Efficient Shortest Distance Query Processing and Indexing on Large Road Network

by

DIAN OUYANG

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Centre for Artificial Intelligence (CAI)
Faculty of Engineering and Information Technology (FEIT)
University of Technology Sydney (UTS)

February, 2019

CERTIFICATE OF AUTHORSHIP/ORIGINALITY

This research is supported by the Australian Government Research Training Program

Signature of Candidate

Production Note:
Signature removed
prior to publication.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my supervisor Dr. Lu Qin for his continuous encouragement and guidance of my PhD career. He is professional, kind, patient and wise. He introduces knowledge of database area and research skill to me. His creative ideas extends thinking and horizon for me. Additionally, Lu is a good mentor and friend for me. He shares his experiment of research and life to me. Thanks to his help and engcouragement, I am always confident and brave when experiencing challenge and trouble. This thesis could not reach its present form without his whole-heartedly instructions.

Secondly, I would like to deliver my great gratitude to my co-supervisor Prof. Ying Zhang for his constant encouragement and guidance, especially supporting for my academic career. His brilliant ideas and inspirations extent my research area and mode of thinking. He is also farsighted and provide efficient advise for career.

Thirdly, I would like to thank Prof. Xuemin Lin, Dr. Lijun Chang and Dr. Long Yuan for supporting the works in this thesis, as most of the works are conducted in collaboration with them. I thank Prof. Lin for letting me join his research group. I learn writing and researching skill from Prof Lin. I thank Dr. Chang for his brilliant ideas and support for academic career to me. His wonderful research works inspires my research work in this thesis. I thank Dr. Long Yuan for his advise of writing in this thesis.

I would also like to thank the following people: Prof. Qin Zhu, Prof. Wenjie Zhang, Dr. Xin Cao, Dr. Zengfeng Huang, Dr. Xiaoyang Wang, Dr. Shiyu Yang, for sharing interesting ideas and knowledge. Thanks to Dr. Fan Zhang, Dr. Longbin Lai, Dr. Xing Feng, Dr. Xiang Wang, Dr. Jianye Yang, Dr. Fei Bi, Mr. Xubo Wang, Mr. Wei Li, Mr. Haida Zhang, Ms. Chen Zhang, Ms. Xiaoshuang Chen, Mr. Yang Yang, Mr. Kai Wang, Mr. You Peng, Mr. Hanchen Wang, Mr. Xuefeng Chen, Mr. Boge Liu, Mr. Dong Wen, Mr. Wentao Li, Mr. Bohua Yang, Mr. Mingjie Li, Ms. Conggai Li, Mr. Daokun Zhang, Mr. Fujin Zhu, Mr. Adi Lin for sharing the happiness with me in my PhD career.

Last but not least, I would like to thank my father Mr. Jianzhi Ouyang and my mother Mrs. Meihua Tan, for giving me great living and studying environment. Thanks for their love and supporting during the Ph.D studying.

ABSTRACT

Computing the shortest distance between two vertices is a fundamental problem on road networks. State-of-the-art indexing-based solutions can be categorized into hierarchy-based solutions and hop-based solutions. However, the hierarchy-based solutions require a large search space for long-distance queries while the hop-based solutions result in a high computational waste for short-distance queries. Moreover, in real life, the weight of edges changes frequently. For example, building a road need several months, but the travel time of road changes frequently such as traffic jam in the morning peak. We model this problem as the shortest path problem on a dynamic road network. The existing solutions are not efficient to update the index for the dynamic condition. Shortest path query on bicriteria road network is another important and practical problem in real life. To compute shortest path between any two vertices, we can get the shortest path set which is called path skyline. We propose an efficient exploring strategy to accelerate path skyline computing.

We propose a novel hierarchical 2-hop index (H2H-Index) which assigns a label for each vertex and at the same time preserves a hierarchy among all vertices. With the H2H-Index, we design an efficient query processing algorithm with performance guarantees by visiting part of the labels for the source and destination based on the vertex hierarchy. We also propose an algorithm to construct the H2H-Index based on distance preserved graphs. The algorithm is further optimized by computing the labels based on the partially computed labels of other vertices.

We use dynamic road network to define the graph model whose topological structure is stable and weight of edges changes frequently. In this model, we have two processing, shortest path query and road update processing, to do on road network. We use Contraction Hierarchies which is one of art-of-the-state index algorithm for shortest path problem to answer queries. And propose an efficient index updating algorithm to update CH index for road updating processing. In contrast to vertex centric algorithm, our shortcut centric algorithm has better theoretical bound.

In the literature, PSQ is a fundamental algorithm for path skyline query and is also used as a building block for the afterwards proposed algorithms. In PSQ, a key operation is to record the skyline paths for each node v that is possible on the skyline paths from s to t. However, to obtain the skyline paths for v, PSQ has to maintain other paths that are not skyline paths for v, which makes PSQ inefficient. Motivated by this, in this chapter, we propose a new algorithm PSQ$^+$ for the path skyline query. By adopting an ordered path exploring strategy, our algorithm can totally avoid the fruitless path maintenance problem in PSQ.

We conducted extensive performance studies using large real road networks including the whole USA road network. The experimental results demonstrate that our approach can make significant improvement to every problem.

PUBLICATIONS

- ***Dian Ouyang***, *Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu, When Hierarchy Meets 2-Hop-Labeling: Efficient Shortest Distance Queries on Road Networks, to apper in SIGMOD 2018.*

- ***Dian Ouyang***, *Long Yuan, Fan Zhang, Lu Qin and Xuemin Lin, Towards Efficient Path Skyline Computation in Bicriteria Networks, to apper in DASFAA 2018.*

- ***Dian Ouyang***, *Long Yuan, Lu Qin, Ying Zhang and Lijun Chang, Efficient Shortest Distance Query Processing on Dynamic Road Networks with Theoretical Guarantees, in submission*

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Computing the shortest network distance between two locations is one of the most fundamental problems on road networks with many applications such as GPS navigation, POI recommendation, and route planning services. A road network can be modelled as a weighted graph $G(V, E)$ with vertex set V and edge set E. Given a source vertex $s \in V$ and a destination vertex $t \in V$, a shortest distance query asks for the shortest network distance from s to t in graph G. The classic approach to answer a shortest distance query is to use the Dijkstra's algorithm. Given a shortest distance query $q = (s, t)$, the Dijkstra's algorithm traverses the vertices in graph G in non-decreasing order of their distances to s and terminates when the destination vertex t is reached. Dijkstra's algorithm can compute the shortest distance between two vertices in polynomial time. Nevertheless, when the network is large, the Dijkstra's algorithm cannot satisfy the real-time requirements for a shortest distance query since it may traverse the whole network when the two query vertices are far away from each other. Consequently, researchers resort to indexing-based approaches by investigating the characteristics of road networks such as the low-degree and planarity properties [35, 28, 54, 11, 26, 53, 56, 55, 1, 43, 2, 68, 8, 75, 3, 10].

In the experiment, our datasets are the real data from the US road network. The largest dataset contains the whole road network in the US which is over 23 million vertices and 58 million edges. The existing solutions are not efficient on large size datasets. Hence, the motivation of this thesis is to propose more efficient algorithms for practical shortest path query problem on road network. Three topics, shortest distance, index update on dynamic network and path skyline query on bicriteria network, are studied in this thesis. There are many important topics of road network. However, we choose three typical topics in this thesis. The other topics are introduced in future work.

In each topic, we mostly focus on efficiency. So we compare our algorithms with state-of-the-art solutions with query time and update time to prove that our algorithms are efficient and effective on the real dataset.

1.1 H2H Index for Shortest Path Distance

The state-of-the-art approaches for shortest distance query processing on road networks mainly fall into two categories, namely, hierarchy-based solutions [26, 75] and hop-based solutions [1, 2, 3]. The hierarchy-based solutions precompute a hierarchy of vertices on the road network and add some shortcuts from lower-hierarchy vertices to higher-hierarchy vertices. Given a shortest distance query $q = (s, t)$, a hierarchy-based solution starts searching from both s and t simultaneously in a bidirectional manner using the bidirectional Dijkstra's algorithm, and the search direction is confined to be only from lower-hierarchy vertices to higher-hierarchy vertices. In this way, the search space is significantly reduced. The hop-based solutions precompute a 2-hop label for each vertex on the road network. The 2-hop label for a vertex includes the shortest distances from the vertex to a subset of vertices on the road network. Given a shortest distance query $q = (s,t)$, the shortest distance of s and t can be answered using only the labels of s and t by joining the common vertices in their labels.

Motivations. By investigating the above two categories of solutions, we make the following observations. First, a hierarchy-based solution is efficient to answer a shortest distance query $q = (s,t)$ when s and t are near each other on the road network. However, when s and t are far away from each other, a hierarchy-based solution still has to exploit a large number of vertices on the road network because the search space of the bidirectional Dijkstra's algorithm increases rapidly when the distance between s and t increases. Second, for the case where s and t are far away from each other, a hop-based solution is much more efficient than a hierarchy-based solution because the hop-based solution can answer a shortest distance query by only exploiting the labels of s and t without searching the road network. However, when s and t are near each other on the road network, a hop-based solution still needs to scan the entire labels of s and t to answer the query. In this case, a large number of useless vertices in the labels are visited. Motivated by these observations, in this chapter, we propose a novel solution that can overcome the drawbacks of both hierarchy-based and hop-based solutions.

Our Idea. Our general idea is simple: Given a road network, we aim to design a label for each vertex as well as a hierarchy among all vertices on the road network. To answer a shortest distance query $q = (s,t)$, instead of exploiting all vertices in the labels of s and t, we only need to visit a subset of vertices in the labels of s and t adaptively with the help of the vertex hierarchy. Intuitively, the number of visited vertices decreases when the distance between s and t decreases. When the distance between s and t is small, only a small subset of vertices in the labels of s and t need to be visited. Existing hierarchy-based and hop-based solutions each focus on one aspect and cannot be simply combined to achieve our goal. To make our idea practically applicable, the following issues need to be addressed: (1) how to find an appropriate hierarchy for all vertices; (2) how to assign labels for each vertex so that the hierarchy can be used to efficiently answer the shortest distance queries; and (3) how to efficiently compute the hierarchy and the vertex labels.

Contributions. In this chapter, we answer the above questions and make the following contributions:

- We investigate the drawbacks of the existing hierarchy-based and the hop-based solutions for shortest distance query processing on road networks. We introduce a new index named Hierarchical 2-Hop Index (H2H-Index) based on the concept of tree decomposition. Based on the H2H-Index, we design an efficient algorithm to answer a shortest distance query in $O(w)$ time in the worst case, where w is the tree width of the tree decomposition which is small for road networks. Compared to the state-of-the-art hop-based algorithms, our algorithm can avoid visiting a large number of unnecessary vertices in the labels with the assistant of the vertex hierarchy, and thus achieves a much faster query processing time while consuming a similar index space.

- We design an algorithm to construct the H2H-Index based on the concepts of distance preserved graph and distance preserved tree decomposition. We further improve the efficiency of the algorithm by computing the new labels using the partial labels generated for other vertices. Our new algorithm enforces a certain vertex visiting order, in order to maximumly reuse the existing information to reduce the overall computational cost. The index construction algorithm has a bounded worst-case time complexity and thus it can handle large road networks efficiently.

- We conduct extensive performance studies to test the performance of the proposed algorithms on real large road networks including the whole road network of the USA. The experimental results demonstrate that our algorithm can achieve a speedup of an order of magnitude in query processing compared to the state-of-the-art while consuming comparable indexing time and index size.

1.2 Shortest Path Query on Dynamic Roadnetwork

Dijkstra algorithm can compute shortest path for both static graph and dynamic graph. Even if weight of road changes frequently, Dijkstra algorithm can update shortest path efficiently. But comparing with other index shortest path query algorithm, the query efficiency of shortest path query is much slowly. The state-of-the-art algorithms, such as Contraction Hierarchies(CH) and Arterial Hierarchy(AH), are efficient for shortest path query. However, both of them need precomputing for construction index. When weight of road changes, the index also need to be updated. There is not an efficient algorithm for updating index so far. Geisberger gives a vertex-centric algorithm based on CH index construction algorithm. When weight of an edge changes, this algorithm tries to compute all potential shortcuts which contain weight-changing edge and recomputes weight of these shortcuts. However, Geisberger's algorithm still computes redundant vertex and shortcut. For example, when extent of changing weight is low, the number of shortcuts which are influenced is small. The potential shorcuts can be checked and pruned heuristically.

Motivations. Comparing the efficiency of shortest path query algorithm with Dijkstra algorithm and CH algorithm, we find CH is much faster to answer shortest path query on dynamic graph than Dijkstra algorithm[27]. In real life, many shortest path query are needed to be answered at the same time. Generally, weight update processing is less than shortest path query. If we can use the efficient shortest path algorithm CH to answer shortest path query and update index quickly, it is more suitable than Dijkstra which answers shortest path slowly. So we propose an algorithm to update CH Index efficiently. In contrast to AH, the construction time and size of AH is much larger than CH[75]. Updating index of AH costs more time than CH. So, we finally choose CH as our shortest path query index. Comparing with Geisberger's CH updating algoirhtm, we propose a shortcut-centric algorithm. When we compute shortcut base on vertex, all pairs shortcut of the vertex should be computed. Hence, redundant computing is created. So, we regard shortcut as basic element. We use heuristic algorithm to update weight of shortcuts. Only necessory shortcuts will be updated in our algorithm. Moreover, we will propose complexity of our algorithm which is base size of shortcuts which are updated.

Our Idea. In this chapter, we use CH to answer shortest path query. For road weight changing, we propose an efficient vertex-centric updating algorithm. We analyse CH's construction procedure and the relationship of shortcuts in CH index. We find that topological structure is stable if we hold the order of vertices and use 1-hop neighbor bounded algorithm. Therefore, we can hold topological structure of CH index and only update weight of shortcut for weight updating processing. To present relationship of shortcut intuitionisticly, we construct an shortcut supporting graph(SS-Graph) to help us check and update weight of shortcut heuristicly. SS-Graph costs extra space which is unaccpeted in large dataset like the US road network. Hence, we propose improved space saving algorithm which computes relationship of shortcut online. In studying case part, the efficiency of improved algorithm is similar to algorithm with SS-Graph.

Contribution. In this chapter, we make the following contributions:

- We are the first one who propose a shortcut centric algorithm for updating CH index. This algorithm focuses on changing shortcut and pruning most redundant computation. We propose shortcut supporting graph to present relationship of shortcuts.

- For index space, we propose a space saving algorithm. This algorithm does not need extra space consumption and updating processing efficency is similar with basic algorithm.

- We conduct extensive performance studies to test the performance of the proposed algorithms on real large road networks including the whole road network of the USA. Our algorithm greatly improve efficiency of CH index updating. In case studying, our algorithm is quicker than state-of-the-art algorithm of at least 2 order of magnitude for average updating time. For the US road network, we cost several millisecond for weight of an edge changing. It is suitable for CH index updating on large scale dataset.

1.3 Efficient Path Skyline Computation in Bicriteria Networks

Computing the shortest path for a given source s and a destination t in a network is one of the most commonly used operation in online location based services.However, using a single cost criterion is often not sufficient in real applications. For example, in a rush hour, besides considering the distance between s and t, people often consider the traffic congestion degree of the path as well. In this application, people prefer to find all the paths that could potentially be optimal under the distance or traffic congestion degree, which leads to the path skyline query problem. Given a source s and a destination t in a network and a

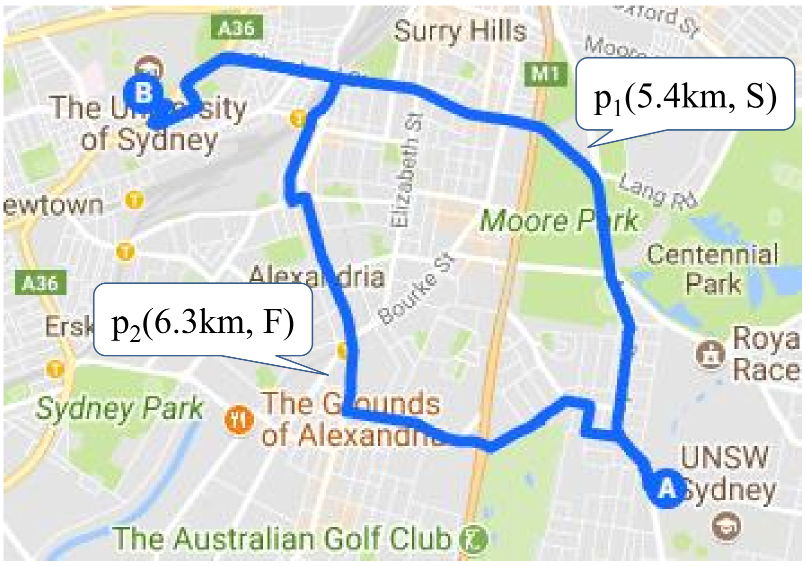


Figure 1.1: Path Skyline Query

set of path cost criteria, a path p_{st} is said to dominate another path p'_{st} when p_{st} is strictly better than p'_{st} for one or more criteria while p_{st} and p'_{st} are equally good for the other criteria. A skyline path from s to t is a path that is not dominated by any other paths from s to t. The path skyline query aims to find all the skyline paths from s to t.

Example 1. *Fig. 1.1 shows a path skyline query in an online location based service. Assume that a user wants to find the path from the University of New South Wales (A) to The University of Sydney (B). Instead of returning the shortest path based on distance, path skyline query returns two paths p_1 and p_2 considering both the distance and the live traffic congestion. p_1 has the shortest distance between A and B but the traffic on p_1 moves slow. Contrarily, p_2 is longer than p_1 but the traffic moves faster. Path skyline query provides the flexility to the user to select the path based on his/her preference.*

Application. Besides the above example, path skyline query has been adopted in a wide range of other application scenarios. For example, in the telecommunication network, the network routing problem aims to find paths that minimize the total number of links while simultaneously minimize the bandwidth [21], which can be modelled as a path skyline query problem. In bicycle trip planning, different routes are recommended based on the distance and the hardness of road conditions [62], which can also be modelled as a path skyline query problem. In earth observing satellite scheduling, path skyline query can be used to select a daily shot sequence to photograph earth landbelts [24].

Motivation. Path skyline query is an NP-hard problem [58]. In general, the number of skyline paths might increase exponentially as a function of the amount of considered cost criteria [59]. In order to keep the amount of potentially skyline paths on a moderate level and make the returned paths easier to interpret by users, in this chapter, we focus on the path skyline query problem in bicriteria networks. In the literature, path skyline query in bicriteria networks has received considerable attention [29, 42, 64, 60, 41, 49]. Among them, PSQ [29] is a fundamental algorithm for this problem and it is also a building block of the afterwards proposed algorithms for the path skyline query problem [42, 64, 49]. To answer the path skyline query for a source s and a destination t, PSQ traverses the nodes of G starting from s and iteratively extends the paths that are possible to lead to skyline paths from s to t. During the traversal, a key operation in PSQ is to record the skyline paths from s to v for each node v which is possible on the skyline paths from s to t. In order to achieve this, PSQ keeps all the paths from s to v which are the skyline paths hitherto as a candidate set. When a new path p_{sv} is explored, PSQ enlarges the candidate set by inserting p_{sv} into it, verifies the dominant relations among the paths in the candidate set and updates the candidate set to ensure that the paths in it are skyline paths from s to v. In this operation, although PSQ only needs to record the skyline paths from s to v, those paths that are not skyline paths are also continuously maintained.

