distribution of the scale free graph follows a power law. This model applies a preferential attachment mechanism, which assigns the nodes with higher degree a higher probability to be the destination for a new link. The graph generated by this model has a long tail distribution, in which a small number of vertices have huge degrees while most of the vertices have small degrees. The parameter is $\langle n, m \rangle$, where n is the node

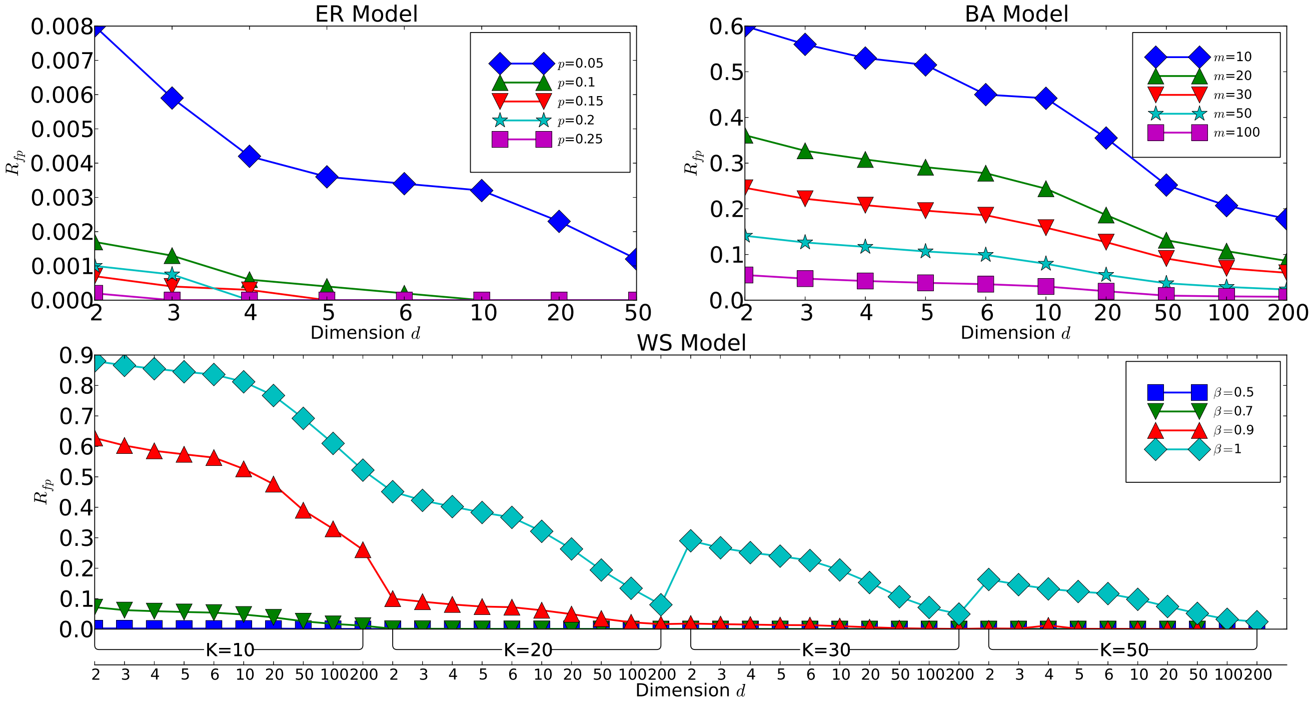


FIGURE 3.2: False Positive Ratio R_{fp} and Accuracy Ratio R_{acc} on Different Graph Models

number and $m(10, 20, 30, 50, 100)$ is the number of edges attached from a new node to the existing nodes.

We do not test the performance in road networks because they are mostly planar and have small index sizes with constant query answering time. So the evaluations are focused on the performance changes on the various complex graph structures.

3.5.2 Evaluation Metrics and Experimental Results

False Positive Ratio

It shows the ratio of false positive pairs in the total positive answers and is denoted as $R_{fp} = \frac{|false\ positive\ query|}{|false\ positive\ query + true\ positive\ query|}$. The lower the R_{fp}, the higher the possibility that the coordinate comparison can give the right answer straightly. When R_{fp} is 0, we can answer the reachability query just by coordinate comparison and no online research is needed.

As shown in Figure 3.2, R_{fp} drops as dimension increases generally. And as the graph becomes denser (p increases in ER Model, K increases in WS Model and m increases in BA Model), R_{fp}

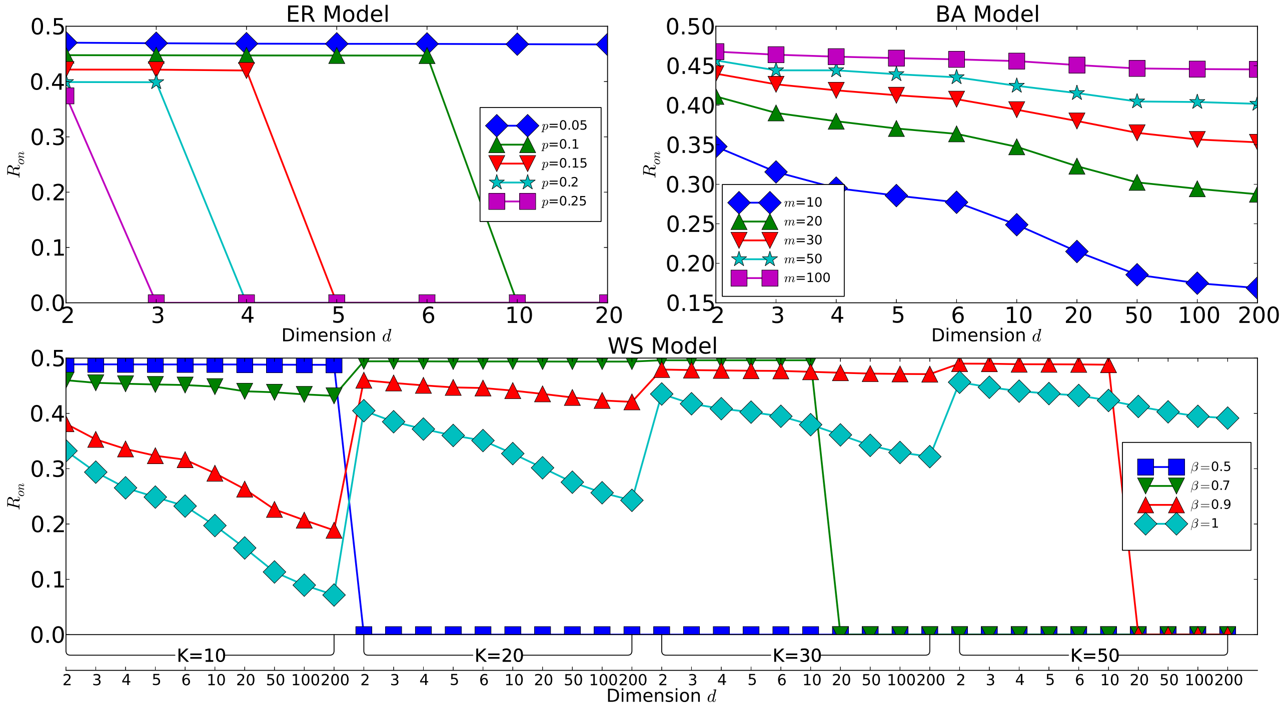


FIGURE 3.3: Online-Search Ratio R_{on} of Different Graph Models

drops dramatically. This indicates the fact that the denser the graph is (*more edges in the graph*), the higher chance two vertices are reachable. As β increases in WS Model, which means the graph becomes more random in the small world structure, R_{fp} soars up. This shows that *HD-GDD* has a better performance in a small world structure graph than the random graphs.

Online Search Ratio

It indicates the ratio of online search queries among the total queries and is denoted as $R_{on} = \frac{|Online\ search|}{|total\ queries|} = \frac{|Positive\ query|}{|total\ queries|}$. When R_{fp} is not 0, R_{on} indicates the ratio of online search needed to answer a reachability query accurately, and it affects query time directly. The higher the R_{on}, the longer the average query time. When $R_{fp} \to 0$, $R_{on} \to \frac{|ReachableVerticesPairs|}{|AllVerticesPairs|}$, which is a constant number of a specific graph. When R_{fp} is 0, R_{on} falls to 0.

As shown in Figure 3.3, R_{on} grows higher as the graph becomes denser. It reveals the fact that as the chance one vertex can reach another grows higher, the times the algorithm needs to finish a *BFS* grows. As shown in the WS Model, although the more random graphs may have a lower R_{on} when the graph is sparse, it always needs a higher dimension to eliminate all the false positive pairs

($\beta = 0.5$ first drops to 0, $\beta = 0.7$ next and then $\beta = 0.9$). This is because the larger the β, the higher randomness. When β is small, the graph is made up of several small worlds. As β grows, these small worlds have more links, which results in higher dimension to remove false positive. On the other hand, as K grows, the graph has more edges and becomes denser. It should be harder to remove false positive, as shown when β is 1 and 0.9. However, when $beta$ is small, a larger K means denser in each small world. Therefore, it is more possible for a small world to becomes a *DAG* if it has more edges within it. So the R_{on} drops to 0 when $K = 20$ and $\beta = 0.5$ when dimension is only 2. Similar for the cases when β is 0.7 and 0.9. When $\beta = 1$, it density increases randomly over the whole graph rather each small world. As for the cases in the BA Model, its R_{on} never reaches 0. This shows the fact that the scale-free graph is hard to eliminate the false positive, thus the $refinedonlinesearch$ methods do not work well on it. And they perform well when on dense random graph and sparse small world graph.

Since the false positive is the reason we need to conduct the online search, R_{on} should decrease as R_{fp} drops. However, R_{on} does not decrease dramatically due to the existence of *true positives*. In fact, R_{on} grows as total positive pair increases. At the same time, R_{on} should always be smaller than 0.5. This is because in DAG, we can guarantee that if $s \rightarrow t$, then $t \nrightarrow s$.

Index Construction

Figure 3.4 confirms that the index construction time of each model and the index size are all almost linear to the dimension number. As p grows higher in ER Model and m becomes larger in BA Model, which means the graph becomes denser, the index construction time becomes larger. This meets our time complexity analysis. As for WS Model, we take $K = 10$ for example. As β increases, the graph becomes more random and the index construction time increases accordingly. This is because in small world model, there are less edges among the vertex clusters when β is small. Thus, the average searching length in topological sorting is small. As β increases, edges among vertex clusters increase and more long search paths appear, which results in longer index construction time. So if the graph has more clusters and less links among the clusters, the index construction time is smaller. Another interesting observation in WS Model is that as β increases, the time curve becomes more non-linear. It accords with the trend of R_{on} that the more random graph takes higher dimension to approximate to the reachable pair ratio of the graph. Moreover, more randomness in small world would cause more out-neighbors' in-degree become 0. Therefore, its candidate set is larger than the less random graph,

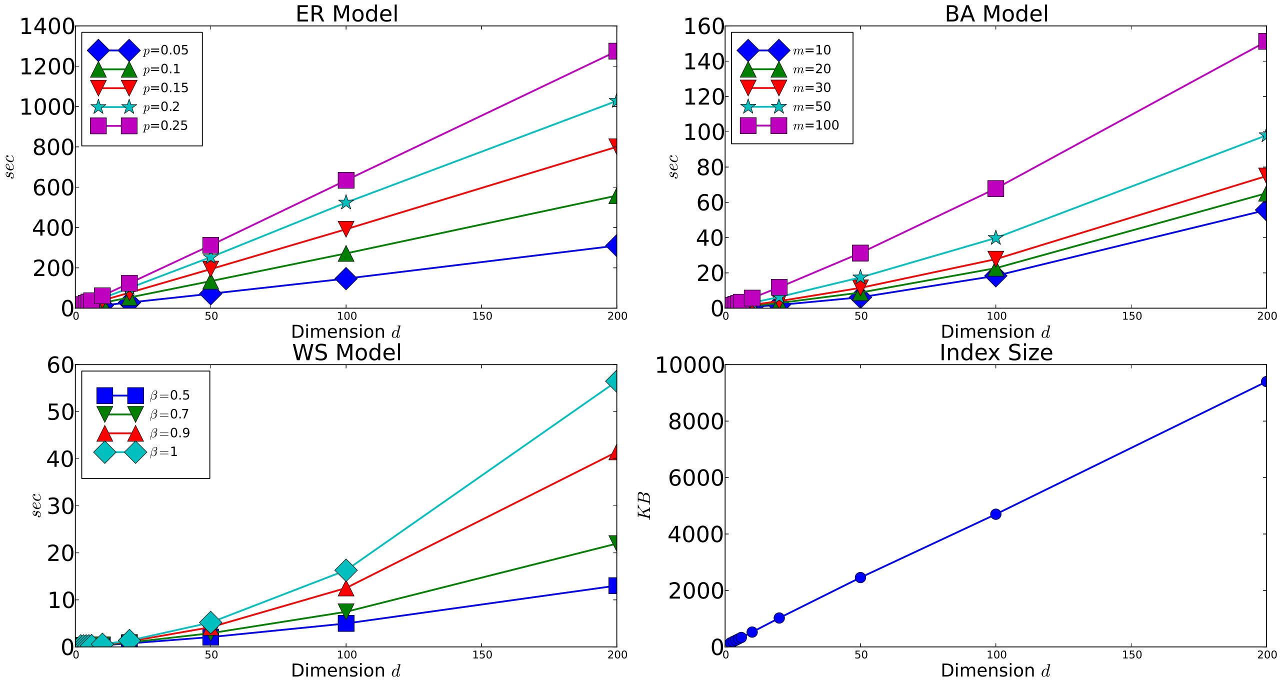


FIGURE 3.4: Reachability Index Construction Time and Size of Different Models

which results in a higher time complexity.

In conclusion, the index construction time grows when graph becomes denser, more random and has more long paths.

Query Time

As shown in BA Model and WS Model in Figure 3.5 , the query time decreases when d is small and soars up when d is large. This is because when d is small, the most time consuming operation is online search, while when d is large, the time is occupied by the coordinate comparison. And when R_{on} falls down to 0, the query time drops to constant as well as shown in ER Model and when K is high in WS Model. Generally, it follows the trends of R_{on}. In BA Model, when m grows, the graph becomes dense and query time becomes longer. This is because the number of positive pairs grows, which needs more online search. So it verifies the conclusion that $refinedonlinesearch$ does not perform well on scale-free graph. In ER Model, as the random graph becomes dense, the query time drops to constant earlier. So our approach works well on dense random graph. And WS Model follows the same trend. Moreover, as β decrease, the query time drops earlier.

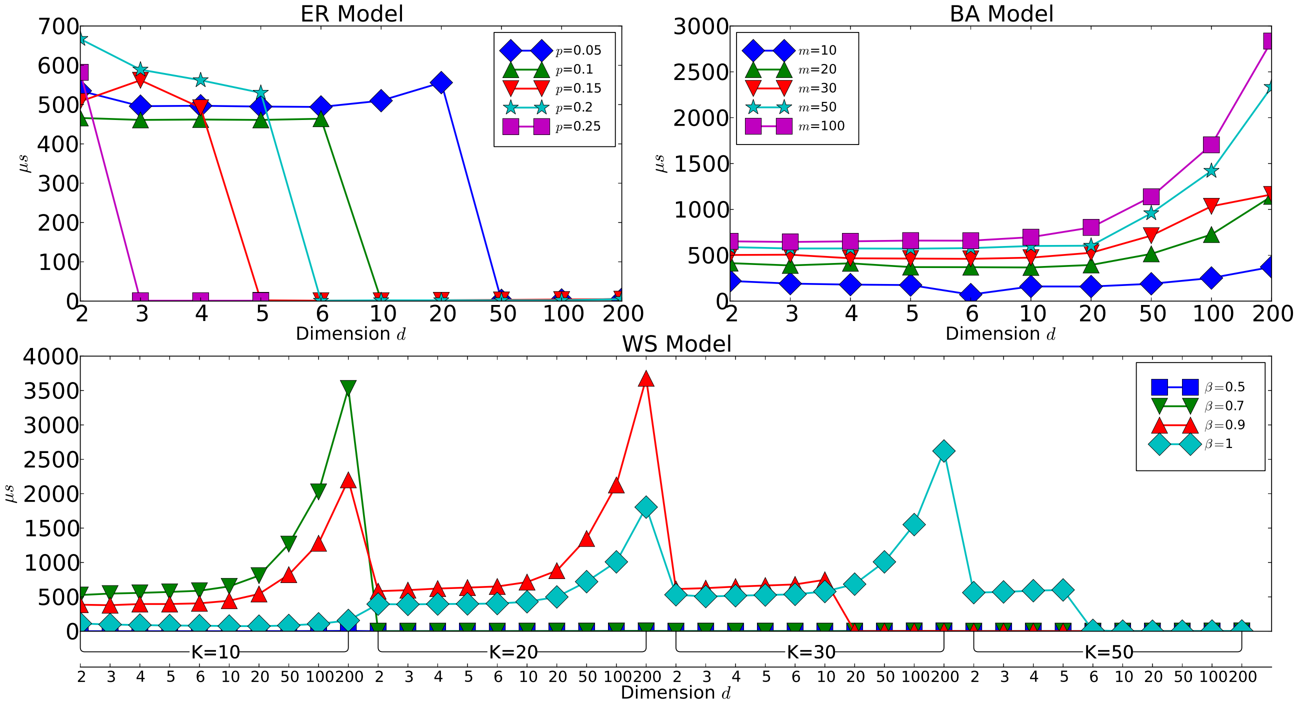


FIGURE 3.5: Query Time on Different Graph Models

3.5.3 Results on Real Graph

Experiment Setup

We compare our method with the state-of-the-art *Refined Online Search* algorithms: *Grail* [12], *Ferrari* [13] and *Feline* [14]. We compare the query time with *HD-GDD* when d is set to 3, 4 and 5. Table 3.2 shows the size and source of five real large graphs.

TABLE 3.2: Information of Real Graphs for Reachability Test

Dataset	$\|V\|$	$\|E\|$	d_{avg}	Source
EU institution Email	265,214	420,045	3.1676	[126, 127, 128]
CiteSeer	384,413	1,751,463	9.1124	[129, 130]
Google Webpage	875,713	5,105,039	11.659	[126, 131, 132]
Baidu Internal	2,141,300	17,794,839	16.621	[126, 133, 134]
US-Patent	3,774,768	16,518,947	8.7523	[126, 135, 136]

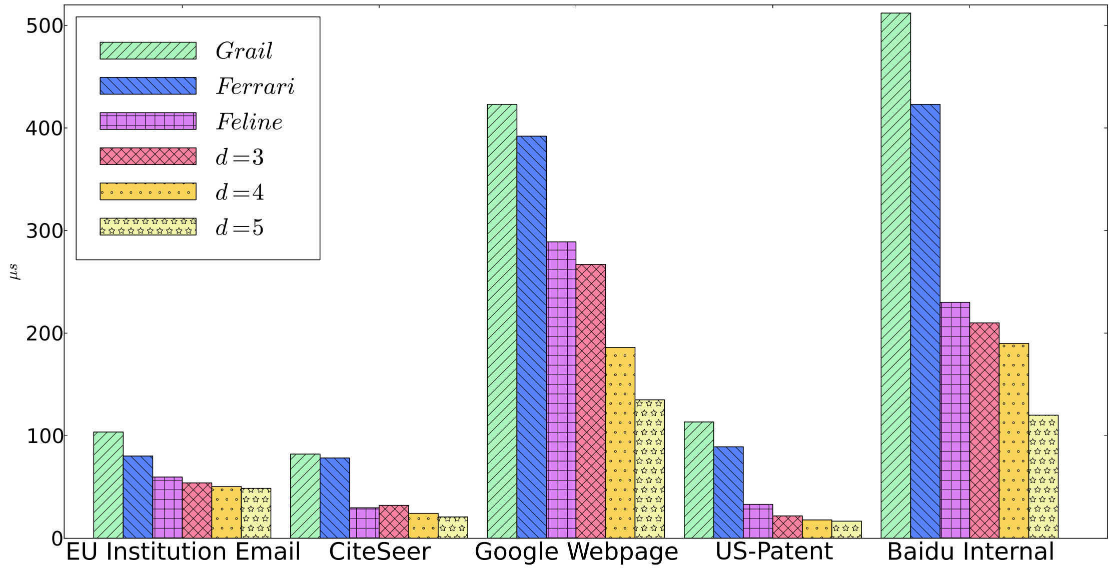


FIGURE 3.6: Reachability Query Time on Real Graphs

Query Time

We randomly test 1 million queries for each dataset and Figure 3.6 shows the average query time. It is worth noting that although *US-Patent* is larger than *Google Webpage*, its query time is obviously smaller. It indicates that the query time is not only dominated by the graph size, but also affected by the evaluation metrics we talked about previously. And *HD-GDD* always has a better performance than the others.

3.6 Summary

In this chapter, we analyze the cause of the false positive problem confronted by existing *Refined Online Search* algorithms, and study the graph drawing approach to answer reachability queries. We propose an efficient and scalable *HD-GDD* approach to improve query performance. It only takes no more than 5 dimensions to bring the false positive ratio in road network to 0, so it can help answer path query in constant time. We conduct extensive experiments on synthetic graphs with different graph structures and some real large graphs. The empirical results demonstrate certain relationship between query performance and graph structure, and verify that our proposal consistently outperforms state-of-the-art approaches on real graphs.

Chapter 4

Speed Profile Generation from Trajectory

In this chapter, we explain how we derive the speed profile from the trajectory data. We first talk about how to obtain the road speed from the trajectory in Section 4.1. Then we present our observations on the effects of different granularities of the speed collections in Section 4.2. After selecting an appropriate time slot size, we use several approaches to estimate missing values in Section 4.3. In Section 4.4, we test three compression algorithms on our speed profiles in order to reduce the storage space. Finally, we present the evaluation results in Section 4.5.

4.1 From Trajectory to Road Speed

First of all, we match the trajectory $tr_i =< (x_1^i, y_1^i, t_1^i), ..., (x_m^i, y_m^i, t_m^i) >$ to the graph G. There are several methods [32, 137, 138] in this field. After that, we obtain a sequence of consecutive edges $E_i =< e_1, ..., e_m >$, with $\forall 1 \leq j \leq m, (x_j^i, y_j^i, t_j^i)$ is on some edge $e_k \in E_i$. It should be noted that an edge could have several points attached to it, while some edges might have no attaching points. For any consecutive pair of points $p_j^i = (x_j^i, y_j^i, t_j^i)$ and $p_{j+1}^i = (x_{j+1}^i, y_{j+1}^i, t_{j+1}^i)$, we can retrieve a set of edges $E_j^i =< e_k, e_{k+1}, ..., e_n >$ between them. We assume the travel between p_j^i and p_{j+1}^i keeps an even speed. The distance $d_{j,j+1}^i$ between them is the sum of corresponding traveled edge length. Thus, the speed is $v_j^i = d_{j,j+1}^i/(t_{j+1}^i - t_j^i)$. Then we attach this speed to the corresponding edges in E_j^i, with time proportional to the distance to p_j^j. By repeating this procedure from the first GPS point to the last one, we can get all the roads' speed along this trajectory, together with their corresponding starting time.

4.2 Speed Data Collection

Before collecting the data into time slots, we first categorized them into weekday and weekend, or by date. Then for each edge e_i in one specific category, it has a set of speed data $< (v_1^i, t_1^i), ..., (v_n^i, t_n^i) >$. The next step is converting it into a usable speed profile.

The most straightforward method is to use these speed data directly, which would result in a set of linear piecewise speed functions. However, it is not practical for the following reasons. Firstly, some speeds are either much smaller than the others because the driver may wait for the traffic light or even stop to wait for a passenger, or bigger than the average due to some emergency cases. If we line up these speed points directly, we will get a zigzag speed profile that apparently cannot describe the road network's actual traffic condition. In fact, it falls into the terrible situation of over-fitting. Secondly, a speed profile with a random bunch of functions is both hard to use and compress. Another approach is approximating the speed data using some regression methods [139, 140]. Although it can represent the speed profile as functions, it is unable to deal with missing value since it estimates the missing speed only by the values on each edge itself, which is highly inaccurate.

To address the problems mentioned above, we use a histogram-based approach to collect the speed data. Specifically, we divide one day's time into Ŧslots with the same length. Then the speed data that fall into the same slot will be added up together to get an average speed. Thus, the influence of the outliers is reduced dramatically. However, the granularity of the histogram is another important issue to consider. If Ŧis small, it cannot reflect the difference of traffic conditions during different time of a day. While if Ŧis big, there will be not enough speed data within each time slot and the size of the speed profile will soar up at the same time. We test the granularity of *1-day, 1-hour, 30-minutes, 15-minutes and 5-minutes* in section 4.5.2. Based on the experiment results, we choose the 5-minutes time slot, whose number is 288 for each edge in a day, to collect the speed.

One may think if we increase the granularity further to 1-minute, we might have a speed profile with better accuracy (lower MAE). However, it is not realistic in our experiment. Firstly, the missing value problem becomes more severe. Much more slot are void now, making the estimation process harder and more time-consuming. Moreover, when we collect the data, we have to remove the outliers of each slot. But we cannot tell one speed data is an outlier or not in a 1-minute speed profile. This would pollute the histogram with abnormal data. In fact, if we have a larger amount of data, we could generate a 1-minute speed profile. But in our case, the 5-minute one is the best we can achieve.

4.3 Missing Value Estimation

Even though the GPS-based trajectory data has a higher coverage of the road network than other approaches, it is still hardly possible to cover every edge. So it also faces the sparsity problem. To make the matter worse, the data becomes even sparer as the number of time slots grows. In our test, although the 5-minutes granularity is not too small to produce too many void time slots, there are still 85% of them have no value. In this section, we propose two approaches to estimate the missing values in the histogram data: *Cosine Similarity* and *Spatial-Temporal Neighboring Average*.

4.3.1 Cosine Similarity

This approach compares the similarity between an edge and its neighbors and uses the similar ones' data to fill its missing values. Each road e_i's speed profile can be viewed as a speed vector SP_i with $\mathcal{T}$ values: $SP_i =< SP_{i,0}, SP_{i,1}, ..., SP_{i,\mathcal{T}-1} >$, where $SP_{i,j}$ is the speed of edge e_i at time slot j. If there is a missing value, we just use 0 to denote it. $|SP_i|$ denotes the number of time slots without missing values. Thus, the similarity between the speed profiles of two edges e_i, e_j can be evaluated by the cosine similarity:

$$\begin{aligned} Similarity(SP_i, SP_j) &= \frac{SP_i \cdot SP_j}{\| SP_i \| \| SP_j \|} \\ &= \frac{\sum\limits_{k=0}^{\mathcal{T}-1} SP_{i,k} \times SP_{j,k}}{\sqrt{\sum\limits_{k=0}^{\mathcal{T}-1} (SP_{i,k})^2} \times \sqrt{\sum\limits_{k=0}^{\mathcal{T}-1} (SP_{j,k})^2}} \end{aligned}$$

We use $SP_i \cap SP_j = \{k | SP_{i,k} \neq 0 \wedge SP_{j,k} \neq 0\}$ to denote the time slots that are not empty on both edges. Furthermore, in order to eliminate the bias from the edges with sparse speed profile, we calculate the similarity only when $|SP_i \cap SP_j| > 25\% \times \mathcal{T}$. For each edge we compute its similarities between its 3-hop neighboring edges and find the top-3 similar ones. Then, it uses the speed in these three profiles to fill its missing speeds. For a specific time slot, if the most similar one is also empty, then we check the second most similar one. If still empty, then check the third.

The computation works iteratively from the edges with higher $|SP_i|$ to lower ones. As the process proceeds, the $|SP_i|$ changes at the same time. Eventually, the edges with $|SP_i|$ larger than $25\% \times \mathcal{T}$ would get fully filled. For those sparse ones, we apply the *Spatial-Temporal Neighboring Average* approach described in Section 4.3.2. This is because the similarity between a pair of sparse vector is not accurate, and for most cases it is not even computable due to the value position's mismatch.

4.3.2 Spatial-Temporal Neighboring Average

This is the simple approach that averages the speed of a road's neighbor and its neighboring time slots.

$$SP_{i,j} = Avg(SP_{k,j}, SP_{i,j-1}, SP_{i,j+1}), \forall e_k \cap e_i \neq \phi$$

where $SP_{i,j}$ is edge e_i's speed at its j^{th} time slot. If its neighbors are also empty at certain time slots, we extend the search to the 3-hop neighbors and 3-hop time slots. The computation also computes iteratively starting from the roads that have fewer missing values. This is because these roads always link to roads that have a relatively complete speed profile. Then it propagates all the roads in the road network eventually.

4.4 Speed Profile Compression

As mentioned previously, the smaller the time slot size, the less space-efficient the speed profile is, especially when the neighboring slots have the same or similar speeds. To save the space for storing the speed profiles on disk and in memory, we propose an *adaptive speed profile*. The term *adaptive* means this speed profile is derived from the histogram-based profile and adapts the occasions where the nearby time slots have similar speed values. In this subsection, we aim to reduce the speed profile size from the perspective of each road. We test three different kinds of *Piecewise Linear Approximation* [45, 46] algorithms to convert the 5-minutes histogram based speed profile to a set of piecewise linear functions. The actual speed of each road at different time can be computed by the corresponding function.

The histogram-based speed profile can be viewed as a type of *Time Series Data* [44], and building the adaptive speed profile from the histogram-based speed profile falls into the category of *Time Series Segmentation* and is defined as below:

Definition 4.1. ***(Speed Profile Segmentation).*** *Given a speed time series $SP_i =< SP_{i,0}, SP_{i,1}, ...,$ $SP_{i,T-1} >$, construct a model $\hat{SP_i} =< SP_{i,0}, ..., SP_{i,\hat{d}} >$ of reduced dimensionality $\hat{d}$, $(\hat{d} \ll T - 1)$ such that $R(\hat{SP_i}, SP_i) < \varepsilon$, where R is a reconstruction function and ε is a given error threshold.*

The reconstruction function R calculates the difference of speed value between adaptive speed profile and the original one. It serves as the evaluation method of compression quality. We choose the *residual error* as the reconstruction function, which adds up the square of the differences. *PLA (Piecewise Linear Approximation)* [45] is the compression approach which aims at transferring the original SP_i into a set of approximate lines while retaining the essential features. There are two ways of approximation:

- *Linear Interpolation*: Use a line connecting the two ending points to approximate.

- *Linear Regression*: Use the linear regression algorithms to find the best fitting line.