Since this operation is a basic procedure of PSQ and is performed many times in each iteration, lots of fruitless paths are maintained during the processing, which makes PSQ inefficient in terms of recording the skyline paths for each node v.

Our Approach. In order to address the drawback of PSQ, in this chapter, we propose a new algorithm, PSQ$^+$, for the path skyline query problem. We observe that since PSQ processes the paths from s to v disorderly, when processing a new path p_{sv}, PSQ cannot determine whether p_{sv} is a skyline path w.r.t s and v. As a result, PSQ has to keep a candidate set of skyline paths and verify the dominant relationship when a new path is explored. On the other hand, if we can process the paths from s to v in a certain order based on the cost criteria, we can determine whether a path is a skyline path w.r.t s and v directly. Consequently, instead of keeping the candidate set and continuously maintaining the candidate set, we are able to identify whether a path is a skyline path directly and record the exactly skyline paths for each node. Following this idea, our proposed algorithm can avoid the fruitless path maintenance problem in PSQ.

Contribution. In this chapter, we make the following contributions:

(1) Investigation of performance bottleneck of a fundamental algorithm for path skyline query. We conduct a comprehensive literature review and investigate the performance bottleneck of a fundamental algorithm PSQ for path skyline query. Through theoretical and experimental analyses, we find that the operation of recording skyline paths for each possible nodes in PSQ restricts its efficiency.

(2) An accurate skyline paths record method. We propose a new method to record the skyline paths for each node which is possible on the skyline paths from s to t. Our new method can guarantee that the paths recorded for each node during the processing are exactly the skyline paths, which avoid the repetitious skyline paths maintenance in the existing solution.

(3) Extensive performance studies on real large networks. We conduct extensive

performance studies using real large networks. The experimental results show that PSQ^+ outperforms PSQ on all the networks used in our experiment. We also evaluate the afterwards proposed algorithm 2P [49] which uses PSQ as a building block for path skyline query problem. The experimental results on the real networks demonstrate that 2P can achieve a significant performance improvement after we substitute PSQ^+ for PSQ.

1.4 Graph Model and Preliminaries

In this thesis, we consider road network as a degree-bounded connected weighted graph $G = (V, E)$, where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges in G. We use V and E to denote $V(G)$ and $E(G)$ when the context is obvious. We denote the number of vertices $|V|$ and the number of edges $|E|$ by n and m respectively. In this thesis, all datasets are undirected graph. But the index of shortest path on dynamic road network is an undirected graph. On undirected graph, for each vertex $v \in V$, the neighbors of v, denoted as $N(v, G)$, is define as $N(v, G) = \{u|(u, v) \in E(G)\}$. We will use $N(v)$ to denote $N(v, G)$ when the context is obvious. Each edge $e = (u, v) \in E$ is associated with a positive integer $\phi(u, v)$. The degree of a vertex $u \in V(G)$, denoted by $\deg(u, G)$, is the number of neighbors of u, i.e., $\deg(u, G) = |N(v, G)|$. For bicriteria graph $G = (V, E)$, the definition is similar with normal road network. But for edge(u, v), two positive integer $\phi_1(u, v)$ and $\phi_2(u, v)$ are associated with this edge. For directed graph, we use N^+ and N^- to present out-neighbors and in-neighbors. We use deg^+ and deg^- to present out-degree and in-degree.

Base on the graph model definition above, in this thesis, we research 3 types of problem shown as follow.

Based on the aforementioned graph definition, we formally propose three

Symbol	**Description**
$N(v)$	the neigbors set of v
$N^+(v)$	the out-neighbors set of v
$N^-(v)$	the in-neighbors set of v
$deg(v)$	number of elments in neighbors set of v
$deg^+(v)$	number of elments in out-neighbors set of v
$deg^-(v)$	number of elments in in-neighbors set of v
d_{max}	the maximum degree of graph G
$dist_G(u,v)$	the distance of u and v in graph G

Table 1.1: Notations

research problems in this thesis.

- Given a graph $G = (V, E)$, two vertices $s \in V$ and $t \in V$, we aim to compute shortest path distance from s to t.
- Given a graph $G = (V, E)$, CH graph G' of G, weight update of edge $e \in E$ to k, we aim to compute the CH graph for graph G whose weight of e updated to k.
- Given a bicriteria graph $G = (V, E)$, two vertices $s \in V$ and $t \in V$, we aim to compute path skyline from s to t.

The rest of this chapter is organized as follows. Chapter 2 gives a literature review about the shortest distance, tree decomposition and bicriteria shortest path. Chapter 3 introduces the first work H2H index for the shortest path distance query. We propose a novel shortest distance query index Hierarchical 2-Hop index. Chapter 4 describes the second work shortest path query on dynamic road networks. We propose an efficient method to update Contraction Hierarchies index. Chapter 5 gives the third work path skyline query on bicriteria network. Chapter 6 concludes the thesis.

Chapter 2

LITERATURE REVIEW

In this section, we review the related work in shortest distance query, shortest path in dynamic road network and path skyline query. We classify the literature to shortest distance on road network Section 2.1, tree decomposition in Section 2.2, shortest distance in other networks in Section 2.3, bicriteria shortest path in Section 2.4 and shortest path in dynamic road network in Section 2.5.

2.1 Shortest distance on road networks.

Shortest path/distance queries are one of the most important types of queries on road networks. When answering shortest path/distance queries in a road network, both topological information and coordinate information are used to accelerate query processing. Existing solutions mainly focus on constructing an effective index to answer shortest path/distance queries. For example, ALT [28] accelerates the A* search by the pre-computation of some shortest distances. The hierarchical technique is an important category for this problem. HiTi [35] divides the graph and constructs a hierarchical structure to accelerate query processing. Highway Hierarchies [54] and Contraction Hierarchies [26] are good methods for pruning the search space. The contraction hierarchies approach relies on a total order of vertices. By adding shortcuts from low-ordered vertices to high-ordered vertices, it constructs a toplogical order of a graph and thus reduces the search space. The hop-based labeling approach is another important category for shortest distance queries on road networks. Of these, hub-based labeling algorithms are proposed in [1, 2] based on the 2-hop index; highway-based labeling [3] is another efficient hop-based algorithm based on highway decomposition, which divides the road network by shortest paths. Given a shortest distance query, hop-based labeling can answer the query using labels only, rather than searching the graph. Transit Node Routing [8], TRANSIT [11], and Arterial Hierarchy [75] are algorithms which integrate coordinate information. They divide the map into grids. Transit Node Routing creates an index which contains all the distances between different grids, TRANSIT precomputes the distances for each vertex to its closest transit node in the grid, and Arterial Hierarchy is an improved algorithm of the contraction hierarchies algorithm. Other works assume a road network as a planar graph without a negative weight, and use the Quadtree and path oracle [53, 55, 56, 43] on planar graphs to solve the shortest path/distance problem. A survey can be found in [10] and some experimental results for different algorithms to answer the shortest path and distance queries on road networks are reported in [68].

2.2 Tree decomposition and treewidth.

Tree decomposition, which is originally introduced by Halin [51] and rediscovered by Robertson and Seymour [52], is a mapping of a graph into a tree that can be used to define the treewidth of the graph and speed up solving certain computational problems on the graph. Many algorithmic problems, such as maximum independent set and Hamiltonian circuits that are NP-complete for arbitrary graphs, may be solved efficiently by dynamic programming for graphs of bounded treewidth, using the tree-decompositions of these graphs. An introductory survey by Bodlaender can be found in [13]. It is NP-complete to determine whether a given graph G has treewidth at most a given variable [6]. However, when the treewidth of a graph is a fixed constant, a tree decomposition for the graph with the minimum treewidth can be constructed in a time linear to the size of the graph but exponential to the treewidth [14]. This algorithm is only practical when the treewidth is very small (e.g., smaller than 10). Many heuristics are proposed to determine the treewidth of a graph, but unfortunately few of them can deal with large graphs (e.g., graphs with more than $1,000$ vertices) [36]. Because of the big challenges in computing the optimal tree decomposition, in this chapter, we adopt a suboptimal algorithm introduced in [15]. The algorithm does not guarantee to compute a treewidth with a bounded approximation ratio. Nevertheless, since the time complexities of both our index construction and query processing algorithms are polynomial to the treewidth of the tree decomposition, the algorithm in [15] is practically applicable for our problem in large real-world road networks.

2.3 Shortest Distance in Other Networks.

Shortest path distance query is also important in other networks such as scale-free networks and temporal networks. In scale-free networks, the most efficient method for distance query is also based on the 2-hop index. Pruned Landmark Labeling [4] and Hop-Doubling Labeling [34] are two state-of-the-art algorithms. Pruned Landmark Labeling calculates a vertex order, and computes the labels of vertices by pruning the landmarks following the vertex order. Hop-Doubling

Labeling regards edges as initial index and calculate the 2-hop labels by doubling the length of paths in each iteration. Tree decomposition is used in shortest distance problems [66, 5, 20, 70]. However, these approaches are not tailored for road networks and therefore cannot outperform the state-of-the-art algorithms such as Hub-based Labeling on road networks. There are also some works for temporal networks. For example, Timetable labeling [65] is an indexing algorithm to solve earlist arrival path query, latest departure path query and shortest duration path query. TopChain [67] improves Timetable Labeling by transforming the input graph into a directed acyclic graph. Some other works focus on shortest path problem with constrained condition such as [31] and [50].

2.4 Bicriteria Shortest Path Problem

Path skyline query in bicriteria networks is an NP-hard problem [58]. [44] suggests that in practical applications we can expect to find a reasonably small number of skyline paths. In the literature, the methods to solve the path skyline query in bicriteria networks can be divided into three subclasses: labelling methods, ranking methods and two-phases methods. Labelling methods [29, 22, 40, 17, 61] maintain a set of non-dominated solutions at each node and the identified skyline paths are represented by the labels at the destination node when the labelling methods finishes. Ranking methods [60, 41] first compute all paths having a length within a certain deviation from the length of the shortest path for one criterion. Then the skyline paths are computed based on the obtained paths by considering the other criterion. The advantage of methods in this subclass is that they do not have to compute the shortest paths from s to every other nodes in the network, but, on the other hand, it is proved to be not competitive, because they perform dominance test only at the destination node

[60, 49]. Two-phases methods [42, 64, 49] handle a path skyline query from s to t by adopting some pruning strategies. They contain two phases. In phase 1, they compute the so called supported skyline paths which are the skyline paths that can be obtained by solving a shortest path problem with a weighted sum as $\min_{P_{st} \in \mathcal{P}_{st}} \{\lambda_1 \cdot \phi_1(P_{st}) + \lambda_2 \cdot \phi_2(P_{st})\}$, where $\lambda_1 > 0, \lambda_2 > 0, \lambda_1 + \lambda_2 = 1$ and $\mathcal{P}_{st}$ represents all the paths from s to t. In phase 2, they enumerate the remaining skyline paths using the labelling methods. During the enumeration, if a path P_{sv} from source node s to a node v is dominated by a supported skyline path, then P_{sv} is pruned and the labelling procedure following this path can be terminated. When the labelling procedure finishes, the skylines can be obtained by the labels starting from destination node. Due to the introduction of supported skyline path and pruning strategies, two-phase methods are generally more efficient than the labelling methods and ranking methods for the path skyline query problem when the network is large and the two query nodes are located remotely w.r.t the cost criteria in the network [49].

In the literature, there exist some works focusing on the path skyline query in multicriteria networks. [37] proposes two algorithms based on different pruning strategies to address this problem. [33] discusses continuous skyline queries on road networks. [71] studies the path skyline query on the multicriteria time-dependent uncertain networks. Surveys on existing solutions to this problem can be found in [23, 63, 25]. Besides, [59, 57] study the efficient algorithms for the supported skyline path problem.

For the skyline operator, [16] first studies it in the context of databases and proposes an SQL syntax for the skyline query. After that, variations of the skyline operator have been explored, including skylines in distributed environments [9, 32], skylines in data streaming environments [39], skylines for partially ordered value domains [18], skyline cubes [48, 69, 72], approximate skylines [19] and skylines on uncertain data [47, 74].

2.5 Shortest path in dynamic road networks

Although many index-based methods for the shortest path queries are proposed, most of them assume the input road network is static and cannot support the dynamic road networks efficiently. The only exception is CH. In [27], an algorithm named $\mathsf{DCH_{vcs}}$ is proposed to maintain the index structure of CH for dynamic networks. We introduce it in Section 4.3 and use it as the baseline solution in our experiment. Besides $\mathsf{DCH_{vcs}}$, [73] studies the dynamic shortest path problem in distributed environments. Since it considers the distributed environments, its objective and proposed techniques are totally different from ours. [30] studies the dynamic edge-constrained short path query problem, which focuses on the social networks and considers the edges associated with multiple labels.

Chapter 3

H2H index for shortest path distance query

3.1 Overview

In this chapter, we aim at shortest distance problem on road network. We classify the existing solution to hiearchy-based solution and hop-based solution. However, hierarchy-based solution is efficient for short distance query. Hop-based solution is efficient for long distance query. To solve the drawback, we introduce our hierarchy-2-hop index solution for shortest path distance query on road network. This work is published in [45]. The rest of this chapter is organized as follow. Section 3.2 introduces our problem statement. Section 3.3 compares the hierarchy-based solution and hop-based solution. Section 3.4 proposes our basic idea of index using tree decomposition. Section 3.5 gives an improvment query processing on h2h index. Section 3.6 introduces our basic and improved index construction algorithms. Section 3.7 practically evaluates h2h-index and shows experimental results. Section 3.8 summarizes this chapter.

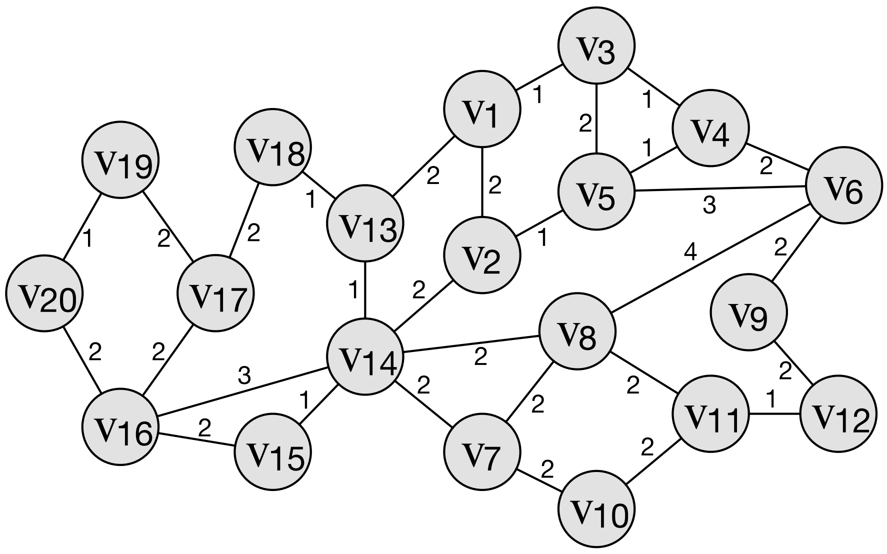


Figure 3.1: An Example of a Road Network G

3.2 Problem Statement

Given two vertices $s, t \in V$, a shortest path p between s and t is a path starting at s and ending at t such that $\phi(p)$ is minimized. The shortest distance of s and t in G, denoted as $dist_G(s, t)$, is the weight of any shortest path between s and t. We use $dist(s, t)$ to denote $dist_G(s, t)$ if the context is obvious. For each of explanation, we consider G as an undirected graph in this chapter, and our techniques can be easily extended to handle directed graphs.

Problem Definition. Given a road network G, a shortest distance query is defined as $q = (s, t)$ where $s, t \in V(G)$, and the answer of q is the shortest distance $dist(s, t)$. In this chapter, we aim to develop effective indexing techniques so that q can be answered efficiently.

Example 2. *Fig. 3.1 shows a road network* $G = (V, E)$ *with* 20 *vertices and* 30 *edges. The weight of each edge is marked beside the corresponding edge. For example, for edge* $(v_6, v_8) \in E$ *we have* $\phi(v_6, v_8) = 4$. *There are many paths* *between v_{12} and v_{19}. For example, two of them are $p_1 = (v_{12}, v_{11}, v_8, v_{14}, v_{13}, v_{18}, v_{17}, v_{19})$ and $p_2 = (v_{12}, v_9, v_6, v_5, v_2, v_{14}, v_{16}, v_{20}, v_{19})$ with $\phi(p_1) = 11$ and $\phi(p_2) = 18$. p_1 is a shortest path between v_{12} and v_{19}. Given a shortest distance query $q = (v_{12}, v_{19})$, the answer of q is $dist(v_{12}, v_{19}) = 11$.*

3.3 Existing Solutions

Given a shortest distance query $q = (s, t)$ on a road network $G = (V, E)$, we can use a traditional single source shortest algorithm (e.g., the Dijkstra Algorithm) to compute the shortest distance between s and t. Nevertheless, the algorithm is inefficient because it requires a time complexity of $O(m + n \cdot \log(n))$ and may traverse the whole network to answer one query in the worst case. Therefore, researchers resort to indexing based solutions to accelerate query processing. In the literature, the state-of-the-art indexing-based solutions for shortest distance query processing on road networks mainly fall into two categories, namely, hierarchy-based solution and hop-based solution. We briefly introduce them below.

3.3.1 Hierarchy-based Solution

The hierarchy-based solution is an indexing approach that imposes a hierarchy of all vertices in G by assigning each vertex v a rank $r(v)$. Based on the vertex hierarchy, it pre-computes the shortest distance among a subset of vertex pairs to reduce the computational cost of online query processing.

Contraction Hierarchies (CH). The hierarchy-based solution is first introduced by Geisberger et al. [26] in which an approach named Contraction Hierarchies (CH) is introduced. In the index construction phase, CH first assigns a total order of vertices in G as the vertex hierarchy by assigning each vertex v a rank $r(v)$. Then CH examines each vertex v_i following the total order. For each vertex v_i, CH visits the neighbors $N(v_i)$ of v_i in G. For each pair $v_j, v_k \in N(v_i)$, if the shortest path between v_j and v_k passes through v_i, an artificial edge (v_j, v_k) (referred to as a shortcut) is added into G with weight $\phi(v_j, v_k) = dist(v_j, v_k)$. Once all pairs of neighbors in v_i are processed, v_i is removed from the network G. The index construction phase terminates after all vertices are examined. The original network G along with all the shortcuts added in the index construction phase form the index of CH.

Given the CH index G_I which includes the network G along with the shortcuts, a shortest distance query $q = (s, t)$ can be answered using a modified bidirectional Dijkstra's algorithm. Specifically, the we start searching from both s and t simultaneously in a bidirectional manner using the Dijkstra's algorithm on the constructed graph G_I. Each time when expanding from a vertex u to vertex v, we only consider those edges (u, v) such that the rank of v is higher than that of u in G_I. When the termination condition of the bidirectional Dijkstra's algorithm is satisfied, the algorithm reports the current best distance as the answer to q. Since CH enforces the expansion direction to be only from lower-ranked vertices to higher-ranked vertices, the search space of CH is significantly reduced compared to the traditional bidirectional Dijkstra's algorithm when answering a shortest distance query q.

Arterial Hierarchy (AH). The Arterial Hierarchy (AH) approach is proposed by Zhu et al. [75]. The AH algorithm is inspired by CH, but it produces shortcuts by imposing the 4×4 grids on the network by exploiting some 2-dimensional spatial properties in the network. Query processing of AH is similar to that of CH which adopts the bidirectional Dijkstra's algorithm by enforcing the expansion direction to be only from lower-ranked vertices to higher-ranked vertices. AH is shown to be more efficient than CH when answering shortest distance queries

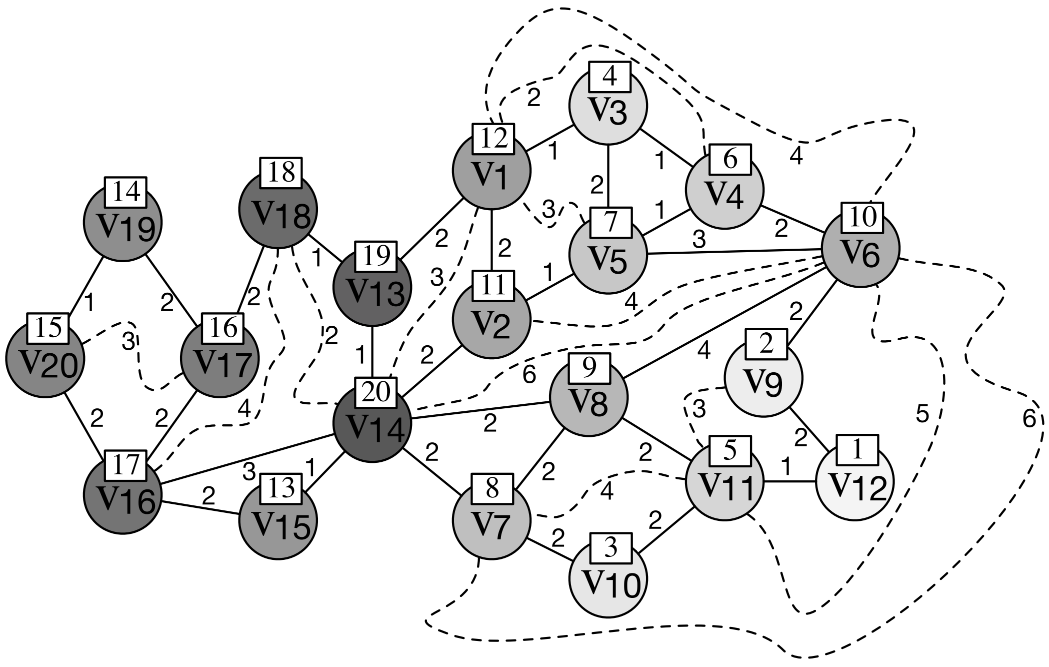


Figure 3.2: The index structure G_I of CH

since both network information and spatial information are utilized in AH. More details of the AH algorithm can be found in [75].

Example 3. *In this example, we illustrate the* CH *algorithm using the road network shown in Fig. 3.1. Suppose the vertex rank is assigned as* $v_{12} < v_9 < v_{10} < v_3 < v_{11} < v_4 < v_5 < v_7 < v_8 < v_6 < v_2 < v_1 < v_{15} < v_{19} < v_{20} < v_{17} < v_{16} < v_{18} < v_{13} < v_{14}$. *We mark the rank of each vertex in a box beside each vertex in Fig. 3.2, and we also use the darkness of each vertex to illustrate its rank: the darker color indicates a vertex with higher rank.*

In index construction, the CH *algorithm first examines* v_{12} *with the lowest rank. For its two neighbors* v_9 *and* v_{11}*, since the shortest path from* v_9 *to* v_{11} *(with weight* 3*) passes through* v_{12}*, we add a short cut* (v_9, v_{11}) *with weight* 3 *in* G_I*. Next, the vertex* v_9 *is examined, which has two neighbors* v_6 *and the newly added neighbor* v_{11} *(*v_{12} *is not neighbor of* v_9 *since it has been marked* *as removed). A shortcut* (v_6, v_{11}) *with weight* 5 *is added in* G_I. *The algorithm continues until all vertices are examined. The final index is shown in Fig. 3.2 where each dashed edge is a shortcut.*

With the index G_I, *given a shortest distance query* $q = (v_3, v_{10})$, *using the modified bidirectional Dijkstra's algorithm, we can expand from* v_3 *to* v_6 *through the path* (v_3, v_4, v_6) *with increasing vertex ranks and expand from* v_{10} *to* v_6 *through the path* (v_{10}, v_{11}, v_6) *with increasing vertex ranks where edge* (v_{11}, v_6) *is a shortcut, and thus obtain the shortest path from* v_3 *to* v_{10} *with* $dist(v_3, v_{10}) = 10$.

3.3.2 Hop-based Solution

Given a road network $G(V, E)$, the hop-based solution aims to assign each vertex $v \in V$ a 2-hop label $L(v)$ which is a collection of pairs $(u, dist(v, u))$ where $u \in V$. Given two vertices $s, t \in V$, the shortest distance of s, t can be calculated as follows:

$$dist(s,t) = \min_{u \in L(s) \cap L(t)} dist(s,u) + dist(u,t) \tag{3.1}$$

In other words, the hop-based solution answers a query $q = (s, t)$ by considering all common vertices in the labels of s and t. For each such common vertex u, a distance is calculated using 2 hops from s to u and from u to t. The labels is assigned in a way to guarantee that the minimum of the two-hop distances for all $u \in L(s) \cap L(t)$ is the shortest distance from s to t. In the literature, two popular hop-based approaches are hub-based labeling approach [1, 2] and Pruned Highway labeling approach [3].

Hub-based Labeling (HL). One of the most efficient hop-based solution for shortest distance query processing in a road network is hub-based labeling proposed by Abraham et al. [1, 2]. The algorithm aims to use the hub vertices as the label of a vertex. It is shown that for road networks, the effective label

V	2-Hop Label
v_1	$\{(v_1,0),(v_2,2),(v_{13},2),(v_{14},3)\}$
v_2	$\{(v_2,0),(v_{14},2)\}$
v_3	$\{(v_1,1),(v_2,3),(v_3,0),(v_4,1),(v_5,2),(v_6,3),(v_{13},3),(v_{14},4)\}$
v_4	$\{(v_1,2),(v_2,2),(v_4,0),(v_5,1),(v_6,2),(v_{13},4),(v_{14},4)\}$
v_5	$\{(v_1,3),(v_2,1),(v_5,0),(v_6,3),(v_{14},3)\}$
v_6	$\{(v_1,4),(v_2,4),(v_6,0),(v_{13},6),(v_{14},6)\}$
v_7	$\{(v_6,6),(v_7,0),(v_8,2),(v_{14},2)\}$
v_8	$\{(v_6,4),(v_8,0),(v_{14},2)\}$
v_9	$\{(v_1,6),(v_2,6),(v_6,2),(v_7,7),(v_8,5),(v_9,0),(v_{11},3),(v_{13},8),(v_{14},7)\}$
v_{10}	$\{(v_6,7),(v_7,2),(v_8,4),(v_{10},0),(v_{11},2),(v_{14},4)\}$
v_{11}	$\{(v_6,5),(v_7,4),(v_8,2),(v_{11},0),(v_{14},4)\}$
v_{12}	$\{(v_1,8),(v_6,4),(v_7,5),(v_8,3),(v_9,2),(v_{11},1),(v_{12},0),(v_{14},5)\}$
v_{13}	$\{(v_{13},0),(v_{14},1)\}$
v_{14}	$\{(v_{14},0)\}$
v_{15}	$\{(v_{14},1),(v_{15},0),(v_{16},2)\}$
v_{16}	$\{(v_{14},3),(v_{16},0),(v_{18},4)\}$
v_{17}	$\{(v_{13},3),(v_{14},4),(v_{16},2),(v_{17},0),(v_{18},2)\}$
v_{18}	$\{(v_{13},1),(v_{14},2),(v_{18},0)\}$
v_{19}	$\{(v_{13},5),(v_{14},6),(v_{16},3),(v_{17},2),(v_{18},4),(v_{19},0),(v_{20},1)\}$
v_{20}	$\{(v_{13},6),(v_{14},5),(v_{16},2),(v_{17},3),(v_{18},5),(v_{20},0)\}$

Table 3.1: 2-hop Labels for the road network in Fig. 3.1

of a vertex $v \in V(G)$ can be obtained from the upward search space of a CH query by considering u as a source vertex. The quality of the labeling depends on the vertex order for CH. The authors observe that a good order order can be obtained by greedily picking the vertices that hit the most number of shortest paths.

Pruned Highway Labeling (PHL). The Pruned Highway labeling approach is proposed by Akiba et al. [3]. In this work, instead of using hubs as the labels for each vertex, the algorithm uses paths as the labels aiming to encode more information in each label. In the indexing phase, the algorithms decomposes the road network into disjoint shortest paths, and then computes a label for each vertex v which contains the distance from v to vertices in a small subset of the computed paths. It guarantees that any shortest distance query $q = (s, t)$ can be answered by hopping from s to a path in $L(s) \cap L(t)$ and then hopping from the path to t. The pruned labeling technique [4] are used in indexing to reduce the label size.

Example 4. *Table 3.1 shows the 2-hop labels for the road network in Fig. 3.1 computed using hub-based labeling algorithm. The label of* v_2 *is* $L(v_2) = \{(v_2, 0), (v_{14}, 2)\}$ *which indicates that* $dist(v_2, v_2) = 0$ *and* $dist(v_2, v_{14}) = 2$*. To answer the shortest distance* $q = (v_5, v_{12})$*, according to Eq. 3.1, we can first compute* $L(v_5) \cap L(v_{12}) = \{v_1, v_6, v_{14}\}$*. Then the shortest distance between* v_5 *and* v_{12} *can be calculated as* $dist(v_5, v_{12}) = \min\{dist(v_5, v_1) + dist(v_{12}, v_1), dist(v_5, v_6) + dist(v_{12}, v_6), dist(v_5, v_{14}) + dist(v_{12}, v_{14})\} = \min\{3 + 8, 3 + 4, 3 + 5\} = 7$*.*

3.4 The Basic Idea

3.4.1 Problem Analysis.

In this section, we analyze the drawbacks of the hierarchy-based solution and the hop-based solution.

Hierarchy-based Solution. The hierarchy-based solution defines a hierarchy for vertices in a road network. To answer a shortest distance query $q = (s, t)$, the hierarchy-based solution explores the road network from s and t simultaneously following the predefined order in the hierarchical index, and thus significantly reduces the search space compared to the strategy by searching from scratch on the road network. The algorithm is efficient for the case when s and t are near. Nevertheless, for a long-distance query $q = (s, t)$, i.e., s is far away from t, the algorithm may still spend a large among of search space and computational cost when searching the hierarchical index bidirectionally. Below, we show an example to demonstrate the above cases.

Example 5. *Consider the road network in Fig. 3.1, the hierarchy-based index* G_I *is shown in Fig. 3.2. To compute the shortest distance from* v_1 *to* v_6*, we only need to search one hop-neighbors from* v_1 *and* v_6 *following the hierarchical order and compute* $dist(v_1, v_6)$ *efficiently using the shortcut* (v_1, v_6) *with weight* 4*.*

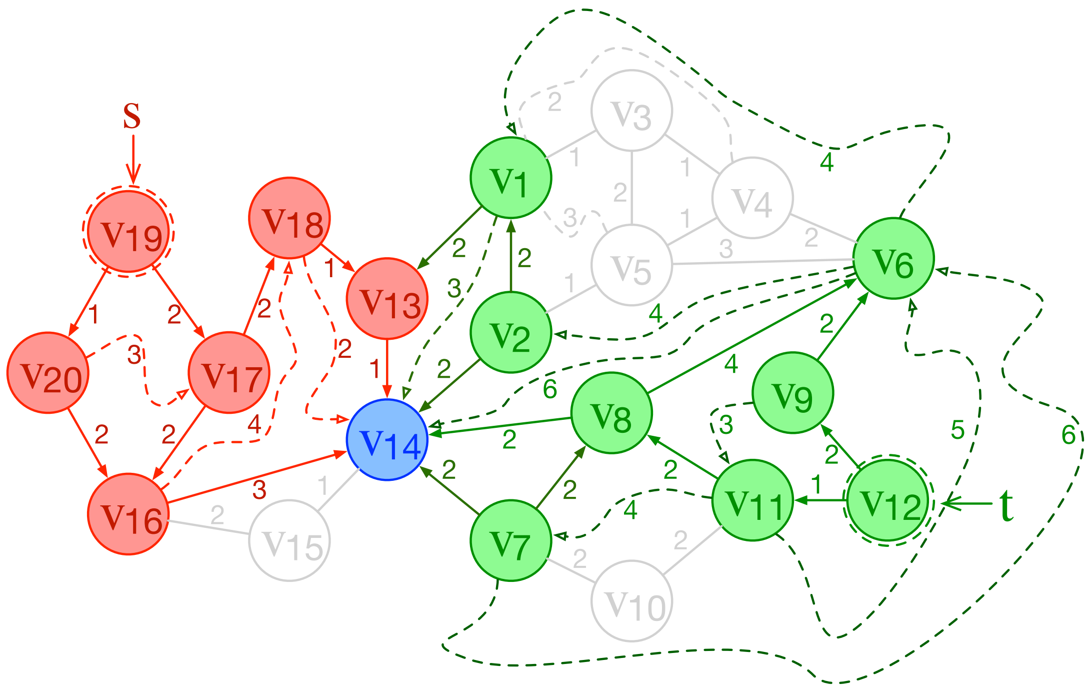


Figure 3.3: The Search Space for CH ($q = (v_{19}, v_{12})$)

However, when we compute the shortest distance from v_{19} to v_{12}, the search space is shown in Fig. 3.3. The vertices and edges marked in the red color indicate the search space from the source vertex v_{19} and the vertices and edges marked in the blue color indicate the search space form the target vertex v_{12}. As illustrated in Fig. 3.2, since v_{19} is far away from v_{12}, more than half of the vertices and edges (including shortcuts) have to be visited during the search, which is costly.

Hop-based Solution. Given a road network G, the hop-based solution assigns a 2-hop label $L(v)$for each $v \in V(G)$. To answer a shortest distance query $q = (s, t)$, the algorithm simply joins $L(s)$ and $L(t)$ to compute the answer to the query. For a long-distance query, the algorithm avoids online searching the intermediate vertices in the path from s to t and therefore is much more efficient compared to the hierarchy-based solution. Nevertheless, for each vertex, the algorithm uses the same label to answer all queries with the vertex. When the distance between s and t is short, the algorithm may result in unnecessary computations when the label sizes of s and t are large. Below, we show an example to demonstrate these situations.

Example 6. *Consider the road network in Fig. 3.1, the 2-hop label $L(v)$ for each vertex v on the road network is shown in Table 3.1. To answer the long-distance query $q = (v_6, v_{16})$, we only need to consider the* 5 *vertices in $L(v_6)$ and* 3 *vertices in $L(v_{16})$, with a smaller search space compared to the hierarchy-based solution. However, to answer the short-distance query $q = (v_9, v_{12})$, we need to consider the* 9 *vertices in $L(v_9)$ and the* 8 *vertices in $L(v_{12})$ with a large amount of useless computations, whereas in the hierarchy-based solution we only need to explore the neighbor of v_{12} to answer the query.*

Our Solution. Based on the above analysis, we can see that the hierarchy-based solution is inefficient to answer long-distance queries and the hop-based solution may result in large useless computations when answering short-distance queries. To handle both cases more efficiently, in this chapter, we propose a hierarchical 2-hop index. Our general idea is as follows: we assign each vertex v a 2-hop label $L(v)$ and we also assign a hierarchy for all vertices on the road network. Given a shortest distance query $q = (s, t)$, instead of using all vertices in $L(s)$ and $L(t)$ to answer the query, we only pick up a subset of vertices in $L(s)$ and $L(t)$ to answer the query according to the hierarchies of s and t. In general, when s and t are far away from each other, most vertices in $L(s)$ and $L(t)$ are selected, and when s and t are near each other, only a small part of $L(s)$ and $L(t)$ are selected. In this way, we can overcome the drawbacks of both hierarchy-based solution and hop-based solution.

To make the idea practically applicable, the following issues need to be addressed: (1) how to find an appropriate hierarchy for all vertices, and (2) how to assign labels for each vertex so that the hierarchy can be used to efficiently answer the shortest distance queries. We will answer the questions in the following sections.

3.4.2 Vertex Hierarchy by Tree Decomposition

Tree decomposition, which is originally introduced by Halin [51] and rediscovered by Robertson and Seymour [52], is a way to map a graph in to a tree to speed up solving certain computational problems in graphs. In this chapter, we use tree decomposition to define the vertex hierarchy, and we will show that the hierarchy is effective to answer shortest distance queries in a road network. Given a road network $G(V, E)$, a tree decomposition of G is defined as follows:

Definition 1. ***(Tree Decomposition)*** *A tree decomposition of a graph $G(V, E)$, denoted as T_G, is a rooted tree in which each node $X \in V(T_G)$ is a subset of $V(G)$ (i.e., $X \subseteq V(G)$) such that the following three conditions hold:*

1. $\bigcup_{X \in V(T_G)} X = V$;
2. *For every $(u, v) \in E(G)$, there exists $X \in V(T_G)$ s.t. $u \in X$ and $v \in X$;*
3. *For every $v \in V(G)$ the set $\{X | v \in X\}$ forms a connected subtree of T_G.*

For any $v \in V(G)$, we use $T(v)$ to denote the subtree induced by the set $\{X | v \in X\}$ in T_G, and use $X(v)$ to denote the root node of $T(v)$ in T_G.

For ease of presentation, we refer to each $v \in V(G)$ on the road network G as a *vertex* and refer to each $X \in V(T_G)$ in the tree decomposition T_G as a *node*. We consider T_G as a rooted tree by picking up a node as a root. Below, we show an example to illustrate the tree decomposition of a road network.