Apparently, linear interpolation has a smooth look while linear regression produces a set of disjoint segments. We choose linear interpolation approach because of the following reasons. Firstly, it is obviously faster to implement and compute. Secondly, it is more space saving than linear regression. Linear interpolation only needs to store the turning points while linear regression has to store all the end points of the segments, which is twice larger. Moreover, since the speeds in profile are all approximate, the more accurate algorithm in this step cannot promise a better approximation. There are three basic categories of *PLA*: *sliding window* [45], *top-down* [47] and *bottom-up* [49]. They are described in the following sections:

Algorithm 3: Sliding Window Algorithm

```
Input: The speed profile of an edge $SP_i =< SP_{i,0}, ..., SP_{i,T-1} >$, error threshold $\varepsilon$
Output: The adaptive speed profile of a road $\hat{SP}_i^{sw} =< SP_{i,0}, ..., SP_{i,\hat{d}} >$
begin
    $\hat{SP}_i^{sw}.insert(SP_{i,0})$
    for $j\ from\ 0\ to\ T-1$ do
        for $k\ from\ j+1\ to\ T-1$ do
            if $R(SP_{i,j,k}\hat{SP}_{i,j,k}^{sw}) > \varepsilon$ then
                $\hat{SP}_i^{sw}.insert(SP_{i,k})$
                $j = k - 1$
                $break$
    if $SP_{i,T-1}\ not\ in\ \hat{SP}_i^{sw}$ then
        $\hat{SP}_i^{sw}.insert(SP_{i,T-1})$
    $return\ \hat{SP}_r^{sw}$
```

4.4.1 Sliding Window Algorithm

The sliding window algorithm is a fast online algorithm whose time complexity is $O(n)$, where n is the speed profile length of an edge. It keeps expanding the approximate line from the left starting point to the right until the *error* surpasses a user specified threshold ε. Then it uses the end point of the last generated segment as the next starting point and repeats until all the points are visited. Since each point in the speed profile is visited only once, it has a linear complexity. The detail is shown in Algorithm 3. $SP_{i,j,k}$ denotes the speed profile segment of edge e_i from its j^{th} speed point to k^{th} speed point.

4.4.2 Top-Down Algorithm

The top-down algorithm finds the best speed point that splits the original speed profile into segments each time (i.e. where the two resulting segments have the smallest combined error). If the any of the two resulting segment's error is larger than threshold ε, the algorithm repeats recursively to find the best splitting speed points in it. The algorithm terminates when all the speed profile segments' errors are smaller than ε.

Algorithm 4: Top Down Algorithm

```
Input: The speed profile of an edge $SP_i =< SP_{i,0}, ..., SP_{i,T-1} >$, error threshold $\varepsilon$
Output: The adaptive speed profile of a road $\hat{SP}_i^{td} =< SP_{i,0}, ..., SP_{i,\hat{d}} >$
begin
    Function TDFindBreakPoint(int low, int high)
        $best\_so\_far = inf$
        for $j\ from\ low + 1\ to\ high - 1$ do
            $BestTmp = R(SP_{i,low,j}, \hat{SP}^{td}_{i,low,j}) + R(SP_{i,j,high}, \hat{SP}^{td}_{i,j,high})$
            if $BestTmp < best\_so\_far$ then
                $best\_so\_far = BestTmp$
                $k = j$
        $\hat{SP}_i^{td}.insert(SP_{i,k})$
        if $R(SP_{i,low,k}, \hat{SP}^{td}_{i,low,k}) > \varepsilon$ then
            $TDFindBreakPoint(low, k)$
        if $R(SP_{i,k,high}, \hat{SP}^{td}_{i,k,high}) > \varepsilon$ then
            $TDFindBreakPoint(k, high)$
```

It breaks the search space into two pieces each time and calls for itself recursively at most twice. At the same time, the R function calculates the difference between the result speed profile segment and the original one, which takes $O(n)$ times. So the overall time complexity of the top-down algorithm is

Algorithm 5: Bottom-Up Algorithm

```
Input: The speed profile of an edge $SP_i =< SP_{i,0}, ..., SP_{i,T-1} >$, error threshold $\varepsilon$
Output: The adaptive speed profile of a road $\hat{SP}_i^{bu} =< SP_{i,0}, ..., SP_{i,\hat{d}} >$
begin
    $\hat{SP}_i^{bu} =< SP_{i,0}, ..., SP_{i,T-1} >$
    do
        $minError = inf$
        for $j\ from\ 1\ to\ T - 1$ do
            $errorTmp = R(SP_{i,j-1,j+1}, SP^{bu}_{i,j-1,j+1})$
            if $errorTmp < minError$ then
                $minError = errorTmp$
                $bp = SP_{i,j}$
        if $minError < \varepsilon$ then
            $\hat{SP}_i^{bu}.erase(bp)$
    while $minError \leqslant \varepsilon$
```

$O(n \log n)$. As the threshold ε grows, less recursion is needed, and the overall running time decreases.

4.4.3 Bottom-Up Algorithm

The bottom-up algorithm is reverse to the top-down algorithm. In the initial step, it connects the points in the original speed profile, so errors are all 0. After that, it merges consecutive lines, by erasing the intermediate point, with the smallest error iteratively until the smallest error exceeds the threshold ε. In the worst case, we have to erase all the intermediate points, which runs $\frac{n(n-1)}{2}$ times. Therefore, the time complexity of bottom-up algorithm is $O(n^2)$. The detail is shown in Algorithm 5.

4.5 Experiment

In this section, we first describe the experiment setup in terms of datasets, online query setup and speed profile evaluation metrics. After that, we present the result of a comprehensive performance study to demonstrate the effectiveness and efficiency of our *MORT* algorithms. Finally, we show the experiment results of the offline speed profile generation.

TABLE 4.1: Trajectory Data Sets

City	Num	4.1	4.2	4.3	4.4	4.5
Beijing	Traj	532868	143998	541650	310976	642390
	GPS	17698668	5164315	17069156	11402483	23614206
Shanghai	Traj	389733	103411	378968	180670	349265
	GPS	10747519	2949734	10039956	5519847	10946567

4.5.1 Experiment Setup

Datasets

We first describe the map datasets we use for the whole system. Then we present the trajectory data we use for the speed profile generation.

We get two maps of Beijing and Shanghai from *Navinfo*[1]. The Beijing map consists of 302,364 intersections and 387,588 roads, which covers a 184km×185km spatial range and has a total length of 51,666km of roads. The road network of Shanghai has 243,842 intersections and 310,058 roads. It covers a 120km×143km spatial range. The total length of road segments is 42,930km. As for the parking vertices, we attach them on maps randomly to test its influence on the algorithms.

We obtain our trajectory data from *DiDi*[2]. It has the trajectory data of five consecutive days from 2015.4.1 to 2015.4.5, collected from taxis in Beijing and Shanghai, respectively. The total data set has 2,171,882 trajectories and 74,948,829 GPS points in Beijing, and 1,402,047 trajectories and 40,203,623 GPS points in Shanghai. The details of each day's basic information is shown in Table 4.1. The trajectory's length distribution of each city on each day is present in Figure 4.1 (a)-(b). It shows that the number of trajectory decreases as the length grows, so most of our trajectories are not too long. As for the last value point that soars up, that is because it is the accumulation of the all the trajectories that have length no shorter than 10km. The starting time distribution along 24 hours is shown in Figure 4.1 (c)-(d). Except for 2015.4.2, which lacks some data, most trajectories are collected during daytime and few trajectory appears after midnight. This distribution corresponds to the people daily behavior, and we build our test speed profile on daily basis.

[1] http://www.navinfo.com/

[2] http://www.xiaojukeji.com/news/newslisten

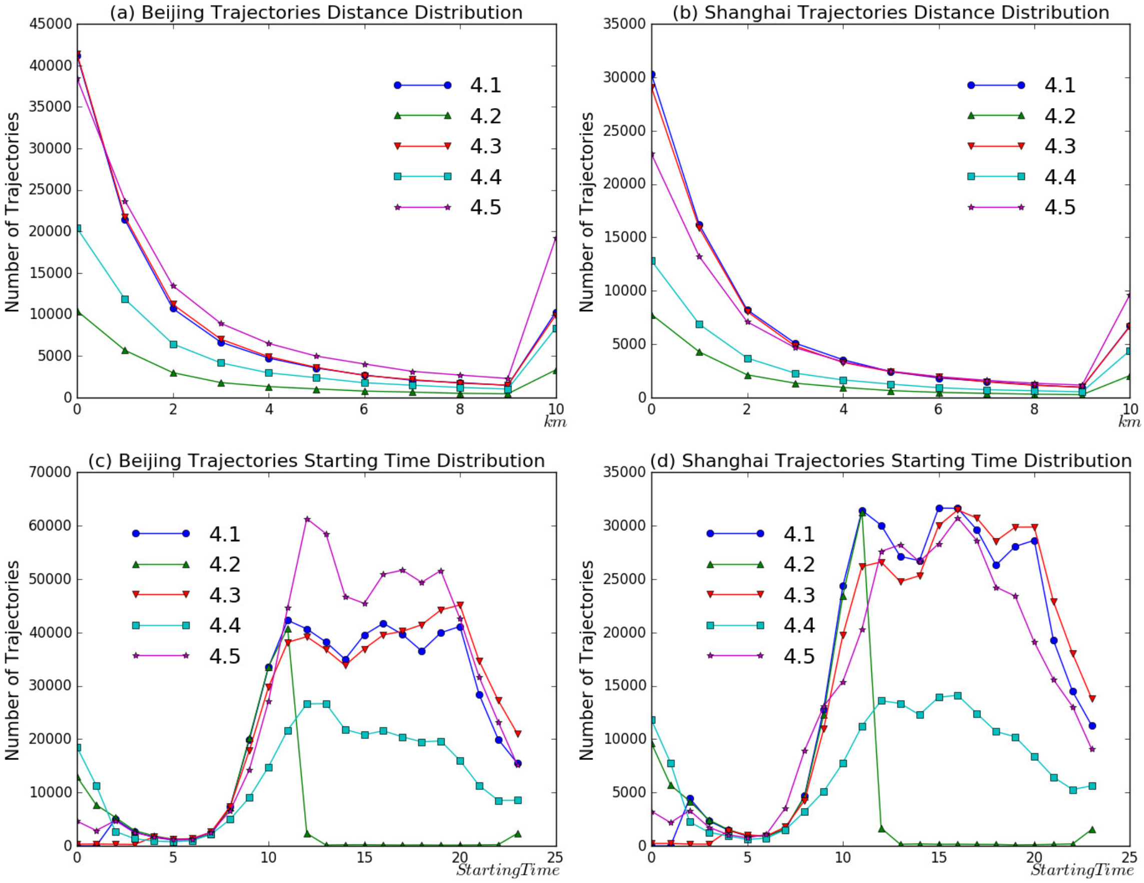


FIGURE 4.1: Trajectory Starting Time and Length Distribution

Speed Profile Evaluation Metrics

We use 80% of the trajectories that are selected randomly to build the speed profiles and test them on the remaining 20% trajectories. In the evaluation, we re-travel the testing trajectories using the speed from the generated speed profile since these trajectories are the only ground truth we have. For any trajectory Tr_i, we first match it on map and convert it into a sequence of consecutive edges like $< Tr_i.e_0, Tr_i.e_1, ..., Tr_i.e_k >$, which starts on time t_0 and stops on t_{k+1}, its average speed along this trajectory is

$$Tr_i.speed = \frac{\Sigma_{j=0}^{k} Len(Tr_i.e_j)}{t_{k+1} - t_0} \tag{4.1}$$

Then we re-travel this trajectory by following exactly the same roads in the same order from t_0 using the testing speed profile, and it will finish traveling $Tr_j.e_k$ on t'_{k+1}. The new average speed is

$$Tr_i.speed' = \frac{\Sigma_{j=0}^{k} Len(Tr_i.e_i)}{t'_{k+1} - t_0} \tag{4.2}$$

Then we can calculate the *mean absolute error (MAE)* of each speed profile as

$$MAE = \frac{\sum_{i=0}^{N} |Tr_i.speed - Tr_i.speed'|}{|Tr|} \tag{4.3}$$

where $|Tr|$ is the number of testing trajectories. The smaller the *MAE*, the better the speed profile. During the test, we omit those trajectories that are short since they are more easily affected by the abnormal driving behavior while the longer ones suffer less from it.

Experiment Environment

We ran all the experiments on a Dell R720 PowerEdge Rack Mount Server which has two Xeon E5-2690 2.90GHz CPUs, 192GB memory, 1TB hard disk, and runs Ubuntu Server 14.04 LTS operating system.

4.5.2 Speed Profile Generation Evaluation

Granularity

We compare the *MAE* of speed profiles under granularities of 1-day(Universal), 1-hour, -30-minutes, 15-minutes and 5-minutes on five days in Beijing and Shanghai, respectively. The results are shown in Figure 4.2 (a)-(b). We can observe clearly that the 5-minutes speed profile outperforms the others. The *MAE* increases as the time slot size grows. The universal granularity, which is actually a static graph, has the largest *MAE* obviously. So for the rest of the tests we only present the results of the 5-minutes speed profile.

Missing Value Estimation

We compare three missing value estimation approaches in this test: *Cosine Similarity*, *Matrix-Factorization based Collaborative-Filtering (MF-CF)*[41] and *Spatial-Temporal Neighboring*. The *MAE* of these three missing value estimation approaches are shown in Figure 4.2 (c)-(d). It is clear that the *MF-CF* approach is much worse than the other two. In the Beijing road network, the *spatial-temporal* approach has a better performance, while in the Shanghai road network, the *cosine similarity* approach is better. The best missing value estimation method of each day has a *MAE* around 1. It means that in a travel of an hour, our speed profile has a travel distance difference about 3 *km*, which is quite acceptable because different drivers have different driving behavior.

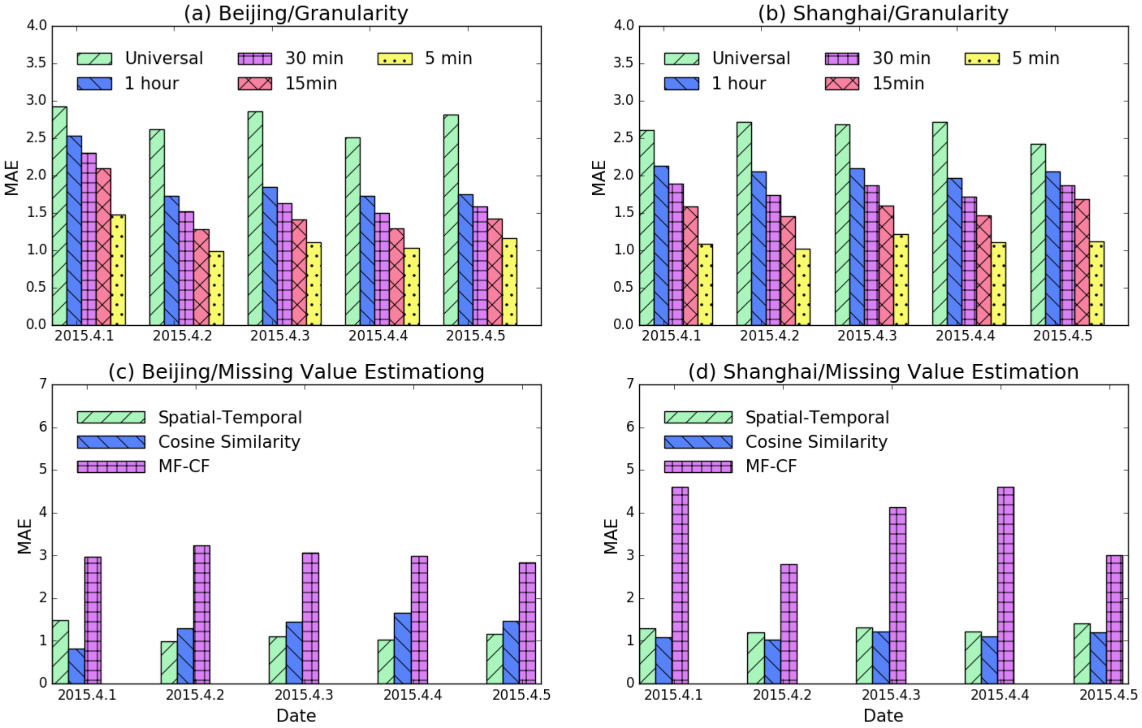


FIGURE 4.2: *MAE* of Speed Profiles under Different Granularity and using different Missing Value Estimation

TABLE 4.2: Size of Speed Profiles under Different Granularity

	1 Day	1 Hour	30 Min	15 Min	5 Min
Beijing	4.9MB	68MB	135MB	369MB	861MB
Shanghai	4.1MB	56MB	112MB	223MB	718MB

Speed Profile Compression

We compare three compression algorithms on Beijing Map 2015.4.1 in this test: *Sliding Window (SW)*, *Top-Down (TD)* and *Bottom-Up (BU)*. The compression result is shown in Figure 4.3. We compare the error *MAE*, compression time and the storage size of each algorithm under error threshold ε of 0.1, 0.5, 1, 2 and 5. As shown in Figure 4.3 (a), the MAE of the three algorithms are nearly the same, while the *Bottom-up* is always slightly better than the other two, and it is almost as good as the original one. As expected, the accuracy becomes worse as the compression error threshold ε grows. When it comes to the construction time and space consumption shown in Figure 4.3 (b)-(c), the *sliding window* algorithm is the fastest to compute and its compression rate is not bad. The *top-down* algorithm is slow and has the worst takes the largest space. The *bottom-up* algorithm takes the longest time to compute but its compression is the best. In fact, it only takes 18% of original

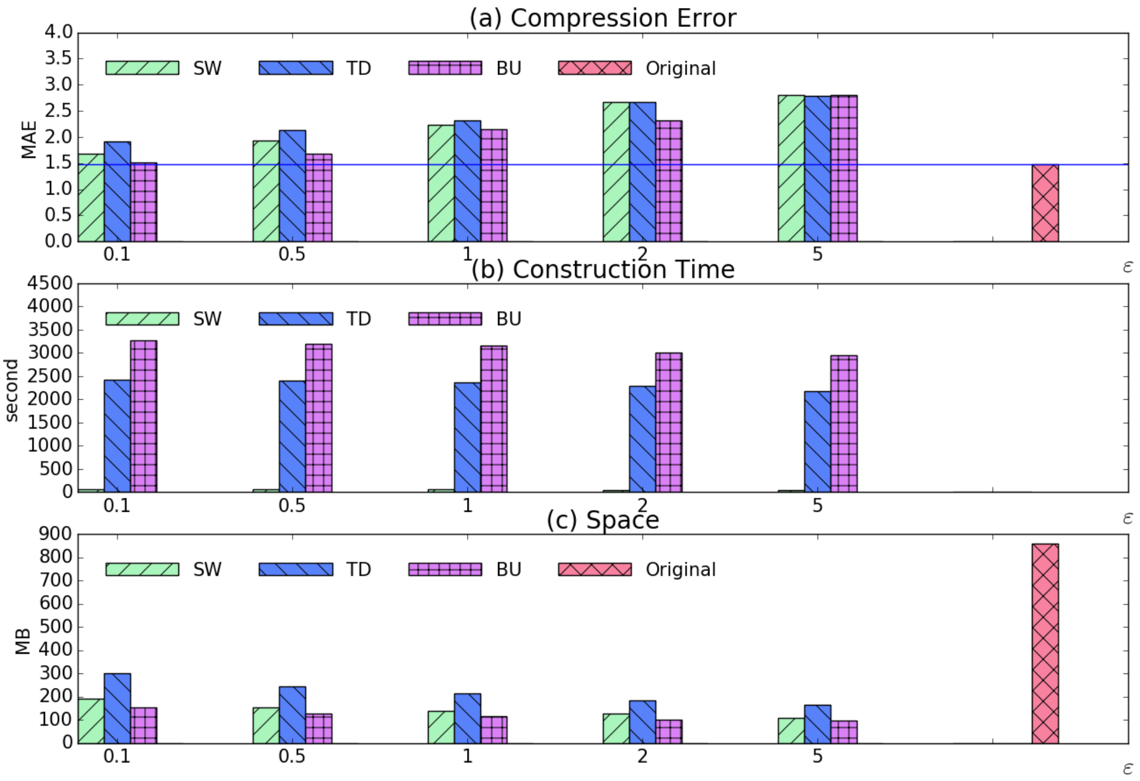


FIGURE 4.3: Compression Performance on Beijing Map 2015.4.1

space. So if the compression time is not a problem, we can use the bottom-up algorithm to compress. Otherwise, the sliding window algorithm is a better choice.

4.6 Summary

In this chapter, we describe how we obtain our speed profile from historical trajectories. It involves map matching, speed data collection, missing value estimation and compression. Each step has several approaches and we test their performance. In conclusion, we use 5 minutes time slot to collect the speed data, use spatial-temporal neighboring average and cosine similarity to estimate the missing values, and use bottom-up *PLA* algorithm to compress the histogram into linear piecewise functions. The output of this chapter serves as the time-dependent function for the next two chapters.

Chapter 5

Minimal On-Road Traveling Time Route Scheduling

5.1 Introduction

With the prevalence of GPS enabled devices and wireless networks, navigation systems have been widely adopted by public transportation, logistics, private vehicles and a broad range of location-based services. Essentially, it is the path planning algorithm that plays the vital role in those navigation systems, which helps people travel more smartly and more predictively. In the past decades, different path planning algorithms are proposed for various application scenarios and requirements. For example, shortest path algorithms [20, 80, 141] find a path with the minimal distance between origin and destination, while fastest path algorithms return a path with the least total travel time given a static traffic condition [92]. If a user is allowed to depart from any time during a certain period, another set of fastest path algorithms can be used [96, 21, 95, 86, 101, 142] to find the optimal departure time with the least total travel time. Moreover, path planning algorithms for earliest arrival and latest departure [82, 90] are also important in transportation.

The common optimization goal of the above path planning algorithms is the total travel time, which is the difference between departure time and arrival time, and is made up of on-road time and waiting time. In a time-dependent road network where the cost associated with road segment can change over time, the existing path planning problem makes use of an important observation known as the *FIFO* property, which means a vehicle enters a road segment first will also reach the end of road segment first in spite of the time-dependent nature [92]. So for an *FIFO* road network, there is no need to consider waiting during travel since waiting can only increase the total time. However, for many users such as logistics companies with heavy trucks, the actual on-road time (i.e., the time when the engine is running) becomes critical as it directly relates to fuel consumption which can be as high as 80% of their operational cost. As long as the goods can be delivered on time, reducing the actual on-road time can be more economic than arriving the destination earlier. On the other hand, tourists would also like to reduce their time spent on road so that they can spend more time on the tourist attractions. On a bigger view, the more cars that reduce their on-road time, the better traffic condition there would be, which would lead to less exhausted emission and a better environment. This motivates us to study a new kind of path planning algorithm that optimizes the on-road time by waiting strategically in certain places along the route in order to avoid predictable traffic jam. To better understand how waiting can shorten the on-road time when traveling, consider the road network with five vertices shown in Figure 5.1. Three of them are ordinary vertices, and two of them are parking vertices that allow waiting. The traveling cost functions are shown in Figure 5.1(b)-(f). Suppose the starting time from v_1 is 0 and the latest arrival time at v_5 is 130. The fastest path takes 105 time units ($v_1 \rightarrow v_2 : 40; v_2 \rightarrow v_3 : 70; v_3 \rightarrow v_5 : 105$), and its on-road travel time is also 105. However, if we still start from v_1 at 0 and arrive v_2 at 40, but travel from v_2 to v_4 and arrive v_4 at 95, the current on-road time is 95. Then we wait on v_4 and depart on 120, the cost from v_4 to v_5 reduces to 5. So the on-road travel time of this path is 100. So by taking advantages of these parking vertices, we can obtain a route that has shorter on-road travel time. More application scenarios are explained in Section 5.4.4 after the algorithm is fully described.

In this work, we model a road network as a time-dependent graph, whereas each edge is associated with a function that returns the time cost of traveling the edge for a given departure time from the starting vertex. There are two types of vertices in this graph: *ordinary vertices* that do not allow waiting, and *parking vertices* that do. This model considers the phenomenon that some vehicles may choose to stop at some places to avoid traffic jams. The proposed query, *minimal on-road time path query* (*MORT*), aims to find a path that consists of not only a consecutive of edges in the road network, but also a *waiting plan* that determines the amount of time to stop at a parking vertex in order to minimize on-road time. So it is actually a route scheduling algorithm rather than a path planning problem. This is different to the previous problems that aim at minimizing *the total travel time* which includes both the on-road time and waiting time. Clearly, a *MORT* query is more complicated than traditional path planning queries that minimize the total travel time. First of all, it needs to decide

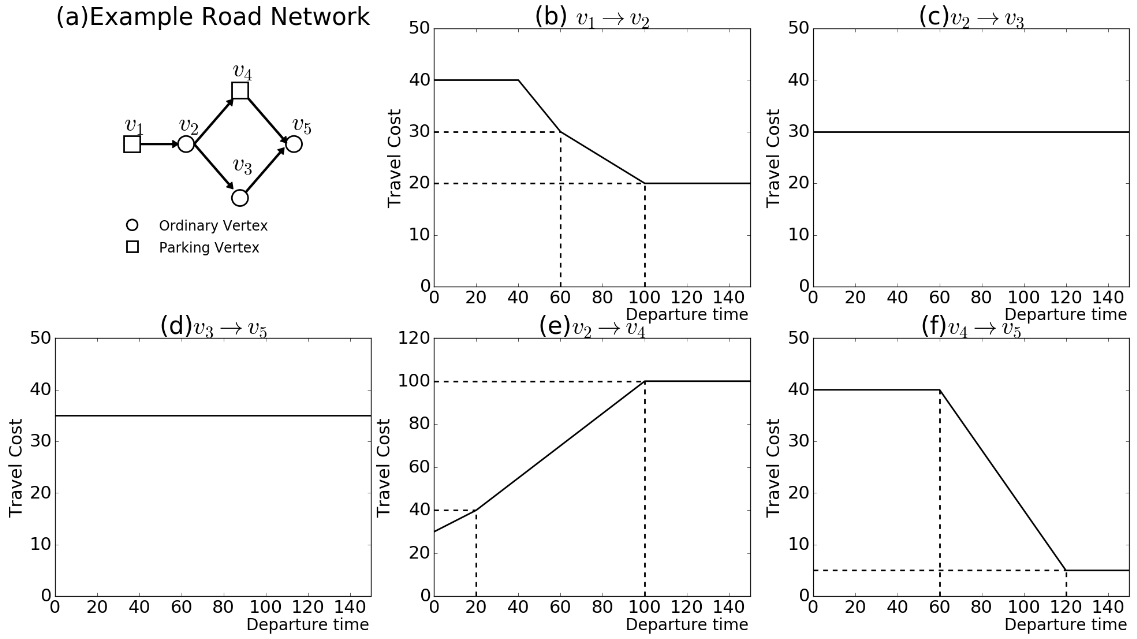


FIGURE 5.1: A Road Network with Parking Vertices

(a) The example graph.

(b) - (f) The corresponding time-dependent weight for each edge over time domain (0 - 150)

whether waiting at certain parking vertices, or even taking a detour to a parking vertex, can save on-road time at all. Secondly, if waiting on this parking vertex has benefit, it needs to further determine the waiting time on it. Finally, because waiting on any vertex is allowed, the graph that *MORT* query runs on does not need to follow *FIFO* property, which is the basis of all the existing algorithms.

In fact, the existing path planning algorithms cannot solve this problem even under *FIFO* setup. First of all, the *shortest path algorithms* [20, 80, 141, 143] only works with static edge weights. Thus, it cannot handle the time-dependent costs. Secondly, the *single starting-time fastest path (SSFP)* algorithm [92] does not allow waiting at any vertex. Even though it has the ability to cope with time-dependent costs, it cannot solve our problem. Finally, the *interval starting-time fastest path (ISFP)* algorithms [96, 21] allow waiting on the starting vertex, but they do not allow waiting on the intermediate vertices since it would simply result in a longer total travel time. One naive approach to find an approximate *MORT* path based on *ISFP* algorithms is to select the optimal waiting time on each parking vertex along the path in a greedy fashion. Firstly, it runs *ISFP* algorithm on the starting vertex to get the optimal departure time t_s^* on starting vertex v_s. Then, it runs *ISFP* algorithm on the first parking vertex v_{p1} along the path with its arrival time from v_s at time t_s^* as the starting time, and

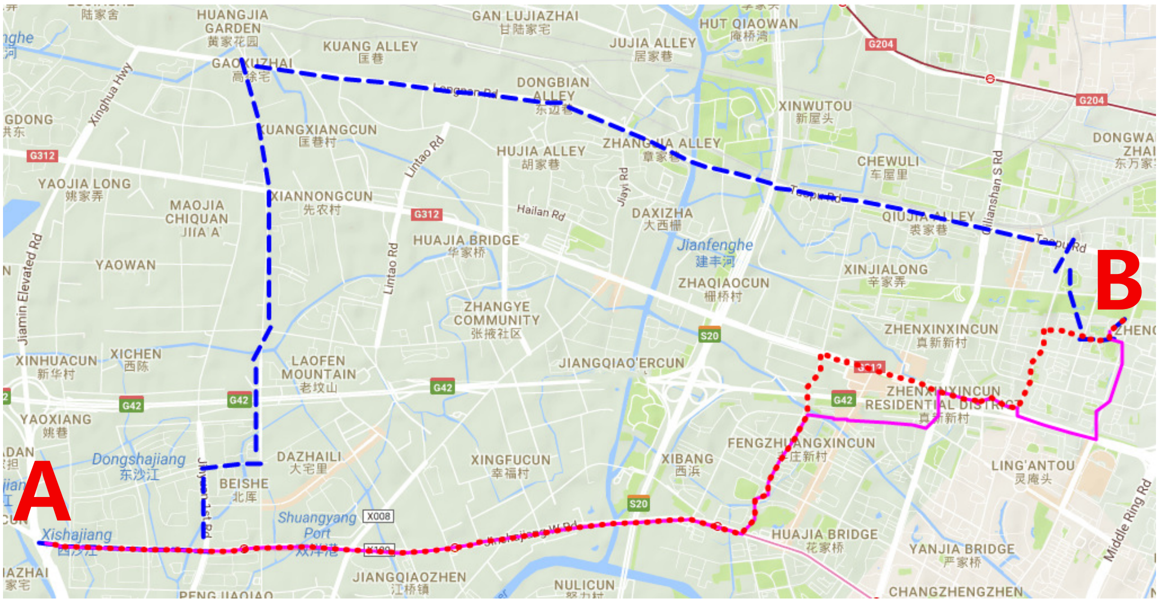


FIGURE 5.2: Comparison between *MORT* Path and Other Paths

MORT Path (Red Dotted), Fastest Path (Blue Dashed) and Recursive Fastest Path (Pink Solid).

gets the optimal departure time t^*_{p1} from v_{p1}. After that, it runs the *ISFP* on the first parking vertex along the new path from v_{p1} again to get its optimal departure time. The procedure runs iteratively until the destination vertex is reached. However, this approach has two problems: Obviously, it runs *ISFP* multiple times, so its computation time is long. A more serious problem is that this approach has no guarantee to find the optimal solution at all as it is a greedy method with no backtracking (the first parking site on a route is just an accidental stop point from a path that has not considered parking as an optimization option). Figure. 5.2 shows a real life example of the comparison of our algorithm, *ISFP* [21] and the iterative approach. The example illustrates paths from location $A(31.2414, 121.304)$ to $B(31.2559, 121.386)$ in Shanghai, whose shortest distance is 10km. The starting time interval is set from 10:00 to 16:00 and the latest arrival time is 19:00. *ISFP* finds a path with a on-road travel time of 1385s, iterative approach finds a path of 1130s, while our algorithm finds a path of 986s.