Example 7. *Fig. 3.4 shows a tree decomposition T_G of the road network G in Fig. 3.1. There are* 20 *nodes in T_G. One of the nodes containing* 3 *vertices is*

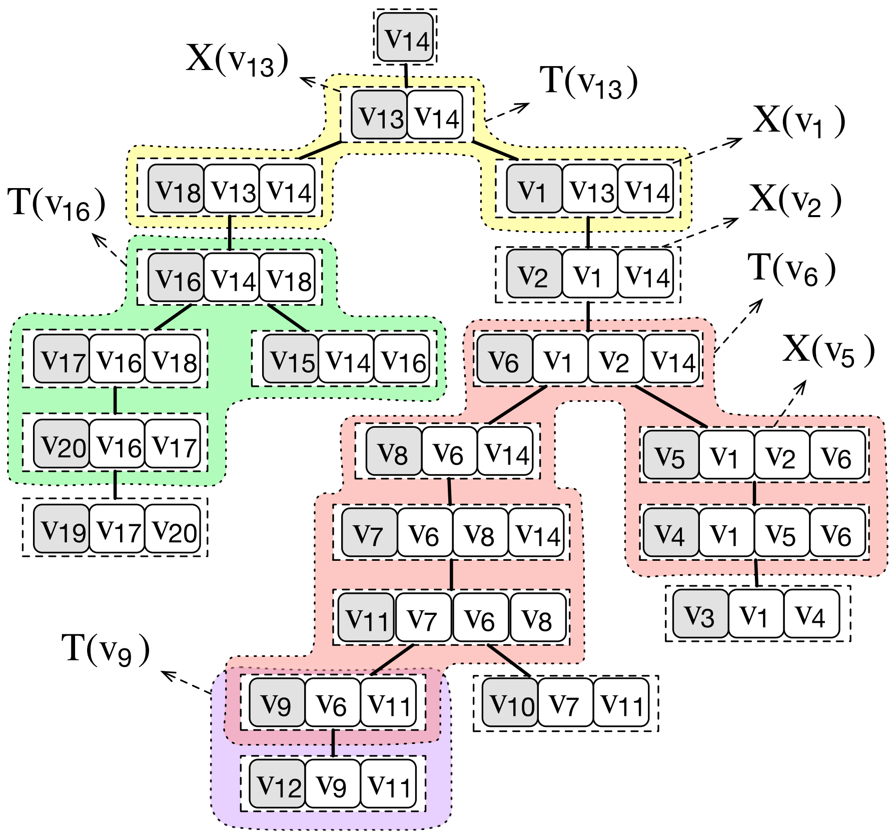


Figure 3.4: Tree Decomposition T_G of the Network in Fig. 3.1

$\{v_{18}, v_{13}, v_{14}\}$. *For the edge* $(v_{17}, v_{18}) \in E(G)$, *there is a node* $\{v_{17}, v_{16}, v_{18}\}$ *in* $V(T_G)$ *containing both* v_{17} *and* v_{18}. *For the vertex* $v_{13} \in V(G)$, *there are three nodes in* $V(T_G)$ *containing* v_{13}. *The three nodes form a subtree* $T(v_{13})$ *as marked by the yellow area in the figure. The root of* $T(v_{13})$ *is denoted as* $X(v_{13})$. *We also mark* $T(v_6)$, $T(v_9)$, $T(V_{16})$, $X(v_1)$, $X(v_2)$, *and* $X(v_5)$ *in the figure.*

Definition 2. ***(Tree Width and Tree Height)*** *Given a tree decomposition* T_G *of graph* G, *the tree width of* T_G, *denoted as* $w(T_G)$ *is the maximum size of all nodes in* T_G, *i.e.,* $w(T_G) = \max_{X \in T_G} |X|$. *The tree height of* T_G, *denoted as* $h(T_G)$, *is the maximum depth of all nodes in* T_G. *Here the depth of a node in* T_G *is the distance from the node to the root node of* T_G. *If the context is obvious, we* *will use w and h to denote the tree width and tree height of the tree decomposition T_G respectively.*

Example 8. *For the road network G in Fig. 3.1, a tree decomposition T_G is shown in Fig. 3.4. The tree width $w(T_G)$ of T_G is 4 since each node in T_G contains at most 4 vertices. The tree high $h(T_G)$ of T_G is 10.*

It is important to note that in a road network, we can obtain a tree decomposition with low tree width and tree height values. For example, on the road network for the City of New York with $264,346$ vertices and $733,846$ edges, we can obtain a tree decomposition with tree width only 132 and tree height only 320. The tree width and tree height values for more road networks including the road network for the whole USA are shown in Section 3.7.

Tree Decomposition Computation. We briefly introduce the tree decomposition algorithm which is proposed in [7]. The pseudocode of the algorithm is shown in Algorithm 1. The algorithm consists of two phases.

- In the first phase (line 2-8), all nodes in T_G are created. We create a fill-in graph H which is initialized as G. Each time, we select a vertex v in H with the smallest degree and create a node in T_G consisting of v and all its neighbors in H (line 3-5). We add edges into H to make all neighbors of v in H connected to each other and then remove v and its adjacent edges from H (line 6-7). We use π to denote a total order of vertices removed from H.

- In the second phase (line 9-12), all edges in T_G are created. For every node $X(v)$ with size larger than 1, i.e., $X(v)$ is a non-root node, we find a vertex u in $X(v) \setminus \{v\}$ with the smallest π value (line 11) and simply assign the parent of $X(v)$ to be $X(u)$ (line 12).

Algorithm 1 TreeDecomposition$(G(V,E))$

Input: A road network $G(V,E)$;
Output: Tree decomposition T_G.

```
1:  $H \leftarrow G$; $T_G \leftarrow \emptyset$;
2:  for $i = 1$ to $|V|$ do
3:      $v \leftarrow$ the node in $H$ with smallest degree;
4:      $X(v) \leftarrow \{v\} \cup N(v,H)$;
5:      Create a node $X(v)$ in $T_G$;
6:      Add edges to $H$ to make every pair of vertices in $N(v,H)$ connected to
        each other in $H$;
7:      Remove $v$ and its adjacent edges from $H$;
8:      $\pi(v) \leftarrow i$;
9:  for each $v \in V(G)$ do
10:     if $|X(v)| > 1$ then
11:         $u \leftarrow$ the vertex in $X(v) \setminus \{v\}$ with smallest $\pi$ value;
12:         Set the parent of $X(v)$ be $X(u)$ in $T_G$;
13: return $T_G$;
```

Note that the algorithm guarantees that every $X(v)$ is unique, i.e., for any $v \neq u$, we have $X(v) \neq X(u)$. Therefore, there is a one-to-one mapping from $V(G)$ to $V(T_G)$. In this chapter, we assume this property holds for a tree decomposition T_G for the ease of presentation. The time complexity of Algorithm 1 is $O(n \cdot (w^2 + \log(n)))$. Below, we use an example to demonstrate the process of tree decomposition.

Example 9. *Suppose we compute the tree decomposition for the road network shown in Fig. 3.1 using Algorithm 1. In the first phase, we first choose v_{12}, and create a node $X(v_{12}) = \{v_{12}, v_9, v_{11}\}$ including v_{12} and its neighbors. We then create an edge between v_9 and v_{11} and remove v_{12}. We then choose v_9. Now v_9 has an original neighbor v_6 and a newly added neighbor v_{11}. Therefore, we create a node $X(v_9) = \{v_9, v_6, v_{11}\}$, add an edge between v_6 and $v_1 1$ and then remove v_9. The process continues until all vertices are removed. In the second phase, we create edges in T_G. For the node $X(v_{12})$, the node with the smallest order in*

$X(v_{12}) \setminus \{v_{12}\}$ is v_9. Therefore, we assign the parent of $X(v_{12})$ to be $X(v_9)$. The final tree decomposition result is shown in Fig. 3.4.

3.4.3 A Naive Solution

In this subsection, we introduce a straightforward solution to solve the shortest distance queries using the tree decomposition based on the concept of a vertex cut.

Definition 3. ***(Vertex Cut)*** *Given a road network $G(V, E)$, a subset of vertices $C \subset V$ is a vertex cut of G if the deletion of C from G splits G into multiple connected components. A vertex set C is called the vertex cut of vertices u and v if u and v are in different connected components by the deletion of C from G.*

Example 10. *For the road network G shown in Fig. 3.1, the set $C = \{v_{13}, v_{14}\}$ is a vertex cut of G. C is a vertex cut of vertices v_1 and v_{18}, but it is not a vertex cut of v_1 and v_2. Because after the removal of C from G, v_1 and v_{18} are in different connected components while v_1 and v_2 are in the same connected component.*

Given a vertex cut C for two vertices s and t, it is obvious that every path from s to t should contain at least one vertex in C. Therefore, we can easily derive the following theorem:

Theorem 1. *Given a road network G, let C be a vertex cut for two vertices s and t, we have:*

$$dist(s,t) = \min_{v \in C} dist(s,v) + dist(v,t)$$

According to Theorem 1, if we can obtain a small cut C for vertices s and t and precompute the distances from s to the vertices in C and from the vertices in C to t, we can calculate $dist(s,t)$ efficiently. Below, we show that the tree decomposition T_G can be used to complete the task based on the following cut property [20] of a tree decomposition.

Property 1. *Given a tree decomposition T_G for a road network G, for any two vertices s and t in $V(G)$, suppose $X(s)$ is not an ancestor/decedent of $X(t)$ in T_G, let X be the lowest common ancestor (LCA) of $X(s)$ and $X(t)$ in T_G, then X is a vertex cut of s and t in G.*

Note that using the techniques in [12], we can use $O(1)$ time to compute the LCA of any pair of nodes in a tree T_G using $O(n)$ index space.

Example 11. *For the road network G (Fig. 3.1) and its tree decomposition T_G shown in Fig. 3.4, given two vertices v_3 and v_9, the lowest common ancestor of $X(v_3)$ and $X(v_9)$ in T_G is $X(v_6) = \{v_6, v_1, v_2, v_{14}\}$. Therefore, $\{v_6, v_1, v_2, v_{14}\}$ is a vertex cut of v_3 and v_9 in G. According to Theorem 1, we can get $dist(v_3, v_9) = \min\{dist(v_3, v_6) + dist(v_6, v_9), dist(v_3, v_1) + dist(v_1, v_9), dist(v_3, v_2) + dist(v_2, v_9), dist(v_3, v_{14}) + dist(v_{14}, v_9)\} = 5$.*

The Naive Solution. Based on Property 1 and Theorem 1, given a tree decomposition T_G of a road network G, we can devise a straightforward solution to answer shortest distance queries as follows:

- *Indexing.* In the indexing phase, for each $v \in V(G)$ and each ancestor $X(u)$ of $X(v)$ in T_G, we precompute the shortest distance $dist(v, w)$ for any $w \in X(u)$ and index them using a hash table. For each vertex v, at most $h \times w$ shortest distances are computed. Therefore, the total index size is bounded by $O(n \cdot h \cdot w)$.

- *Query Processing.* In the query processing phase, given a query $q = (s, t)$, we can first compute the LCA X of $X(s)$ and $X(t)$ in T_G using $O(1)$

time [12]. Then we can compute $dist(s,t)$ according to Property 1 and Theorem 1. i.e.,

$$dist(s,t) = \min_{v \in X} \; dist(s,v) + dist(v,t) \tag{3.2}$$

Here $dist(s,v)$ and $dist(v,t)$ for every $v \in X$ have been precomputed in the indexing phase. Obviously, the query processing time is $O(c \cdot w)$ where c is the time cost to look up the distance for a specific pair of vertices in the hash table.

3.5 Query Processing by H2H-Index

In this section, we first analyze the drawbacks of the naive solution, followed by the introduction of a more effective indexing and query processing mechanism.
Drawbacks of the Naive Solution. Below, we show the drawbacks for the naive solution in terms of index size and query processing time.

- *Index Size.* The index size of $O(n \cdot h \cdot w)$ for the naive solution can be very large, which makes the approach impractical to handle large road networks, such as the whole USA road network.

- *Query Processing Time.* Although the query processing complexity of the naive solution is $O(c \cdot w)$ where c is a constant theoretically, it can be a large value and the query processing may require random accesses in the hash table. Therefore, the query processing of the naive approach is inefficient in practice.

Hierarchical 2-Hop Index (H2H-Index). To overcome the above drawbacks, we introduce a new index structure called hierarchical 2-hop index (H2H-Index). The index is designed based on the following property:

Property 2. *Given a road network G, its tree decomposition T_G, and an arbitraty node $X(u) \in V(T_G)$, for any $v \in X(u) \setminus \{u\}$, $X(v)$ is an ancestor of $X(u)$ in T_G.*

Proof Sketch: According to condition 3 of Definiton 1, the nodes containing v is a connected subtree $T(v)$ in T_G, and the root of the subtree is $X(v)$ according to Definiton 1. Since $X(v)$ is an ancestor of any non-root node in $T(v)$, $X(v)$ is an ancestor of $X(u)$ in $T(v)$. As a result, $X(v)$ is an ancestor of $X(u)$ in T_G. □

Next, we define the ancestor array for a node in a tree decomposition as follows:

Definition 4. ***(Ancestor Array)*** *Given a tree decomposition T_G of G, for any node $X(v) \in V(T_G)$, let $(X(w_1), X(w_2), \ldots, X(w_l))$ be the path from the root of T_G to $X(v)$ in T_G, here $X(w_1)$ is the root of T_G and $X(w_l)$ is $X(v)$. The ancestor array of $X(v)$, denoted as $X(v).anc$ is defined as $X(v).anc = (w_1, w_2, \ldots, w_l)$. We use $X(v).anc_i$ to denote the i-th element in $X(v).anc$ for any $1 \le i \le l$.*

Example 12. *Given the tree decomposition T_G shown in Fig. 3.4, for the node $X(v_7) = \{v_7, v_6, v_8, v_{14}\}$, according to Property 2, the nodes $X(v_6)$, $X(v_8)$, and $X(v_{14})$ are all ancestors of $X(v_7)$ in T_G. The ancestor array of $X(v_7)$ is $X(v_7).anc = (v_{14}, v_{13}, v_1, v_2, v_6, v_8, v_7)$.*

Based on Property 2 and Definiton 4, we now define the structure of the H2H-Index. The H2H-Index is defined on top of the tree decomposition T_G of road network G. For each node $X(v) \in V(T_G)$, suppose $X(v) = \{u_1, u_2, \ldots, u_k\}$ and $X(v).anc = (w_1, w_2, \ldots, w_l)$, according to Property 2, $X(v)$ is a subset of $X(v).anc$, i.e., $X(v) \subseteq X(v).anc$. The H2H-Index for $X(v)$ consists of two components: distance array and position array.

- *Distance Array:* the distance array for $X(v)$, denoted as $X(v).dis$, is an array defined as $X(v).dis = (dist(v, w_1), dist(v, w_2), \ldots, dist(v, w_l))$. In other words, the distance array of $X(v)$ is the array of distances from v to every vertex in $X(v).anc$. We use $X(v).dis_i$ to denote the i-th value in $X(v).dis$ for any $1 \leq i \leq l$. We have $X(v).dis_i = dist(v, X(v).anc_i)$.

- *Position Array:* the position array for $X(v)$, denoted as $X(v).pos$, is an array defined as $X(v).pos = (p_1, p_2, \ldots, p_k)$ where $p_i (1 \leq i \leq k)$ is the position of u_i in $X(v).anc$, i.e., $X(v).anc_{p_i} = u_i$. For efficiency consideration, we sort values in $X(v).pos$ in their increasing order. We use $X(v).pos_i$ to denote the i-th value in $X(v).pos$ for any $1 \leq i \leq k$.

Example 13. *Fig. 3.5 shows the H2H-Index for the road network in Fig. 3.1 based on the tree decomposition shown in Fig. 3.4. For the node* $X(v_7) = \{v_7, v_6, v_8, v_{14}\}$ *with* $X(v_7).anc = (v_{14}, v_{13}, v_1, v_2, v_6, v_8, v_7)$*, we can calculate the distance array of* $X(v_7)$ *as:* $X(v_7).dis = (dist(v_7,\ v_{14}),\ dist(v_7,\ v_{13}),\ dist(v_7,\ v_1),\ dist(v_7,\ v_2),\ dist(v_7,\ v_6),\ dist(v_7,\ v_8),\ dist(v_7,\ v_7)) = (2, 3, 5, 4, 6, 2, 0)$*. We can calculate the position array of* $X(v_7)$ *as* $X(v_7).pos = (7, 5, 6, 1)$ *since* v_7*,* v_6*,* v_8 *and* v_{14} *are the* 7*-th,* 5*-th,* 6*-th, and* 1*-st elements in* $X(v_7).anc$ *respectively. After sorting, we get* $X(v_7).pos = (1, 5, 6, 7)$.

The following theorem shows the space complexity of the H2H-Index. We will introduce how to construct the H2H-Index efficiently in the next section.

Theorem 2. *The space complexity of H2H-Index for the road network* G *is* $O(n \cdot h)$*, where* h *is the tree height of the tree decomposition* T_G.

Proof Sketch: The space complexity of H2H-Index depends on the size of the two components distance array and position array. For each node $X(v) \in T_G$, the size of $X(v).dis$ is no larger than l since the path from the root to $X(v)$ in T_G is no larger than l. The size of $X(v).pos$ is no larger than h since $X(v).pos$ is a

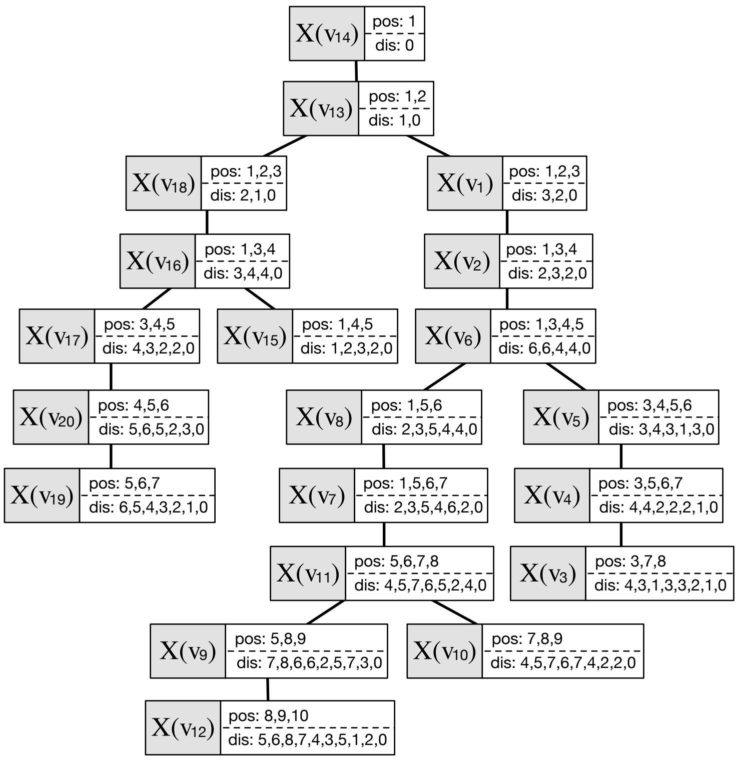

X(v14) pos: 1 dis: 0
X(v13) pos: 1,2 dis: 1,0
X(v18) pos: 1,2,3 dis: 2,1,0
X(v1) pos: 1,2,3 dis: 3,2,0
X(v16) pos: 1,3,4 dis: 3,4,4,0
X(v2) pos: 1,3,4 dis: 2,3,2,0
X(v17) pos: 3,4,5 dis: 4,3,2,2,0
X(v15) pos: 1,4,5 dis: 1,2,3,2,0
X(v6) pos: 1,3,4,5 dis: 6,6,4,4,0
X(v20) pos: 4,5,6 dis: 5,6,5,2,3,0
X(v8) pos: 1,5,6 dis: 2,3,5,4,4,0
X(v5) pos: 3,4,5,6 dis: 3,4,3,1,3,0
X(v19) pos: 5,6,7 dis: 6,5,4,3,2,1,0
X(v7) pos: 1,5,6,7 dis: 2,3,5,4,6,2,0
X(v4) pos: 3,5,6,7 dis: 4,4,2,2,2,1,0
X(v11) pos: 5,6,7,8 dis: 4,5,7,6,5,2,4,0
X(v3) pos: 3,7,8 dis: 4,3,1,3,3,2,1,0
X(v9) pos: 5,8,9 dis: 7,8,6,6,2,5,7,3,0
X(v10) pos: 7,8,9 dis: 4,5,7,6,7,4,2,2,0
X(v12) pos: 8,9,10 dis: 5,6,8,7,4,3,5,1,2,0

Figure 3.5: The H2H-Index for the Road Network in Fig. 3.1

subset of $X(v).anc$, and the size of $X(v).anc$ is no larger than h. Consequently, the size of the H2H-Index is bounded by $O(n \cdot h)$. □

Since the tree hight h for a road network is usually small, the space $O(n \cdot h)$ is practical for large real road networks.