In this work, we propose two algorithms to find the minimal on-road travel route. Both of them construct and maintain a set of *Minimum Cost Functions* to record the minimal on-road time from the starting vertex to the other vertices at different arrival time. The first algorithm builds the minimum cost functions over the whole query time interval iteratively in a *Dijkstra* way, while the second algorithm constructs it sub-time-interval by sub-time-interval instead. We observe a *non-increasing property* for the parking vertices, which integrates the waiting time benefit into the minimum cost function. Both of them support user specifying different minimum staying times when waiting on parking vertices. We also provide a route retrieval solution to return routing schedule satisfying user's requirement on the arrival time. It is worth noting that our *MORT* algorithm is more general than the existing time-dependent path algorithms. First of all, if we treat the parking vertices as normal vertices, our algorithm can solve the *ISFP* problem. Moreover, if we further prohibit waiting on starting vertex, our algorithm can solve the *SSFP* problem. In fact, both *ISFP* and *SSFP* are the special cases of *MORT*.

In summary, our contributions are listed as follows:

- We identify a general form of time-dependent route scheduling problem, called *MORT*, to make use of parking facilities in a road network to minimize the on-road travel time, instead of the total travel time.

- We propose a *minimum cost function* and two novel algorithms to solve the *MORT* route scheduling problem efficiently. Our algorithms can handle real-life road network with dynamic and complex speed profiles. Both of them are able to address other existing types of time-dependent path planning problems if no parking vertices are considered.

- The *Basic MORT Algorithm* performs the *MORT* search for a vertex after each iteration, until the destination is reached. We show that its time complexity is $O(T|V|\log|V| + T^2|E|)$. The *Incremental MORT Algorithm* runs *MORT* search for each vertex starting from a small subinterval to fill the full time interval incrementally, and its time complexity is $O(L(|V|\log|V|$ $+|E|))$. Both algorithms require $O(T(|V| + |E|))$ space. T is the average number of turning points in minimum cost functions, and $L > T$ is the average number of subintervals created during computation.

- We evaluate the effectiveness and efficiency of our *MORT* algorithms with extensive experiments in road network and small world graphs, measuring both the reduction of the minimal on-road time and the algorithm running time.

The rest of the chapter is organized as follows. Section 5.2 discusses the related work. We formally define the minimal on-road time problem in Section 5.3. Section 5.4 presents the two *MORT* algorithms with correctness and complexity analysis. We present our approximation method in Section 5.5. An empirical study is shown in Section 5.6. Our summary can be found in Section 5.7.

5.2 Related Work

In this section, we review the previous works on modeling time-dependent road network and position our work by discussing the difference from the fastest path problems.

The simplest model of the time-dependent road network is the discrete time-dependent graph (or "timetable" graph), of which the existence of each edge is time-dependent. A few path planning algorithms such as *earliest arrival time path, latest departure time path, shortest path* and *shortest duration time path* have been proposed on such graphs. [50] proved that these queries could be solved with a modified version of the *Dijkstra* algorithm. However, it does not scale well with the size of the network. Several techniques are proposed to improve the efficiency [89, 82, 90], but they only work on timetable graphs.

A more precise way to describe a time-dependent road network is to use the continuous time-dependent cost function. Fastest path query has been well studied that aims to find a path with the minimum w_{TOT} including waiting time. Dreyfus [92] first showed the time-dependent fastest path problem was solvable in polynomial time if the graph is restricted to have *FIFO* property. Other early theoretical works on this problem include [93] and [94]. However, these algorithms are very difficult to implement, and no empirical evaluation results were reported. Most of the recent path planning algorithms in road network share a common assumption that the travel along a road follows *FIFO* property, which means a vehicle starting earlier will not arrive destination later regardless of the time cost of edges. Due to this property, waiting on a vertex always results in a longer total travel time. So these algorithms do not consider waiting on vertices actually. We briefly discuss some representative fastest path algorithms below.

Single Starting-Time Fastest Path (SSFP) algorithm does not allow waiting on the starting vertex. This problem can be solved in $O(|V|\log|V| + |E|)$ time by minor modification on *Dijkstra's Algorithm* if *FIFO* property holds [92]. The algorithm can answer both *Earliest Arrival Path* and *Latest Departure Path*, with the same computational complexity.

Interval Starting-Time Fastest Path (ISFP) algorithm allows waiting on the starting vertex in a given starting time interval. But once departing, no waiting is allowed along the path. The difference between *ISFP* and *MORT* is illustrated in Figure 5.3. Moreover, *ISFP* only returns the optimal departure time from starting vertex v_s, while *MORT* needs to determine the optimal departure time from each parking vertex along the path. It is proved in [144] that the theoretical lower-bound of *ISFP* is

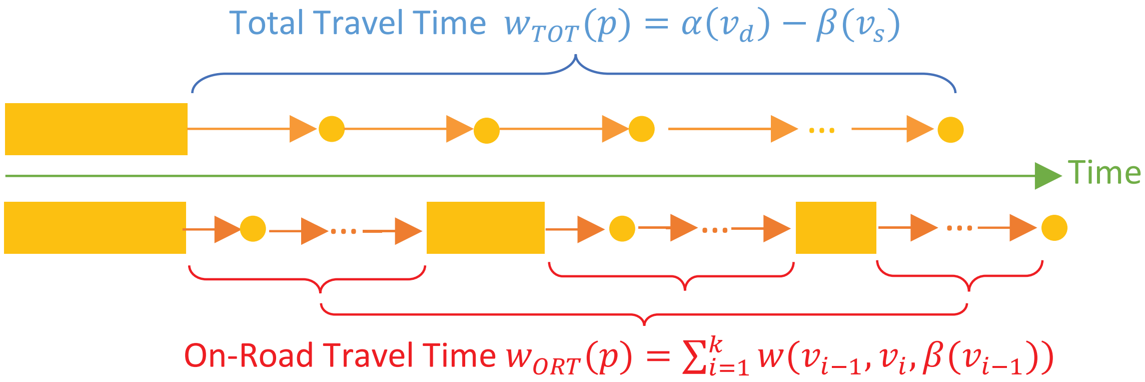

Total Travel Time $w_{TOT}(p) = \alpha(v_d) - \beta(v_s)$
Time
...
On-Road Travel Time $w_{ORT}(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i, \beta(v_{i-1}))$

FIGURE 5.3: Comparison between *total travel time* and *on-road travel time*.
Thick bar: Waiting time on a parking vertex; Circle: No waiting on the vertex; Arrow: Travel time from one vertex to another

$\Omega(T(|V|\log|V| + |E|))$ [144], where T is the average number turning points in the result functions if the weight functions are piecewise linear. Currently no existing algorithm can achieve this bound because T could be large and it is hard to find the departure time points that would result in the T turning points. Some early works like *DOI* [95] and [145, 142] select $k \ll T$ starting time points in the starting time interval and run *SSFP* k times. Obviously, this approach has no guarantee to find the optimal departure time, and both the running time and accuracy highly depend on the choice of k. [96] proposed a path selection and time refinement approach using the heuristic of A*-algorithm. They computed an arrival time function for each vertex iteratively and used A*-algorithm to reduce the searching space. However, it is hard to find an appropriate heuristic condition on a time-dependent graph. [21] applied a more precise refinement approach that expanded the time interval step by step rather than computing the entire time interval iteratively. It could avoid unnecessary computations and achieve better performance, although time complexity remained the same. It has a complexity of $O(\alpha(\hat{T})(|V|\log|V|+|E|))$, where $\hat{T}$ is the size of the whole time domain, and $\alpha(\hat{T})$ is the complexity to maintain the time-dependent functions. Although it is not pointed out in their paper, $\alpha(\hat{T})$ actually has a much larger value than the turning point number in the final functions. Other works further build different kinds of indexes to speed up fastest path query, such as *time-dependent CH* [105] and *time-dependent SHARC* [107].

Although *ISFP* is different from *MORT*, we can adopt it as our baseline algorithm by invoking the algorithms in [96, 21] recursively to get an approximate result. [25, 146] take waiting on intermediate vertices into consideration in their problems. But they allow waiting on any vertex, which does not make sense in real life. In fact, [25] cannot solve our problem directly and has a time complexity of $O(|V|\log|V| + T|V| + T^2|E|)$, which means it cannot guarantee the optimal result actually since each vertex is visited once. As for [146], they define a time-dependent weight function $w(v_i, v_j, t)$ and a cost function $c(v_i, v_j, t)$ for each edge (v_i, v_j), and aim to find the path with minimum cost, not the minimum weight. But they set the cost functions to linear constants. So rather than confronting with the complex linear piecewise weight functions, they only have to deal with a small set of constant values, which actually simplifies the problem by converting the complex functions to constant values, even though the problem description looks more complicated. Thus, their algorithm cannot find the minimum on-road time (or the minimum weight under their scenario).

From the network point of view, the road network with parking vertices can be treated as a kind of *graph with special nodes*. *Electric vehicle shortest walk problem*[147, 148] adopted this model but on static road network. In this problem setting, an electric vehicle has a driving distance limit, and it has to recharge its battery at a power station before it is running out. Given a source vertex and a destination vertex, the problem aims to find the shortest path that the vehicle is able to travel through it. Both [148] and [147] build a sub-network of power station first to solve this problem. It is possible to do this since the network is static and the driving limit is predefined. Essentially it is a special case of *Constraint Shortest Path Problem*[149]. If the problem is generalized to be independent on driving distance, the problem becomes *NP-H*. [150, 151] use *Lagrange Relaxation* to find approximate result. Although network model is similar to ours, they do not consider time-dependent cost on edges, which makes impossible to pre-built a sub-network just as what they do on static graph.

5.3 Problem Definition

A time-dependent road network can be represented as a directed graph $G(V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of ordered pairs of vertices, with a weight function $w : (E, t) \to \mathbb{R}$ mapping edges to time-dependent real-valued weights. The weight of an edge $e(u, v) \in E$ at time t in a time domain $\mathcal{T}$ is $w(u, v, t)$, which represents the amount of time required to reach v starting from u at time t. In this paper, we only consider the case where the weight of an edge can change over time, but not the case where the structure of a graph can change over time (i.e., V and/or E remain to be static over time). This is a reasonable assumption, as the structure of a road network changes much less frequently compared with the traffic situations. We also define $w(u, v, t) = \infty$ if there is no edge from u to v.

A path from u to v in G can be represented as $p =< v_0, v_1, \ldots, v_k >$, where $v_0 = u$, $v_k = v$, and $(v_{i-1}, v_i) \in E$ for any $1 \leq i \leq k$. Let $\alpha(v_i)$ and $\beta(v_i)$ be the arrival and departure time at $v_i \in p$, the *time-dependent* cost of p is the sum of the time-dependent weights of its edges $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i, \beta(v_{i-1}))$. This cost is ∞ by definition if there is no path from u to v in G.

Now let us differentiate two different types of cost for a path: the *total travel time* $w_{TOT}(p) = \alpha(v_k) - \beta(v_0)$ and the *on-road travel time* $w_{ORT}(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i, \beta(v_{i-1}))$. Although $w_{ORT}(p)$ looks identical to $w(p)$ above, the difference here is that for a vertex $v_i \in p$, it is no longer necessary to have $\alpha(v_i) = \beta(v_i)$. In other words, the traveler can stop at a vertex if that can help to reduce the on-road travel time. It is trivial to see that $\alpha(v_i) = \beta(v_{i-1}) + w(v_{i-1}, v_i, \beta(v_{i-1}))$ for $i > 0$, and $\beta(v_0)$ is the selected depart time by a path planning algorithm.

The problem to find shortest/fastest path from u to v is to find such a path $p(u, v)$ with minimum cost $w(p)$. Most existing works on this topic have an implicit assumption that for any vertex $v \in p$, $\alpha(v) = \beta(v)$ (e.g., a traveler cannot stop at any vertices along the path). These algorithms focus on w_{TOT} cost. In that case, a traveler departs earlier will always get to the destination earlier (known as the *FIFO* property [92]). With this setting, travelers always keep $\beta(v) = \alpha(v)$ for any vertex v on a path to achieve optimal w_{TOT}. Some recent works have noticed that, in order to optimize w_{ORT} instead of w_{TOT}, it can be beneficial to delay the departure time at the starting vertex [96, 21]. However, there are more vertices than just the source vertex in a road network where a vehicle can stop for a period of time. Let $V' \subseteq V$ be a set of *parking vertices* in G where a vehicle can wait *voluntarily* for a minimum amount of time t_{min} before traveling again. In other words, $\beta(v) - \alpha(v) \geq v.t_{min}$ if $v \in V'$, and $\beta(v) = \alpha(v)$ if $v \in V - V'$. This should not be confused with the case that a vehicle stops in a traffic jam or in front of a traffic light; these forced stops are captured by the weight function of $w(u, v, t)$ already.

We are ready to define the problem we address in this paper as follows.

Definition 5.1. ***(Minimal On-Road Time Route Scheduling Problem)****. Given a directed graph $G = (V, E)$ with a set of parking vertices $V' \subseteq V$, each of which has a minimum staying time $v_i.t_{min}$ and a time-dependent edge weight function w, a query $Q_{MORT}(v_s, v_d, t_{s1}, t_{s2}, t_d)$ is to find a path from v_s to v_d, represented as $p =< v_0, v_1, \ldots v_k >$, such that: (1) $v_s = v_0$ and $v_d = v_k$; (2) $\beta(v_i) = \alpha(v_i)$ if $v_i \in V - V'$ and $\beta(v_i) - \alpha(v_i) \geq v_i.t_{min}$ if $v_i \in V'$; (3) $t_{s1} \leq \beta(v_s) \leq t_{s2}$; (4) $\alpha(v_d) \leq t_d$; and (5) $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i, \beta(v_{i-1}))$ is minimal among all possible paths meeting the conditions (1),* *(2), (3) and (4).*

Condition (1) means that p is a path from v_s to v_d and condition (2) allows the traveler to stop and wait only at a parking node for a minimum period of time. Conditions (3) and (4) define that the traveler must depart v_s during the specified time interval and must arrive at v_d before the given latest arrival time t_d. If there does not exist a path meeting these four conditions, the cost to travel from v_s to v_d is defined as ∞. Condition 5 requires the path to have the minimal on-road travel time.

If the edge weight is not time-dependent (i.e., the weight for each edge is static), a *MORT* query reduces to traditional shortest path queries in a static road network [20]. Besides, the time-dependent query studied in [96, 21] is a special case of the *MORT* query where parking node set $V' = \{v_s\}$.

5.4 Algorithm

In this section, we describe our *MORT* algorithms in detail. The key idea is that we define and maintain a variational piecewise *Minimum Cost Function* $C_i(t)$ for each vertex v_i. $C_i(t)$ returns different minimal on-road travel time from v_s to v_i given different arrival time t, so it has the potential to model traffic tendency more accurately. Based on the new cost function, we design two algorithms to expand the *MORT* path step by step in a *Dijkstra* way: (1) the *Basic MORT Algorithm* constructs $C_d(t)$ by updating $C_i(t)$ of each visited vertex over the whole time interval, and finishes expanding until $C_d(t)$ is stable; (2) the *Incremental MORT Algorithm* decomposes $C_d(t)$ into different parts according to the query time sub-intervals, and finishes expanding until each part of $C_d(t)$ is complete. Both of these algorithms do not require the graph to follow *FIFO* property. Although our path expanding algorithms are able to find the *MORT* time, its result is not a route schedule, which is the expected output of *MORT* problem. To address that, *path retrieval* is introduced to generate the final results. Considering scalability is important for route scheduling, we present the correctness and complexity analysis of the proposed method at the end of each subsection.

5.4.1 Algorithm Outline

Given a time-dependent graph $G(V, E)$ and a *MORT* query $Q_{MORT}(v_s, v_d, t_{s1}, t_{s2}, t_d)$, the proposed algorithm generates the minimal on-road time $R_{p^*_{s,d}}$ and the corresponding route with traveling schedule $p^*_{s,d}$. The whole process can be divided into three parts as below:

1. ***Active Time Interval Profiling (ATI)*** computes the active time interval T_i for each vertex v_i, which is bounded by a pair of earliest arrival time $v_i.t_{EA}$ and latest departure time $v_i.t_{LD}$.
2. ***Path Expansion*** finds the path with minimum on-road travel time in a *Dijkstra* way and produces the *Minimum Cost Functions* of the visited vertices.
3. ***Route Retrieval*** returns the actual route schedule with user specified arrival time.

In the following subsections, we will introduce each part of the proposed algorithm thoroughly except for the path expansion part. The full details of the path expansion which are the major contributions in this work will be presented in Section 4.2 and 4.3, respectively. We further explain how to apply our algorithms to different scenarios in Section 5.4.4.

Active Time Interval Computation (ATI)

The MORT query specifies a departure interval $[t_{s1}, t_{s2}]$ on v_s and a latest arrival time t_d on v_d. With these constraints, the route schedule is roughly outlined but loose for other vertices. If the graph does not follow *FIFO*, we have to use this loose time interval. Otherwise, we could reduce the computation load by computing an *active time interval (ATI)* for each vertex in the proposed algorithms. An *ATI* of a vertex v_i is denoted as $T_i = [v_i.t_{EA}, v_i.t_{LD}]$, which is bounded by a earliest arrival time $v_i.t_{EA}$ (we cannot arrive v_i any earlier) and a latest departure time $v_i.t_{LD}$ (we will never arrive v_d before t_d if it departs from v_i any later). It models a vehicle's possible occurrence interval on the corresponding vertex under the query constraints (t_s and t_d). *ATI* is very important for the proposed algorithm since it is the basis of the other parts. In the following, we will introduce how the *ATI* is computed for each vertex.

ATI, as well as all the following calculations, are computed from speed profile. In a speed profile, each edge (v_i, v_j) is associated with a function $w(v_i, v_j, t)$ whose parameter is t and output is time cost. Compared to [146], function $w(v_i, v_j, t)$ is a combination of consecutive linear functions rather than constant values. It obeys the *FIFO* and serves in the *path expansion*. Notice that when t is given, we use $w(v_i, v_j, t)$ to represent the time cost of travelling from v_i to v_j at time t. The speed profile is then instantiated as $\{(t_0, w(v_i, v_j, t_0)), \ldots, (t_k, w(v_i, v_j, t_k))\}$, and the intermediate values between points are computed linearly. Figure 5.1(b)-(f) illustrate an example of speed profile.

Given the proposed speed profile, the earliest arrival time of each vertex is computed by performing *SSFP* from v_s at t_{s1}. As for the latest departure time, we have to compute from v_d at t_d reversely,

TABLE 5.1: MORT Important Notations

Notation	Description
T_i	Active Time Interval of v_i
I_i	$[v_i.t_{EA}, \tau_i] \subseteq T_i$
τ_i	upper bound of I_i
$C_i(t)$	minimum cost function of v_i
$g_{f,i}(t)$	$C_f(t) + w(v_f, v_i, t)$
$g'_{f,i}(t)$	non-increasing version of $g_{f,i}(t)$
$C'_i(t)$	$min(C_i(t)), g_{f,i}(t)$

both in time and in vertex order, respectively. After two rounds of *SSFP*, each vertex obtains its active time interval, and all the future computations will be based on the active time intervals. The *ATI* has the same time complexity as *Dijkstra*, which is $O(|V|\log|V| + |E|)$.

We query the road network in Figure 5.1 with Q_{MORT} $(v_1, v_5, 0, 30, 130)$ as an example. $ATI(v_1, v_5, 0, 30, 130)$ generates the following active time intervals: $T_1 = [0, 25]$, $T_2 = [40, 65], T_3 = [70, 95], T_4 = [95, 125]$ and $T_5 = [105, 130]$.

Minimum Cost Function

In order to model the correlations between time and cost, we construct a *minimum cost function* whose value varies with arrival time for each vertex, instead of defining the minimum cost which is constant over time in [146]. Accordingly, the output of path expansion in our work is the minimal of v_d's minimum cost function. Since the minimum cost function is the basis of the two proposed path expansion algorithms, we present the definition and construction of the minimum cost function in this part.

The minimum cost function, denoted as $C_i(t)$, monitors the minimum on-road cost of traveling from v_s to v_i that arrives on time t. The minimum value of $C_i(t)$ is equivalent to the minimum on-road time (*MORT*) from v_s to v_i . For example, $C_i(300) = 50$ means when it starts traveling from v_s at t_s and arrives on v_i at time 300, the minimum on-road travel time (*MORT*) is 50. Accordingly, for the destination vertex v_d, the *MORT* is $min(C_d(t))$. In addition, for a parking vertex v_i^p, the value of dependent variable of $C_i^p(t)$ has a *non-increasing* property:

Lemma 5.1. $\forall v_i \in V'$ *and* $\forall v_i.t_{EA} \leq t_a < t_b \leq v_i.t_{LD}, C_i^p(t_a) \geq C_i^p(t_b)$

Proof. Suppose route schedule p_a arrives parking vertex v_i at t_a with cost $C_i(t_a)$, and another route schedule p_b arrives v_i at t_b with cost $C_i^p(t_b)$, $t_a < t_b$. If $C_i^p(t_a) \geq C_i^p(t_b)$, then the lemma holds. If $C_i^p(t_a) < C_i^p(t_b)$, the vehicle can wait on v_i from t_a to t_b until $C_i^p(t_b) = C_i^p(t_a)$. Thus, the non-increasing property still holds. □

The non-increasing property reveals a natural fact: If one route schedule arriving at t_b takes higher cost than another arriving at t_a, we should choose the latter one and wait from t_a to t_b, which reduces the on-road time from $C_i^p(t_b)$ to $C_i^p(t_a)$. The non-increasing property indicates that waiting is necessary to decrease the on-road travel time.

$C_i(t)$ is linear piecewise because it is constructed from the speed profile which is also linear piecewise. Thus, a minimum cost function $C_i(t)$ equals a set of consecutive discrete linear functions. These functions share the end points and are maintained in the ascending order of time. Based on that, the cost function of a vertex is denoted as an ordered point set $S_i = \{(t_0, C_i(t_0)), ..., (t_k, C_i(t_k))\}$. The update of S_i is achieved by merge. For instance, suppose $C_i'(t)$ is the current minimum cost function of v_i, and $C_i''(t)$ is another minimum cost function provided by another path to v_i, the new $C_i(t)$ is formed by merging the smaller parts of these two functions: $min(C_i'(t), C_i''(t))$.

Route Retrieval

The route retrieval generates the route schedule based on the user specified arrival time using the minimum cost functions. For each turning point in the ordinary vertices' minimum cost functions, we store its predecessors. For the parking vertices, apart from the predecessors for the turning points, we also need to store the points that happen to have the same value as the current cost (no turning point added because it is not smaller). This predecessor cache has the same space complexity as the minimum cost functions.

If t is a user-specified arrival time, we can traverse the vertices back from v_d at time t. In this backward traversal, suppose we are visiting v_i at t_i. Firstly, if v_i is an ordinary vertex, we find the latest turning point $(t_i', C_i(t_i'))$ in $C_i(t)$ such that $t_i' \leq t_i$, and use its predecessor as the next visiting vertex. The arrival time is the same as t_i. Secondly, if v_i is a parking vertex, we also find the latest turning point $(t_i', C_i(t_i'))$ in $C_i(t)$ with $t_i' \leq t_i$. However, the arrival time is t_i' rather than t_i. If the turning point has more than one predecessor, or the parking vertex has more than one points with the same cost, we can traverse the graph in a *DFS* way to output more than one routes for users to choose. Obviously, this approach takes $O(k)$ time, where k is the number of vertices along the route.

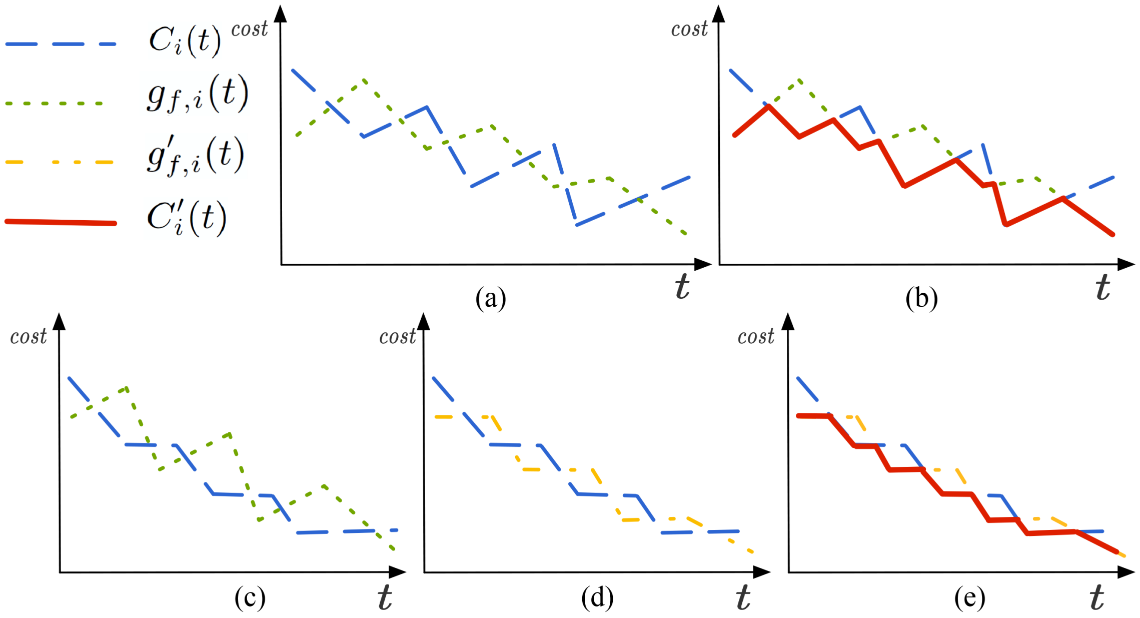


FIGURE 5.4: Minimum Cost Function Update

(a) $g_{f,i}(t)$ and $C_i(t)$ for ordinary vertex v_i. (b) Result of $min(g_{f,i}(t), C_i(t))$ for ordinary vertex v_i. (c) $g_{f,i}(t)$ and $C_i(t)$ for parking vertex v_i, $C_i(t)$ is non-increasing. (d) $g_{f,i}(t)$ applies non-increasing. (e) Result of $min(g_{f,i}(t), C_i(t))$ for parking vertex v_i

5.4.2 Basic MORT Algorithm

The *Path Expansion* in *Basic MORT* algorithm uses a *Dijkstra* way to find the *MORT* from v_s to other vertices. Instead of using the *shortest distance* as the sorting key, we use the minimum value of each vertex's $min(C_i(t))$. Each time we visit a vertex, we update its neighbors' $C_i(t)$ over their *ATI*, until $C_d(t)$ is guaranteed stable. We first describe how to update the minimum cost function in *Minimum Cost Function Updat* , then present path expansion in *Basic Path Expansion Algorithm*. Correctness and complexity are proved after them.

Minimum Cost Function Update (MCFU)

Each time we visit a vertex, we update its out-neighbor's $C_i(t)$. From v_i's point of view , its $C_i(t)$ can only be updated by its in-neighbors. Suppose v_f is v_i's in-neighbor, $C_f(t)$ is v_f's minimum cost function and $w(v_f, v_i, t)$ is the weight function on edge (v_f, v_i). We use $g_{f,i}(t') = C_f(t) + w(v_f, v_i, t), t' = t + w(v_f, v_i, t)$ to denote the cost to travel from v_s to v_i via v_f. Depending on whether v_i is a parking vertex or not, we update $C_i(t)$ differently.

The update of ordinary $C_i(t)$ has two steps as shown in Figure 5.4(a)-(b). We first calculate $g_{f,i}(t)$(dot line). Then we compare $g_{f,i}(t)$ with original $C_i(t)$ (dash line) and use the smaller parts of the two functions as the new minimum cost function $C_i'(t)$ (solid line). We use the line segment intersection detection technique to compute $C_i'(t) = min(C_i(t), g_{f,i}(t))$.

However, if v_i is a parking vertex, we cannot use $g_{f,i}(t)$ directly since the result of $min(C_i(t), g_{f,i}(t))$ may not follow non-increasing property. So we convert $g_{f,i}(t)$ to its non-increasing version $g'_{f,i}(t)$ first before computing $C_i'(t)$. Figure 5.4(c) shows the non-increasing $C_i(t)$ and an ordinary $g_{f,i}(t)$. We convert $g_{f,i}(t)$ into its non-increasing version $g'_{f,i}(t)$ in Figure 5.4(d), and then compute $C_i'(t)$ in Figure 5.4(e). The correctness is guaranteed by the following lemma.

Lemma 5.2. *If both $C_i(t)$ and $g'_{f,i}(t)$ are non-increasing, then $C_i'(t) = min(C_i(t), g'_{f,i}(t))$ is also non-increasing.*

Proof. $\forall t_a < t_b \Rightarrow C_i(t_a) \geq C_i(t_b), g_{f,i}(t_a) \geq g_{f,i}(t_b)$. (1) If $min(C_i(t_a), g_{f,i}(t_a)) = C_i(t_a)$ and $min(C_i(t_b), g_{f,i}(t_b) = C_i(t_b)$, $C_i(t_a) \geq C_i(t_b)$, non-increasing holds. (2) If $min(\ C_i(t_a), g_{f,i}(t_a)) = g_{f,i}(t_a)$ and $min(C_i(t_b), g_{f,i}(t_b) = C_i(t_b)$, $g_{f,i}(t_a) \neg g_{f,i}(t_b) \neg C_i(t_b)$, non-increasing holds. The remaining two situations are similar. □

In order to guarantee the minimum staying time on the parking vertices, we attach a user specified value $v_i.t_{min}$ on each $v_i \in V'$. When computing $g_{f,i}(t)$ from a parking vertex v_f to v_i, the departure time from v_f is changed to $t' = t + v_f.t_{min}$. Thus, the arrival time on v_i further grows to $t'' = t' + w(v_f, v_i, t')$. So $g_{f,i}(t'') \leftarrow C_f(t') + w(v_f, v_i, t')$.

The details of *MCFU* is shown in Algorithm 6. Suppose v_f is the current visiting vertex and v_i is v_f's out-neighbor. *MCFU* computes the updated $C_i'(t)$ using $C_f(t)$ and the edge weight $w(v_f, v_i, t)$. It works in a *sweeping-line* way. Line 2-6 compute the cost to v_i via v_f. If v_f is a parking vertex, then minimum staying time is applied. If v_i is a parking vertex, a non-increasing version $g'_{f,i}(t)$ is generated (Line 7-8). Then it visits the line segments in the $C_i(t)$ and $g'_{f,i}(t)$ together one by one. Initially, it retrieves the first line segment in $C_i(t)$ and $g'_{f,i}(t)$ (Line 9-10), and their corresponding end points (p_1, p_2) and (p_1', p_2') (Line 12-13). Line 14-17 use the line segment intersection technique, which tells the position relation of two lines by computing d_1, d_2, d_3 and d_4, as illustrated in Figure 5.5. If $d_1 > 0, d_2 < 0, d_3 < 0$ and $d_4 > 0$ (Line 18), it is guaranteed that the line segments has an intersection point p' and line segment (p_1, p') should appear in $C_i'(t)$. If $d_1 < 0, d_2 > 0, d_3 > 0$ and $d_4 < 0$ (Line 22), the line segment (p_1', p') should appear in $C_i'(t)$. Then the corresponding points are

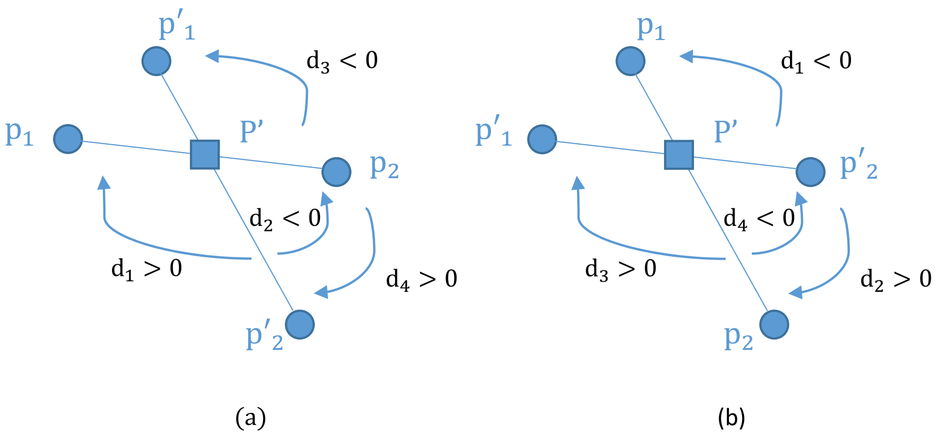


FIGURE 5.5: Line Segment Intersection

updated in Line 21 or Line 25. The loop recurs until it reaches the last end points.Given the active time interval has T time units. In the worst case, there are T end points in the cost function. Within the update of each line segment, it only costs constant time. So the time complexity of the Algorithm 6 is $O(T)$.

Basic Path Expansion Algorithm

Path expansion algorithm maintains a priority queue Q that uses $min(C_i(t))$ as keys to store all the vertices. Each time we pop out the top vertex and update its out-neighbors' $C_i(t)$. This procedure runs on until $C_d(t)$ is guaranteed stable. The details are described in Algorithm 7. Line 2-5 initialize the minimum cost function of each vertex by adding the two end points $(v_i.t_{EA}, v_i.t_{EA} - t_{s1})$ and $(v_i.t_{LD}, \infty)$. Obviously, the source vertex's cost is alway 0. Then these minimum cost functions are organized into a priority queue Q ordered by their $min(C_i(t))$. Each time we pop up the vertex v_i with the smallest $min(C_i(t))$ value in Q and use it to update the minimum cost functions of its out-neighbors v_j using algorithm 6 (Line 12). If $C_j(t)$ has changed and v_j is out of Q, we insert the new function back to Q. If it is changed but still in Q, we just update its key (Line 13-17). The algorithm terminates either when Q becomes empty (Line 7) or when the top function's smallest value is larger than v_d's minimum on road cost (Line 9-10).

Correctness

Theorem 5.3. *Algorithm 7 finds the MORT.*

Algorithm 6: Minimum Cost Function Update(MCFU)

```
Input: $v_i$'s minimum cost function $C_i(t)$, $v_f$'s minimum cost function $C_f(t)$, the cost
       function from $v_f$ to $v_i$: $w(v_f, v_i, t)$ and minimum staying time $v_f.t_{min}$ on $v_f$
Output: $v_i$'s new minimum cost function $C'_i(t)$
begin
    if $v_f \in V'$ then
        $g_{f,i}(t'') \leftarrow C_f(t') + w(v_f, v_i, t')$
        $t' \leftarrow t + v_f.t_{min}, t'' \leftarrow t' + w(v_f, v_i, t')$
    else
        $g_{f,i}(t') \leftarrow C_f(t) + w(v_f, v_i, t), t' \leftarrow t + w(v_f, v_i, t)$
    if $v_i \in V'$ then
        $g'_{f,i}(t) \leftarrow Non - Increase((g_{f,i}(t))$
    $t_1 \leftarrow S_i[0], t'_1 \leftarrow S_i[1]$ //$S_i$: time points in $C_i(t)$
    $t_2 \leftarrow S_f[0], t'_2 \leftarrow S_j[1]$ //$S_f$: time points in $g'_{f,i}(t)$
    while $t_1 \neq S_i.end$ and $t_2 \neq S_j.end$ do
        $p_1 \leftarrow (t_1, C_i(t_1)), p_2 \leftarrow (t_2, C_i(t_2))$
        $p'_1 \leftarrow (t'_1, g'_{f,i}(t'_1)), p'_2 \leftarrow (t'_2, g'_{f,i}(t'_2))$
        $d_1 \leftarrow Direction(p'_1, p'_2, p_1)$
        $d_2 \leftarrow Direction(p'_1, p'_2, p_2)$
        $d_3 \leftarrow Direction(p_1, p_2, p'_1)$
        $d_4 \leftarrow Direction(p_1, p_2, p'_2)$
        if $d_1 > 0$ and $d_2 < 0$ and $d_3 < 0$ and $d_4 > 0$ then
            $(t', C_i(t')) \leftarrow$ intersection point
            $C'_i(t).insert(t', C_i(t'))$
            $t_1 \leftarrow t', t'_1 \leftarrow t'_2, t'_2 \leftarrow S_j.next$
        else if $d_1 < 0$ and $d_2 > 0$ and $d_3 > 0$ and $d_4 < 0$ then
            $(t', C_i(t')) \leftarrow$ intersection point
            $C'_i(t).insert(t', C_i(t'))$
            $t'_1 \leftarrow t', t_1 \leftarrow t_2, t_2 \leftarrow S_i.next$
    return $C'_i(t)$
    Function Direction($p_i, p_j, p_k$)
        return $(p_k - p_i) \times (p_j - p_i)$
```

Proof. Initially, the top of Q is $min(C_s(t))$, which is 0 because v_s is the starting vertex. Then, its out-neighbors can all get their *MORT* after updated from v_s. Suppose v_i is the current top item of Q and v_j is v_i's out-neighbor. If $min(C_j(t)) < min(C_i(t))$, then $\forall \Delta > 0, min(C_i(t)) + \Delta > min(C_j(t))$. So v_i cannot update $C_j(t)$'s minimum value. In fact, v_j has already found its *MORT* that no vertex in Q can reduce it. But the other parts of $C_j(t)$ could be changed. So if $C_j(t)$ is changed, it is inserted back to Q. If $min(C_i(t)) < min(C_j(t))$, v_j might find a better path via v_i and gets updated. And since $min(C_i(t)) < min(C_k(t)), \forall v_k \in Q$, it is ensured that $min(C_i(t)) < min(C_j(t)) + \Delta, \forall \Delta > 0$. Thus, v_i has found its *MORT* that no vertex in Q can reduce it. Finally, after the $min(C_i(t)) > min(C_d(t))$ pops out from Q, it is guaranteed that no vertex in Q can update $min(C_d(t))$. Thus, v_d has found its *MORT*. □

Algorithm 7: Path Expansion Algorithm

```
Input: $G(V, E)$, $Q_{MORT}(v_s, v_d, t_{s1}, t_{s2}, t_d)$
Output: $R_{p^*_{s,d}}$
begin
    for $v_i \in V$ do
        $C_i(v_i.t_{EA}) \leftarrow v_i.t_{EA} - t_{s_1}$
        $C_i(v_i.t_{LD}) \leftarrow \infty$
    Let $Q$ be a priority queue initially containing pairs $(min(C_it), v_i)$, ordered by $min(C_it)$
      in ascending order
    $Q.insert(min(C_s(t)), v_s)$
    while $Q$ is not empty do
        $v_i \leftarrow Q.pop()$
        if $min(C_i(t)) \geq min(C_d(t))$ then
            $break$
        for $v_j \in v_i$'s out-neighbors do
            $C_j'(t) = MCFU(C_j(t), C_i(t), w(v_i, v_j, t))$
            if $C_j'(t) \neq C_j(t)$ then
                if $v_j \in Q$ then
                    $Q.Update(min(C_j(t)), v_j)$
                else
                    $Q.insert(min(C_j(t)), v_j)$
    return $min(C_d(t))$
```

Complexity Analysis

As mentioned previously, the time complexity of the *ATI* algorithm is $O(|V|\log|V|+|E|)$. As for the *Path Expansion* algorithm, we use *Fibonacci Heap* [51] to implement the priority queue. T is used to denote the average number of turning points in $C_i(t)$, which indicates the average number of times a vertex's minimum cost function would be updated among all the vertices. So on average, $C_i(t)$ could be updated T times, which means v_i is visited T times. The maximum number of elements in Q is $|V|$, and it takes $\log|V|$ time to pop out the top element. So it takes $O(T|V|\log|V|)$ time in total to retrieve the top elements in Q. Each edge might be visited T times to update the corresponding minimum cost function, And *MCFU* also takes $O(T)$ time. So the update part of the algorithm takes $O(T^2|E|)$ time. Thus, the total time complexity of *Basic MORT Algorithm* is $O(T|V|\log|V| + T^2|E|)$.

As for the space complexity, the speed profile takes $O(T|E|)$ space, the minimum cost function takes $O(T|V|)$ space, and the graph itself takes $O(|V|+|E|)$ space. Hence, the total space complexity is $O(T(|V| + |E|))$.

5.4.3 Incremental MORT Algorithm

Unlike *Basic MORT* which updates the minimum cost function on the whole active time interval repeatedly, *Incremental MORT Algorithm* uses *Incremental Path Expansion* to build the minimum cost function for each vertex v_i in its $T_i = [v_i.t_{EA}, v_i.t_{LD}]$ sub-interval by sub-interval incrementally, which could reduce unnecessary computations.

Incremental Path Expansion Algorithm

Suppose for a subinterval $I_i = [v_i.t_{EA}, \tau_i] \subseteq T_i = [v_i.t_{EA},\ v_i.t_{LD}]$, we have already computed its minimum cost function $C_i(I_i)$. Then we extend I_i to a larger sub-interval $I'_i = [v_i.t_{EA}, \tau'_i] \subseteq T_i$ where $\tau'_i > \tau_i$ and make sure $C_i(I')$ is refined. It should be noted that the current $C_i(t)$ is constructed by v_i's in-neighbors, and refinement means specifying a larger sub-interval within which the minimum cost function is stable. After that, we update v_i's out-neighbor v_j's $C_j(t)$ in its corresponding time interval $[\tau_j^1, \tau_j^2]$. $v'_j s$ $C_j(t)$ will be refined when we visit them. When τ_i reaches $v_i.t_{LD}$, $C_i(t)$ is guaranteed to be refined over T_i. When τ_d reaches t_d, the algorithm terminates. The details are shown in Algorithm 8. It is made up of two main parts: *Arrival Time Interval Extension* to determine the next sub-interval to refine, and *Minimum Cost Function Update*.

Initially, we set v_s's cost function to 0 in its active time interval and set τ_s to the query's starting time (Line 2). Then we set the other vertices' cost functions to their earliest arrival time minus t_s and the corresponding τ_i to their earliest arrival time $v_i.t_{EA}$ (Line 3-4). At this stage, the subintervals of the vertices are empty. So all cost functions are refined. We use a priority queue Q to organize the information. The elements we insert into Q are pairs of $(\tau_i, C_i(t))$ ordered by τ_i. The while loop (Line 6-28) updates the minimum cost functions and refines the subintervals. For each element in Q, it is ensured that its minimum cost function is well refined in its subinterval $[v_i.t_{EA}, \tau_i]$.

Arrival Time Interval Extension (Line 7-9): Each time we pop out the top pair $(\tau_i, C_i(t))$ from Q. As defined, $C_i(t)$ is well refined within subinterval $[v_i.t_{EA}, \tau_i]$. Then we need to expand this subinterval to a later arrival time such that its well refined claim still holds. Recall that the elements in Q are sorted by τ which is the arrival time of each vertex. It is obvious that τ_i is no bigger than any τ in Q, and the current top pair $(\tau_k, C_k(t))$ has the smallest τ in Q. Thus, for any v_i's in-neighbor v_f, its refined time interval's upper bound $\tau_f \geq \tau_k$. If $C_i(t)$ needs to be updated by v_f, it would be later than $\tau_f + w(v_f, v_i, \tau_k)$. Suppose v_f has the smallest travel cost at τ_k among all v_i's in-neighbors,

Algorithm 8: Incremental Path Expansion Algorithm

```
   Input: $G(V, E)$, $Q_{MORT}(v_s, v_d, t_{s1}, t_{s2}, t_d)$
   Output: $R_{p^*_{s,d}}$
1  begin
2      $C_s(t_s) \leftarrow 0, C_s(v_s.t_{LD}) \leftarrow 0, \tau_s \leftarrow t_s$
3      for $v_i \in V/\{v_s\}$ do
4          $C_i(v_i.t_{EA}) = v_i.t_{EA} - t_s, \tau_i \leftarrow v_i.t_{EA}$
5      Let $Q$ be a priority queue initially containing pairs $(\tau_i, C_i(t))$, ordered by $\tau_i$ in ascending
         order
6      while $|Q| \geq 2$ do
7          $(\tau_i, C_i(t)) \leftarrow Q.pop()$
8          $(\tau_k, C_k(t)) \leftarrow Q.top()$
9          $\tau'_i \leftarrow \tau_k + min\{w(v_f, v_i, \tau_k)|v_f$ is $v_i$'s in-neighbor$\}$
10         for $v_j$ is $v_i$'s out-neighbor do
11             if $v_i \in v'$ then
12                 $C'_j(t'') \leftarrow C_i(t') + w(v_i, v_j, t')$
13                 $t' \leftarrow t + w(v_i, v_j, t), t'' \leftarrow t' + v_i.t_{min}$
14             else
15                 $C'_j(t') \leftarrow C_i(t) + w(v_i, v_j, t)$
16                 $t' \leftarrow t + w(v_i, v_j, t)$
17             $t \in [\tau_i, \tau'_i]$
18             if $v_j \in V'$ then
19                 $C'_j(t) \leftarrow Non - Increase(C'_j(t))$
20             $\tau_j^1 = \tau_i + w(v_i, v_j, \tau_i)$
21             $\tau_j^2 = \tau'_i + w(v_i, v_j, \tau'_i)$
22             $C_j(t) \leftarrow min(C_j(t), C'_j(t), t \in [\tau_j^1, \tau_j^2]$
23             $Q.update(\tau_j, C_j(t))$
24         $\tau_i \leftarrow \tau'_i$
25         if $v_i = v_d$ and $\tau_i \geq t_d$ then
26             return $min(C_i(t))$
27         else if $\tau_i < v_i.t_{LD}$ then
28             $Q.insert((\tau_i, C_i(t)))$
29     $R_{p^*_{s,d}} = min(C_d(t))$
```

then no vertex can change $C_i(t)$ before $\tau_k + w(v_f, v_i, \tau_k)$). That is to say, $C_i(t)$ is well refined in subinterval $[\tau_i, \tau'_i]$, where $\tau'_i = \tau_k + w(v_f, v_i, \tau_k)$ (Line 9).

Minimum Cost Function Update (Line 10-23): For each out-neighbor v_j of v_i, we compute its $C_j(t)$ that departs from v_i within $[\tau_i, \tau'_i]$. This part is similar to *Basic MORT*'s but it works on a smaller time interval. If v_i is a parking vertex, we apply minimum staying time on it (Line 11-13). If its neighbor v_j is a parking vertex, we apply the non-increasing property on it. Then we compute the corresponding new subinterval: lower bound τ_j^1 is $\tau_i + w(v_i, v_j, \tau_i)$ and upper bound τ_j^2 is $\tau'_i + w(v_i, v_j, \tau'_i)$. Finally, we compare the new $C'_j(t)$ with the existing $C_j(t)$ and use the smaller one as the newly computed $C_j(t)$, and update v_j's function in Q. It should be noted that although we have updated $C_j(t)$ in a new subinterval, it is still not well refined within it. It is only when we actually visit v_j as the top element in Q that its refined subinterval can be expanded.

After updating, we go back to see v_i itself. We first set τ_i to its new value τ_i' (Line 24). If τ_i has already reached its latest departure time, then $C_i(t)$ is fully refined and we will not need it anymore. Otherwise, it is still not well refined and thus we insert it back to Q with the new τ_i as the sorting key (Line 28). If v_d is fully refined within its active time interval, the algorithm terminates. As for the minimum value of $C_d(t)$, it is trivial to maintain.

Running Example

We continue with the example used in Section 5.4.1. After running $ATI(v_1, v_5, 0, 30, 130)$, we can get the corresponding initial τ values (earliest arrival times): $\tau_1 = 0, \tau_2 = 40, \tau_3 = 70, \tau_4 = 95$ and $\tau_5 = 105$. Thus, the initial elements in Q are $< (\tau_1 = 0, C_1(t)), (\tau_2 = 40, C_2(t)), (\tau_3 = 70, C_3(t)), (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) >$. $C_0(t)$ has two points $(0, 0)$ and $(25, 0)$, and the other $C_i(t)$ only has one point (τ_i, τ_i).

In the first iteration, v_1 has the smallest τ in Q, so we pop v_1 out of Q. The current top element in Q is $(\tau_2 = 40, C_2(t))$, which has the earliest refined arrival time in Q. Thus, we use $\tau_2 = 40$ as the base time. v_1 has no in-neighbor, so $min(w(v_f, v_1, 40)) = \infty > v_1.t_{LD}$. Then v_1 is well refined in its active time interval. Now we update v_1's out-neighbors in the refined time interval [0,25]. Because v_2 is v_1's only out-neighbor and the edge cost function is $w(v_1, v_2, t)$, we compute $C_2(t)$ on time interval $[0 + w(v_1, v_2, 0), 25 + w(v_1, v_2, 25)] = [40, 65]$. It should be noted that although $C_2(t)$ is newly computed, τ_2 remains 40, which means the $C_2(t)$ from $t = 40$ is still unrefined and might be changed by other vertices.

In the second iteration, the current Q is $< (\tau_2 = 40, C_2(t)), (\tau_3 = 70, C_3(t)), (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) >$. We pop out the top element v_2 and visit it. The current top element is $\tau_3 = 70$, so none of the in-queue vertices' refined latest arrival time is earlier than 70, which means all the vertices' time interval before 70 has been used to update their out-neighbors. For v_2's in-neighbor v_1, if it departs at $t = 70$, it will arrive v_2 at $70 + w(v_1, v_2, 70) = 97.5$. So it is guaranteed that no vertices can change $C_2(t)$ in time interval $[40, 97.5]$. Thus, $C_2(t)$ is refined in $[40, 97.5]$, and its new τ_2 is extended to 97.5. However, since $97.5 > v_2.t_{LD}$, v_2 is also well refined in its active time interval. Then we update v_2's out-neighbors (v_3 and v_4). First we consider v_3. The new time interval for v_3 is $[40 + w(v_2, v_3, 40), 65 + w(v_2, v_3, 65)] = [70, 95]$. Since the previous

$C_3(t)$ has no value in [70,95], we use the new one directly. Then we update v_4 in time interval $[40 + w(v_2, v_4, 40), 65 + w(v_2, v_4, 65)] = [95, 138.75]$. However, since v_4 is a parking vertex, it has to follow the non-increasing property.

In the third iteration, Q becomes $< (\tau_3 = 70, C_3(t)), (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) >$. We pop out top element and visit v_3. The current top is $\tau_4 = 95$ and $w(v_2, v_3, 95) = 30$. So v_3's refined time interval is extended to $[70, 95 + 30] = [70, 125]$, which is larger than v_3's active time interval. So v_3 is also well refined. v_3' out-neighbor v_5's minimum cost function will be computed in time interval $[70 + w(v_3, v_5, 70), 95 + w(v_3, v_5, 95)] = [105, 130]$. τ_5 remains 105. The current Q is $< (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) >$.

In the fourth iteration, we visit v_4 and the top element is $\tau_5 = 105$. $w(v_2, v_4, 105) = 100$ and it extends τ_4 to 205, which exceeds v_4's active time interval, so v_4 is also well refined. We update v_4's out-neighbor v_5 in time interval $[95 + w(v_4, v_5, 95), 125 + w(v_4, v_5, 125)] = [108.75, 130]$. The new $C'_5(t)$ has some lower values compared with the previous one, so we take the lower one as the $C_5(t)$. Finally, the Q has only one element, and we can guarantee that no vertex can update v_5 now. So the minimum on-road travel time from v_1 to v_5 is 100.

Correctness

Before we prove the correctness of *Incremental MORT Algorithm* in Theorem 5.6, we first prove the minimum cost function is correctly computed. Lemma 5.4 proves Line 7-9 is correct. Lemma 5.5 proves Line 10-23 is correct.

Lemma 5.4. *When v_i is popped out and visited, it is guaranteed that $C_i(t)$ will not change in $[\tau_i, \tau'_i]$.*

Proof. Suppose τ_j is the current top τ in Q. Thus, $\forall \tau_k \in Q, \tau_k \geq \tau_j \Rightarrow C_k(t)$ is well refined before τ_k, which means $\forall v_k \rightarrow v_o, C_o(t)$ has been updated from v_k before τ_k. In other words, no update before time τ_j is possible from now on. The earliest possible time to update from v_k to v_o is τ_j. Suppose $v_f \rightarrow v_i$, so the earliest possible time to update from v_f to v_i is also τ_j. If we depart from v_f at τ_j, the earliest arrival time at v_i is $\tau_j + w(v_f, v_i, \tau_j)$. Suppose $w(v_f, v_i, \tau_j)$ is the smallest among all in-neighbors of v_i, then the earliest change of $C_i(t)$ will not happen before $\tau'_i = \tau_j + w(v_f, v_i, \tau_j)$. So $C_i(t)$ will not change in $[\tau_i, \tau'_i]$. □

Lemma 5.5. *$C_i(t)$, where $t \in [\tau_i, \tau'_i]$, has been updated before it is refined.*

Proof. $\tau_i = min\{\tau_j + w(v_f, v_i, \tau_j)|\forall v_i\}$. If v_f is not in Q, then $C_f(t)$ is already refined. So when we finish refining $C_f(t)$, we will update $C_i(t)$ from v_f. If v_f is in Q, then $\tau_f \geq \tau_j \geq \tau_i$. Otherwise we should have visited v_f earlier than v_i. Thus, v_f's refinement lower bound is no earlier than τ_j, so $C_i(t)$ has been updated from v_f at τ_f, which leads to $\tau_f + w(v_f, v_i, \tau_f) \geq \tau_i'$. Hence, $C_i(t)$ has been updated in subinterval $[\tau_i, \tau_i']$. □

Theorem 5.6. *Algorithm 8 finds the MORT.*

Proof. Lemma 5.5 guarantees each $C_i(t)$ is fully updated, and Lemma 5.4 ensures the final $C_i(t)$ is validated incrementally. When v_d's τ_d reaches the latest arrival time t_d, v_d's minimum cost function $C_d(t)$ is fully refined and will not be changed even if the while loop runs on. All the $C_i(t)$ are updated by its in-neighbors, so they are the same as *Basic MORT*'s minimum cost functions. Therefore, the minimum value of $C_d(t)$ is the minimal on-road travel time. □

Complexity Analysis

The *ATI* takes $O(|V|\log|V| + |E|)$ time. The initialization phase (Line 2-5) takes $O(V)$ time. We use *Fibonacci Heap* [51] to implement the priority queue. The size of Q is at most $|V|$, so the extract-min operation on Q takes $O(\log|V|)$ time. Since each vertex v_i's minimum cost function is constructed incrementally, we use L_i to denote the number of its subintervals. Therefore, L_i is actually the number of times v_i would be extracted from Q, which takes $L_i \log|V|$ time. The update and insert on *Fibonacci Heap* take $O(1)$ time, so the maintaining of Q takes $O(\Sigma_{i=0}^{|V|} L_i |V| \log|V|) = O(L(|V|\log|V|))$ time, where L is the average number of subintervals. On the other hand, during the update, we visit all v_i's in-neighbors, which is the same as in-edges E_i^{in}. So if we visit all the in-neighbors of all the vertices, we actually visit every edge. Thus, $\Sigma_{i=0}^{|V|}|E_i^{in}| = |E|$. So the total time complexity is $O(\Sigma_{i=0}^{|V|} L_i(\log|V| + |E_i^{in}|)) = O(L(|V|\log|V| + |E|))$.

Now let's analyze the lower-bound of L_i. Firstly, suppose τ_i^j is the top value in Q and τ_k is the head value, $\tau_i^j \leq \tau_k$. Then $\tau_k + min(w(v_f, v_i, \tau_k)) = \tau_i^{j+1}$, so $\tau_k < \tau_i^{j+1}$. Eventually, we can have a L_i such that $\tau_i^{L_i} \geq v_i.t_d$. Next, we define $\eta_i^0 = v_i.t_s$ and $\eta_i^{j+1} = \eta_i^j + min(w(v_f, v_i, \eta_i^j))$. Eventually, we can get a J_i such that $\eta_i^{J_i} \geq v_i.t_d$. Since for the same j, τ_i^j is always smaller than η_i^j, so we can get $L_i > J_i$. If we use J to denote the average number of J_i, then the lower-bound of L is J. Obviously, $J > T$, so L is also bigger than T.

For the space complexity, the time-dependent parking graph takes $O(|V| + T|E|)$ space. Each minimum cost function $C_i(t)$ takes $O(T)$ space. Q has at most $|V|$ elements, so the size of Q is $O(T|V|)$. Hence, the overall space complexity is $O(T(|V| + |E|))$.

5.4.4 Application Scenarios

In this section, we provide three examples to explain how our algorithm works in different scenarios. It should be noted that the graph structure and time-dependent information are crucial for finding the desired results.

First, suppose a commuter wants to arrive office faster and depart later. In fact, this is just an *ISFP* problem, so we can run our algorithm on a road network that only allows waiting on the departure vertex with a user predefined departure time interval.

Second, consider a truck driver who needs a forced rest every period of time at the service stations along the highway. In this case, the graph is a network of highway, and the parking vertices are some service stations, each has a pre-defined minimum staying time. The traveling time between these stations roughly equals to the driver's maximum driving time. Therefore, the force waiting is included in the computation and the minimum rest time is guaranteed for safe driving.

Finally, suppose a traveler is planning a journey from one city to another in several weeks time and wants to visit national parks along the route. In this case, the graph should only contain the national parks as vertices and allows waiting on all of them, which is another extreme case of our model. The graph structure should express the rough traveling order. In this case, it could be organized into a layered graph, and we only visit one of the vertices on the same layer. In an extreme case when the traveler wants to visit every park, the graph should be organized as a linear line. The edges only exist between the vertices in neighboring layers. It should be noted the graph structure can also reflect the distribution of waiting schedule. We can set the distribution of parking vertex manually to meet users' waiting requirement (e.g. a forced rest every period of time). Next, we should not use the traffic condition as the only parameter to determine the time-dependent weight functions. In fact, the functions should take both travel cost and drivers' willingness into account. For instance, it is a journey rather than hurrying on the way, so we should avoid the unsafe night driving. Thus, the weights during night time should be set much higher even though the traffic condition is good. In fact, all the weights for the time that are not suitable for driving, either due to bad traffic condition or due to travelers' preference, should be set higher. After that, our algorithm could find a *MORT* traveling schedule on this time-dependent graph.

5.5 α-MORT Approximation

In this section, we present several approximation methods to solve the *MORT* problem faster with a guaranteed lower bound α. As analyzed in Section 18, the time complexity is significantly affected by the number of turning points in $C_i(t)$. What is worse it grows larger as the expansion grows, which makes the computation slower and slower. So the key to speed up is decreasing the number of turning points, especially the useless ones. However, we cannot determine if one turning point will end up with the optimal result until the final $C_d(t)$ is constructed. Therefore, we design an approximation approach that can guarantee the final result is no less than $\alpha C_d^*(t)$, $\alpha \in (0, 1]$. Section 5.5.1 introduce the approximation error α and how the error grows as the path grows, Section 5.5.2 to 5.5.4 describe three approximate methods in detail.

5.5.1 Error Bound α and Turning Point Pruning

Given an input approximation ratio α, we aim to compute a path whose *MORT* time $A_d^*(t) \geq \alpha C_d^*(t)$. However, the approximation cannot be applied on each edge along the path directly.

Suppose a path is made up of a series of consecutive edges $\hat{E} =< e_1, e_2, ..., e_n >$ and $||\hat{E}||$ is the length of $\hat{E}$. If we apply an approximation factor α_1 on e_1, α_2 on e_2 and so on, the error of the final result does not grow linearly, as shown below.

$$\begin{aligned}
||\hat{E}||' &= ((((e_1\alpha_1 + e_2)\alpha_2) + e_3)\alpha_3 + ...e_n)\alpha_n \\
&= \alpha_1\alpha_2\alpha_3...\alpha_n\, e_1 + \alpha_2\alpha_3...\alpha_n e_2 + ... + \alpha_n e_n \\
&= \prod_{j=1}^{n} \alpha_j e_1 + \prod_{j=2}^{n} \alpha_j e_2 + ... + \prod_{j=n}^{n} \alpha_j e_n \\
&= \sum_{i=1}^{n} \prod_{j=i}^{n} \alpha_j e_i
\end{aligned}$$

To achieve $||\hat{E}||' \geq ||\hat{E}||$, we have to guarantee $\prod_{j=1}^{n} \alpha_j \geq \alpha$. In another word, we can view α as a total budget of pruning power along the path, the larger the budget assigned to a vertex, the stronger pruning power it has to reduce the turning points. Because the turning point numbers of the earlier visited vertices are much smaller than those of the latter visited ones, we concentrate the pruning power to the latter vertices by setting a global turning point number threshold ρ: Only those vertices whose turning point numbers are larger than ρ will be pruned. Because in the *ATI* computation we already have two functions of *earliest arrival path* and *latest departure path*, we

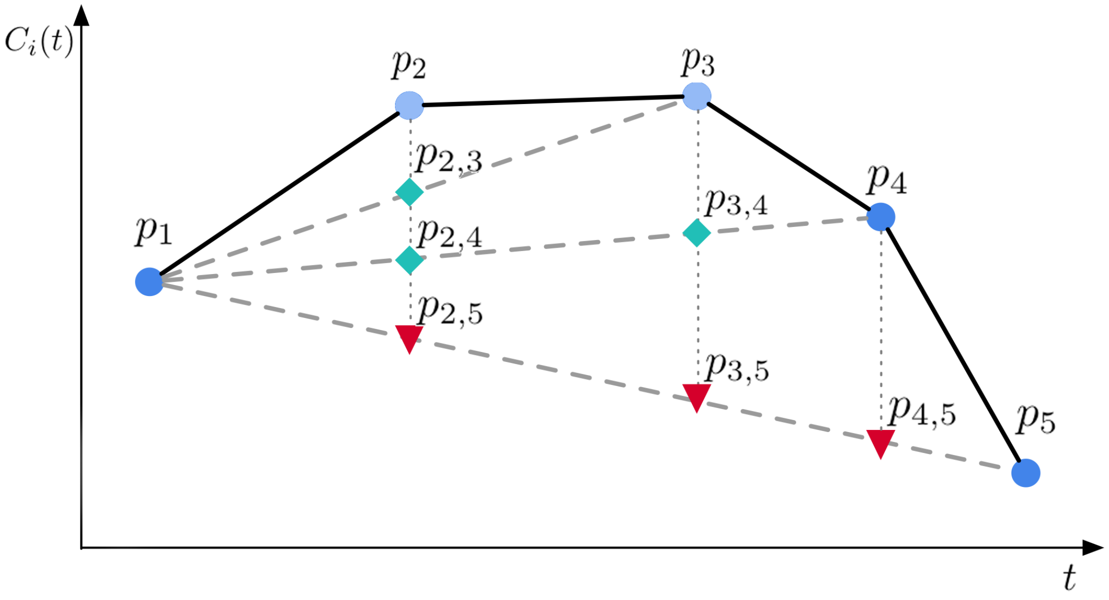


FIGURE 5.6: $C_i(t)$ Turning Points Pruning Example
p_2 can be pruned because its new point $p_{2,3}$ represented by line (p_1, p_3) is larger than $\alpha_i p_2$. p_3 can be pruned because $p_{2,4} > \alpha_i p_2$ and $p_{3,4} > \alpha_i p_3$. But p_4 cannot be pruned because their new values on the new approximate line (p_1, p_5) are smaller than $\alpha_i p_2$, $\alpha_i p_3$ and $\alpha_i p_4$.

use $\frac{min(|EA(v_d)|,|LD(v_s)|)}{3}$ as a heuristic threshold value, where $|EA(v_d)|$ and $|LD(v_s)|$ are the turning point numbers of those two paths. The details of how to assign pruning power α_j are discussed from Section 5.5.2 to Section 5.5.4.