Query Processing using H2H-Index. We now show how to use the H2H-Index to process a shortest distance query $q = (s, t)$. The algorithm is based on the following theorem:

Theorem 3. *Given the tree decomposition T_G of a road network G, the H2H-Index, and a query $q = (s, t)$, let X be the LCA of $X(s)$ and $X(t)$ in T_G, we have:*

$$dist(s, t) = \min_{i \in X.pos} X(s).dis_i + X(t).dis_i$$

Proof Sketch: Based on the definition of H2H-Index, suppose $X = \{u_1, u_2, \ldots, u_k\}$ and elements in X are sorted in increasing order of their positions in $X.anc$. For any decedent $X(v)$ of X in $T(G)$, $X.anc$ is a prefix of $X(v).anc$, i.e., $X.anc_i = X(v).anc_i$ for any $1 \leq i \leq |X.anc|$. Therefore, we have $dist(v, X.anc_i) = dist(v, X(v).anc_i) = X(v).dis_i$ for any $1 \leq i \leq |X.anc|$. According to Theorem 1 and Property 1, we know that $dist(s,t) = \min_{v \in X} dist(s,v) + dist(v,t)$. Since both $X(s)$ and $X(t)$ are decedents of X in T_G, we can derive that:

$$\begin{aligned} dist(s,t) &= \min_{v \in X} dist(s,v) + dist(t,v) \\ &= \min_{i \in X.pos} dist(s, X.anc_i) + dist(t, X.anc_i) \\ &= \min_{i \in X.pos} dist(s, X(s).anc_i) + dist(t, X(t).anc_i) \\ &= \min_{i \in X.pos} X(s).dis_i + X(t).dis_i \end{aligned}$$

Therefore, the theorem holds. □

With Theorem 3, we are now ready to design the query processing algorithm. Given a shortest distance query $q = (s,t)$, the algorithm to answer q is shown in Algorithm 2. The algorithm first computes the LCA of $X(s)$ and $X(t)$ in T_G using $O(1)$ time [12] (line 1), then based on Theorem 3, we can answer q using only the distance array of $X(s)$ and $X(t)$ and position array of X (line 2-5). Below, we show the time complexity of Algorithm 2.

Theorem 4. *The time complexity of query processing using Algorithm 2 in a road network G is $O(w)$, where w is the tree width of the tree decomposition T_G.*

Proof Sketch: First, line 1 costs $O(1)$ time according to [12]. Second, line 2-4 costs $O(w)$ time since $|X.pos| \leq w$ in the tree decomposition T_G. Therefore, the overall time complexity of Algorithm 2 is $O(w)$. □

It is important to note that compared to the naive query processing algorithm

Algorithm 2 H2H-Query(T_G, H2H-Index, $q = (s, t)$)

```
Input:  the tree decomposition $T_G$, the H2H-Index,
        and the query $q = (s, t)$;
Output: the shortest distance $dist(s, t)$.
1: $X \leftarrow$ the LCA of $X(s)$ and $X(t)$ in $T_G$;
2: $d \leftarrow +\infty$;
3: for each $i \in X.pos$ do
4:     $d \leftarrow \min(d, X(s).dis_i + X(t).dis_i)$;
5: return $d$;
```

introduced in Section 3.4.3, in Algorithm 2, we can avoid looking up vertices using hash tables, and instead, we only need to sequentially scan $X.pos$, $X(s).dis$ and $X(t).dis$ to compute the result, since the positions in $X.pos$ are sorted in their increasing order. Therefore, Algorithm 2 is much more efficient than the naive algorithm. Below, we use an example to demonstrate the query processing algorithm.

Example 14. *Given the H2H-Index shown in Fig. 3.5 for the road network G, the query processing for queries $q = (v_{12}, v_3)$ and $q = (v_{12}, v_{19})$ is shown in Fig. 3.6 (a) and Fig. 3.6 (b) respectively. For query $q = (v_{12}, v_3)$, we first get LCA(v_{12}, v_3) $= X(v_6)$ in T_G. Since $X(v_6).pos = (1, 3, 4, 5)$, we can compute $dist(v_{12}, v_3) = \min(X(v_{12}).dis_1 + X(v_3).dis_1,\ X(v_{12}).dis_3 + X(v_3).dis_3,$ $X(v_{12}).dis_4 + X(v_3).dis_4,\ X(v_{12}).dis_5 + X(v_3).dis_5) = \min(5+4,\ 8+1,\ 7+3,$ $4+3) = 7$. Similarly, for $q = (v_{12}, v_{19})$ we can first get LCA(v_{12}, v_{19}) $=$ $X(v_{13})$ in $T(G)$ with $X(v_{13}).pos = (1, 2)$, and then compute $dist(v_{12}, v_{19}) =$ $\min(X(v_{12}).pos_1 + X(v_{19}).pos_1,\ X(v_{12}).pos_2 + X(v_{19}).pos_2) = 11$.*

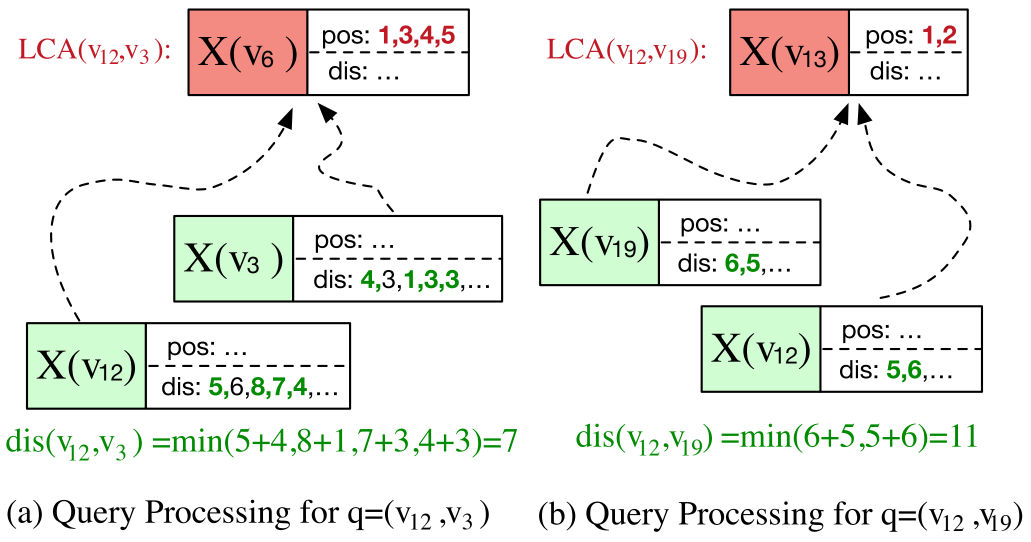


Figure 3.6: Query Processing Examples using the H2H-Index

3.6 H2H-Index Construction

In this section, we show how to construct the H2H-Index efficiently. We first introduce the distance preserved graph (DP-Graph) and show how to compute the H2H-Index using the DP-Graphs. Then we propose an improved algorithm with the assistance of the existing computed labels.

3.6.1 Index Construction using DP-Graph

In order to design an efficient algorithm to construct the H2H-Index, we first introduce the concept of distance preserved graph below.

Definition 5. ***(DP-Graph)*** *Given a road network G, a graph G' is called a Distance Preserved Graph (DP-Graph) if $V(G') \subseteq V(G)$, and for any pair of vertices $u \in V(G')$ and $v \in V(G')$, we have $dist_{G'}(u,v) = dist_G(u,v)$. We use $G' \sqsubseteq G$ to denote that G' is a DP-Graph of G.*

We can easily derive the following lemma:

Algorithm 3 operator $\ominus(G, v)$

Input: graph G and vertex $v \in V(G)$;
Output: the graph of $G \ominus v$.

```
1: $H \leftarrow G$;
2: for each pair of vertices $u, w \in N(v)$ do
3:     if $(u, w) \notin E(G)$ then
4:         insert edge $(u, w)$ with $\phi(u, w) \leftarrow \phi(u, v) + \phi(v, w)$ in $H$;
5:     else if $\phi(u, v) + \phi(v, w) < \phi(u, w)$ then
6:         update $\phi(u, w) \leftarrow \phi(u, v) + \phi(v, w)$ in $H$;
7: return $H$;
```

Lemma 1. *$G_1 \sqsubseteq G_2$ and $G_2 \sqsubseteq G_3 \Rightarrow G_1 \sqsubseteq G_3$.*

Proof Sketch: For any pair of vertices $u, v \in V(G_1)$, since $G_1 \sqsubseteq G_2$, we have $dist_{G_1}(u, v) = dist_{G_2}(u, v)$. Since $G_2 \sqsubseteq G_3$, we further have $dist_{G_2}(u, v) = dist_{G_3}(u, v)$. Consequently, $dist_{G_1}(u, v) = dist_{G_3}(u, v)$. Therefore, the lemma holds. $\square$

DP Vertex Elimination. Given a graph G and a vertex $v \in V(G)$, the Distance Preserved Vertex Elimination (DP Vertex Elimination) operation for v in G transform G into another graph as follows: For every pair of neighbors (u, w) of v in G, if $(u, w) \notin E(G)$, a new edge (u, w) with weight $\phi(u, w) = \phi(u, v) + \phi(v, w)$ is inserted. Otherwise, if $\phi(u, v) + \phi(v, w) < \phi(u, w)$, the weight of edge (u, w) is updated as $\phi(u, v) + \phi(v, w)$. Then v is removed. We use $G \ominus v$ to denote the DP Vertex Elimination operation for v in G. The algorithm for the $\ominus$ operator is shown in Algorithm 3. The following lemma shows that the DP Vertex Elimination operation results in a DP-Graph of the original graph G.

Lemma 2. *$G \ominus v \sqsubseteq G$.*

Proof Sketch: Let G' be $G \ominus v$. For any $u \in V(G')$ and $w \in V(G')$, we consider two cases:

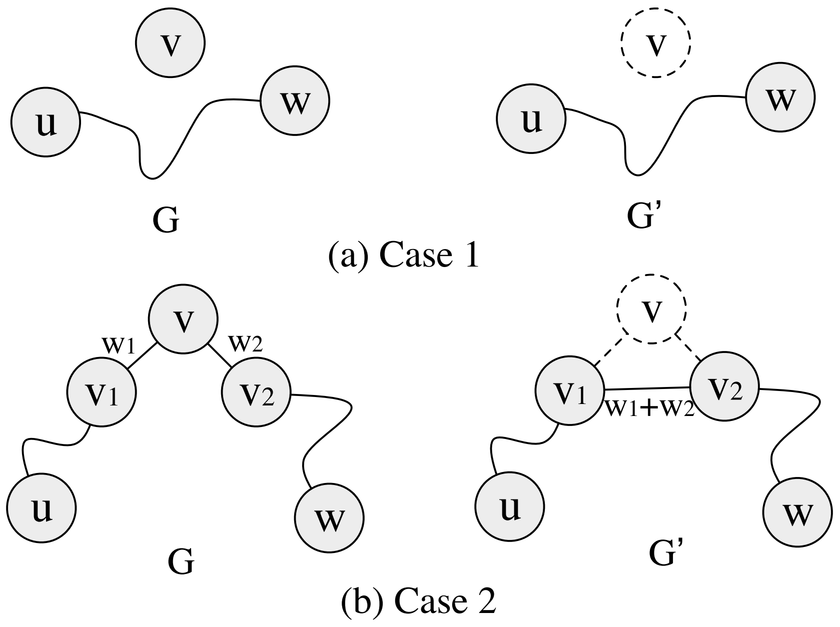


Figure 3.7: Proof of Lemma 2

- *Case 1: the shortest path from u to w in G does not pass through v.* In this case, we have $dist_G(u, w) = dist_{G'}(u, w)$ because the shortest path from u to w in G is also a shortest path from u to w in G'. Therefore, the distance from u to w is preserved. This case is illustrated in Fig. 3.7 (a).

- *Case 2: the shortest path from u to w in G passes through v.* In this case, suppose the shortest path from u to w in G is $(u, \ldots, v_1, v, v_2, \ldots, w)$. In G', v is eliminated, and a new edge (v_1, v_2) with weight $\phi(v_1, v) + \phi(v, v_2)$ is inserted. Therefore, the path $(u, \ldots, v_1, v_2, \ldots, w)$ is a shortest path from u to w in G' with $dist_{G'}(u, w) = dist_G(u, w)$. Therefore, the distance from u to w is preserved. This case is illustrated in Fig. 3.7 (b).

From the above two cases, we conclude that $G' = G \ominus v$ is a DP-graph of G.

$\square$

DP Tree Decomposition. We introduce the Distance Preserved Tree Decomposition (DP Tree Decomposition) using the DP vertex elimination operator. Generally speaking, DP tree decomposition is similar to tree decomposition introduced in Section 3.4.2. The only difference is that in each tree node $X(v)$ in the tree decomposition T_G, we not only keep the vertices in $X(v)$, but also preserve the weights from v to vertices in $X(v)$.

The detailed algorithm for DP tree decomposition is shown in Algorithm 4. Similar to Algorithm 1, the algorithm iteratively eliminate the vertex v with the smallest degree in graph H (line 3). For each v, we create a node $X(v)$ in T_G. Here, $X(v)$ not only contains the vertices $\{v\} \cup N(v, H)$, but also the edges (v, u) with weight $\phi(v, u)$ for all $u \in N(v, H)$ (line 4-5). In line 6, we eliminate v from H using the DP vertex elimination operator $\ominus$, and in line 7, we maintain the order π for all vertices. The process to construct the tree is the same as that in Algorithm 1 (line 8). After that, for efficiency consideration, for each node $X(v)$ in T_G, we sort vertices in $X(v)$ in decreasing order of their π values (line 10), and we use an array $X(v).\phi$ to preserve the weights from v to the vertices in $X(v)$ (line 11). Here, we suppose $\phi(v, v) = 0$. We use $X(v).\phi_i$ to denote the distance from v to the i-th vertex in $X(v)$. Finally, we return T_G as the tree decomposition.

According to Lemma 1 and Lemma 2, we know that after every vertex elimination in Algorithm 4, the graph H is a DP-graph of G, i.e., $H \sqsubseteq G$.

Lemma 3. *The time complexity of Algorithm 4 is $O(n \cdot (w^2 + \log(n)))$ and the space complexity of Algorithm 4 is $O(n \cdot w)$.*

Proof Sketch: For time complexity, the dominant cost for each node (line 3-7) include the operation to maintain and select the node with the smallest degree, which costs $O(m + n \cdot \log(n))$ time, and the $\ominus$ operation (Algorithm 3) which costs $O(|N(v, H)|^2) = O(w^2)$ time. Therefore, the overall time complexity of

Algorithm 4 DPTreeDecomposition$(G(V, E))$

Input: A road network $G(V, E)$;
Output: DP Tree decomposition T_G.

```
1: $H \leftarrow G$; $T_G \leftarrow \emptyset$;
2: for $i = 1$ to $|V|$ do
3:     $v \leftarrow$ the node in $H$ with smallest degree;
4:     $X(v) \leftarrow$ the star from $v$ to each $u \in N(v, H)$;
5:     Create a node $X(v)$ in $T_G$;
6:     $H \leftarrow H \ominus v$;
7:     $\pi(v) \leftarrow i$;
8: Line 9-12 of Algorithm 1;
9: for each $v \in V(G)$ do
10:    Sort vertices in $X(v)$ in decreasing order of $\pi$ values;
11:    $X(v).\phi_i \leftarrow \phi(v, x_i)$ where $x_i$ is the $i$-th vertex in $X(v)$ for all $1 \leq i \leq |X(v)|$;
12: return $T_G$;
```

Algorithm 4 is $O(n \cdot (w^2 + \log(n)) + m) = O(n \cdot (w^2 + \log(n))$. For space complexity, for each node, we maintain a star which costs $O(w)$ space. Therefore, the space complexity of Algorithm 4 is $O(n \cdot w)$. □

Example 15. *Fig. 3.8 shows the DP tree decomposition T_G for the road network G in Fig. 3.1. To construct T_G, we first select v_{12}, and create a node $X(v_{12})$ in T_G which contains a star with two edges (v_{12}, v_9) and (v_{12}, v_{11}) with $\phi(v_{12}, v_9) = 2$ and $\phi(v_{12}, v_{11}) = 1$. We eliminate v_{12} by adding an edge (v_9, v_{11}) with $\phi(v_9, v_{11}) = 3$, and set $\phi(v_{12}) = 1$. The process stops when all vertices are eliminated. For the node $X(v_{12})$, after sorting vertices in $X(v_{12})$ in decreasing order of their π values, we obtain the order v_{11}, v_9, v_{12} with $\phi(v_{12}, v_{11}) = 1$, $\phi(v_{12}, v_9) = 2$, and $\phi(v_{12}, v_{12}) = 0$. Therefore, $X(v_{12}).\phi = (1, 2, 0)$ as shown in Fig. 3.8.*

Given the DP tree decomposition T_G, for each $v \in V(G)$, we define a special graph $G(v)$ as follows.