At this stage, we assume $|C_i(t)| > \rho$ and it has a pruning power α_i. The pruning process visits the turning points one by one in a sliding window way, as shown in Algorithm 9. Each time we visit a turning point p_i, we put it into a *PointList* (line 4). It can be pruned only if all the points p_j in *PointList* can be safely represented by point $p_{j,i+1}$ on line (p_{i-1}, p_{i+1}) (line 7). The safe representation has two conditions (Line 8). Firstly, $p_{j,i+1}$ has to be no smaller than $\alpha_i p_j$, as required by approximation bound. Secondly, p_j is no smaller than $p_{j,i+1}$, because the smaller value has a higher possibility to result in the final optimal result. If any p_j does not satisfy these two conditions, p_i cannot be pruned and we empty the *PointList*. When all the points are visited, we return the remaining points as the approximate function $A_i(t)$. Since $p_{j,i+1} \geq \alpha_i p_j$ is strictly required, $A_i(t) \geq \alpha_i C_i(t)$. In the worst case when all the points within $C_i(t)$ is pruned, the testing in line 8 has to run $O(|C_i(t)|^2)$ times. However, it has a near linear running time in practice.

Algorithm 9: $C_i(t)$ Pruning Algorithm

```
Input: $C_i(t) = (p_1, p_2, ..., p_n)$, pruning power $\alpha_i$
Output: $\alpha_i$-approximate $A_i(t)$
begin
    $i \leftarrow 2$
    while $i \leq |C_i(t)| - 1$ do
        $PointList.insert(p_i)$
        for $p_j \in PointList$ do
            //compute $p_{j,i+1}$ on $(p_{i-1}, p_{i+1})$
            $p_{j,i+1} \leftarrow Compute(p_{i-1}, p_{i+1}, p_j)$
            if $p_{j,i+1} \geq \alpha_i p_j$ and $p_j \geq p_{j,i+1}$ then
                $i \leftarrow i + 1$
                $PointList.clear()$
                $break$
        $C_i(t).prune(p_i)$
    return $A_i(t) \leftarrow C_i(t)$
```

Figure 5.6 shows a pruning example.

5.5.2 Even Distribution

The first way to assign pruning power is distributing them evenly. Suppose $|\hat{E}|$ is number of edges in path $\hat{E}$. The most straightforward way is to assign $\alpha_i = \alpha^{\frac{1}{|\hat{E}|}}$. Obviously,

$$\sum_{i=1}^{n}\prod_{j=i}^{n}\alpha_j e_i = \sum_{i=1}^{n}\prod_{j=i}^{n}\alpha^{\frac{1}{|\hat{E}|}} e_i = \sum_{i=1}^{n}\alpha^{\sum_{j=i}^{n}\frac{1}{|\hat{E}|}} \; > \; \alpha\sum_{i=1}^{n} e_i$$

However, pruning power $\alpha^{\frac{1}{|\hat{E}|}}$ becomes weaker when $|\hat{E}|$ is larger, which makes the pruning insufficient. Therefore, we restrict the pruning power α shared by only $|\check{E}|$ vertices along the path, where $|\check{E}| \ll |\hat{E}|$. Thus, the vertices has larger pruning power when their turning point numbers surpass the threshold ρ.

5.5.3 Exponential Distribution

The second way to distribute pruning power is decreasing the power exponentially. In this way, the first pruning vertex can have a much larger power than those in then even distribution. We assign

$\alpha_1 = \alpha^{\frac{1}{2}}$, $\alpha_2 = \alpha^{\frac{1}{2^2}}$ and so on. In this way, the approximate bound is guaranteed, as proved below:

$$\begin{aligned} ||\hat{E}||' &= \alpha_1\alpha_2\alpha_3...\alpha_n\; e_1 + \alpha_2\alpha_3...\alpha_n e_2 + ... + \alpha_n e_n \\ &= \alpha^{\frac{1}{2}}\alpha^{\frac{1}{2^2}}...\alpha^{\frac{1}{2^n}} e_1 + \alpha^{\frac{1}{2^2}}...\alpha^{\frac{1}{2^n}} e_2 + ... + \alpha^{\frac{1}{2^n}} e_n \\ &= \alpha^{\sum_{i=1}^{n}\frac{1}{2^i}} e_1 + \alpha^{\sum_{i=2}^{n}\frac{1}{2^i}} e_2 + ... + \alpha^{\sum_{i=n}^{n}\frac{1}{2^i}} e_n + \\ &> \alpha e_1 + \alpha^{\frac{1}{2}} e_2 + ... + \alpha^{\frac{1}{2^n}} e_n > \alpha \sum_{i=1}^{n} e_i = \alpha||\hat{E}|| \end{aligned}$$

Although the pruning is large in the beginning, it decreases fast as the pruned vertices grows. So similar to the *Even Distribution*, we also set a small upper-bound of n to avoid useless pruning.

5.5.4 Dynamic Exponential Distribution

The previous two methods assign fixed pruning power to each vertices and do not care whether the power is fully utilized or not. In fact, most of the vertices only use part of their power, which is a waste of the precious budget. In order to take the most advantages of the precious pruning budget, we propose the *Dynamic Exponential Distribution* method.

Like the *Exponential Distribution*, the pruning power also decreases exponentially. However, instead of dividing the previous pruning power's logarithm by 2, we divide the remaining pruning power's logarithm by 2. Initially, the algorithm keeps pruning power's logarithm $\mathit{\Delta}_i$ for each vertex and set them to 1. The first pruning vertex v_k has the pruning budget $\alpha^{\frac{\mathit{\Delta}_k}{2}} = \alpha^{\frac{1}{2}}$. During the pruning, we can get its actual pruning usage by $\beta = max(p_{i,j}/p_i)$, where p_i is the pruned point. Then the remaining pruning power logarithm for $v'_k s$ out-neighbor v_j is $\delta_k - log_\alpha \beta$. If v_j is to be pruned, its pruning budget is $\alpha^{\frac{\delta_k - log_\alpha \beta}{2}}$. We also set a lower bound for α_i to avoid the useless pruning.

The proof of bound guarantee is similar to *Exponential Distribution*.

5.6 Experiments

In this section, we present the results of a comprehensive performance study on one real-world road networks and a small-world graph with different speed profiles, to demonstrate the effectiveness and efficiency of our algorithms.

5.6.1 Experiment Setup

We use the same Beijing and Shanghai maps as in Section 4. The corresponding speed profiles are compressed using *Bottom-Up PLA* Algorithm.

We test the algorithms under four variations. The first one is the distance of two vertices in road network. The second one varies the starting time interval size from 1 hour, 2 hours, 3 hours to 4 hours. The next one tests the performance under different speed profiles (50, 100, 200, 400 turning points), and the last one varies the percentage of parking vertices (5%, 10%, 50%, 100%). Except for the third test, which uses synthetic speed profile, all the experiments use the speed profiles generated from the trajectory data.

We ran all the experiments on a Dell R720 PowerEdge Rack Mount Server which has two Xeon E5-2690 2.90GHz CPUs, 192GB memory, 1TB hard disk, and runs Ubuntu Server 14.04 LTS operating system.

5.6.2 Comparison with Existing Algorithms

In this section, we compare the minimal on-road time routes computed by our algorithm with paths generated by the other path planning algorithms under different configurations. We compared our methods with the following algorithms: 1) *SP (Shortest Path)* which computes the shortest path between two vertices. We set the departure time randomly within the time interval. 2) *EAP (Earliest Arrival Path)* and *LDP (Latest Departure Path)*, which are two bypass results when computing the minimal on-road time. 3) *FP (Fastest Path)* [21]. 4) *IFP (Iterative Fastest Path)* which uses the *FP (Fastest Path)* algorithm iteratively to get the approximate minimal on-road time path, as described in Section 5.1. The results achieved by our algorithms are labeled with *MORT*. We do not distinguish the two versions of our algorithms in this experiment since they produce the same on-road travel time.

In the first test, we change the distance between v_s and v_d. We randomly select four sets of vertex pairs with the approximate distance of around 10km, 20km, 30km, and 50km in the two maps. The starting time interval is set to be 4 hours. 10% of the vertices are selected as parking vertices. The results are shown in Figure 5.7(a)-(b). It is obvious that our algorithms always produce the shortest on-road travel time, followed by *IFP* and *FP*. As for the other three algorithms, they do not have a chance to achieve a shorter on-road time by changing the departure or waiting time, so their performance is unstable and worse than the previous algorithms in average.

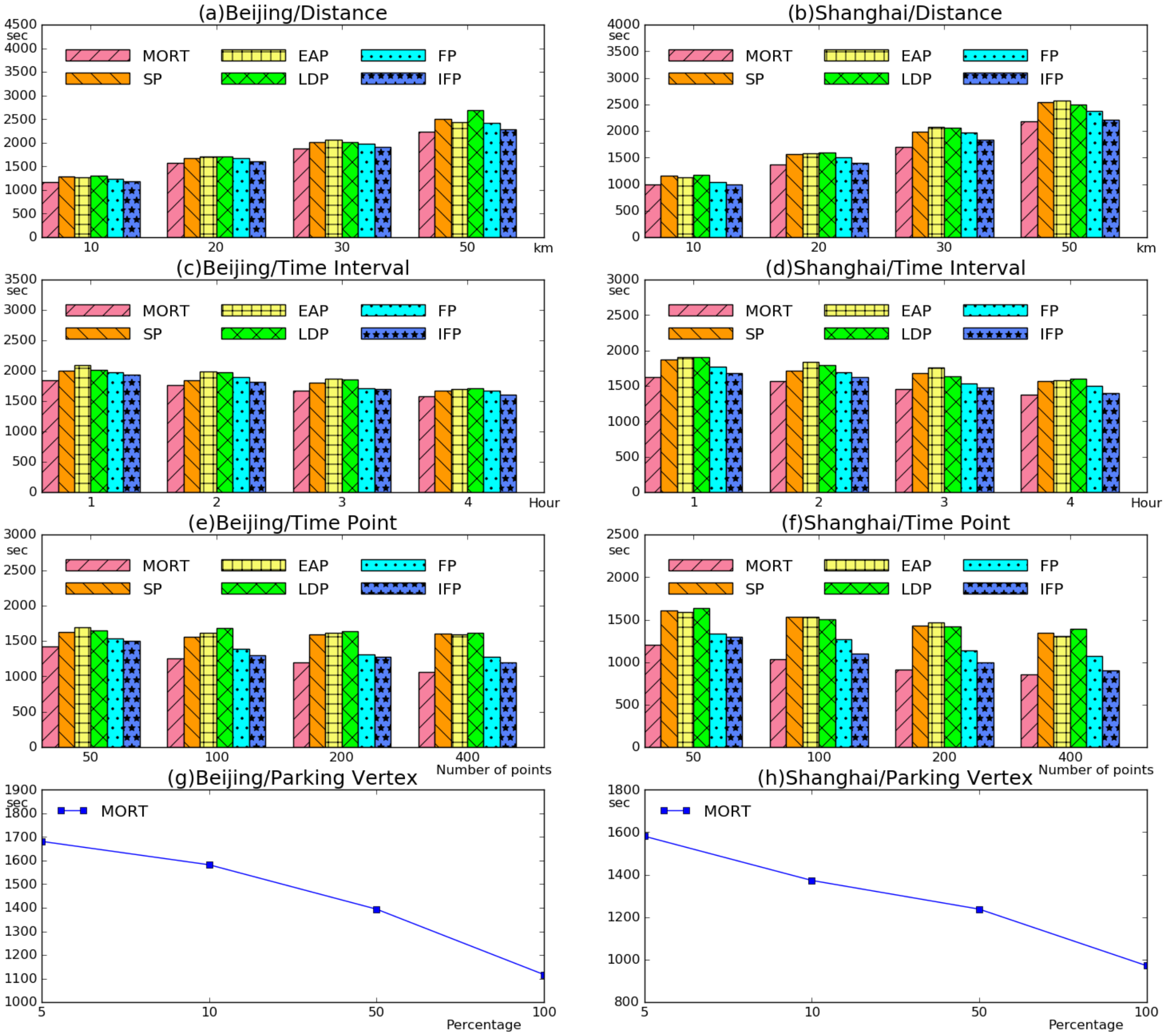


FIGURE 5.7: Results of Minimal On-Road Time

The second test varies the length of starting time interval from 1 hour to 4 hours. The distance is set to be 20, speed profile is 100 and parking vertex is 10%. Figure 5.7(c)-(d) show the results. As the length of the time interval grows, more possible starting time emerge, so the on-road time of *FP* and *IFP* decrease. As for *MORT*, it also decreases because it has a longer time to wait for a faster path on the parking vertices. And it decreases faster than *FP* because it can get more benefits. As for the other algorithms, they do not change much correspondingly due to the same reason as the previous test.

The third test evaluates the influence of the speed profile granularity, whose turning point numbers are 50, 100, 200 and 400. The distance is also 20, parking vertex is 10% and the starting time interval is 4 hours. We can see from Figure 5.7(e)-(f) that as the total number of turning points grows, the number of the turning points that have smaller traveling cost also increases. So there is a higher chance for *FP* and *IFP* to find paths with smaller on-road time. And *MORT* also decreases more distinctly for the same reason.

The last test studies the influence of the park vertex percentage, which varies from 5%, 10%, 50% to 100%. The distance is 20, the speed profile has 100 turning points and time interval is 4 hours. Figure 5.7(g)-(h) only show the on-road time of *MORT* because the results of all the other methods do not change along with the percentage of parking vertices. It is easy to draw the conclusion that as the percentage rises, the on-road time drops accordingly since it has more vertices able to wait for a shorter on-road time.

5.6.3 Algorithm Running Time

In this section, we compare the running time of our algorithms on the three graphs under the same setting of the previous experiments. Apart from the running time of our *Basic* and *Incremental* algorithms, we also show the performance of *IFP* in the first test, and *Fastest Path* in the second and third tests.

Firstly, Figure 5.8(a)-(b) show the results under different distances. As the distance grows, the numbers of the visited vertices and edges also grow, so the running time increases. Not surprisingly, the running time of *IFP* soars up, so we demonstrate it in exponential step. Secondly, the impact of time interval is illustrated in Figure 5.8(c)-(d). As the interval grows longer, the active time interval also grows, which makes the minimum cost function longer. Both algorithms run slower because more turning points appear in the minimum cost functions.

Furthermore, we demonstrate the running time on different speed profiles in Figure 5.8(e)-(f). If the density of the speed profile rises, the number of the turning points in the minimum cost function also increases. However, different from the growth of the time interval, which increases the turning points linearly, the growth of time points in speed profile raises the point number in minimum cost functions more dramatically. And the *Basic* algorithm has higher cost on maintaining larger cost function, so it becomes slower than the *Incremental* algorithm. In addition, as shown in Figure 5.8(c)-(f), *FP* is always slower than *MORT*. The reason is that *FP* cannot apply *non-increasing*, so it always has more turning points in the minimum cost functions.

Finally, we present the influence of the percentage of parking vertices in Figure 5.8(g)-(h). Since

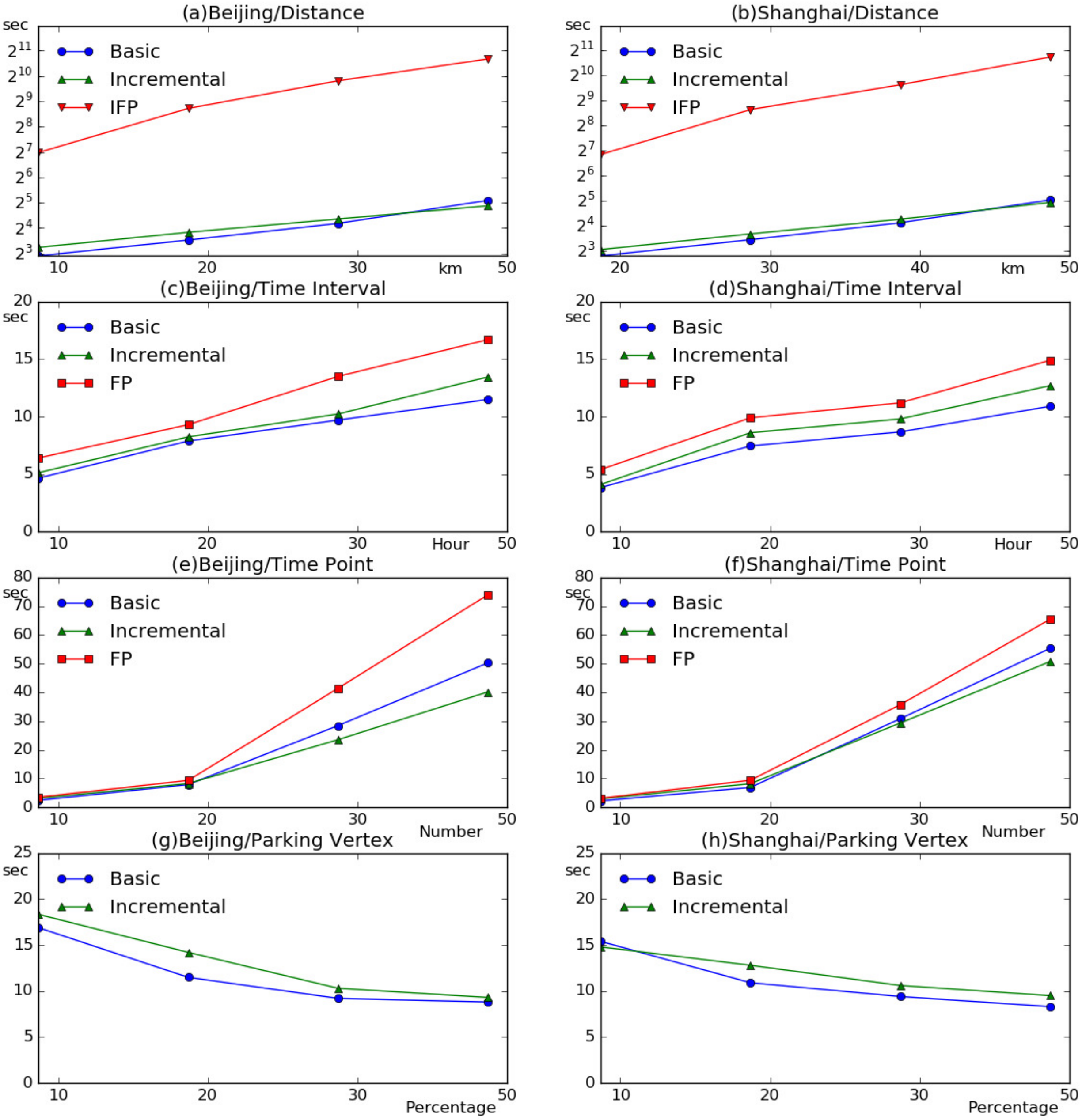


FIGURE 5.8: Algorithm Running Time

the minimum cost function of a parking vertex is non-increasing, the number of its turning point is smaller than the ordinary vertices. Therefore, as the percentage of the parking vertices increases, the total number of the turning points decreases. So the running time drops correspondingly. We do no present the running time of *FP* because its running time is not affected by the parking vertices.

Even if our algorithms are faster than the state-of-art fastest path algorithm, it is still slow for the long distance query. So we will present an index to answer the time-dependent path queries under a

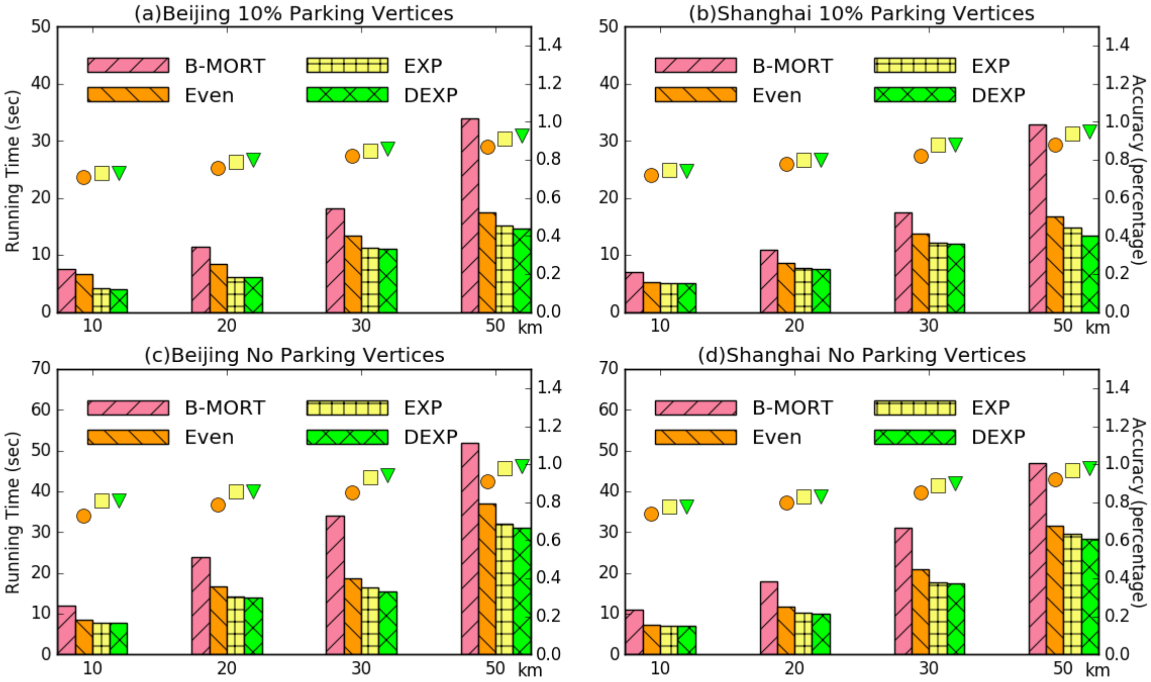


FIGURE 5.9: Running time and accuracy of $\alpha-$MORT

second in the future work. But algorithms in this paper are the basis for the index.

5.6.4 Approximation Algorithm

In this section, we test the running time and accuracy of our approximation algorithms on two road networks. The results are shown in Figure 5.9. As analyzed in Section 5.5.1, the error decreases as the path grows longer. In fact, we can still get a good approximation result even if the initial error bound is small. We show the results of $\alpha = 0.2$ in this test. First of all, as the distance grows, the approximation performs better. For example, the running time is nearly half of the original algorithm in the 50km test, while the accuracy is around 97%. This is because the longer paths have much more turning points than the short ones, and pruning those points could lead to more benefit. And the pruning power are the same for all the paths regardless of their lengths, so the longer paths have higher accuracy. Secondly, even though the *Even Distribution* can reduce the running time dramatically, the *Exponential Distribution* has an even better performance, while the *Dynamic Exponential Distribution* is only slightly better the *Exponential Distribution*. The reason of it is that although *Dynamic Exponential* makes better use of the pruning budget, its dynamic mechanism takes extra costs. Finally, the percentage of parking vertices also affects the approximation performance.

The no parking tests have higher accuracy and speedup. The reason is the same as the distance: less parking vertices along the route results in more turning points.

5.7 Summary

In this chapter, we have studied a new route scheduling problem called *MORT* query that aims to minimize on-road time in time-dependent graphs with parking vertices. *MORT* query further generalizes the path planning problem studied before in time-dependent graphs from allowing the traveler to choose the optimal departure time to minimize on-road travel time that allows multiple stops at parking vertices. From theoretical point of view, *MORT* is the most general type of time-dependent route scheduling problem, which covers all previous problems both in terms of problem formulation and also algorithms. From practical point of view, *MORT* query is useful in many applications, to name a few, minimizing fuel consumption for trucks and advising people to stop and do other things to avoid getting stuck in heavy traffic. From algorithm design and database query processing points of view, *MORT* queries are significantly more complex than time-dependent shortest/fastest path queries. We have proposed two algorithms to do *MORT* route scheduling. The *Basic MORT Algorithm* computes a *minimum cost function* directly and takes $O(T|V|\log|V| + T^2|E|)$ time. The *Incremental MORT Algorithm* reduces the time complexity by computing the *minimum cost function* incrementally and takes $O(L(|V|\log|V| + |E|))$ time. Our extensive studies in road network and small-world graph have confirmed that our algorithms could find minimal on-road time paths more efficiently.

Chapter 6

Time Dependent 2-Hop Labeling

Route scheduling on time-dependent road network is slow due to it has a problem complexity of $\Omega(T(|V|\log|V|+|E|)$, where T is the number of turning points in the result's minimum cost function, $|V|$ is the number of vertices and $|E|$ is the number of edges. To make things worse, T grows larger as the route becomes longer or the query time interval becomes bigger, especially for a *fastest path profile* query where the length of query's time interval is 24 hours. In this chapter, we aim to answer the *fastest path profile* query on time-dependent road network faster by extending the *2-hop labeling* approach, which is fast in answering shortest distance query on the static graph. However, building an index on a time-dependent graph is both time and space consuming, so currently only online-search approaches exist, like *time-dependent CH* and *time-dependent SHARC*. Apparently, their query answering power is limited by the online searching. To further speed up the query answering, we first propose the *time-dependent 2-Hop* on the small graph. To expand on the large road network, we partition the road network into smaller subgraphs, and build time-dependent 2-hops within and between partitions. Furthermore, we propose an approximation version to reduce the index size several times down and accelerate the query answering even more, with a user given error bound. Experiments on a real road network and its linear piecewise speed profile show that our approach outperforms the state-of-art fastest path index approaches and can speed up the query answering by hundreds of times.

6.1 Introduction

With the fast development of GPS technology and mobile network, traffic condition is easier to obtain than ever. For example, *Google Map* has provided a feature called *Google Traffic*, which displays traffic conditions in real time on major roads and highways by analyzing the GPS-determined locations transmitted to Google from a large number of mobile phone users. GPS navigators like *TomTom* also have collected traffic data during the past decade to provide traffic prediction feature. In addition to that, lots of approaches and applications are developed to infer traffic speed on roads from the drivers' trajectory data [38, 36, 41, 33]. With traffic condition information at hand, the measurement used in path planning has changed from the distance to travel time. So finding the fastest path from one place to another has become an essential query. In fact, it has a much wider range of applications other than trip planning, such as the spreading of information [152] and disease [153], metabolic pathway analysis in biochemist[154] and within cells [155], just to name a few.

It is fast to compute the *single starting time fastest path (SSFP)*, whose departure time is given as a single time point, because its time complexity is same as *Dijkstra*'s $O(|V|\log|V| + |E|)$ [20, 92], where $|V|$ is the number of vertices and $|E|$ is the number of edges. However, when the departure time is a time interval, its complexity lower-bound grows to $\Omega(T(|V|\log|V| + |E|))$ [144], where T is the number of turning points in the result's function. We call it *interval starting time fastest path (ISFP)*. Such a query is often used to find an optimal departure time that would result in a path with least traveling time. If we extend the departure time interval to the whole time span, we are facing the *fastest path profile query (FPP)*, which is useful in bulk query answering and is a fundamental query for traffic analysis. In fact, it is a general query that can answer any *SSFP* and *ISFP*. Unfortunately, it is even slower than *ISFP* because it has a larger T. Therefore, in this paper, we study the problem of how to answer *FPP* query faster. Additionally, we use term*fastest path* to denote *FPP* throughout this paper because *FPP* is essentially a fastest path, and *fastest path* is widely used and can self-explain. Furthermore, all the edges' time-dependent weight functions follow *FIFO* property, which forbids overtaking.

In this work, we aim to speed up the fastest path query answering even faster by taking advantages of *2-hop* labeling [6]. By assigning two sets of labels for each vertex's in- and out-distance information to other vertices, static *2-hop* labeling can answer the shortest path query only using the labels [67, 156, 68, 65, 62]. However, no existing work ever tries to extend it to the time-dependent

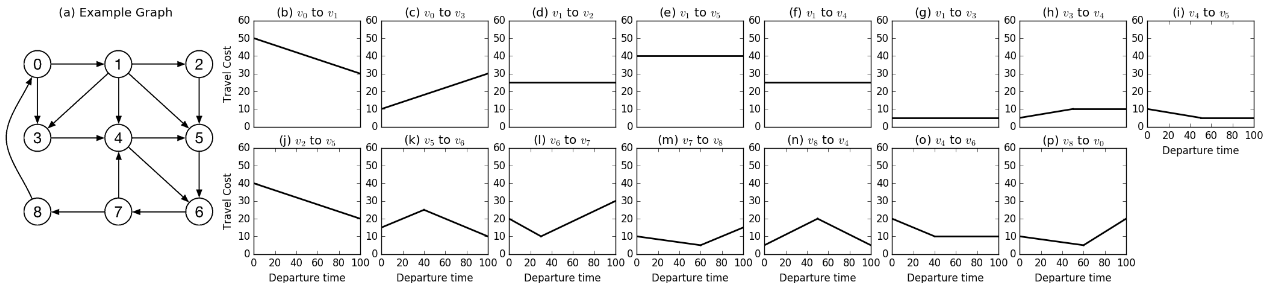


FIGURE 6.1: Example of a Time-Dependent Graph

(a):Directed graph. (b)-(p):Linear piecewise cost functions for each edge, time span is $[0, 100]$

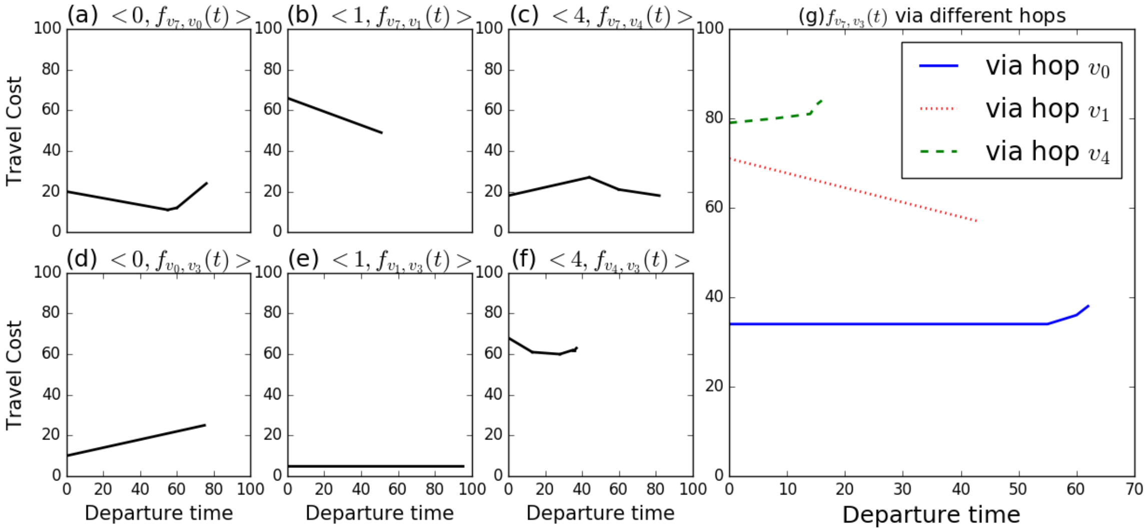


FIGURE 6.2: Time-Dependent 2-Hop Query Example

(a)-(c):Out label functions of v_7. (d)-(f):In label functions of v_3. (g):Result functions via hop v_0, v_1 and v_4.

scenario due to the following challenges: Firstly, achieving the minimum label size ($\Omega(|V||E|^{1/2})$ on the static graph is already *NP-H*, adding another temporal dimension would definitely make it even harder. Moreover, there is no time-dependent index ever exists actually using linear piecewise function. It is already hard enough for online speed-up approaches to stay away from it and only test on histogram-based speed profile. Last but not least, unlike in the static version, where the shortest path between a vertex pair is fixed, the fastest path between a vertex pair changes over time. In another word, there could be a set of hops instead of only one, for the query result. Therefore, answering a query is no longer finding the minimum value, but constructing a result minimum function from a set of functions.

TABLE 6.1: Time Dependent 2 Hop Labels of v_7 and v_3

Node	Out Label	In Label
v_7	$(v_0, f_{v_7,v_0}(t))$	$(v_4, f_{v_4,v_7}(t))$
	$(v_1, f_{v_7,v_1}(t))$	$(v_5, f_{v_5,v_7}(t))$
	$(v_4, f_{v_7,v_4}(t))$	$(v_6, f_{v_5,v_7}(t))$
	$(v_8, f_{v_7,v_8}(t))$	
v_3	$(v_4, f_{v_3,v_4}(t))$	$(v_0, f_{v_0,v_3}(t))$
		$(v_1, f_{v_1,v_3}(t))$
		$(v_4, f_{v_4,v_3}(t))$
		$(v_5, f_{v_5,v_3}(t))$
		$(v_6, f_{v_6,v_3}(t))$

Figure 6.1 shows an example of the time-dependent graph we use in this paper. Figure 6.1-(a) is a directed graph with nine vertices, Figure 6.1-(b)-(p) are its corresponding time-dependent cost functions of the edges. To save space, Table 6.1 only presents the labels of v_7 and v_3 we precomputed. Suppose we are answering the fastest query from v_7 to v_3, we only need to use v_7's out-labels and v_3's in-labels. The intersection of these two label sets is (v_0, v_1, v_4), and the corresponding cost functions are illustrated in Figure 6.2-(a)-(f). Thus, we only need to compute the cost functions from v_7 to v_3 via v_0, v_1 and v_4, and return the minimum of them as the result. The three result functions are shown in Figure 6.2-(g). Since the one that uses v_0 as hop dominates the others over the whole time period, the query result only uses one hop. Also, it is the same as the result returned by the other fastest path algorithms. It should be noted that this might not always be the case. If for some time intervals, v_1's functions is smaller than v_0's, while bigger for the rest, the final result should be the combination of v_0's and v_1's functions.

In this work, we propose a *time-dependent 2-hop labeling* approach to answer the fastest path query efficiently. Depending on the graph size, we present two ways to construct it. Both of them using *pruned landmark approach* to build *2-hop cover*. Such an approach can reduce the construction time significantly [65]. The first one works on small graphs, typically under 5K vertices. We build the time-dependent 2-hop on it directly. The second one works on the large road network, which is often planar. We first split the graph into partitions that each has only a small number of boundary vertices. Then we build time-dependent 2-hop between the boundary vertices over the original road network.

After that, the time-dependent 2-hop within each partition is built with the help of the boundary 2-hop. Furthermore, since we have separate labels for partitions and boundary, we can answer a query in parallel. Finally, we reduce the label size by applying a *piecewise linear approximation* approach, which has a guaranteed error bound. Extensive experiments using real road network and speed profile shows that we can speed up fastest path query answering by hundreds of times. We also re-construct the state-of-art online search approaches time-dependent *CH* and time-dependent *SHARC* using linear piecewise functions, and shows that our approach outperforms them by hundreds of times.

Our contributions are listed as follows:

- We propose a time-dependent 2-hop labeling approach to speed up the fastest path query answering on small graph by hundreds of times.
- We propose a partition-based time-dependent 2-hop labeling to the answer fastest path query on the large road network.
- We apply a piecewise linear approximation approach on the label set to reduce the index size and further speed up the query answering.
- We thoroughly evaluate our approach with extensive experiments on the real-life road network and linear-piecewise-function-based speed profile. Results show than our approach outperforms the linear piecewise version of time-dependent *CH* and *SHARC*.

The rest of this chapter is organized as following: We discuss the related works in Section 6.2. Section 6.3 defines the problem and its corresponding notions. Section 6.4 presents the algorithm of time-dependent 2-hop on small graph, while Section 6.5 extends it on large road network and presents the approximation method. We evaluate our methods in Section 6.6 and summarize the chapter in Section 6.7.

6.2 Related Work

6.2.1 Fastest Path Speed-up Techniques

Just like many speed-up techniques are proposed for answering reachability and shortest distance queries, there are some time-dependent online search speedup techniques, which are extended from their static versions, to support fastest path queries on the time-dependent graph.

The first category of approaches apply simple augmentation on their original static versions. *Bidirectional A^* Search* [102] augments *A^* Search Using Landmarks(ALT)* [103, 80, 104] by applying the landmarks on the *lower bound graph $\underline{G}$*, whose edge weights are the lower bounds of their corresponding time-dependent functions. Although the correctness is guaranteed by using the lower bound, it is a loose approximation of the original time-dependent graph so its effectiveness is weak. [107] extends *Arc-Flag* [108, 109] by counting an edge as important if it appears in a fastest path at least once. Therefore, unlike the original *Arc-Flag*, this approach has to label much more edges because the fastest path changes over time. And it is time-consuming to construct so [107] applies approximation on preprocessing. *Time-dependent contraction hierarchies* [105, 106] expands the original *CH* [66] by adding a shortcut when it is the fastest path for at least once within the whole time period. The second category of approaches combine the previous methods in different ways. *Core-ALT* [110, 111] is the combination of time-dependent *Landmark, bidirectional search* and *CH*, while time-dependent *SHARC* [157, 107] is the combination of time-dependent *CH* and *Arc-Flag*.

It is obvious that all of the above speed-up approaches only apply basic extension to time-dependent scenario by projecting the time dimension to the static graph. In this way, the time-dependent graph is treated as a denser static graph, and those speed-up techniques can be applied directly. Although the query has to adapt to the time-dependent version, the speed-up performance is limited by these static structures since the temporal information is actually discarded. The reason of it is that the number of the interpolation points is much larger than the edge number and the vertex number, which would result in a much longer preprocessing time, although it would speed up the query.

Finally, all of them are online search approaches, which are actually different search space pruning strategies, so they still have to run a search on the time-dependent graph when answering a query.

6.2.2 2-Hop Labeling

The 2-Hop labeling was originally proposed by [6] to answer the shortest distance and reachability query efficiently on the static graph. The basic idea of 2-hop labeling is to pre-compute a set of labels for each vertex and answer the query only by the labels of the involved vertices. For example, on an undirected graph, each vertex v_i keeps a label set $L(v_i) = \{(v_j, d_{v_i,v_j})\}$, which stores its shortest distance to v_j (d_{v_i,v_j}). When answering a shortest distance query from v_s to v_d, we only need to find the vertices $\{v_j\}$ that exist in both of $L(v_s)$ and $L(v_d)$, and selects the smallest $d_{v_s,v_j} + d_{v_j,v_d}$ as the result. By deriving it into a *3-SAT* problem, they proved that finding a 2-hop labeling with the minimal size $\Omega(nm^{1/2}))$ is NP-H. So they provided an approximate algorithm that can return a label size of $\log(n)$ times the optimal lower bound.

From then on, several works are proposed to build the label faster while keeping the size small. However, not too many of them work on the large graph. *Highway-centric labeling* [68] is inspired by the highway structures in the road network and the existing works of the online search approaches like [158]. By selecting a set of vertices as *highway vertices*, and can answer the query between them in constant time, the *2-hop* is converted into three parts: the distance from the source vertex to one of the *entries* in highway, the distance from one *entry* to *exit*, and the distance from the *exit* to the destination. *IS-Label* [159] breaks the graph into a set of *independent sets*, such that the graph can be represented in a hierarchical structure. The distance information is preserved in the process of vertex hierarchy construction. Thus, the vertex order is generated directly from the hierarchical structure. By visiting the vertices in this order, and only visiting the vertices of higher level, we can attach labels to the starting vertex. *Pruned Labeling* [65] mainly focuses on undirected and unweighted graph. By running *BFS* vertices one by one, the labels are created incrementally. During the search, if the distance to the current vertex can be answered by the existing labels, the searching would not expand from it. The label size can be further compressed in bit, which could speed up the query processing and index construction. *Hop Doubling* [62] can only works on unweighted scale-free graph. Under the constraint of scale-free, a vertex's betweenness centrality can be approximated by its degree, which derives an effective vertex order naturally.

6.3 Preliminary

6.3.1 Time-Dependent Road Network

A time-dependent road network is represented as a directed graph $G(V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of directed edges, which has a weight function $w(E, t) \to \mathbb{R}$ mapping edges to time-dependent real-valued weights. The weight of an edge $e(u, v) \in E$ at time t in a time domain $\mathcal{T}$ is $w(u, v, t)$, which represents the amount of time required to reach v starting from u at time t. $w(u, v, t) = \infty$ if there is no edge from u to v.

A path p from u to v in G is represented as $p =< v_0, v_1, \ldots, v_k >$, where $v_0 = u$, $v_k = v$ and $(v_{i-1}, v_i) \in E$, $\forall 1 \leq i \leq k$. Let $\alpha(v_i)$ be the arrival time of v_i, and $\beta(v_i)$ be the departure time of v_i. The time-dependent cost of p is the sum of the weights on edges at each v_i's departure time: $w(p) = \Sigma_{i=1}^{k} w(w_{i-1}, w_i, \beta(v_{i-1}))$. The single starting time fastest path problem on time-dependent road network is defined below.

Definition 6.1. *(Single Starting-Time Fastest Path Problem (SSFP)). Given a time-dependent graph $G(V, E)$ and a query $Q(v_s, v_d, t)$, where t is the departure time, SSFP is to find a path with minimal $w(p, t) = \Sigma_{i=1}^{k} w(w_{i-1}, w_i, \beta(v_{i-1}))$, where $\beta(v_s) = t, v_0 = v_s$ and $v_k = v_d$.*

The road network naturally follows *FIFO* property: $\forall t_1 \leq t_2, t_1 + w(u, v, t_1) \leq t_2 + w(u, v, t_2)$, which means no overtaking in reality. Such a property ensures waiting on a vertex would increase $w(p)$. In another word, $\beta(v_i) = \alpha(v_i)$ is essential to minimize $w(p)$.

If we extend the departure time $\beta(v_s)$ to the whole time span, we can get a minimum cost function from v_s to v_d $f_{v_s,v_d}(t)$ that records the least travel time at different departure time.

Definition 6.2. *(Fastest Path Profile Problem (FPP)). Given a time-dependent graph $G(V, E)$ and a query $Q(v_s, v_d)$, FPP is to compute a minimum cost function $f_{v_s,v_d}(t)$ such that $f_{v_s,v_d}(\beta(v_s)) = min(w(p, \beta(v_s)))$, $\forall \beta(v_s) \in \mathcal{T}$.*

Problem Definition: We study the following problem: given a time-dependent graph $G(V, E)$, construct an index L, for processing fastest path profile problem that given any pair of (v_s, v_d), return $f_{v_s,v_d}(t)$ only using L.

6.3.2 Time-Dependent 2-Hop Cover

For each vertex $v \in V$, we pre-compute two sets of labels: out-labels $L_{out}(v_i) = \{(v_j, f_{v_i,v_j}(t))\}$ and in-labels $L_{in}(v_i) = \{(v_j, f_{v_j,v_i}(t)\}$, where v_j is a hop vertex and $f_{v_i,v_j}(t)$ returns the minimal cost from v_i to $_j$ at different departure time t. We use $L = \{(L_{out}(v_i), L_{in}(v_i)) | \forall v_i \in V\}$ to denote the set of all the labels. If L can answer all the queries on G, then we say L is a time-dependent 2-hop cover.

A minimum cost function query $Q_f(v_s, v_d, L)$ returns $f_{v_s,v_d}(t)$ only using the labels, as shown

TABLE 6.2: Time-Dependent 2-Hop Labeling Important Notations

Notation	Description
$f_{v_s,v_d}(t)$	Minimum cost function from v_s to v_d
$f_{v_s,v_d,v_i}(t)$	Minimum cost function from v_s to v_d via hop v_i
$V_{s,d}$	Set of vertices along paths from v_s to v_d
$\oplus$	Concatenation of two minimum cost functions
$Min()$	Min function of a set of minimum cost functions
$L_{out}(v_i) = \{v_j, f_{v_i,v_j}(t)\}$	Out-label of vertex v_i
$L_{in}(v_i) = \{v_j, f_{v_j,v_i}(t)\}$	In-label of vertex v_i
$Q(v_s, v_d)$	Fastest path profile query from v_s to v_d
$Q_f(v_s, v_d, L)$	Minimum cost function query using hop cover L
H_{v_s,v_d}	Hop vertex set $L_{out}(v_s) \cap L_{in}(v_d)$
G_i	A subgraph of G
$\hat{L}^{G_i}$	Labels of subgraph G_i
$\hat{L}^B$	Labels of boundary vertices

below:

$$\begin{aligned} Q_f(v_s, v_d, L)) &= Min(f_{v_s,v_d,v_i}(t)) \\ &= Min(f_{v_s,v_i}(t) \oplus f_{v_i,v_d}(t)) \\ &= Min(f_{v_i,v_d}(f_{v_s,v_i}(t))) \\ &= f_{v_s,v_d}(t), t \in \mathcal{T} \\ \forall v_i \in H_{v_s,v_d} &= L_{out}(v_s) \cap L_{in}(v_d) \end{aligned}$$

If v_i exists in both v_s's out-label set and v_d's in-label set, v_i is a hop vertex of this query. For all the hop vertices $v_i \in H_{v_s,v_d}$, we compute the minimum cost functions from v_s to v_d via v_i: $\{f_{v_s,v_d,v_i}(t)\}$ by the concatenation operation $\oplus$ of two time-dependent functions, which is computed as $f_{v_s,v_d,v_i}(t) = f_{v_s,v_i}(t) \oplus f_{v_i,v_d}(t) = f_{v_i,v_d}(f_{v_s,i}(t) + t)$. $Min()$ is a function that takes all the $\{f_{v_s,v_d,v_i}(t)\}$ as input, and combines the smallest function pieces at each sub-time interval as the minimum cost function $f_{v_s,v_d}(t)$. An example is shown in Figure 6.3. As for the example of Figure 6.2, the input functions are $\{f_{7,3,0}(t), f_{7,1,0}(t), f_{7,4,0}(t)\}$. Because $f_{7,3,0}(t)$ is always smaller than the others, $f_{7,3}(t) = Min(\{f_{7,3,i}(t)\}) = f_{7,3,0}(t)$.

Within each minimum cost function piece $f_{v_s,v_d,v_i}(t)$, we use $V_{s,d,i}$ to denote the set of vertices

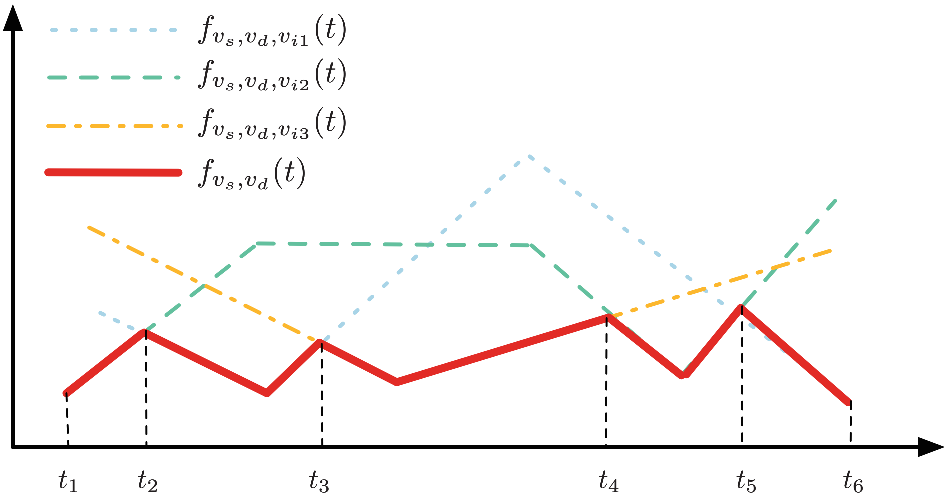


FIGURE 6.3: Example of $Min()$ function

Hop vertex set $H_{v_s,v_d} = \{v_{i1}, v_{i2}, v_{i3}\}$. Three dashed lines are the inputs, the solid red line is the output $f_{v_s,v_d}(t)$. Different parts of it come from different input lines: (1)$f_{v_s,v_d}(t) = f_{v_s,v_d,v_{i1}}(t)$ for $t \in [t_2, t_3] \cup [t_5, t_6]$; (2)$f_{v_s,v_d}(t) = f_{v_s,v_d,v_{i2}}(t)$ for $t \in [t_1, t_2] \cup [t_4, t_5]$; (3)$f_{v_s,v_d}(t) = f_{v_s,v_d,v_{i3}}(t)$ for $t \in [t_3, t_4]$.

along the fastest paths from v_s to v_d via v_i. The number of hops in the 2-hop cover L is $|L| = \Sigma_{i=1}^{|V|}(|L_{in}(v_i)| + |L_{out}(v_i)|)$.

In the following two sections, we present the construction and query of time-dependent 2-hop labels on the small graph and large road network, respectively.

6.4 Time-Dependent 2-Hop Labeling on Small Graph

6.4.1 Index Construction

In order to help understand our approach to construct a time-dependent 2-hop labeling, we first present a naive one. Suppose the vertices are ordered in $V = \{v_1, v_2, \ldots, v_n\}$ and we visit them in this order. Intuitively, we run a single source fastest path search from each $v_i \in V$ to get all the minimum cost functions $f_{v_i,v_j}(t)$ from v_i to all the other vertices $v_j \in V - \{v_i\}$, and run another reverse backward fastest path search to get all the minimum cost functions $f_{v_j,v_i}(t)$ from all the other vertices v_j to v_i. We use $L^i = \{(L^i_{out}(v_j), L^i_{in}(v_j)) | \forall v_j \in V\}$ to denote the label set we get after the searchings from v_i. Initially, $L^0 = \phi$. Suppose after the searchings from v_{k-1}, we obtain a label set $L^{k-1} =$

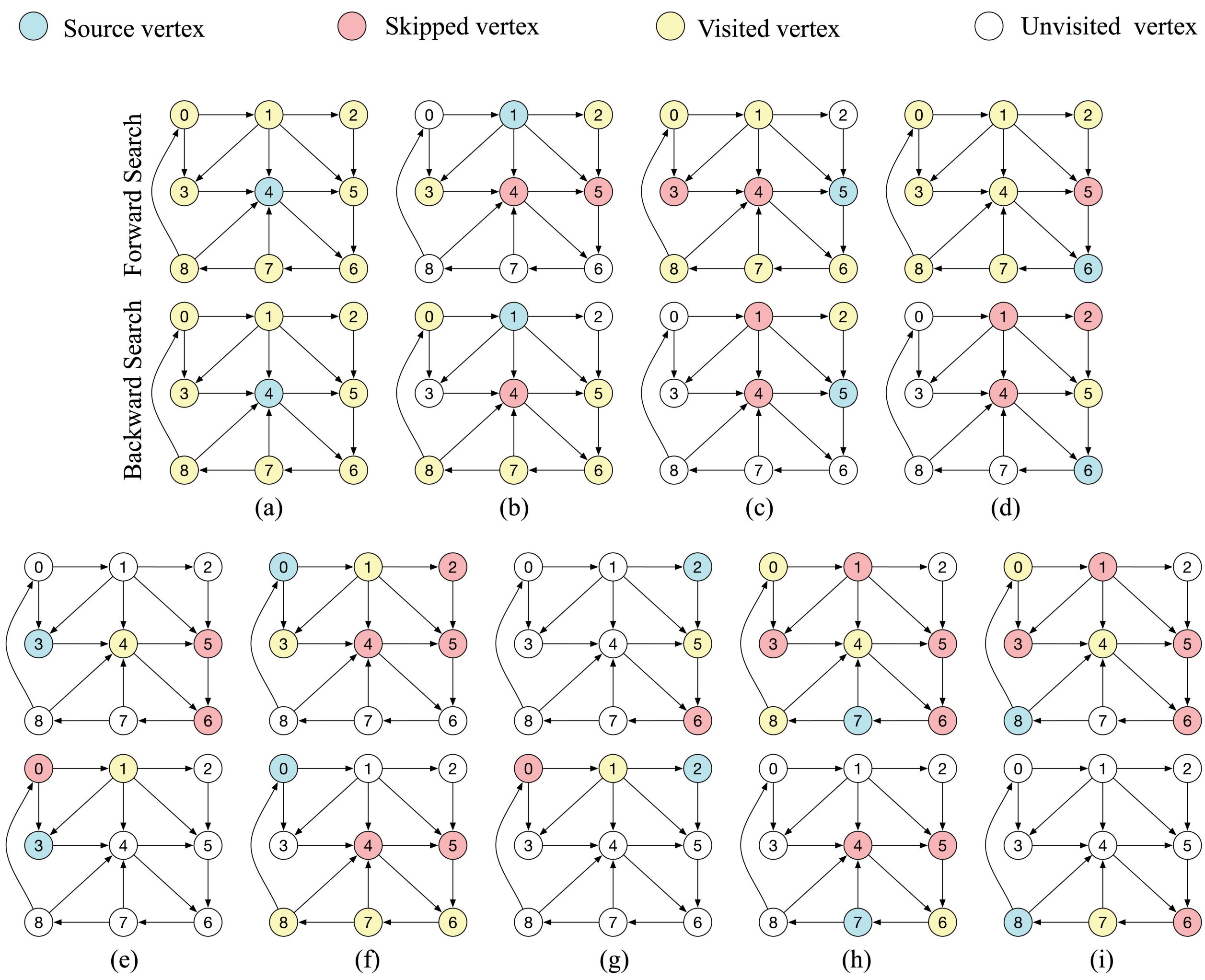


FIGURE 6.4: Continuous Time Dependent 2-Hop Labeling Construction Example

$\{(L_{out}^{k-1}(v_j), L_{in}^{k-1}(v_j))|\forall v_j \in V\}$. Then we run a fastest path search from v_k, and get all the minimum cost functions from v_k: $\{f_{v_k,v_j}(t)\}$. Therefore, we update the in-label sets $L_{in}^{k}(v_j) = L_{in}^{k-1}(v_j) \cup (v_k, f_{v_k,v_j}(t)), \forall v_j \neq v_k$. Meanwhile, we run a reverse backward fastest path search from v_k and get all the minimum cost functions from v_j to v_k: $\{f_{v_j,v_k}(t)\}$. Correspondingly, we update the out-label sets $L_{out}^{k}(v_j) = L_{out}^{k-1}(v_j) \cup (v_k, f_{v_j,v_k}(t)), \forall v_j \neq v_k$. This procedure continues until the last vertex v_n finishes the searchings.

Obviously, the result of this approach is exactly the same as *all-pair pre-computation*, so it has generated a 2-hop cover that can answer query correctly. However, it is inefficient in both construction time and label size. Therefore, we apply a pruning approach to reduce the search space, which could further reduce the construction time and $Size(L)$. We use $\hat{L}^i$ to denote the result after the i^{th} pruning search on v_i. Suppose we just finish visiting v_{k-1} and get a label set of $\hat{L}^{k-1} = \{(\hat{L}^{k-1}_{out}(v_j), \hat{L}^{k-1}_{in}(v_j))|\forall v_j \in V\}$. Then we start a fastest path searching from v_k. During the search, each time when we settle a vertex v_i, we can obtain a minimum cost function $f_{v_k,v_i}(t)$. At the same time, we also run a minimum cost function query $Q_f(v_k, v_i, \hat{L}^{k-1})$ to obtain the intermediate minimum cost function $f^{k-1}_{v_k,v_i}(t) = Min(\{f_{v_k,v_j}(t) \oplus f_{v_j,v_i}(t)|v_j \in \hat{L}^{k-1}_{out}(v_k) \cap \hat{L}^{k-1}_{in}(v_i)\})$. If $f_{v_k,v_i}(t)$ is dominated by query result $f^{k-1}_{v_k,v_i}(t)$, i.e., $f_{v_k,v_i}(t) \geq f^{k-1}_{v_k,v_i}(t)$, $\forall t \in \mathcal{T}$. Then we prune the search from v_i, which means we do not add $(v_i, f_{v_k,v_i}(t))$ to $\hat{L^k}$ and we also do not visit the edges from v_i. The same pruning strategy also applies on the reverse backward fastest path search. The detailed is illustrated in Algorithm 10.

Algorithm 10: Time-Dependent 2-Hop Construction

```
Input: $G(V, E)$
Output: $L$
begin
    for $i = 1, \ldots, |V|$ do
        //Forward search, create in-labels $FM$ stores all temporay labels $FM[v_d] = f_{v_i,v_d}(t)$
        //$FQ$ is a priority queue, key is $f_{v_i,v_j}t.min$
        $FQ.push(v_i)$
        while $!FQ.empty()$ do
            $v_j \leftarrow FQ.pop()$
            if $Q_f(v_i, v_j, \hat{L}^{i-1})$ domintes $MF[v_j]$ then
                continue
            for $v_k \leftarrow v_j$'s out-neighbor do
                $f_{v_i,v_k,v_j}(t) \leftarrow f_{v_i,v_j}(t) \oplus f_{v_j,v_k}(t)$
                $FM[v_k] = Min(FM[v_k], f_{v_i,v_k,v_j}(t))$
                $FQ.push/update(v_k)$
        for $v_d \in FM$ and $v_d$ was not skipped do
            $\hat{L}^i_{in}(v_d) = \hat{L}^{i-1}_{in} \cup (v_i, FM[v_d])$
        //Backward search, create out-labels $BM$ stores all temporay labels:
          $BM[v_d] = f_{v_d,v_i}(t)$
        //$BQ$ is a priority queue, key is $f_{v_i,v_j}t.min$
        $BQ.push(v_i)$
        while $!BQ.empty()$ do
            $v_j \leftarrow BQ.pop()$
            if $Q_f(v_j, v_i, \hat{L}^{i-1})$ domintes $BM[v_j]$ then
                continue
            for $v_k \leftarrow v_j$'s in-neighbor do
                $f_{v_k,v_i,v_j}(t) \leftarrow f_{v_k,v_j}(t) \oplus f_{v_j,v_i}(t)$
                $BM[v_k] = Min(BM[v_k], f_{v_k,v_i,v_j}(t))$
                $BQ.push/update(v_k)$
        for $v_d \in BM$ and $v_d$ was not skipped do
            $\hat{L}^i_{out}(v_d) = L^{i-1}_{out} \cup (v_i, BM[v_d])$
    return $\hat{L}^{|V|}$
```

Figure 6.4 demonstrates an example of the index construction using the time dependent graph of Figure 6.1. We visit the vertices in order $< v_4, v_1, v_5, v_6, v_3, v_0, v_2, v_7, v_8 >$ (from Figure 6.4-(a) to Figure 6.4-(i)). The first line shows the behavior of pruned forward search while the second line shows the pruned backward search. We use blue vertex to denote the vertex we start the search from. Red vertex denotes the vertices that are dominated by the existing time-dependent 2-hop labels. We do not update the red vertex's neighbors. The yellow ones are the vertices that we visit during the search, and the white vertices are not visited in the current search. For example, Figure 6.4-(e) demonstrates the fifth searches starting from v_3. The forward search only visits v_4 and stops on v_5 and v_6 since their minimum cost functions are dominated by query results $f^4_{v_3,v_5}(t) = Q_f(v_3, v_5, \hat{L}^4)$ and $f^4_{v_3,v_6}(t) = Q_f(v_3, v_6, \hat{L}^4)$, respectively. $f^4_{v_3,v_5}(t)$ is created by $(v_4, f_{v_3,v_4}(t))) \in L_{out}(v_3)$ and $(v_4, f_{v_4,v_5}(t)) \in L_{in}(v_5)$ via v_4, which are introduced by the first searches from v_4 as shown in Figure 6.4-(a). $f^4_{v_3,v_6}(t)$ is also created by hop v_4. The backward search from v_3 only visits v_1 and pruned v_0 because $f_{v_0,v_3}(t)$ is dominated by $f^4(v_0, v_3)(t)$, which is created by hop v_1 in $L_{out}(v_0)$ and $L_{in}(v_3)$ introduced in the second searches, as shown in Figure 6.4-(b). It is obvious that the searching spaces of latter searches are smaller than the earlier ones.

6.4.2 Query

A straightforward way to answer a minimum cost function query $Q_f(v_s, v_d, \hat{L})$ is checking the hops H_{v_s,v_d} one by one. Firstly, we use a hash table to retrieve the common hop vertex set $H_{v_s,v_d} = \hat{L}_{out}(v_s) \cap \hat{L}_{in}(v_d)$. This process takes $O(min(\hat{L}_{out}(v_s), \hat{L}_{in}(v_d)))$ time. If v_s exists in $\hat{L}_{in}(v_d)$ or v_d exists in $\hat{L}_{out}(v_s)$, we can answer the query by using the hop function directly. Otherwise, we construct cost functions $f_{v_s,v_d,v_i}(t)$ using the concatenation operation $f_{v_s,v_i}(t) \oplus f_{v_i,v_d}(t) = f_{v_i,v_d}(f_{v_s,v_i}(t) + t)$ on each $v_i \in H_{v_s,v_d}$. The final result $f_{v_s,v_d}(t)$ is generated by $Min(\{f_{v_s,v_d,v_i}(t)\})$. So the query process takes $O(|T| * |H_{v_s,v_d}|)$ time, where $|T|$ is the average number of linear piecewise functions of $f_{v_s,v_d}(t)$.