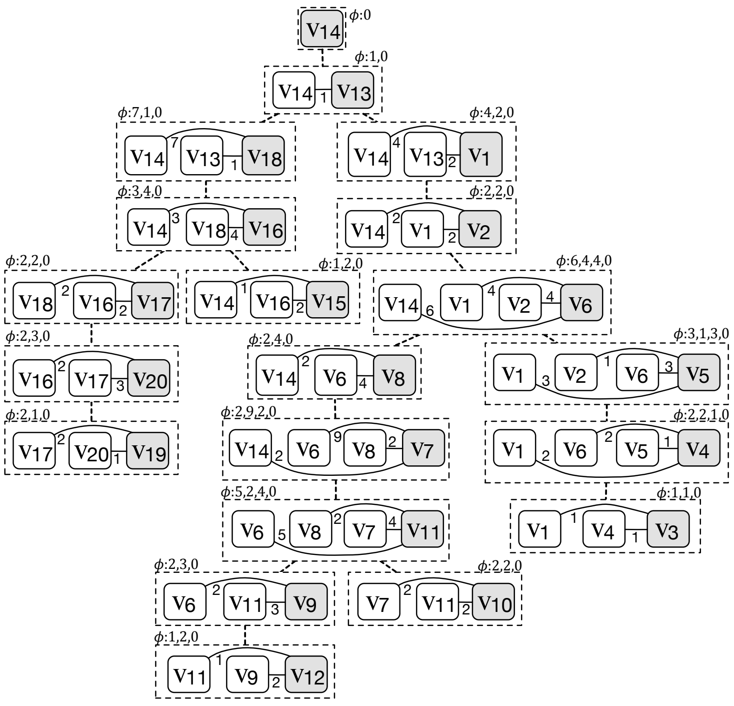


Figure 3.8: Distance Preserved Tree Decomposition T_G

Definition 6. *Given a DP tree decomposition T_G for the road network G, for each $v \in V(G)$, the graph $G(v)$ is defined as the union of all the stars for nodes in the path from $X(v)$ to the root of T_G.*

The number of edges in each $G(v)$ is at most $(w-1) \times h$ since each start has at most $w-1$ edges and the number of nodes from $X(v)$ to the root of T_G is at most h. Therefore, we can use $O(w \cdot h)$ time to construct $G(v)$ for each vertex v. Below we show an example to illustrate $G(v)$.

Example 16. *Given the DP tree decomposition T_G in Fig. 3.8, for vertex v_{20}, the path from $X(v_{20})$ to the root $X(v_{14})$ consists of 6 nodes $X(v_{20})$, $X(v_{17})$, $X(v_{16})$, $X(v_{18})$, $X(v_{13})$, and $X(v_{14})$. We merge the stars in the 6 nodes and we will obtain the graph $G(v_{20})$ shown in Fig. 3.9 (a) with 6 vertices and 9 edges. Similarly, we can obtain the graph $G(v_3)$ as shown in Fig. 3.9 (b).*

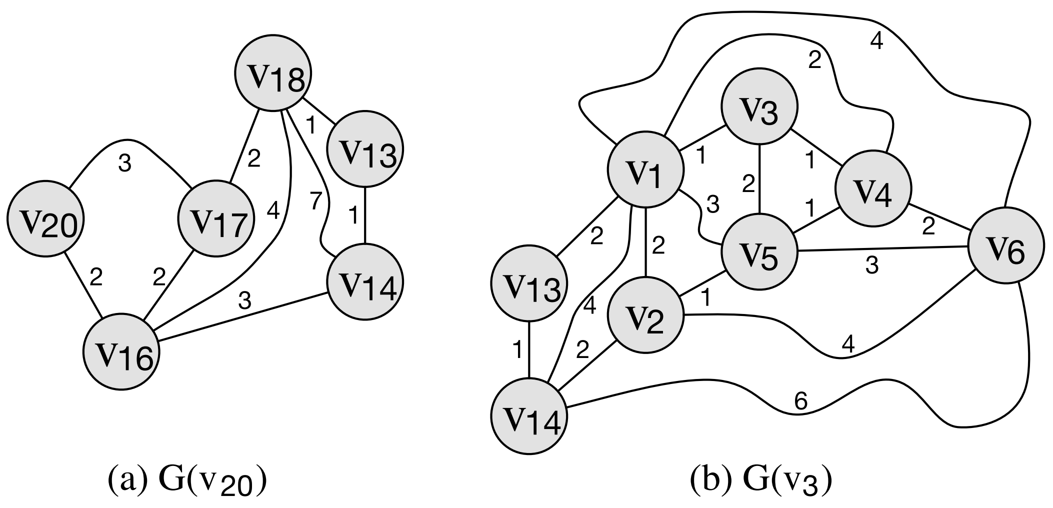


Figure 3.9: DP-Graphs $G(v_{20})$ and $G(v_3)$

The following lemma shows the property that $G(v)$ is a DP-graph of G for any $v \in V(G)$:

Lemma 4. *$G(v) \sqsubseteq G$ for any $v \in V(G)$.*

Proof Sketch: We prove the lemma using induction by the length of the path from $X(v)$ to the root of T_G, i.e., the depth of $X(v)$ in T_G. When the depth of $X(v)$ is 0, i.e., $X(v)$ is the root of T_G, $G(v)$ only contains one vertex. Therefore, $G(v) \sqsubseteq G$. Assume that the lemma holds when the depth of $X(v)$ is no larger than i, we now prove that the lemma holds when the depth of $X(v)$ is $i+1$. Suppose $X(v')$ is the parent of $X(v)$ in T_G, we know the depth of $X(v')$ is i. For any pair of vertices u and w in $G(v)$, we consider two cases:

- *Case 1: $u \neq v$ and $w \neq v$.* In this case, u and w are nodes in $G(v')$ and $dist_{G(v')}(w,u) = dist_G(w,u)$ by assumption. Since $G(v')$ is a subgraph of $G(v)$, we know that $dist_{G(v)}(w,u) = dist_{G(v')}(w,u) = dist_G(w,u)$.

- *Case 2: $u = v$ or $w = v$.* Without loss of generality, we suppose $w = v$.

Let H be the graph in Algorithm 4 before eliminating v, we know that $X(v) \setminus \{v\}$ is the neighbor set of v in H. Since $X(u)$ is a ancestor of $X(v)$ in T_G, we know that u has not been eliminated when eliminating v from H, i.e., $u \in V(H)$. Therefore, we have $dist_H(v,u) = \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x) + dist_H(x,u)\}$. According to Lemma 2, we know that H is a DP-graph. Therefore, we have $dist_G(v,u) = dist_H(v,u) = \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x)+dist_H(x,u)\} = \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x)+dist_G(x,u)\}$. In addition, $X(v)\setminus\{v\}$ is also the neighbor set of v in $G(v)$. Therefore, we have $dist_{G(v)}(v,u) = \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x) + dist_{G(v)}(x,u)\}$. Since $G(v')$ is a subgraph of $G(v)$, we have $\min_{x\in X(v)\setminus\{v\}}\{\phi(v,x) + dist_{G(v)}(x,u)\} = \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x) + dist_{G(v)'}(x,u)\}$. Here, $X(v) \setminus \{v\}$ is a subset of vertices in $G(v')$, thus by assumption, we have $\min_{x\in X(v)\setminus\{v\}}\{\phi(v,x) + dist_{G(v')}(x,u)\} = \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x)+dist_G(x,u)\}$. Based on the above analysis, we can derive:

$$\begin{aligned}
dist_{G(v)}(w,u) =&\ dist_{G(v)}(v,u)\\
=&\ \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x)+dist_{G(v)}(x,u)\}\\
=&\ \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x)+dist_{G(v')}(x,u)\}\\
=&\ \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x)+dist_{G}(x,u)\}\\
=&\ \min_{x\in X(v)\setminus\{v\}}\{\phi(v,x)+dist_{H}(x,u)\}\\
=&\ dist_H(v,u) = dist_G(v,u) = dist_G(w,u)
\end{aligned}$$

Based on the above analysis, we know $dist_{G(v)}(w,u) = dist_G(w,u)$ for any pair of vertices w and u in $G(v)$. Therefore, the lemma holds by induction. □

The Index Construction Algorithm. With Definiton 6 and Lemma 4, we

Algorithm 5 H2H-Naive(G)

Input: The road network $G(V, E)$;
Output: The H2H-Index.

```
1: $T_G \leftarrow$ DPTreeDecomposition($G$);
2: for each $v \in V(G)$ do
3:   $X(v).pos_i \leftarrow$ the position of $v_i$ in $X(v).anc$ where $v_i$ is the $i$-th vertex in $X(v)$ for all $1 \leq i \leq |X(v)|$;
4:   Construct $G(v)$ according to Definiton 6;
5:   Compute $dist_{G(v)}(v, u)$ for each vertex $u$ in $G(v)$;
6:   for $i = 1$ to $|X(v).anc|$ do
7:     $X(v).dis_i \leftarrow dist_{G(v)}(v, X(v).anc_i)$;
8: return $X(v).dis$ and $X(v).pos$ for all $v \in V(G)$;
```

are now ready to design the index construction algorithm. The pseudocode of the algorithm is shown in Algorithm 5. Recall that for each $v \in V(G)$, the H2H-Index consists of two components, the position array $X(v).pos$ and and distance array $X(v).dis$. After computing the DP tree decomposition T_G (line 1), we construct the two arrays for each $v \in V(G)$ (line 2). We first probe each vertex v_i of $X(v)$ in the sorted order in T_G, and assign $X(v).pos_i$ be the position of v_i in the ancestor array $X(v).anc$ in T_G (line 3). Then, we construct $G(v)$ according to Definiton 6 (line 4). After that, we can compute the single source shortest distances $dist_{G(v)}(v, u)$ from v to all vertices u in $G(v)$ using the Dijkstra's Algorithm (line 4). By Definiton 6 and Lemma 4, for each i from 1 to $|X(v).anc|$, we can assign $X(v).dis_i$ as the shortest distance from v to $X(v).anc_i$ in $G(v)$ (line 6-7). Finally, the distance array and position array for all vertices in G are returned as the H2H-Index.

Theorem 5. *The time complexity of Algorithm 5 to construct the H2H-Index is* $O(n \cdot (h \cdot w \cdot \log(h) + \log(n)))$.

Proof Sketch: As proved in Lemma 3, line 1 requires $O(n \cdot (w^2 + \log(n)))$ time. In the for loop (line 2-7), for each vertex v, line 3 requires $O(w \cdot h)$ time.

Line 4 requires $O(w \cdot h)$ time since $G(v)$ is the union of at most h stars each of which has at most w vertices. Line 5 requires $O(w \cdot h \cdot \log(h))$ time, and line 6-7 requires $O(h)$ time. Therefore, the overall time complexity of Theorem 5 is $O(n \cdot (w^2 + h \cdot w \cdot \log(h) + \log(n))) = O(n \cdot (h \cdot w \cdot \log(h) + \log(n)))$ since $w \leq h$.

$\square$

Below, we show an example to illustrate Algorithm 5:

Example 17. *Suppose the DP tree decomposition T_G in Fig. 3.8 is computed for the road network G in Fig. 3.1. We show how to compute $X(v_{20}).pos$ and $X(v_{20}).dis$. From T_G we know that $X(v_{20}) = (v_{16}, v_{17}, v_{20})$ and $X(v_{20}).anc = (v_{14}, v_{13}, v_{18}, v_{16}, v_{17}, v_{20})$. The positions of v_{16}, v_{17}, and v_{20} in $X(v_{20}).anc$ are 4, 5, and 6, respectively. Therefore, we have $X(v_{20}).pos = (4, 5, 6)$. The constructed $G(v_{20})$ is shown in Fig. 3.9 (a). Using the Dijkstra's Algorithm, we can compute the shortest distance from v_{20} to all vertices in $X(v_{20}).anc$ in graph $G(v_{20})$ as $dist(v_{20}, v_{14}) = 5$, $dist(v_{20}, v_{13}) = 6$, $dist(v_{20}, v_{18}) = 5$, $dist(v_{20}, v_{16}) = 2$, $dist(v_{20}, v_{17}) = 3$, $dist(v_{20}, v_{20}) = 0$. Therefore, we can get $X(v_{20}).dis = (5, 6, 5, 2, 3, 0)$.*

3.6.2 Index Construction using Partial Labels

In this subsection, we show how to further optimize the index construction algorithm in Algorithm 5. We will consider two aspects to improve the algorithm. First, for each vertex v, we need to construct the DP-graph $G(v)$ and compute the single source shortest distances from v to all other vertices in $G(v)$, which is costly. We will consider how to compute the labels without explicitly building the DP-graph $G(v)$. Second, the vertex labels are computed independently in Algorithm 5. To save the computational cost, we will consider the cost sharing when computing the labels for different vertices. Note that the graph $G(v)$ is a supergraph of $G(u)$ for any $u \in X(v).anc$. It means that we can reuse the label information in $X(u)$ when computing the label for $X(v)$. To do this, we need to compute the vertex labels in a top-down manner in T_G. We first introduce the following lemma:

Lemma 5. *For any* $1 \leq i < |X(v).anc|$, *we have*

$$X(v).dis_i = \min_{1 \leq j < |X(v)|} X(v).\phi_j + dist(x_j, X(v).anc_i)$$

Here, x_j *is the* j*-th vertex in* $X(v)$*.*

Proof Sketch: Since $G(v)$ is a DP-graph with all vertices in $X(v).anc$, we only need to show that the lemma holds in $G(v)$. Note that $X(v) \setminus \{v\}$ is the set of neighbors in $G(v)$. Therefore, we can derive that $X(v).dis_i = dist_{G(v)}(v,$ $X(v).anc_i) = \min_{x \in X(v) \setminus \{v\}} \phi(v, x) + dist_{G(v)}(x, X(v).anc_i) = \min_{1 \leq j < |X(v)|} X(v).\phi_j$ $+ \, dist(x_j,$ $X(v).anc_i)$. □

Based on Lemma 5, we need to compute $dist(x_j,$ $X(v).anc_i)$ for each $x_j \in X(v)$ and $1 \leq i < |X(v).anc|$. In order to compute $dist(x_j, X(v).anc_i)$ without exploring the graph $G(v)$, we consider two cases:

- *Case 1:* $X(v).pos_j > i$. In this case, $X(X(v).anc_i)$ is an ancestor of $X(x_j)$ in T_G. Therefore, we have

$$dist(x_j, X(v).anc_i) = dist(x_j, X(x_j).anc_i) = X(x_j).dis_i$$

  In addition, since we traverse $X(v)$ in a top-down manner, $X(x_j).dis_i$ has been computed when computing the label of $X(v)$. Therefore, $X(x_j).dis_i$ can be directly used as $dist(x_j, X(v).anc_i)$.

- *Case 2:* $X(v).pos_j \leq i$. In this case, $X(x_j)$ is an ancestor of $X(X(v).anc_i)$

in T_G. Therefore, we have

$$dist(x_j, X(v).anc_i) = X(X(v).anc_i).dis_{X(v).pos_j}$$

Similar to case 1, since we traverse $X(v)$ in a top-down manner, $X(X(v).anc_i).dis_{X(v).pos_j}$ has been computed when computing the label of $X(v)$. Therefore, $X(X(v).anc_i).dis_{X(v).pos_j}$ can be directly used as $dist(x_j, X(v).anc_i)$.

Example 18. *Given the DP tree decomposition T_G in Fig. 3.8, for vertex v_4, we have $X(v_4) = (v_1, v_6, v_5, v_4)$, $X(v_4).\phi = (2, 2, 1, 0)$ and $X(v_4).anc = (v_{14}, v_{13}, v_1, v_2, v_6, v_5, v_4)$. We can compute that $X(v_4).pos = (3, 5, 6, 7)$. Now let us compute $X(v_4).dis_4$, i.e., the shortest distance from v_4 to v_2. Suppose the distance arrays for all ancestors of $X(v_4)$ in T_G has been computed. Based on Lemma 5, we can get that $X(v_4).dis_4 = \min\{2 + dist(v_1, v_2), 2 + dist(v_6, v_2), 1 + dist(v_5, v_2)\}$. To compute $dist(v_1, v_2)$, we know $X(v_4).pos_1 = 3 < 4$, which is case 2. Therefore, we can derive that $dist(v_1, v_2) = X(X(v_4).anc_4).dis_{X(v_4).pos_1} = X(v_2).dis_3 = 2$. To compute $dist(v_6, v_2)$, we know $X(v_4).pos_2 = 5 > 4$, which is case 1. Therefore, we can derive that $dist(v_6, v_2) = X(v_6).dis_4 = 4$. Similarly, to compute $dist(v_5, v_2)$, we know $X(v_4).pos_3 = 6 > 4$, which is case 1. Therefore, we can get $dist(v_5, v_2) = X(v_5).dis_4 = 1$. Consequently, we can get $X(v_4).dis_4 = \min\{2 + 2, 2 + 4, 1 + 1\} = 2$.*

With Lemma 5, we are ready to design the optimized algorithm to construct the H2H-Index. The pseudocode of the algorithm is shown in Algorithm 6. After computing the DP tree decomposition T_G, we examine all nodes $X(v)$ in T_G in a top-down manner (line 2). Suppose x_i is the i-th vertex in $X(v)$ (line 3), we first compute the position array $X(v).pos$ by its definition (line 4-5). Then, for each $1 \leq i < |X(v).anc|$, we compute $X(v).dis_i$ (line 6). We use Lemma 5 to

Algorithm 6 H2H(G)

```
Input:  The road network $G(V, E)$;
Output: The H2H-Index.
1:  $T_G \leftarrow$ DPTreeDecomposition($G$);
2:  for each $X(v) \in V(T_G)$ in a top-down manner do
3:      Suppose $X(v) = (x_1, x_2, \ldots, x_{|X(v)|})$;
4:      for $i = 1$ to $|X(v)|$ do
5:          $X(v).pos_i \leftarrow$ the position of $x_i$ in $X(v).anc$;
6:      for $i = 1$ to $|X(v).anc| - 1$ do
7:          $X(v).dis_i \leftarrow +\infty$;
8:          for $j = 1$ to $|X(v)| - 1$ do
9:              if $X(v).pos_j > i$ then $d \leftarrow X(x_j).dis_i$;
10:             else $d \leftarrow X(X(v).anc_i).dis_{X(v).pos_j}$;
11:             $X(v).dis_i = \min\{X(v).dis_i, X(v).\phi_j + d\}$;
12:     $X(v).dis_{|X(v).anc|} \leftarrow 0$;
13: return $X(v).dis$ and $X(v).pos$ for all $v \in V(G)$;
```

compute $X(v).dis_i$ and consider the two cases based on the comparison between $X(v).pos_j$ and i (line 7-11). We set the last value in the distance array to be 0 since it is the distance from v to itself (line 12). Finally, we return the distance and position arrays for all nodes as the H2H-Index (line 13). We can derive the following theorem:

Theorem 6. *The time complexity of Algorithm 6 to construct the H2H-Index is* $O(n \cdot (\log(n) + h \cdot w))$.