However, the size of $|H_{v_s,v_d}|$ could be large when the graph contains more vertices and $\hat{L}$ grows bigger, which not only slows down the query answering, but also prolongs the index construction time. Therefore, we propose a pruning strategy to reduce the actual concatenation and minimization times. First of all, along with the linear piecewise function's turning points, we also store the minimum value $f_{v_s,v_d}(t).min$ and maximum values $f_{v_s,v_d}(t).max$. Then we organize the hop vertices in a heap, by using the estimated lower bound $f_{v_s,v_i}(t).min + f_{v_i,v_d}(t).min$ as key. At this stage, no concatenation or minimization has conducted. After that, we visit the hops in heap one by one. Each time we visit a hop v_i, the temporary result $f'_{v_s,v_d}(t)$ is updated (by Min). For a normal query, if $f_{v_s,v_i}(t).min + f_{v_i,v_d}(t).min > f'_{v_s,v_d}(t).max$, we are safe to stop and use the current $f'_{v_s,v_d}(t)$ as the query result. For a query during index construction, we compare the temporary result $f^k_{v_s,v_d}(t)$ with the direct searching result $f_{v_s,v_d}(t)$. If the current $f^k_{v_s,v_d}(t)$ already is able to dominate $f_{v_s,v_d}(t)$, we stop traversing the heap and avoid searching from v_d.

6.4.3 Correctness and Minimality

In this part, we prove $\hat{L}$ is a *time-dependent 2-hop cover*. Firstly, the following lemma gives an instance of correct time-dependent 2-hop cover.

Lemma 6.1. *L^n is a time-dependent 2-hop cover.*

Proof. Since L^n is an all pair fastest path index, then $\forall v_s, v_d \in V$, both of $(v_d, f_{v_s,v_d}) \in L^n_{out}(v_s)$ and $(v_s, f_{v_s,v_d}) \in L^n_{in}(v_d)$ hold. Therefore, $L^n_{out}(v_s) \cap L^n_{in}(v_d)$ contains at least $\{(v_s, f_{v_s,v_d}), (v_d, f_{v_s,v_d})\}$, and their functions are the same. Thus, $Q_f(v_s, v_d) = Q_f(v_s, v_d, L^n) = f_{v_s,v_d}(t)$. So L^n is a time-dependent 2-hop cover. □

Then, to prove $\hat{L}$ is a time-dependent 2-hop cover, we only need to prove $Q_f(v_s, v_d, \hat{L}^n) = Q_f(v_s, v_d, L^n), \forall v_s, v_d \in V$. In fact, we prove it by showing the both label sets created in iteration k has the same cover capability in Theorem 6.2.

Theorem 6.2. $Q_f(v_s, v_d, \hat{L}^k) = Q_f(v_s, v_d, L^k), \forall k \in [0, n]$.

Proof. We prove it by math induction on the search iteration number i. Initially, $i = 0 \Rightarrow L^0 = \hat{L}^0 = \phi$. Then suppose $Q_f(v_s, v_d, \hat{L}^i) = Q_f(v_s, v_d, L^i)$ for $0 \leq i \leq k-1$. Now we prove it still holds for $i = k$.

First of all, suppose the minimum cost function query $Q_f(v_s, v_d, L^{k-1})$ that using label set L^{k-1} can return a result $f^{k-1}_{v_s,v_d}(t)$. Otherwise, just simply ignore the query and use the search result as the hop labels. Thus, we need to prove the result of $Q_f(v_s, v_d, \hat{L}^{k-1})$, $\hat{f}^{k-1}_{v_s,v_d}(t)$, is exactly same as $f^{k-1}_{v_s,v_d}(t)$. Secondly, as mentioned in section 6.3.2 which describes $Min()$ function, $f^{k-1}_{v_s,v_d}(t)$ is made up of a set of functions from different vertices in H_{v_s,v_d}. Thus, we only need to prove one of the functions $f_{v_s,v_d,v_h}(t)$, which is created by hop vertex v_h, can also be created by query using $\hat{L}^{k-1}$. In this way, we can prove each part of $\hat{f}^{k-1}_{v_s,v_d}(t)$ equals to the corresponding part of $f^{k-1}_{v_s,v_d}(t)$, which leads to $\hat{f}^{k-1}_{v_s,v_d}(t) = f^{k-1}_{v_s,v_d}(t)$.

Although the paths of minimum cost function change over time, we can still break it into pieces based on the time interval such that within each piece, the vertices along each fast path is fixed. As simple example was illustrated in Figure 4. For any piece of $f^{k-1}_{v_s,v_d}(t)$, suppose h is the smallest vertex order (or earliest search iteration) that creates that piece. Otherwise, we use a smaller one to replace v_h for proof. Obviously, $(v_h, f_{v_s,v_h}(t)) \in L^{k-1}_{out}(v_s)$ and $(v_h, f_{v_h,v_d}(t)) \in L_{in}(v_d)^{k-1}$. All we need to prove is that both of $(v_h, f_{v_s,v_h}(t))$ and $(v_h, f_{v_h,v_d}(t))$ also exist in the pruned sets $\hat{L}^{k-1}_{out}(v_s)$ and $\hat{L}^{k-1}_{in}(v_d)$, respectively.

We prove $(v_h, f_{v_s,v_h}(t)) \in \hat{L}^{k-1}_{out}(v_s)$ first. In fact, we prove $\forall v_x \in V_{v_s,v_h}, (v_h, f_{v_x,v_h}(t)) \in \hat{L}^{k-1}_{out}(v_x)$, where V_{v_s,v_h} is the set of vertices along the path from v_s to v_h during that piece of time interval. In anther word, we prove labels $(v_h, f_{v_x,v_h}(t))$ are added to $\hat{L}_{out}(v_x)$ at the h^{th} iteration. Firstly, these labels that contain v_h can only be inserted when we run the h^{th} search from v_h. Secondly, $V_{v_x,v_h} \subseteq V_{v_s,v_h}$ obviously, because v_s is the source vertex along the path v_s to v_h while v_x is an inner vertex. Since v_h is the smallest hop vertex that results in $f^{k-1}_{v_s,v_d,v_h}(t)$, then $\forall i < h$, hop vertex $v_i \notin V_{v_x,v_h}$. In another word, no hop vertex exists along the path v_s to v_h, so label set $\hat{L}^{h-1}$ cannot return a result same as $f_{v_x,v_h}(t)$. Therefore, the result of $Q_f(v_x, v_h, \hat{L}^{h-1})$ cannot dominate $f_{v_x,v_h}(t)$, so $(v_h, f_{v_x,v_h}(t))$ is added to $\hat{L}^{h-1}_{out}(v_x)$ during the h^{th} backward search. Thus, $(v_h, f_{v_s,v_h}(t))$ exists in $\hat{L}^{k-1}_{out}(v_s)$. As for $(v_h, f_{v_h,v_d}(t)) \in \hat{L}^{k-1}_{in}(v_d)$, it can be proved symmetrically. □

So when $k = n$, we can safely draw the conclusion that $Q_f(v_s, v_d, \hat{L}^n) = Q_f(v_s, v_d, L^n)$, which means the result of the pruned approach is the same as all pair approach.

Corollary 6.3. *The label set $\hat{L}$ constructed from Section 6.4.1 is a time-dependent 2-hop cover.*

Theorem 6.4. *If we remove any hop v_h from any vertex v_i's $\hat{L}_{out}(v_i)$ or $\hat{L}_{in}(v_i)$, then $\hat{L}$ is not a time-dependent 2-hop cover.*

Proof. Suppose we remove $(v_h, f_{v_s,v_h}(t))$ from $\hat{L}_{out}(v_s)$. This would directly cause $f_{v_s,v_h}(t) \neq Q_f(v_s, v_h, \hat{L})$. First of all, $\forall i < h$, $Q_f(v_s, v_h, \hat{L}^i)$ cannot dominate $f_{v_s,v_h}(t)$, otherwise $(v_h, f_{v_s,v_h}(t)$ would not be added to $\hat{L}_{out}(s)$ during the h^{th} iteration. Secondly, because v_h was in v_s's out-label, v_h appeared earlier during the construction. Therefore, v_s would not be added in v_h's in-label set during the s^{th} iteration since it was dominated by $Q_f(v_s, v_h, \hat{L}^{s-1})$ and $(v_h, f_{v_s,v_h}(t)) \in \hat{L}^{s-1}_{out}(v_s)$. Thus, the incomplete $\hat{L}$ is not a time-dependent 2-hop cover. □

6.4.4 Complexity Analysis

As proved in [6], the static graph's 2-hop label size is bounded by $\Omega(|V||E|^{1/2})$ and $O(|V|^2)$. A direct extension by time-dependent information would result in a loose lower bound $\omega(T|V||E|^{1/2})$, where T is the number of turning points in the minimum cost function. This is because when we construct a minimum cost function from the existing label sets, it has to completely dominate $f_{v_s,v_d}(t)$ at all time points. If any piece of the function cannot be dominated, we cannot prune the search and also have to add $f_{v_s,v_d}(t)$ to the label set. As for the upper bound, it is $O(T|V|^2)$.

Next, we analyze the querying answering time. The average label number of each vertex is $|L|/2|V|$. During the query time, we check the common hop vertices of v_s and v_d. Obviously, $|H_{v_s,v_d}| = O(min(|L|/|V|))$. For each hop vertex, it takes $O(T)$ time to concatenate and minimize. Therefore, the worst case is $O(T|L|/|V|)$.

Finally, we analyze the index construction time. The iteration runs $|V|$ times. Within each iteration, there are two fastest path searches. Each of them take $O(T|V|\log|V| + T^2|E|)$ time as proved in [24]. Therefore, the construction takes $O(T|V|^2\log|V| + T^2|V||E|))$ time in the worst case.

6.5 Partition-based Time-Dependent 2-Hop on Road Network

As analyzed in Section 6.4.4, both the construction time and index size grow fast as the graph becomes larger. It could work well on a graph with thousands of vertices, but cannot scale to an ordinary city road network with hundreds of thousands of vertices. Therefore, we propose a partition-based structure to extend the time-dependent 2-hop. Section 6.5.1 describes the partitions and overall structure; Section 6.5.2 explains the index construction of the boundaries vertices of the partitions; Section 6.5.3 explains how to construct the index inside of each partition; Section 6.5.4 elaborates on query processing; Section 6.5.5 proves the correctness and 6.5.6 analyzes the complexity. We provide an approximation method to reduce the space complexity with a guaranteed error bound in Section 6.5.7. Finally, we provide the query extension on any time interval in Section 6.5.8

6.5.1 Graph Partition

Given an input graph G, we partition it into a set $C = \{G_1, G_2, \dots, G_{|C|}\}$ of edge-disjoint subgraphs of G, such that $\bigcup_{i\in[1,|C|]} G_i = G$. Each subgraph contains thousands of vertices. If a vertex appears in more than one subgraphs in C, then it is a *boundary vertex*. We use the *Natural Cut* method [160] to partition our road network. Such a approach can create only a small number of boundary vertices.

Based on if a vertex is a boundary vertex or not, we construct the time-dependent 2-hop differently. For all the boundary vertices, we create a time-dependent 2-hop for them, where all the hops are boundary vertices, and the path information is still computed from the original graph. For all those vertices that belong to the same subgraph, we construct a time-dependent 2-hop for each subgraph, which also includes the boundary vertices it contains. When answering a query, we use three sets of labels to compute the result: labels from v_s's subgraph $\hat{L}^{G_s}$, labels from v_d's subgraph $\hat{L}^{G_d}$ and labels of the boundary vertices $\hat{L}^B$.

6.5.2 Boundary 2-Hop Construction

This subsection describes the construction of boundary time-dependent 2-Hop. Since the boundary vertices separate the vertices from different subgraphs, all the paths that involve different subgraphs have to pass through them. Therefore, the boundary vertices are naturally hop vertices. Although the number of the boundary vertices is on the same level of the subgraphs', its space consumption is larger. This is because the paths between boundary vertices are longer than the paths within each subgraph, and longer paths have more turning points in their minimum cost functions.

One straightforward approach is to construct the labels directly like what we described in Section 6.4 on the small graph. However, it would be much slower than running on the small graph because the fastest path searches actually run on the entire large graph. And the space consumption is also too huge to tolerate. Moreover, searches of different iterations can only run one by one rather than in parallel, which further prolongs the construction time.

Our boundary time-dependent 2-hop construction actually runs in 2 phases: *parallel all-pair construction phase* and *pruning phase*. During the first phase, we take advantage of the modern multi-thread computing and compute the all-pair fastest paths between all the boundary vertices in parallel. The searches still origins from each boundary vertex. The result of each fastest path search is written down to disk right after it finishes. When all the searches finish, we have cached all the searching results of each boundary vertex. Then in the *pruning phase*, we load these searching results in a pre-defined order and prune them in a similar way as described in Section 6.4.1. We use $\hat{L}_i^B$ to denote the result after processing the i^{th} boundary vertex. Suppose we just finish visiting v_{k-1}^B and get a label set of $\hat{L}_{k-1}^B$. Then we load results of v_k and test each fastest path $f_{v_k,v_d}(t)$ (or $f_{v_d,v_k}(t)$). If it is dominated by the result of query $Q_f(v_k, v_d, \hat{L}^B_{k-1})$ (or $Q_f(v_d, v_k, \hat{L}^B_{k-1})$), we can safely drop it. Otherwise, it is added to $\hat{L}^B_k$. After the last result is processed, we get a time-dependent 2-hop cover on boundary vertices $\hat{L}^B$.

6.5.3 Inner 2-Hop Construction

One may think the construction of inner 2-hop is the same as Section 6.4.1 at first glance. However, although the vertex number is the same and the vertices used as hop are also the same, they still have a little difference.

Unlike a unique small graph, which computes over its own vertices and edges, a subgraph G_i still has connections to other subgraphs in G. Then a fast path may not only traverse the vertices in G_i, but also traverses vertices in other subgraphs. In another word, the fastest path on the original graph G between two vertices in the same subgraph cannot be answered only by the vertices within that subgraph.

One straightforward approach is running the construction on G directly and stops searching when all the vertices in that subgraph are visited. However, it involves other vertices in searching, which increases the number of vertices involved. Moreover, the searching time is much longer for those vertices near the boundary of the subgraph.

To avoid stepping into other subgraphs, we need to make full use of boundary vertices. Suppose v_s and v_d are in subgraph G_i, and their fastest path needs to visit another subgraph G_j. Obviously, there are at least two boundary vertices exist along that path (one from G_i to G_j, another from G_j to G_i). Thanks to the previous subsection, we can take advantages of the existing boundary time-dependent 2-hop. When the search from v_s reaches a boundary vertex v_b, we use labels to compute all the minimum cost functions from v_b to all the other boundaries vertices in the G_i. Then we concatenate $f_{v_s,v_b}(t)$ to these minimum cost functions and get minimum cost functions from v_s to all the boundary vertices in G_i via v_b. In this way, all the searching related to G_j are settled by fast query answering.

Since the constructions within each subgraph are unrelated to each other, we can also construct them in parallel, with each thread is responsible for the time-dependent 2-hop labels of one subgraph.

6.5.4 Query

The query answering has two situations. If the v_s and v_d are in the same subgraph, we can use the inner 2-hop to answer the query directly. If they are from different subgraphs G_i and G_j, we first break the query answering in three parts: v_s to its boundary vertices G_i^B, G_i^B to G_j^B, and G_j^B to v_d. Then we concatenate the results and construct the final result.

The query from v_s to G_i^B is made up of queries from v_s to each boundary vertex in G_i^B. These queries can run in parallel, so the actual query running time is the same as single inner query's. The result set is denoted as $F_{v_d,G_i^B}(t)$. The result set of the query from G_j^B to v_d is denoted as $F_{G_j^B,v_d}(t)$. Its construction procedure is symmetric to the previous one, and we do not discuss it. The query between G_i^B and G_j^B can also run in parallel, and we denote its result set as $F_{G_i^B,G_j^B}(t)$.

After obtaining three result sets, we first sort the functions in $F_{G_i^B,G_j^B}(t)$ based on their minimum values. Then we retrieve the boundary function in that order. This is based on the observation that those boundary functions that is made up of two boundary vertices from opposite directions of the search would result in much longer results and are useless, and we try to avoid visiting them. Each time we retrieve a $f_{v_i^B,v_j^B}(t)$, we concatenate it with two inner results: $f_{v_s,v_i^B}(t) \oplus f_{v_i^B,v_j^B}(t) \oplus f_{v_j^B,v_d}(t)$. Then use this result to update the current existing $f_{v_s,v_d}(t)$. For the next boundary function, if its minimum value is larger than the $f_{v_s,v_d}(t).min$ and can be dominated by $f_{v_s,v_d}(t)$, we skip it. If its minimum value is larger than $f_{v_s,v_d}(t).max$, we can drop it directly. After no function can update $f_{v_s,v_d}(t)$, we return it as the query result.

6.5.5 Correctness

The correctness of the inner and boundary time-dependent 2-hops are the same as 6.4.3 with simple variations. Because these results are correct, and the assembling of them to get the query result is nothing but time-dependent function expansion used in the fastest path search, the query result of vertices from different subgraphs is also correct.

6.5.6 Complexity Analysis

Suppose the graph is partitioned into $|C|$ subgraphs, and these subgraphs have $|B|$ boundary vertices. Then each subgraph has an average number of $2|B|/|C|$ boundary vertices because each boundary vertex is shared by two subgraphs. Each subgraph has $|V|/|C|$ vertices and $|E|/|C|$ edges. The size of inner 2-hop of each subgraph is $\omega(T\frac{|V|}{|C|}(\frac{|E|}{|C|})^{1/2})$. The total size of the inner 2-hop on the whole graph is $\omega(T|C|\frac{|V|}{|C|}(\frac{|E|}{|C|})^{1/2}) = \omega(T|V|(\frac{|E|}{|C|})^{1/2})$. We derive the boundary 2-hop from the all pair result, which is actually a complete graph that has $|B|$ vertices and $|B|^2$ edges. Although we use pruning strategy to reduce its size in practice, the theoretic size is still $\theta(|T|B|^2|)$. Therefore, the overall space complexity is $\omega(T(\frac{|V|}{|C|}(\frac{|E|}{|C|})^{1/2} + |B|^2))$.

The total inner 2-hop construction time is same as that of each individual: $O((T|V|^2\log(\frac{|V|}{|C|} + T^2|V||E|)/|C|^2)$, because they are constructed in parallel. The boundary 2-hop construction time is made up of parallel all-pair generation and pruning: $O((T|V|\log|V| + T^2|E|) + |B|^2T)$.

The query time of inner subgraph query is $O(T|L_C||V|/|C|)$, where $|L_C|$ is label size of the subgraph. The query time of the boundary query is $O(T|L_B| \times (2|B|/|C|)^2) = O(\frac{T|B|^2|L_B|}{|C|^2})$. The concatenation and update operation takes $O(\frac{T|L_B|^2}{|C|^2})$ time. So the overall query time is the sum of inner subgraph query time, boundary query time and update time.

6.5.7 Approximation

Although the partition approach can reduce the label size of the direct time-dependent 2-hop, it is still large for many users. Therefore, we provide a *Piecewise Linear Approximation* method to reduce the label size, while the error bound is guaranteed.

As observed in the actual label sets, we find many of the turning points in the label functions have similar values among nearby time points. If we remove some of them while preserving the important ones, we can have a cost function with fewer turning points, which would result in smaller index size.

The detail of the approximation is shown in Algorithm 5. It takes a minimum cost function $f_{v_s,v_d}(t)$ and an error bound as input, and returns the approximated function $\hat{f}_{v_s,v_d}(t)$. The algorithm runs in iterations. With each iteration, it removes the turning points that would introduce the smallest error. It stops when no remaining turning point can produce an error smaller than the error bound. We use an array B to store the indexes of the remaining turning points, initially from 0 to the number of turning points in $f_{v_s,v_d}(t)$ (line 2-3). For all the turning points except the two ending points, we compute the error that introduced by removing every single one of them at a time (line 6-7), and actually remove the one introducing the minimum error. The removal is implemented by erasing the index in B (line 11-12). The introduced error is computed by a function R (line 7). It first constructs

Algorithm 11: Approximation Algorithm

```
   Input: $f_{v_s,v_d}(t)$, error threshold $\varepsilon$
   Output: $\hat{f}_{v_s,v_d}(t)$
1  begin
2      $size \leftarrow |f_{v_s,v_d}(t)|$
3      $B =< 0, 1, ..., size - 1 >$
4      do
5          $minError = inf$
6          for $j\ from\ 1\ to\ |B| - 1$ do
7              $errorTmp = R(f_{v_s,v_d}(t), B[j-1], B[j+1])$
8              if $errorTmp < minError$ then
9                  $minError = errorTmp$
10                 $breakPoint = j$
11         if $minError < \varepsilon$ then
12             $B.erase(breakPoint)$
13     while $minError \leqslant \varepsilon$
14     return $f_{v_s,v_d}(B)$
15
16     Function $R(f_{v_s,v_d}(t), B[i], B[j])$
17         $error \leftarrow 0$
18         //compute a linear function with points
19         //$(t[B[i]], f_{v_s,v_d}(t[B[i]]))$ and $(t[B[j]], f_{v_s,v_d}(t[B[j]]))$
20         $f'(t) \leftarrow linear < f_{v_s,v_d}(t[B[i]]), f_{v_s,v_d}(t[B[j]]) >$
21         for $k\ from\ B[i] + 1\ to\ B[j]$ do
22             $error \leftarrow error + f'(t[k]) - f_{v_s,v_d}(t[k])$
23         return $error$
```

a linear function $f'(t)$ with two ending points $(t[B[i]], f_{v_s,v_d}(t[B[i]]))$ and $(t[B[j]], f_{v_s,v_d}(t[B[j]]))$, where $t[B[i]]$ is the $B[i]^{th}$ time point (line 20). It should be noted that although $j-1$ and $j+1$ is separated by only one number, their actual values $B[j-1]$ and $B[j+1]$ could be separated by multiple numbers because they were erased in the previous iteration. We need to accumulate the errors of all these intermediate values, by computing the differences between original value and its corresponding value on the linear function $f'(t)$, to get the total error (line 21-22). When no error is smaller than the input error bound, the iteration stops and a new minimum cost function is returned by only keeping the remaining turning points in B.

Because the query answering considers two labels, the error bound of inner partition result is 2ε. As for the query between partitions, it is 6ε obviously.

6.5.8 Fastest Path Query

It is easy to use our time-dependent 2-hop to answer any fastest path query. Suppose the query is issued in time interval $[t_1, t_2]$. A straightforward approach is getting the profile result first, then retrieve the sub-function during $[t_1, t_2]$. A more efficient way is constructing the intermediate functions only during $[t_1, t_2]$. Because the turning points are sorted in temporal order, we can locate the desired positions (t_1 and t_2) in $O(\log T)$ time. All the other procedures are the same, only on a smaller time interval.

6.6 Experiment

In this section, we experimentally evaluate the proposed time-dependent 2-hop labeling on both small graph and real-life road network against the current state-of-the-art methods. Section 6.6.1 describes the experiment settings. Section 6.6.2 and Section 6.6.3 present the evaluations on small graph and large road network, respectively.

6.6.1 Experiment Setup

All the algorithms are implemented in C++, compiled with full optimizations, and tested on a Dell R730 PowerEdge Rack Mount Server which has two Xeon E5-2630 2.2GHz (each has 10 cores and 20 threads) and 378G memory. The data are stored on a 12 × 4TB raid-50 disk.

Dataset. The dataset are the same as Chapter 5, as generated from Chapter 4. We partition it into 83 subgraphs using *Natural Cut* [160]. These subgraphs have various sizes between 1000 and 4000 vertices, and 3011 boundary vertices totally (less than 40 boundary vertices per subgraph averagely). We use these subgraphs as testing small graphs. The average turning point number is 26. Most roads in the inner city have more than 60 turning points, while the rural roads typically have less than 20 turning points.

Query sets. The experiment on small graph is first categorized by the graph size: 1000, 2000, 3000 and 4000 vertices. Within each category, they are further categorized into three sub-categories based on the densities of their speed profiles: *Sparse* (around 10 turning points per edge), *Medium* (around 30 turning points per edge) and *Dense* (around 50 turning points per edge). On each sub-category, we run three sets of tests based on the distance between the source and destination. Because

TABLE 6.3: Time-Dependent 2-Hop Experiment Result of Small Graph

Time unit is ms. Speed up value is the running time compared with *FP*'s. Sparse graphs' speed profiles have around 10 turning points per edge, medium graphs' are around 30 and dense graphs' are around 50.

Size	Algorithm		Sparse				Medium				Dense			
	Distance		5km	10km	15km	Size	2km	4km	6km	Size	6km	8km	10km	Size
1k	T2Hop	Time	0.88	0.88	0.87	181M	3	3.6	3.7	736m	13.7	17	22	5.1G
		Speedup	26.58	37.57	39.55		40.2	84	231		92.7	151.4	176.41	
	T2HopA20	Time	0.32	0.33	0.33	37M	0.76	0.92	0.96	138M	3.3	2.9	4.1	883M
		Speedup	41.5	97	102		160	329	890		386	628	896	
	T2HopA30	Time	0.31	0.32	0.33	33M	0.67	0.8	0.86	117M	2.67	2.33	3.44	742M
		Speedup	75.7	100.4	103		181	375	993		480	771	996	
	TCH	Time	28	31	32	15M	97	151	141	60M	305	350	399	131M
		Speedup	0.84	1.07	1.1		1.25	2	6.1		4.2	5.13	8.6	
	FP	Time	23.4	33.22	34.4		121	304	854		1280	1797	3423	
2k	Distance		10km	20km	30km	Size	3km	5km	7km	Size	3km	4km	5km	Size
	T2Hop	Time	1.48	1.54	1.33	736M	18.8	17.8	18.9	8.5G	26.8	26.7	28.2	11G
		Speedup	71.8	423	3099		16	52	159		20	43	104	
	T2HopA20	Time	0.55	0.51	0.38	139M	0.47	0.45	0.48	1.6G	7	6.9	7.1	1.9G
		Speedup	194	1273	10820		666	2060	6304		77	165	417	
	T2HopA30	Time	0.5	0.48	0.36	123M	0.39	0.38	0.4	1.3G	6.1	5.9	6.2	1.6G
		Speedup	216	1369	11356		802	2439	7565		88.4	193	477	
	TCH	Time	22.3	25.8	23.9	16M	344	527	497	206M	379	470	457	254M
		Speedup	4.75	25.23	173		0.91	1.76	6.1		1.42	2.42	6.74	
	FP	Time	106	651	4136		313	927	3026		539	1140	2959	
3k	Distance		20km	40km	60km	Size	10km	15km	20km	Size	4km	5km	6km	Size
	T2Hop	Time	4.8	6	4.7	2.9G	7.9	9.7	9.2	6.3G	38.7	48.9	47.7	27G
		Speedup	29	1098	8179		32	30	52		12	42	79	
	T2HopA20	Time	1.7	1.5	1.3	543M	1.88	2.3	2.2	1.2G	8.2	10.3	10.2	4.8G
		Speedup	82	4393	30061		136	129	222		57.7	202	367	
	T2HopA30	Time	1.5	1.4	1.4	466M	1.68	2	1.9	1002M	7.8	9.3	9.1	4G
		Speedup	93	4707	27914		154	148	257		60.6	224	412	
	TCH	Time	53	57	72	40M	300	289	284	159M	581	1106	1350	599M
		Speedup	2.64	115.6	543		0.85	1.02	1.72		0.84	1.88	2.78	
	FP	Time	140	6590	39080		256	297	489		473	2083	3751	
4k	Distance		10km	15km	20km	Size	6km	8km	10km	Size	3km	4km	5km	Size
	T2Hop	Time	2.7	2.8	2.9	1.5G	24	26	31	19G	53.9	64.5	50.1	33G
		Speedup	345	432	321		15	28	45		28	42	185	
	T2HopA20	Time	0.73	0.59	0.69	288M	7.2	7.9	8.2	3.3G	9.4	9.9	9.2	5.5G
		Speedup	1275	2103	1353		52.7	93	175		160	275	1021	
	T2HopA30	Time	0.65	0.49	0.57	252M	6.3	6.7	6.9	2.8G	7.7	8.1	7.4	4.6G
		Speedup	1432	2534	1638		60	110	207		195	336	1267	
	TCH	Time	99.6	95.2	85.7	53M	381	516	731	298M	1109	1566	1918	669M
		Speedup	9.3	13	10.9		0.99	1.43	2		1.36	1.74	4.9	
	FP	Time	931	1242	934		379	738	1434		1505	2721	9379	

TABLE 6.4: Time-Dependent 2-Hop Experiment Result of Road Network

Time unit is *second*. *Size* is the total index size, and *Space* is the index actually used for query. *T* is short for *Time* and *S* is short for *Speedup*.

Algorithm		Sparse				Medium				Dense				Size
Distance		70km	80km	90km	Space	40km	50km	60km	Space	15km	20km	25km	Space	Size
T2Hop	T	0.388	0.368	0.37	5.4G	1.65	1.71	1.81	32.3G	6.26	6.12	6.6	82.4G	1.48T
	S	505	766	986		262	262	256		31.5	33.17	44.1		
T2HopA20	T	0.021	0.0137	0.0141	760M	0.025	0.026	0.026	5.3G	0.115	0.117	0.118	14G	224G
	S	9333	20583	25886		17320	17230	17846		1713	1735	2466		
T2HopA30	T	0.0153	0.0149	0.016	557M	0.023	0.025	0.026	4.2G	0.077	0.081	0.083	11.4G	208G
	S	12810	18926	22813		18826	17920	17846		2558	2506	3506		
TCH	T	38.7	35.53	39.3	85G	24.8	23.85	22.4	85G	24.3	19.67	18.5	85G	85G
	S	5.1	7.9	9.2		17.5	18.88	20.7		8.1	10.3	15.7		
TSHARC	T	31.9	29.4	33.2	86.8G	18.7	19.3	18.2	86.8G	19.3	16.4	15.9	86.8G	86.8G
	S	6.1	9.6	11		23	23.2	25.5		10.2	12.4	18.3		
FP	T	196	282	365		433	448	464		197	203	291		

the actual regions of each small graph vary, the distance ranges are different. For each test set, we generate 100 vertex pairs randomly.

The experiment in road network tests the source and destination vertex pairs from different subgraphs. Based on the speed profile density of their origin subgraphs, we categorize the tests into three sets: *Sparse, Medium* and *Dense*. For example, in the *Sparse* test, both of the source and destination come from sparse subgraphs. So do *Medium* and *Dense*. Under each category, we also run three sets of tests based on the distance between the source and destination. Each set contains 100 randomly generated vertex pairs.

Methods. Our time-dependent 2-hop method is denoted as *T2Hop*. We also applied approximation on it with error bound of 20 and 30. They are denoted as *T2HopA20* and *T2HopA30*. We implement the time-dependent CH (*TCH*) using the heuristic method described in[105], with hop limit set to 20. The time-dependent SHARC (*TSHARC*) is implemented using the method of [107]. The baseline approach is the fastest path algorithm (*FP*) [24].

6.6.2 Evaluation on Small Graph

The experiment results are shown in Table 6.3. The four big rows are the graph sizes, and the three big columns are the speed profile density. For each method, we list its index size, the actual query running time (in ms) on different distance sets, and the speedup gain compared with the *fastest path*.

Firstly, all three time-dependent 2-hop approaches have shorter query time than the direct fastest path algorithm and the two online speedup approaches. As the approximation error bound grows, the index size drops, which further leads to shorter query time and higher speedup. This is because their hop sets are the same, but the approximate ones have smaller minimum cost function of each hop, and the cost of linear piecewise function operations drops.

Secondly, the speed profile density has a higher impact on query time than the vertex number and distance. For example, the 1k graph with medium density has queries around 4km length, but its query time is shorter than the 4k graph with sparse density running querier on 20km tests. However, the *FP* time on 20km-4k-sparse graph is similar to 10km-4k-sparse graph. This is because this sparse graph is in the rural region, the longer queries are derived from the rural roads that are long and have sparser speed profile, while the shorter queries are derived from the near-city roads which are shorter but have denser speed profile.

Thirdly, the query answering time increases a little bit as the distance increase, but on a much smaller scale than *FP*. Thus, even though the query answering time does not increase a lot, the speedup increases dramatically. In fact, it is also affected by the order to visit the vertices when we create the labels.

Lastly, the graph size also affects the query performance. For the graphs with similar density, the larger ones have longer query time. This is because the larger ones have more hop labels than the smaller ones. Moreover, for the *FP*, it also needs to visit more vertices through shorter edges.

Finally, the distance between two vertices does not have a significant effect on *TCH* as expected. For some short queries, it is even slower than *FP*. As the distance grows, its speedup performance becomes better but still worse than time-dependent 2-hop. This is because it contains many shortcuts on the vertices. Among these shortcuts, some link to the vertices far away, which help reduce the searching space for long queries. However, for those short queries, we still have to visit those shortcuts, as long as they still have the possibility to contribute to the final result. Therefore, *TCH* might have to visit more vertices than *FP* when the query is short, which makes it slower. On the other hand, it could have speedup performance up to several hundred times, as long as the query distance is long. Nevertheless, it is still slower than our approach.

6.6.3 Evaluation on Large Road Network

The experiment results are shown in Table 6.4. The three big columns are the speed profile densities, and the last small column shows the total index sizes. Under each density category, we test three sets of different distances. Unlike the evaluation on the small graph, the query running time here is in second. The speedup gain is also compared with *FP*.

Apparently, our time-dependent 2-hop approach has a shorter query time than the online speedup approaches, and the approximate methods are even much faster. Like the evaluation on the small graph, the speed profile density also plays an important role. Even when the testing distances are shorter than the sparse tests, the dense tests still have longer query time. On the other hand, the query time does not change too much as the distance increases, because it involves concatenation of three result sets, which takes much longer time than the queries in the subgraphs. Therefore, the main factor that affects the query time is the size of minimum cost functions.

The running time of *TCH* and *TSHARC* does not change too much. Because they need to visit a large number of shortcuts for each vertex, no matter the path is short or long. For the short paths, they suffer from the wasteful computation; For the long paths, they benefit from it. The *TSHARC* is slightly faster than *TCH*, because it is essentially a pruning strategy of *TCH* that avoids visiting some of the apparently useless shortcuts. And its index size is also slightly bigger.

The *Space* columns within each category are the actual space it takes to answer the query. Because our approach is partition based, we only need to load the used indexes of the subgraphs and their corresponding boundary labels. Therefore, the actual space that our approaches consume is smaller than *TCH* and *TSHARC*, which have to load the entire index into memory.

6.7 Summary

In this chapter, we extend the 2-hop label approach to time-dependent environment and use it to answer the *fastest path profile* query quickly. Unlike the case under static environment, where the shortest path indexes are well studied and widely used, little work has been done on the time-dependent environment. Even the state-of-art approaches fall into the easier and smaller but slower *online search* category, and only tested on the simple histogram-based speed profile rather than the common linear piecewise function. In this work, we first propose a time-dependent 2-hop labeling approach to answer fastest path profile query on small graph, using the linear piecewise function. After that, we extend it to real-life large road network by exploiting graph partition. Both of the construction and query answering can work in parallel. To further reduce the space consumption and boost the query answering, we provide a approximation method using *PLA*. Our extensive experiments show the query answering time is sped up to hundreds of times and also much faster than the *online search* approaches.

Chapter 7

Conclusion and future work

7.1 Conclusion

In this thesis, we study the route scheduling in road network thoroughly. From map data preparation and speed profile generation, to *MORT* path finding and fast query answering.

In Chapter 3, we study the reachability problem on general graphs. By extending the graph dominance drawing to higher dimension space, we propose our *HD-GDD* method. It has smaller index size and faster construction time than the accurate indexes, and is faster in query answering than the online search approaches. Evaluations on various graphs with different configurations have fully tested its performance. When it is applied in road network, which is mostly planar, it can achieve constant time query answering with a small dimension number like four or 5. Therefore, we can use it to prune out the isolated vertices on a raw map data, and tests if a pair of query vertices are reachable or not before the actual search starts. In fact, its best using scenario is the near-planar graph like road network.

In Chapter 4, we present our solution to generate speed profile from trajectory data. Because it is expensive and actually impossible to use traffic sensors to generate the speed profile of an entire city, we take advantage of the pervasive trajectories that generated by drivers everyday. However, few work exists to cover the whole process from trajectory to speed profile, so most of the works in the time-dependent road network field just generate a speed profile synthetically. We first map the trajectories on the background road network. Then we test different granularities to collect the speed values. After that, we propose two simple but effective methods to estimate the missing values. Finally, we introduce the *piecewise linear approximation* algorithms to compress the big speed profile. The generated speed profile is used in the latter part of our research.

In Chapter 5, we propose a general form of time-dependent path problem, the *MORT* problem. By allowing waiting on some predefined vertices, the optimization object converts to the actual driving time on road. Various users like logistic companies, tourists and urban planning facilities have needs to find paths like this. Moreover, by configuring some parameters, our problem can reduce to all the other existing single-objective path problems. We propose two algorithm to solve it accurately and analyze their complexity. To further speed up the computation, we provide several approximation techniques with worst case error guaranteed. Extensive experiments are conducted to fully study the algorithm performance.

In Chapter 6, we propose a *time-dependent 2-hop labeling* approach to answer the slow time-dependent path problem queries faster in road network. Similar approaches are studied extensively on static, but no extension on time-dependent environment ever exists. This is because the index size and construction time would soar up. Therefore, only extension on *online search* approaches were proposed in the past ten years. To make the *2-hop* applicable, we first propose a pruning based approach on small graph. After that, we take advantages of graph partition methods to split the large road network into subgraphs. By constructing indexes with and between these subgraphs, we obtain our *time-dependent 2-hop*. To further reduce the index size and speedup query answering, we propose an approximation method with error bounded by a small number. Comparison with the direct path algorithms and existing online speedup approaches shows than our approach is hundreds or thousands times faster.

7.2 Future work

In the future, we plan to continue to explore our research work on time-dependent road network towards the following directions:

- The reachability problem is still not well solved. In our study, we analyze the cause of *false positive* and the importance of *topological ordering* of the existing approaches. However, theoretical breakthrough still lies ahead for us to find out. Without a definite problem complexity analysis related to the graph structure, any approach is just heuristic and only works best on some specific scenarios (like ours in road network). A deep investigation into graph structures is essential to solve it.

- The time-dependent path problem is still not well solved, because no algorithm can reach the lower bound of the problem complexity.
- The path planning query is on the level of millions per hour nowadays. Although the static index is able to answer them quickly, we lose the new information brought by these queries. How to take advantages of this large amount of queries to keep the index up-to-date, and use them to help answering future queries, is a research topic and also an actual need of industry field. A cache like structure would be helpful to answer similar queries.
- All of our time-dependent path methods can be used to extend the static path-related problems, like group trip planning, constraint shortest path, taxi driver-customer assignment and many more. It is not just simply making the problem more complex, but actually improve the real-life services with higher accuracy.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish, *Efficient management of transitive relationships in large data and knowledge bases*, vol. 18.

[2] S. J. van Schaik and O. de Moor, "A memory efficient reachability data structure through bit vector compression," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 913–924, ACM, 2011.

[3] Y. Chen and Y. Chen, "An efficient algorithm for answering graph reachability queries," in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pp. 893–902, IEEE, 2008.

[4] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 595–608, ACM, 2008.

[5] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu, "Tf-label: a topological-folding labeling scheme for reachability querying in a large graph," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 193–204, ACM, 2013.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM Journal on Computing*, vol. 32, no. 5, pp. 1338–1355, 2003.

[7] H. He, H. Wang, J. Yang, and P. S. Yu, "Compact reachability labeling for graph-structured data," in *Proceedings of the 14th ACM international conference on Information and knowledge management*, pp. 594–601, ACM, 2005.

[8] R. Schenkel, A. Theobald, and G. Weikum, "Efficient creation and incremental maintenance of the hopi index for complex xml document collections," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pp. 360–371, IEEE, 2005.

[9] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computing reachability labelings for large graphs with high compression rate," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pp. 193–204, ACM, 2008.

[10] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 813–826, ACM, 2009.

[11] R. Jin and G. Wang, "Simple, fast, and scalable reachability oracle," *Proc. VLDB Endow.*, vol. 6, pp. 1978–1989, Sept. 2013.

[12] H. Yildirim, V. Chaoji, and M. J. Zaki, "Grail: Scalable reachability index for large graphs," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 276–284, 2010.

[13] S. Seufert, A. Anand, S. Bedathur, and G. Weikum, "Ferrari: Flexible and efficient reachability range assignment for graph indexing," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pp. 1009–1020, IEEE, 2013.

[14] R. R. Veloso, L. Cerf, W. Meira Jr, and M. J. Zaki, "Reachability queries in very large graphs: A fast refined online search approach.," in *EDBT*, pp. 511–522, 2014.

[15] T. Kameda, "On the vector representation of the reachability in planar directed graphs," *Information Processing Letters*, vol. 3, no. 3, pp. 75–77, 1975.

[16] P. Eades, H. ElGindy, M. Houle, B. Lenhart, M. Miller, D. Rappaport, and S. Whitesides, "Dominance drawings of bipartite graphs," 1994.

[17] E. M. Kornaropoulos and I. G. Tollis, "Overloaded orthogonal drawings," in *Graph Drawing*, pp. 242–253, Springer, 2012.

[18] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, 2014.

[19] L. Fratta, M. Gerla, and L. Kleinrock, "The flow deviation method: An approach to store-and-forward communication network design," *Networks*, vol. 3, no. 2, pp. 97–133, 1973.

[20] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[21] B. Ding, J. X. Yu, and L. Qin, "Finding time-dependent shortest paths over large graphs," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pp. 205–216, ACM, 2008.

[22] L. Li, W. Hua, and X. Zhou, "Hd-gdd: high dimensional graph dominance drawing approach for reachability query," *World Wide Web*, vol. 20, no. 4, pp. 677–696, 2017.

[23] L. Li, K. Zheng, S. Wang, W. Hua, and X. Zhou, "Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 27, no. 3, pp. 321–345, 2018.

[24] L. Li, W. Hua, X. Du, and X. Zhou, "Minimal on-road time route scheduling on time-dependent graphs," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1274–1285, 2017.

[25] L. Li, X. Zhou, and K. Zheng, "Finding least on-road travel time on road network," in *Australasian Database Conference*, pp. 137–149, Springer, 2016.

[26] H. Jagadish, "A compression technique to materialize transitive closure," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 4, pp. 558–598, 1990.

[27] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pp. 75–75, IEEE, 2006.

[28] R. Jin, N. Ruan, S. Dey, and J. Y. Xu, "Scarab: scaling reachability computation on large graphs," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 169–180, ACM, 2012.

[29] E. M. Kornaropoulos and I. G. Tollis, "Weak dominance drawings and linear extension diameter," *arXiv preprint arXiv:1108.1439*, 2011.

[30] U. Demiryurek, B. Pan, F. Banaei-Kashani, and C. Shahabi, “Towards modeling the traffic data on road networks,” in *Proceedings of the Second International Workshop on Computational Transportation Science*, pp. 13–18, ACM, 2009.

[31] F. C. Pereira, H. Costa, and N. M. Pereira, “An off-line map-matching algorithm for incomplete map databases,” *European Transport Research Review*, vol. 1, no. 3, pp. 107–124, 2009.

[32] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang, “Map-matching for low-sampling-rate gps trajectories,” in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 352–361, ACM, 2009.

[33] B. Yang, C. Guo, and C. S. Jensen, “Travel cost inference from sparse, spatio temporally correlated time series using markov models,” *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 769–780, 2013.

[34] P. Widhalm, M. Piff, N. Brändle, H. Koller, and M. Reinthaler, “Robust road link speed estimates for sparse or missing probe vehicle data,” in *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*, pp. 1693–1697, IEEE, 2012.

[35] X. Xin, C. Lu, Y. Wang, and H. Huang, “Forecasting collector road speeds under high percentage of missing data.,” in *AAAI*, pp. 1917–1923, 2015.

[36] Z. Shan, D. Zhao, and Y. Xia, “Urban road traffic speed estimation for missing probe vehicle data based on multiple linear regression model,” in *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on*, pp. 118–123, IEEE, 2013.

[37] M. T. Asif, N. Mitrovic, L. Garg, J. Dauwels, and P. Jaillet, “Low-dimensional models for missing data imputation in road networks,” in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 3527–3531, IEEE, 2013.

[38] T. Erdelić, S. Vrbančić, and L. Rošić, “A model of speed profiles for urban road networks using g-means clustering,” in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, pp. 1081–1086, IEEE, 2015.

[39] K. Fitzpatrick and J. Collins, "Speed-profile model for two-lane rural highways," *Transportation Research Record: Journal of the Transportation Research Board*, no. 1737, pp. 42–49, 2000.

[40] S. R. Eddy, "Hidden markov models," *Current opinion in structural biology*, vol. 6, no. 3, pp. 361–365, 1996.

[41] J. Shang, Y. Zheng, W. Tong, E. Chang, and Y. Yu, "Inferring gas consumption and pollution emission of vehicles throughout a city," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1027–1036, ACM, 2014.

[42] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th international conference on World Wide Web*, pp. 285–295, ACM, 2001.

[43] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, no. 8, pp. 30–37, 2009.

[44] P. Esling and C. Agon, "Time-series data mining," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 12, 2012.

[45] H. Shatkay and S. B. Zdonik, "Approximate queries and representations for large data sequences," in *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pp. 536–545, IEEE, 1996.

[46] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "Segmenting time series: A survey and novel approach," *Data mining in time series databases*, vol. 57, pp. 1–22, 2004.

[47] C.-S. Li, P. S. Yu, and V. Castelli, "Malm: A framework for mining sequence database at multiple abstraction levels," in *Proceedings of the seventh international conference on Information and knowledge management*, pp. 267–272, ACM, 1998.

[48] S. Park, D. Lee, and W. W. Chu, "Fast retrieval of similar subsequences in long sequence databases," in *Knowledge and Data Engineering Exchange, 1999.(KDEX'99) Proceedings. 1999 Workshop on*, pp. 60–67, IEEE, 1999.

[49] E. J. Keogh and M. J. Pazzani, "An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback.," in *KDD*, vol. 98, pp. 239–243, 1998.

[50] K. L. Cooke and E. Halsey, "The shortest route through a network with time-dependent internodal transit times," *Journal of mathematical analysis and applications*, vol. 14, no. 3, pp. 493–498, 1966.

[51] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[52] E. V. Denardo and B. L. Fox, "Shortest-route methods: 1. reaching, pruning, and buckets," *Operations Research*, vol. 27, no. 1, pp. 161–186, 1979.

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

[54] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[55] R. Bellman, "On a routing problem," tech. rep., DTIC Document, 1956.

[56] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[57] M. L. Fredman, "New bounds on the complexity of the shortest path problem," *SIAM Journal on Computing*, vol. 5, no. 1, pp. 83–89, 1976.

[58] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 1–13, 1977.

[59] D. Peleg, "Proximity-preserving labeling schemes," *Journal of Graph Theory*, vol. 33, no. 3, pp. 167–176, 2000.

[60] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz, "Distance labeling in graphs," in *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pp. 210–219, Society for Industrial and Applied Mathematics, 2001.

[61] M. Thorup and U. Zwick, “Approximate distance oracles,” *Journal of the ACM (JACM)*, vol. 52, no. 1, pp. 1–24, 2005.

[62] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu, “Hop doubling label indexing for point-to-point distance querying on scale-free networks,” *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1203–1214, 2014.

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

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

[65] T. Akiba, Y. Iwata, and Y. Yoshida, “Fast exact shortest-path distance queries on large networks by pruned landmark labeling,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 349–360, ACM, 2013.

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

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

[68] R. Jin, N. Ruan, Y. Xiang, and V. Lee, “A highway-centric labeling approach for answering distance queries on large sparse graphs,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 445–456, ACM, 2012.

[69] 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*, vol. 10, no. 3, pp. 409–432, 1998.

[70] S. Jung and S. Pramanik, “An efficient path computation model for hierarchically structured topographical road maps,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 5, pp. 1029–1046, 2002.

[71] 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*, pp. 43–54, ACM, 2008.

[72] 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*, pp. 200–209, ACM, 2005.

[73] L. Tang and M. Crovella, "Virtual landmarks for the internet," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pp. 143–152, ACM, 2003.

[74] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. d. C. Reis, and B. Ribeiro-Neto, "Efficient search ranking in social networks," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pp. 563–572, ACM, 2007.

[75] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *Proceedings of the 18th ACM conference on Information and knowledge management*, pp. 867–876, ACM, 2009.

[76] W. Chen, C. Sommer, S.-H. Teng, and Y. Wang, "A compact routing scheme and approximate distance oracle for power-law graphs," *ACM Transactions on Algorithms (TALG)*, vol. 9, no. 1, p. 4, 2012.

[77] T. Akiba, C. Sommer, and K.-i. Kawarabayashi, "Shortest-path queries for complex networks: exploiting low tree-width outside the core," in *Proceedings of the 15th International Conference on Extending Database Technology*, pp. 144–155, ACM, 2012.

[78] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas, "Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, pp. 1785–1794, ACM, 2011.

[79] M. Qiao, H. Cheng, L. Chang, and J. X. Yu, "Approximate shortest distance computing: A query-dependent local landmark scheme," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 1, pp. 55–68, 2014.

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

[81] F. Wei, "Tedi: efficient shortest path query answering on graphs," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 99–110, ACM, 2010.

[82] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 721–732, 2014.

[83] R. Bauer, D. Delling, and D. Wagner, "Experimental study of speed up techniques for timetable information systems," *Networks*, vol. 57, no. 1, pp. 38–52, 2011.

[84] B. B. Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *International Journal of Foundations of Computer Science*, vol. 14, no. 02, pp. 267–285, 2003.

[85] D. Kempe, J. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pp. 504–513, ACM, 2000.

[86] A. Orda and R. Rom, "Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length," *Journal of the ACM (JACM)*, vol. 37, no. 3, pp. 607–625, 1990.

[87] R. K. Pan and J. Saramäki, "Path lengths, correlations, and centrality in temporal networks," *Physical Review E*, vol. 84, no. 1, p. 016105, 2011.

[88] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, "Intriguingly simple and fast transit routing," in *Experimental Algorithms*, pp. 43–54, Springer, 2013.

[89] R. Geisberger, "Contraction of timetable networks with realistic transfers," in *Experimental Algorithms*, pp. 71–82, Springer, 2010.

[90] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 967–982, ACM, 2015.

[91] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger, "Fast routing in very large public transportation networks using transfer patterns," in *Algorithms–ESA 2010*, pp. 290–301, Springer, 2010.

[92] S. E. Dreyfus, "An appraisal of some shortest-path algorithms," *Operations research*, vol. 17, no. 3, pp. 395–412, 1969.

[93] J. Halpern, "Shortest route with time dependent length of edges and limited delay possibilities in nodes," *Zeitschrift fuer operations research*, vol. 21, no. 3, pp. 117–124, 1977.

[94] A. Orda and R. Rom, "Minimum weight paths in time-dependent networks," *Networks*, vol. 21, no. 3, pp. 295–319, 1991.

[95] I. Chabini, "Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time," *Transportation Research Record: Journal of the Transportation Research Board*, no. 1645, pp. 170–175, 1998.

[96] E. Kanoulas, Y. Du, T. Xia, and D. Zhang, "Finding fastest paths on a road network with speed patterns," in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pp. 10–10, IEEE, 2006.

[97] E. H.-C. Lu, C.-C. Lin, and V. S. Tseng, "Mining the shortest path within a travel time constraint in road network environments," in *Intelligent Transportation Systems, 2008. ITSC 2008. 11th International IEEE Conference on*, pp. 593–598, IEEE, 2008.

[98] J. Borges and M. Levene, "Data mining of user navigation patterns," in *Web usage analysis and user profiling*, pp. 92–112, Springer, 2000.

[99] C. H. Cheong and M. H. Wong, "Mining popular paths in a transportation database system with privacy protection," in *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, p. 122, IEEE, 2006.

[100] E. H.-C. Lu, W.-C. Lee, and V. S. Tseng, "Mining fastest path from trajectories with multiple destinations in road networks," *Knowledge and information systems*, vol. 29, no. 1, pp. 25–53, 2011.

[101] U. Demiryurek, F. Banaei-Kashani, C. Shahabi, and A. Ranganathan, "Online computation of fastest path in time-dependent spatial networks," in *International Symposium on Spatial and Temporal Databases*, pp. 92–111, Springer, 2011.

[102] G. Nannicini, D. Delling, L. Liberti, and D. Schultes, "Bidirectional a search for time-dependent fast paths," in *International Workshop on Experimental and Efficient Algorithms*, pp. 334–346, Springer, 2008.

[103] T. Ikeda, M.-Y. Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh, "A fast algorithm for finding better routes by ai search techniques," in *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994*, pp. 291–296, IEEE, 1994.

[104] A. V. Goldberg and R. F. F. Werneck, "Computing point-to-point shortest paths from external memory.," in *ALENEX/ANALCO*, pp. 26–40, 2005.

[105] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, "Time-dependent contraction hierarchies," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pp. 97–105, Society for Industrial and Applied Mathematics, 2009.

[106] G. V. Batz, R. Geisberger, S. Neubauer, and P. Sanders, "Time-dependent contraction hierarchies and approximation," in *International Symposium on Experimental Algorithms*, pp. 166–177, Springer, 2010.

[107] D. Delling, "Time-dependent sharc-routing," *Algorithmica*, vol. 60, no. 1, pp. 60–94, 2011.

[108] E. Köhler, R. H. Möhring, and H. Schilling, "Acceleration of shortest path and constrained shortest path computation," in *International Workshop on Experimental and Efficient Algorithms*, pp. 126–138, Springer, 2005.

[109] U. Lauther, "An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background," *Geoinformation und Mobilität-von der Forschung zur praktischen Anwendung*, vol. 22, pp. 219–230, 2004.

[110] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, "Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm," *Journal of Experimental Algorithmics (JEA)*, vol. 15, pp. 2–3, 2010.

[111] D. Delling and G. Nannicini, “Bidirectional core-based routing in dynamic time-dependent road networks,” in *International Symposium on Algorithms and Computation*, pp. 812–823, Springer, 2008.

[112] H. Wang, J. Li, W. Wang, and X. Lin, “Coding-based join algorithms for structural queries on graph-structured xml document,” *World Wide Web*, vol. 11, no. 4, pp. 485–510, 2008.

[113] J. Van Helden, A. Naim, R. Mancuso, M. Eldridge, L. Wernisch, D. Gilbert, and S. J. Wodak, “Representing and analysing molecular and cellular function in the computer,” *Biological chemistry*, vol. 381, no. 9-10, pp. 921–935, 2000.

[114] K. Anyanwu and A. Sheth, “P-queries: enabling querying for semantic associations on the semantic web,” in *Proceedings of the 12th international conference on World Wide Web*, pp. 690–699, ACM, 2003.

[115] S. Kutty, R. Nayak, and L. Chen, “A people-to-people matching system using graph mining techniques,” *World Wide Web*, vol. 17, no. 3, pp. 311–349, 2014.

[116] M. Fernandez, D. Florescu, A. Levy, and D. Suciu, “A query language and processor for a website management system,” in *Proc. of the Workshop on Semi-structured Data, Tucson, Arizona*, pp. 26–33, Citeseer, 1997.

[117] M.-F. Chiang, W.-C. Peng, and S. Y. Philip, “Exploring latent browsing graph for question answering recommendation,” *World Wide Web*, vol. 15, no. 5-6, pp. 603–630, 2012.

[118] B. Berendt and M. Spiliopoulou, “Analysis of navigation behaviour in web sites integrating multiple information systems,” *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 9, no. 1, pp. 56–75, 2000.

[119] Y. Cui, J. Pei, G. Tang, W.-S. Luk, D. Jiang, and M. Hua, “Finding email correspondents in online social networks,” *World Wide Web*, vol. 16, no. 2, pp. 195–218, 2013.

[120] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[121] H. Cai, V. W. Zheng, and K. Chang, “A comprehensive survey of graph embedding: problems, techniques and applications,” *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[122] D. Harel and Y. Koren, "Graph drawing by high-dimensional embedding," in *International symposium on graph drawing*, pp. 207–219, Springer, 2002.

[123] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, (Pasadena, CA USA), pp. 11–15, Aug. 2008.

[124] P. ERDdS and A. R&WI, "On random graphs i.," *Publ. Math. Debrecen*, vol. 6, pp. 290–297, 1959.

[125] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[126] J. Kunegis, "KONECT – The Koblenz Network Collection," in *Proc. Int. Conf. on World Wide Web Companion*, pp. 1343–1350, 2013.

[127] "Eu institution network dataset – KONECT," Oct. 2014.

[128] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowledge Discovery from Data*, vol. 1, no. 1, pp. 1–40, 2007.

[129] "Citeseer network dataset – KONECT," May 2015.

[130] K. Bollacker, S. Lawrence, and C. L. Giles, "CiteSeer: An autonomous Web agent for automatic retrieval and identification of interesting publications," in *Proc. Int. Conf. on Autonomous Agents*, pp. 116–123, 1998.

[131] "Google network dataset – KONECT," Oct. 2014.

[132] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Statistical properties of community structure in large social and information networks," in *Proc. Int. World Wide Web Conf.*, pp. 695–704, 2008.

[133] "Baidu internal links network dataset – KONECT," Oct. 2014.

[134] X. Niu, X. Sun, H. Wang, S. Rong, G. Qi, and Y. Yu, "Zhishi.me – weaving Chinese linking open data," in *Proc. Int. Semantic Web Conf.*, pp. 205–220, 2011.

[135] "Us patents network dataset – KONECT," Oct. 2014.

[136] B. H. Hall, A. B. Jaffe, and M. Trajtenberg, "The NBER patent citations data file: Lessons, insights and methodological tools," in *NBER Working Papers 8498, National Bureau of Economic Research, Inc*, 2001.

[137] J. Yuan, Y. Zheng, C. Zhang, X. Xie, and G.-Z. Sun, "An interactive-voting based map matching algorithm," in *Proceedings of the 2010 Eleventh International Conference on Mobile Data Management*, pp. 43–52, IEEE Computer Society, 2010.

[138] M. A. Quddus, W. Y. Ochieng, and R. B. Noland, "Current map-matching algorithms for transport applications: State-of-the art and future research directions," *Transportation Research Part C: Emerging Technologies*, vol. 15, no. 5, pp. 312–328, 2007.

[139] D. R. Cox, "The regression analysis of binary sequences," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.

[140] H. L. Seal, *The historical development of the Gauss linear model.* Yale University, 1968.

[141] 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*, vol. 5, no. 5, pp. 406–417, 2012.

[142] X. Cai, T. Kloks, and C. Wong, "Time-varying shortest path problems with constraints," *Networks*, vol. 29, no. 3, pp. 141–150, 1997.

[143] B. Zheng, H. Su, W. Hua, K. Zheng, X. Zhou, and G. Li, "Efficient clue-based route search on road networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 9, pp. 1846–1859, 2017.

[144] L. Foschini, J. Hershberger, and S. Suri, "On the complexity of time-dependent shortest paths," *Algorithmica*, vol. 68, no. 4, pp. 1075–1097, 2014.

[145] X. Cai, T. Kloks, and C. Wong, "Shortest path problems with time constraints," in *International Symposium on Mathematical Foundations of Computer Science*, pp. 255–266, Springer, 1996.

[146] Y. Yang, H. Gao, J. X. Yu, and J. Li, "Finding the cost-optimal path with time constraint over time-dependent graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 673–684, 2014.

[147] J. D. Adler, P. B. Mirchandani, G. Xue, and M. Xia, "The electric vehicle shortest-walk problem with battery exchanges," *Networks and Spatial Economics*, vol. 16, no. 1, pp. 155–173, 2016.

[148] T. Ichimori, H. Ishii, and T. Nishida, "Routing a vehicle with the limitation of fuel.," *J. OPER. RES. SOC. JAPAN.*, vol. 24, no. 3, pp. 277–281, 1981.

[149] Y. Xiao, K. Thulasiraman, G. Xue, and A. Jüttner, "The constrained shortest path problem: algorithmic approaches and an algebraic study with generalization," *AKCE International Journal of Graphs and Combinatorics*, vol. 2, no. 2, pp. 63–86, 2005.

[150] D. Blokh and G. Gutin, "An approximate algorithm for combinatorial optimization problems with two parameters," *Australasian Journal of Combinatorics*, vol. 14, pp. 157–164, 1996.

[151] A. Juttner, B. Szviatovski, I. Mécs, and Z. Rajkó, "Lagrange relaxation based method for the qos routing problem," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 859–868, IEEE, 2001.

[152] M. Karsai, N. Perra, and A. Vespignani, "Time varying networks and the weakness of strong ties," *arXiv preprint arXiv:1303.5966*, 2013.

[153] J. Stehlé, N. Voirin, A. Barrat, C. Cattuto, V. Colizza, L. Isella, C. Régis, J.-F. Pinton, N. Khanafer, W. Van den Broeck, *et al.*, "Simulation of an seir infectious disease model on the dynamic contact network of conference attendees," *BMC medicine*, vol. 9, no. 1, p. 87, 2011.

[154] S. A. Rahman, P. Advani, R. Schunk, R. Schrader, and D. Schomburg, "Metabolic pathway analysis web service (pathway hunter tool at cubic)," *Bioinformatics*, vol. 21, no. 7, pp. 1189–1193, 2005.

[155] T. M. Przytycka, M. Singh, and D. K. Slonim, "Toward the dynamic interactome: it's about time," *Briefings in bioinformatics*, p. bbp057, 2010.

[156] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, “Hierarchical hub labelings for shortest paths,” in *European Symposium on Algorithms*, pp. 24–35, Springer, 2012.

[157] R. Bauer and D. Delling, “Sharc: Fast and robust unidirectional routing,” *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 4, 2009.

[158] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner, “Computing many-to-many shortest paths using highway hierarchies,” in *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 36–45, SIAM, 2007.

[159] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong, “Is-label: an independent-set based labeling scheme for point-to-point distance querying,” *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 457–468, 2013.

[160] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck, “Graph partitioning with natural cuts,” in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 1135–1146, IEEE, 2011.