PROOF SKETCH: As proved in Lemma 3, line 1 requires $O(n \cdot (w^2 + \log(n)))$ time. In the for loop (line 2-12), for each vertex v, line 4-5 requires $O(w \cdot h)$ time. In the for loop from line 6 to line 12, line 8-11 requires $O(w)$ time and the loop terminates in at most h iterations. Therefore, the for loop (line 6-12) requires $O(w \cdot h)$ time. In summary, the overall time complexity of Theorem 6 is $O(n \cdot (w^2 + h \cdot w + \log(n))) = O(n \cdot (\log(n) + h \cdot w))$ since $w \leq h$. □

Algorithm 6 improves Algorithm 5 in two aspects. First, it totally avoids

$X(v_4).anc$	v_{14}	v_{13}	v_1	v_2	v_6	v_5	$X(v_4).\phi$
v_{14}	0	1	3	$X(v_2).dis_1$:2	6	3	-
v_{13}	1	0	2	$X(v_2).dis_2$:3	6	4	-
v_1	3	2	0	$X(v_2).dis_3$:2	4	3	ϕ_1:2
v_2	2	3	2	$X(v_2).dis_4$:0	4	1	-
v_6	6	6	4	$X(v_6).dis_4$:4	0	3	ϕ_2:2
v_5	3	4	3	$X(v_5).dis_4$:1	3	0	ϕ_3:1
$X(v_4).dis$:	4	4	2	2	2	1	-

Figure 3.10: Process to Calculate $X(v_4).dis$

materializing the DP-graphs $G(v)$ for each $v \in V(G)$. Second, by re-using the partial labels, Algorithm 6 reduce the $\log(h)$ factor compared to Algorithm 5.

Example 19. *We show how to compute $X(v_4).dis$ using Algorithm 6 for the road network G in Fig. 3.1. The DP tree decomposition T_G is shown in Fig. 3.8 with $X(v_4) = (v_1, v_6, v_5, v_4)$. First, we can get $X(v_4).anc = (v_{14}, v_{13}, v_1, v_2, v_6, v_5, v_4)$ and $X(v_4).pos = (3, 5, 6, 7)$. In Fig. 3.10, the first 7 lines and 7 columns show a 6×6 matrix M. For each $1 \leq j \leq 6$ and $1 \leq i \leq 6$, if $j \leq i$, we set $M_{j,i}$ to be $X(X(v_4).anc_j).dis_i$. Otherwise, we set $M_{j,i}$ to be $X(X(v_4).anc_i).dis_j$. We also show $X(v_4).\phi = (2, 2, 1)$ for vertices v_1, v_6, and v_5 in $X(v_4)$. As shown in Example 18, we can compute $X(v_4).dis_4 = \min\{X(v_4).\phi_1 + X(v_2).dis_3,\ X(v_4).\phi_2 + X(v_6).dis_4,\ X(v_4).\phi_3 + X(v_5).dis_4\} = \min\{4,\ 6,\ 2\} = 2$. The cells that involve the calculation have been marked gray. In the same way, we can calculate $X(v_4).dis_i$ for $i \in \{1,\ 2,\ 3,\ 5,\ 6\}$. We also know $X(v_4).dis_7 = 0$. Therefore, we get $X(v_4).dis = (4,\ 4,\ 2,\ 2,\ 2,\ 1,\ 0)$.*

3.7 Performance Studies

We conduct extensive experiments to evaluate the proposed algorithms.

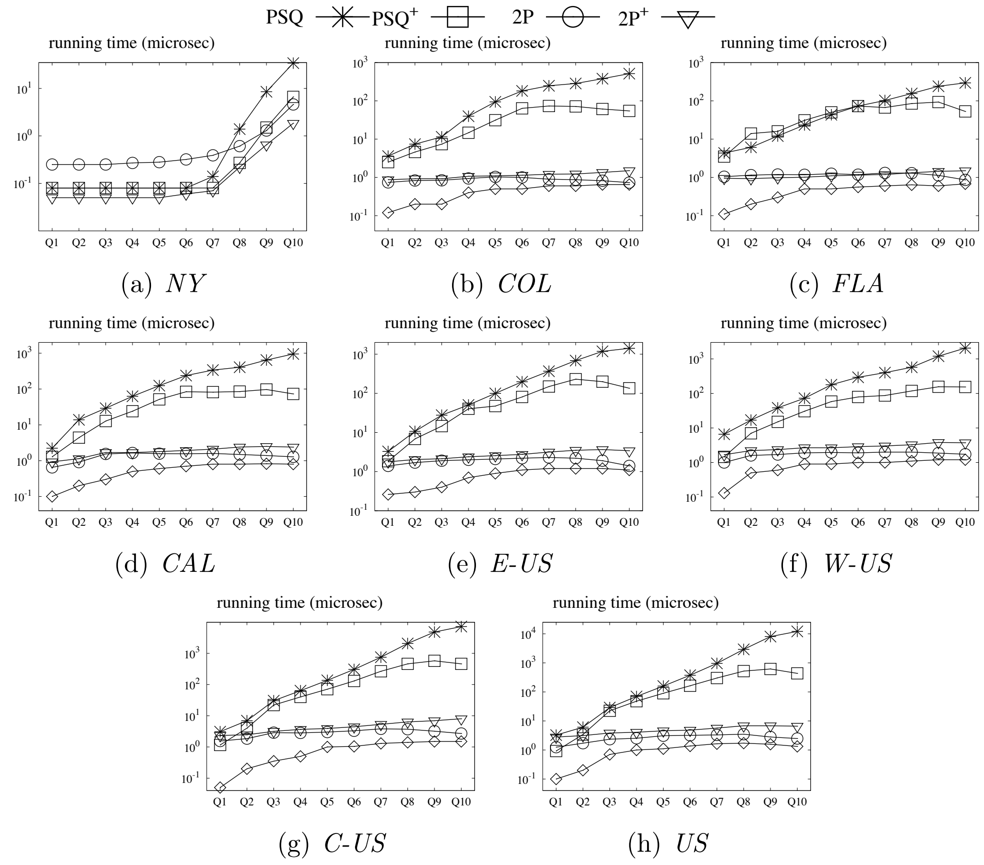
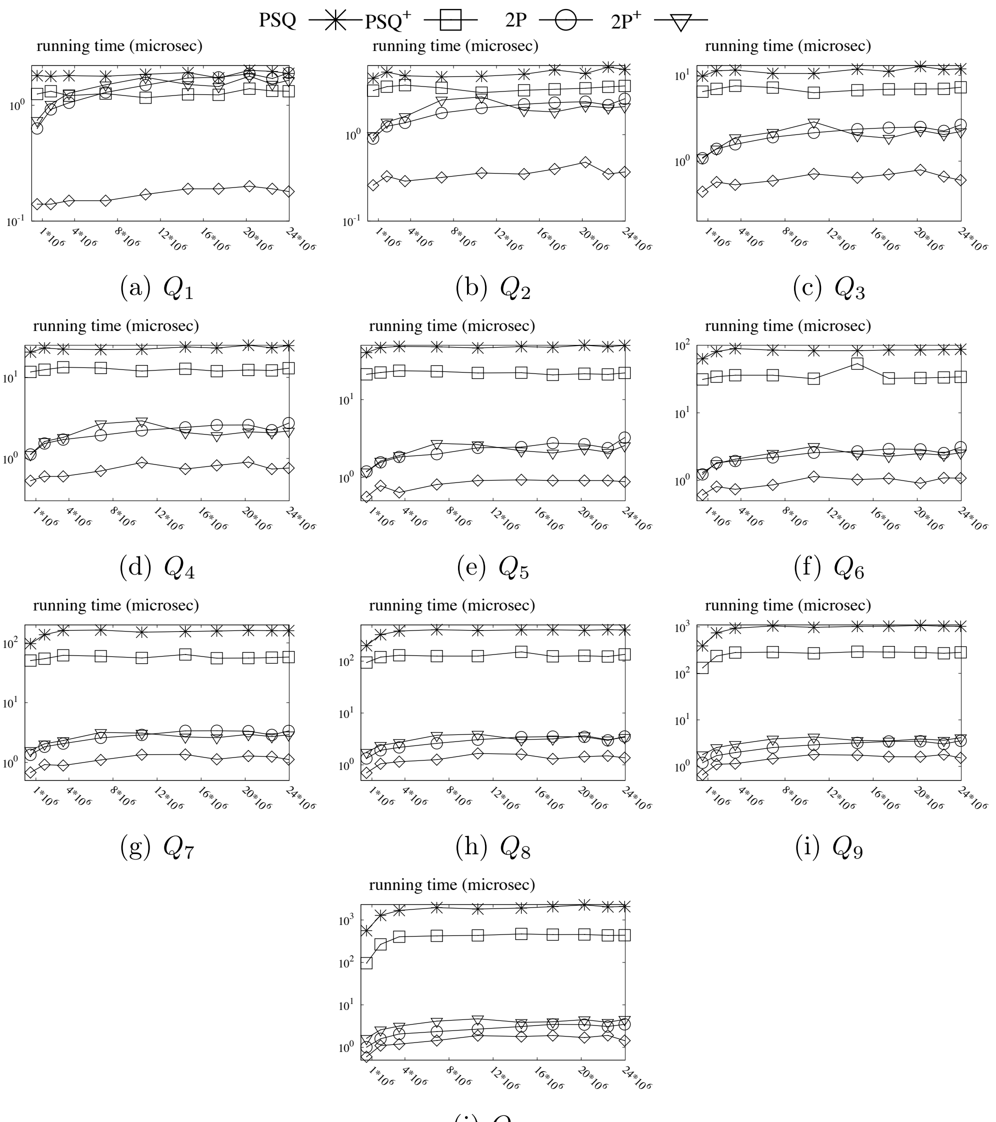


Figure 3.11: Query Processing Time (Varying Query Distance)

Algorithms. We compare our proposed algorithm with the state-of-the-art algorithms for shortest distance query processing on road networks. We implement and compare six algorithms:

- CH: Contraction Hierarchy [26];
- AH: Arterial Hierarchy [75];
- HL: Hub-based Labeling [2];
- PHL: Pruned Highway Labeling [3];
- H2H-Naive: H2H-Index Construction using Algorithm 5.



(a) Q_1 (b) Q_2 (c) Q_3

(d) Q_4 (e) Q_5 (f) Q_6

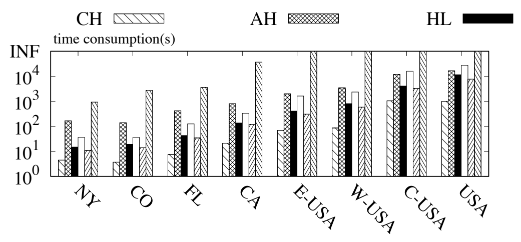
(g) Q_7 (h) Q_8 (i) Q_9

(j) Q_{10}

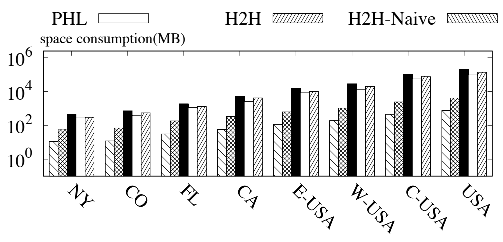
Figure 3.12: Query Processing Time (Varying Dataset Size)

- H2H: H2H Query Processing (Algorithm 2) and Index Construction (Algorithm 6);

All algorithms are implemented in C++ and compiled with GNU GCC 4.4.7. All experiments are conducted on a machine with an Intel Xeon 2.8GHz CPU and 256 GB main memory running Linux (Red Hat Linux 4.4.7, 64bit).



(a) Indexing Time



(b) Index Size

Figure 3.13: Indexing Time and Index Size Testing

Name	**Corresponding Region**	# Nodes	# Edges
NY	New York City	264,346	733,846
COL	Colorado	435,666	1,057,066
FLA	Florida	1,070,376	2,712,798
CAL	California and Nevada	1,890,815	4,657,742
E-US	Eastern US	3,598,623	8,778,114
W-US	Western US	6,262,104	15,248,146
C-US	Central US	14,081,816	34,292,496
US	United States	23,947,347	58,333,344

Table 3.2: Datasets used in the Experiments

Datasets. We use ten publicly available real road networks in the US downloaded from DIMACS[1]. Table 3.2 provides the data details in which the largest dataset is the whole road network in US. We use the distance as the edge weight for each dataset.

Exp-1: Varying Query Distance. In this experiment, we test the query efficiency of the algorithms by varying the distance between the source and target vertices in the query. Specifically, for each dataset, we generate 10 groups of queries Q_1, Q_2, …, Q_{10} as follows: We set l_{min} to be 1 kilometer, and set l_{max} to be the maximum distance of any pair of vertices in the map. Let $x = (l_{max}/l_{min})^{1/10}$. For each $1 \leq i \leq 10$, we generate $10,000$ queries to form Q_i, in which the distance of the source and target vertices for each query fall in the

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

range $(l_{min} \times x^{i-1}, l_{min} \times x^{i}]$. For each algorithm, we record the average query processing time for the $10,000$ queries in the corresponding query set.

The experimental results when varying the query from Q_1 to Q_10 for the 8 datasets are shown in Fig. 3.11 (a) - (h) respectively. We have the following observations. First, when the distance increases, the query processing time tend to increase in all datasets for all the five algorithms CH, AH, HL, PHL, and H2H. This is because when the distance between two vertices s and t increases, more edges are involved in the shortest path between s and t. Therefore, longer time is needed to answer the shortest distance query (s, t). Second, when the distance between s and t is relatively large, AH is 3-5 times faster than CH, HL is more than an order of magnitude faster than AH, PHL is 2-3 times faster than HL, and our algorithm H2H is 1.5-2 times faster than PHL. As analyzed in Section 3.4, the hop-based solutions HL and PHL are much faster than the hierarchy-based solutions CH and AH when $dist(s, t)$ is large, since it can avoid probing the graphs by using the labels directly. Our solution H2H is faster than the hop-based solution since H2H can further reduce the processing cost by visiting only a subset of the labels. Third, when the distance between s and t is relatively small, the two hop-based solutions HL and PHL have similar performance and the two hierarchy-based solutions CH and AH have similar performance. In some datasets such as *C-US* (Fig. 3.11 (g)) and *US* (Fig. 3.11 (h)), the hierarchy-based solutions CH and AH can be even faster than the hop-based solutions HL and PHL. This result is consistent with the analysis in Section 3.4. Remarkably, in this case, our algorithm H2H can achieve a speed-up of more than an order of magnitude compared to all the other approaches. This is because when the distance s and t is small, the vertex cut of s and t on the road network becomes small. Consequently, H2H only needs to visit a small portion of the labels for s and t to answer a query. This result demonstrates the advantage of our approach.

Exp-2: Varying Dataset Size. In this experiment, we test the query scalability when the size of the dataset increases. To do this, we divide the map of the whole US into 10×10 grids. We select a 1×1 grid in the middle to generate 10 groups of queries Q_1, Q_2, …, Q_{10} using the same method in Exp-1 each of which contains $10,000$ shortest distance queries. Then we generate 10 road networks using the 1×1, 2×2, …, 10×10 grids in the middle of the map and denote them as G_1, G_2, …, G_{10} respectively. G_1 is the selected 1×1 grid to generate the queries, and G_{10} is the whole road network of US. We have $G_1 \subset G_2 \subset \ldots \subset G_{10}$. We test the query processing time for each algorithm on each $G_i(1 \leq i \leq 10)$. For each query group $Q_i(1 \leq i \leq 10)$, we record the average processing time of the $10,000$ queries in the group.

The experimental results for Q_1 to Q_{10} when varying the dataset size are shown in Fig. 3.12 (a) to (j) respectively. The x-axis for each figure is the number of vertices for each dataset. From the experimental results, we have the following observations. First, when $dist(s,t)$ is small (Fig. 3.12(a)-(d)), the query processing time for the hop-based solutions HL and PHL has an obvious increasing trend when the data size increases. The query processing time for CH, AH, and H2H is relatively more scale-independent to the data size. This is because when $dist(s,t)$ is small, the size of the vertex cut for s and t does not significantly increase with the increasing of the dataset size. Second, when $dist(s,t)$ is large (Fig. 3.12(e)-(j)), the query processing time for all five algorithms increases when the graph size is no larger than 8×10^6, and then becomes stable when the size of the network further increases. This is because for a large $dist(s,t)$, when the size of the network is small, the size of the vertex cut between s and t has a significant increase when the network size increases, and when the size of the network is large, the size of the vertex cut between s and t becomes stable when the network size further increases. Third, the gap between H2H and the hop-based solutions HL and PHL increases when the size of the network increases for all queries. This is because when the size of the network is larger, H2H can skip visiting more vertices in the vertex labels of s and t. H2H outperforms the best in all cases.

Exp-3: Indexing Time. In this experiment, we test the indexing time for CH, AH, HL, PHL, H2H (Algorithm 6), and H2H-Naive (Algorithm 5). The experimental results for the 8 road networks are shown in Fig. 3.13 (a). From the experimental results, we can see that when the size of the networks increases, the indexing time for all algorithms will increase. CH is most efficient in indexing. Our algorithm H2H is the second most efficient algorithm in indexing. HL is slower than H2H. AH is 2-10 times slower than HL. H2H-Naive is more than two order of magnitude slower than H2H. For the four datasets *E-US*, *W-US*, *C-US*, and *US*, H2H-Naive cannot terminate within 30 hours and therefore we denote the time as INF. This shows the advantages of using the partial labels in the H2H algorithm.

Exp-4: Index Size. In this experiment, we test the index size for the five approaches CH, AH, HL, PHL, and H2H. The experimental results for the 8 road networks are shown in Fig. 3.13 (b). According to the figure, we can see that when the size of the network increases, the index size for all approaches will increase. CH has the smallest index size, followed by AH. The index sizes for the three approaches HL, PHL, and H2H are comparable. This is because they all generate the vertex labels and the label should be large enough to guarantee that the query can be answered correctly using only the information from the labels for s and t for each query $q = (s, t)$. The index size of H2H is 1.5-2 times smaller than HL, because in H2H, we only need to maintain the position array and distance array without keeping the vertex information, which make it a more compact index compared to HL.

3.8 Chapter Summary

In this chapter, we study the problem of point-to-point distance query processing on road networks. We propose a novel hierarchical 2-hop index (H2H-Index) which can overcome the drawbacks of the hierarchy-based solution and hop-based solution - the two state-of-the-art solutions for shortest distance query processing on road networks. We can achieve $O(w)$ query processing time and $O(n \cdot h)$ index size where w and h are the tree width and tree height for the tree decomposition of the road network which are small in practice, and n is the number of vertices on the road network. We propose an efficient algorithm using distance preserved graphs and partial labels which constructs the H2H-Index in $O(n \cdot (log(n) + w \cdot h))$ time. The experimental results on real large road networks demonstrate that our approach can achieve a speedup of an order of magnitude in query processing compared to the state-of-the-art while consuming comparable indexing time and index size.

Chapter 4

Efficient Shortest Distance Query on Dynamic Road Networks

4.1 Overview

In this chapter, we introduce how to solve shortest path on dynamic road networks using CH index. The rest of this chapter is organized as follow. Section 4.2 gives definition of shortest path query on dynamic road network. Section 4.3 introduces vertex centric algorithm which is state-of-the-art algorithm. Section 4.4 proposes our shortcut centric algorithm. Section 4.5 gives performance studies.Section 4.6 summarizes this chapter.

4.2 Preliminaries

This section paves the foundation for our technical discussion by clarifying the basic concepts and contraction hierarchies and providing a formal definition of the problem studied in this chapter.

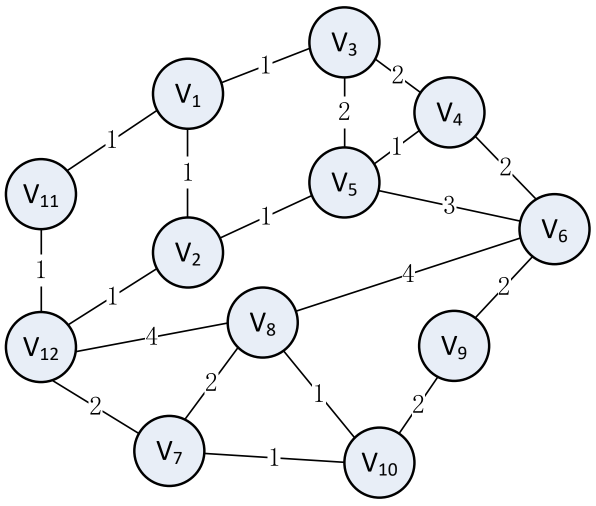


Figure 4.1: A Road Network

4.2.1 Contraction Hierarchies

Contraction Hierarchies (CH) is a hierarchy-based solution of shortest distance query on road network introduced by Geisberger et al. [26]. Given a road network G, CH is defined in two phases. The first phase constructs an index structure G' named *shortcut index* based on a total order of vertices γ in G according to their relative importance. The second phase answers the shortest distance query based on the shortcut index G'. Next we explain the two phases in detail.

Phase 1: Shortcut Index Construction. In the first phase, CH focuses on constructing the shortcut index, which is based on the node contraction operator $\odot$.

Definition 7. ***(Vertex contraction operator $\odot$)*** *Given a graph G, a total vertex order γ and a vertex v, vertex contraction operator performing on v in G, denoted by $G \odot v$, transforms G into $\mathcal{G}$ as follows: For every pair of neighbors (u, w) of v where u and w are ranked higher than v in γ, if $(u, w) \notin E(G)$, a new edge (u, w) with weight $\phi(u, w) = \phi(u, v) + \phi(v, w)$ is inserted. Otherwise, if*

Algorithm 7 ShortcutIndexConstruction

```
Input: A road network $G(V,E)$ and a total vertex order $\gamma$
Output: A shortcut index $G'$ of $G$
 1: $G' \leftarrow G$;
 2: for each node $v \in V(G)$ in the predefining total order $\gamma$ do
 3:     $G' \leftarrow G' \odot v$;
 4: return $G'$;
 5: procedure operator $\odot G, \gamma, v$
 6: $\mathcal{G} \leftarrow G$;
 7: for all pair of vertices $u, w \in N(v, G)$ with $\gamma(u) > \gamma(v)$ and $\gamma(w) > \gamma(v)$ do
 8:     if $(u,w) \notin E(G)$ then
 9:         insert edge $(u,w)$ with $\phi(u,w) \leftarrow \phi(u,v) + \phi(v,w)$ in $\mathcal{G}$;
10:     else
11:         if $\phi(u,w) > \phi(u,v) + \phi(v,w)$ then
12:             update $\phi(u,w) \leftarrow \phi(u,v) + \phi(v,w)$ in $\mathcal{G}$;
13:
14: return $\mathcal{G}$;
```

$\phi(u,v)+\phi(v,w) < \phi(u,w)$, the weight of (u,w) is updated with $\phi(u,v)+\phi(v,w)$. The new inserted or shorten edge is referred to as a shortcut.

CH first assigns a total order of vertices in G by assigning each vertex v a rank $\gamma(v)$ based on the importance of the nodes. Then CH examines each vertex v_i following the total order. For each vertex v_i, CH applies the node contraction operator on v_i. The shortcut index construction phase terminates after all vertices are examined. The edges G whose weight is not decreased along with all the shortcuts added in the index construction phase form the index structure of CH. We call the index structure as *shortcut index* and denote it as G'. Clearly, the shortcut index is a weighted graph. For the ease of distinction, we refer all the edges in the shortcut index as *shortcuts* and the edges in the given road network as *edges*. The detailed algorithm of shortcut index construction is shown in Algorithm 7.

Example 20. *Consider the graph G shown in Fig. 4.1 and assume that the vertices are ranked as $v_9 < v_{11} < v_{10} < v_7 < v_8 < v_4 < v_1 < v_2 < v_3 < v_5 < v_6 <$*

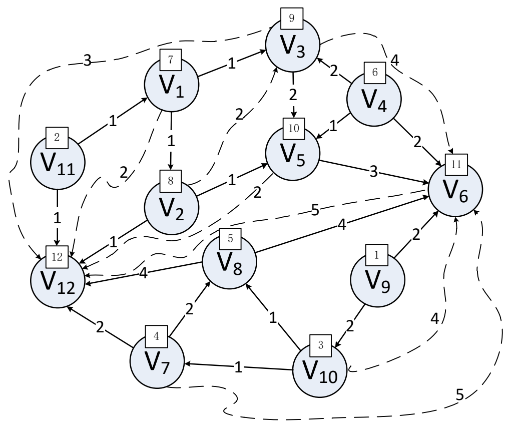


Figure 4.2: Shortcut Graph G_{sc}

v_{12}. We mark the rank of each vertex in a box beside each vertex in Fig. 4.2. In the shortcut graph construction phase, CH *first examines v_9 with the lowest rank. For its two neighbours v_6 and v_{10}, since the there is not an edge from v_6 to v_{10}, we add a shortcut (v_6, v_{10}) with weight 4 in G'. Then, the we examine vertex v_{11} which has two neighbours v_1 and v_{12}. We add a shortcut (v_1, v_{12}) with weight 2. Next, we deal with v_{10}. When we examine v_{10} which has three neighbours v_6, v_7 and v_8, where v_6 is new neighbour by additional shortcut. v_9 is not neighours since it was examined before. We add shortcut (v_6 and v_7) with weight 5. Because it exists edge (v_7, v_8) and (v_6, v_8) whose weight is less, we do not need to update weight. We continuelly examine vertex by order γ. After examine all vertices, the edges and shortcuts which are added consist of shortcut graph. The shortcut graph is shown in Fig. 4.2 where each dashed edge is the new added shortcut.*

Phase 2: Query Processing. For a shortest distance query $q(s, t)$, CH answers the query using the bidirectional Dijkstra's algorithm on the shortcut index G'

with some minor modifications. Specially, it starts two instances of Dijkstra's algorithm simultaneously from s and t, respectively. During the traversal of Dijkstra's search, CH only considers edges and shortcuts that connect a visited vertex v to an unvisited vertex v' whose rank is higher than v, i.e., $\gamma(v) < \gamma(v')$. The two traversals terminate when the any distance between current vertex and initial vertex is larger than current result. Let V_s (resp.V_t) be the set of vertices visited by the traversal that starts from s (resp.t). The shortest distance between s and t equals the smallest value among $dist(s, u)+dist(u, t)$, for any two adjacent vertices $v_s \in V_s$ and $v_t \in V_t$.

Example 21. *Given graph G in Fig. 4.1, CH index in Fig. 4.2 and a distance query (v_7, v_{11}), we compute shortest path distance by Phase 2 of CH. At first, we start 2 Dijkstra instance from v_7 and v_{11}. We push v_7 and v_{11} to priority queue Q. Then, we pop the top element of Q which is v_7. We search neighbors of v_7 and get distance $dist_{G'}(v_7, v_{12}) = 2$, $dist_{G'}(v_7, v_8) = 2$ and $dist_{G'}(v_7, v_6) = 5$. We push them to Q. Next, we pop v_{11} and compute distance of v_{11}'s neighbors. We get distance $dist_{G'}(v_{11}, v_1) = 1$ and $dist_{G'}(v_{11}, v_{12}) = 1$. v_{12} is the first common vertex we get from both two initial vertices. We can compute the current distance of $dist_{G'}(v_7, v_{11})$ by $dist_{G'}(v_7, v_{12}) + dist_{G'}(v_{11}, v_{12}) = 3$. After that, we continue Dijkstra algorithm and pop v_1. We got $dist_{G'}(v_{11}, v_3) = 2$ and $dist_{G'}(v_{11}, v_2) = 2$. We pop v_{12} and there is no neighbor u with $\gamma(u) > \gamma(v_{12})$. We pop and deal with v_2, v_3 and v_{10} continuesly. There is not any common vertex and the closest distance is larger or equal than current result* 3*. So we can terminate algorithm and get the result $dist_G(v_7, v_{11}) = 3$.*

4.2.2 Problem Statement

We now provide a formal definition of the shortest distance query problem on dynamic road network: Given a dynamic road network $G = (V, E, \phi)$, where the weights of edges in G are dynamically updated, for a shortest distance query $q = (s, t)$, it returns the shortest distance $dist(s, t)$ on the latest graph. In this chapter, we aim to develop efficient indexing techniques so that q can be answered efficiently. We illustrate weight updating scenarios: *streaming weight updates* in which edge weight updates are continuously arriving and the processing happens on each update of an edge e.

In the literature, CH is widely adopted for shortest distance query on static road network since it achieves a balanced trade-off between index size and query processing time [68, 38]. Therefore, in this chapter, we resort to CH to address shortest distance query on dynamic road network as well. Our object is to design an algorithm by refitting CH such that it can support shortest distance query efficiently when the weights of edges in G are updated dynamically. For a given road network, we *assume that the total order of vertices γ is fixed.* This assumption is reasonable and practical since the total vertex order is generally determined by the importance of the vertices and the edge weight updates preserve the topology of the graph and have little influence on the importance of the vertices. This assumption is also adopted in the-state-of-the approach [27].

Example 22. *Given graph in Fig. 4.1, we can run CH index construction algorithm to obtain G' in Fig. 4.2. For a weight updating of (v_4, v_6) changing to* 1*, running shortest path query algorithm on Fig. 4.2 can not get the shortest path for graph whose weight of (v_4, v_6) is updated. To maintain the G', we get a new G' in Fig. 4.3.*

4.3 The State of the art

The state-of-the-art algorithm to support the shortest distance query on dynamic road networks, denoted by $\mathsf{DCH}_{\mathsf{vcs}}$, is proposed by Geisberger et al. in [27].

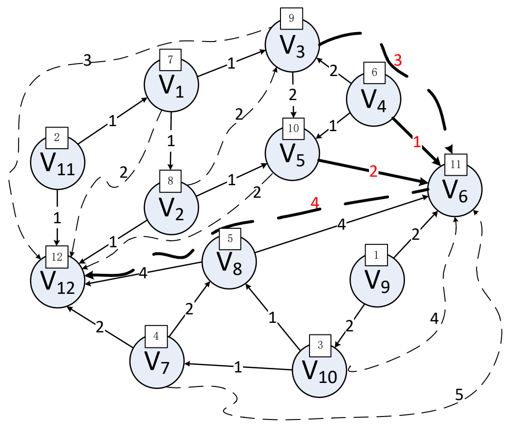


Figure 4.3: new CH graph after (v_4, v_6)'s weight decreasing

$\mathsf{DCH_{vcs}}$ focuses on the streaming weight updates can be addressed by treating them as a series of streaming weight updates.

A vertex-centric algorithm. $\mathsf{DCH_{vcs}}$ (vcs means $\underline{v}$ertex-$\underline{c}$entric for $\underline{s}$treaming weight update) is based on CH and assumes that the total vertex order γ for the graph keeps the same when the weights of edges are updated. Since the query processing phase of CH is query-dependent and is performed on the fly, to support the shortest distandce query on dynamic networks, we only need to maintain the correct shortcut index G' when the weight of e is updated and process the query the same as CH based on the maintained shortcut index. To achieve this goal, a naive approach is to reconstruct the shortcut index from scratch. Obviously, this approach is inefficient. To improve the efficiency, $\mathsf{DCH_{vcs}}$ adopts a vertex-centric paradigm and maintains G' incrementally by leveraging the vertex contraction operator: it first finds a set of vertices that are incident to shortcuts whose weights may be changed due to the weight update of e. Then, it recontracts these vertices to build the new shortcut index for the updated graph. Specifically, it contains two steps:

Step 1. Affected Vertices Identification. When the weight of edge $e = (u, v)$ changes, without lose of generality, let w be another neighbor of u and we assume that $\gamma(u) < \gamma(v) < \gamma(w)$. Based on the shortcut index contraction phase of CH, the weight of shortcut (v, w) may be changed due to the vertex contraction operation on u. Recursively, the weight change of (v, w) may further affect the weight of edges incident to node v for the same reason. Therefore, in this step, starting from the edge e with weight changes, $\mathsf{DCH_{vcs}}$ explores the shortcuts generated upon e in a depth-first search manner on G' and for an explored shortcut (u, v), the incident node with $\min\{\gamma(u), \gamma(v)\}$ is recorded as an affected vertex.

Step 2. Vertices Recontraction. After obtaining all the affected vertices, $\mathsf{DCH_{vcs}}$ updates the weight of e and applies the vertex contraction operator $\odot$ on all the vertices identified in Step 1 following the total vertex order γ and the new generated shortcut index is the shortcut index for the updated graph.

Example 23. *Given graph in Fig. 4.1 and G' in Fig. 4.2, we deal with weight updating of (v_9, v_{10}) from* 2 *to* 4*. In step* 1*, we need to identify all affected vertices. Because of $\gamma(v_9) < \gamma(v_{10})$, v_9 is added as an affected vertex. Then we search all vertices which may be affected by v_9. They are the neighbors of v_9, which are v_6 and v_{10}. Next, we compute affected vertices by v_6 and v_{10}. They are v_7, v_8, v_{12}. Finally, there is no more affected vertices. The affected vertices set S is $\{v_9, v_{10}, v_6, v_7, v_8, v_{12}\}$. In step* 2*, we reconstract all vertices of S by order γ from low to high. The maintained CH index can be obtained after all steps.*

Theorem 7. *Given the shortcut index G' of G, for an edge weight update, the time complexity of $\mathsf{DCH_{vcs}}$ to maintain G' is $O(n \cdot d_{\max}^2)$, where $d_{\max}$ is the maximum degree of G.*

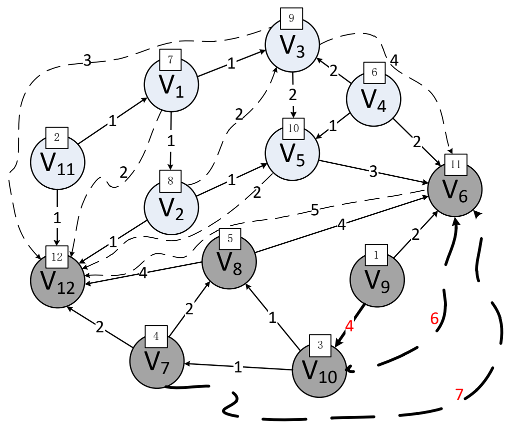


Figure 4.4: new CH graph after (v_9, v_{10})'s weight increasing

Drawbacks of DCH$_{vcs}$. Comparing with directly reconstructing the shortcut index from scratch, DCH$_{vcs}$ contracts less nodes. However, it has the following two drawbacks:

- *Drawback 1: Loose theoretical bound.* DCH$_{vcs}$ contracts less nodes than the naive approach, but the number of nodes contracted of DCH$_{vcs}$ cannot be well bounded. Theoretically, the number of contracted nodes in DCH$_{vcs}$ for each update can only be bounded by $O(n)$, which means DCH$_{vcs}$ shares the same worst case time complexity with the naive approach. This loose theoretical bound restricts the applicability of DCH$_{vcs}$.

- *Drawback 2: Poor practical performance.* Besides its loose theoretical bound, the practical performance of DCH$_{vcs}$ is poor. Based on the results of our experiment, even on the dataset with only 264346 vertices and 733846 edges, the processing time of DCH$_{vcs}$ for an edge weight update is over 100 millionsecond for weight increasing. Obviously, this expensive processing time cannot satisfy the realtime requirements in real application scenarios.

4.4 Streaming Updates Algorithm

In this section, we present our approach for streaming weight updates scenario. Similar to $\mathsf{DCH}_{\mathsf{vcs}}$, our approach *keeps the query processing phase of* CH *to answer the query online and focuses on efficiently maintaining the shortcut index when the weights of edges are updated.* Henceforth, we concentrate on maintaining the shortcut index and the query processing based on the shortcut index can be referred to Section 4.2.1. In this section, we first analyse the reasons causing the drawbacks of $\mathsf{DCH}_{\mathsf{vcs}}$, and then introduce our approach based on the new proposed shortcut-centric paradigm to reduce the computational cost from both theory and practice.

4.4.1 A Shortcut-Centric Paradigm

$\mathsf{DCH}_{\mathsf{vcs}}$ adopts a vertex-centric paradigm and maintains the shortcut index from the vertices perspective following an identification-recontraction approach. However, vertex-centric paradigm is not suitable for maintaining the shortcut index and the identification-recontraction approach leads to drawbacks of $\mathsf{DCH}_{\mathsf{vcs}}$ analysed in Section 4.3. Specifically, two kinds of unnecessary computation involves in $\mathsf{DCH}_{\mathsf{vcs}}$: (1) In Step 1, $\mathsf{DCH}_{\mathsf{vcs}}$ considers all the vertices incident to the explored shortcuts built upon the edge e as the affected vertices. However, not all of the weights of these explored shortcuts will be changed after the weight change of e. As a result, unaffected vertices are carelessly identified as affected vertices in $\mathsf{DCH}_{\mathsf{vcs}}$. (2) In Step 2, for an identified vertex u, all pairs of u's neighbors with higher rank than u are checked due to the vertex contraction operation applied on u. However, some shortcut weights may keep the same after the weight of e changes, which means checking all pairs of neighbors of u with higher rank than u in Step 2 of $\mathsf{DCH_{vcs}}$ involves lots of unnecessary computations.

We show an example to demonstrate the above two problems.

Example 24. *Given graph in Fig. 4.1 and G' in Fig. 4.2, we deal with weight updating of (v_9, v_{10}) from* 2 *to* 4. *In step 1 of* $\mathsf{DCH_{vcs}}$*, the vertices which are regards as affected vertex are* $\{v_9, v_{10}, v_6, v_7, v_8, v_{12}\}$. *However,* v_6*,* v_8 *and* v_{12} *are redundant vertices. Reconstructing redundant vertices do not update CH graph. For* v_7*, when we reconstruct it and compute all pairs of* v_7*, we find that* (v_8, v_{12}) *is unnecessary computation because the* $\phi((v_7, v_{12}), G')$ *and* $\phi((v_7, v_8), G')$ *are not changed.*

Based on above analysis, vertex-centric paradigm is not suitable for maintaining the shortcut index. On the other hand, revisiting the problem of maintaining the shortcut index when the weight of an edge e is changed, its essence is to correct the weights of shortcuts whose weights are changed after the weight change of e. Therefore, in this chapter, we adopt a shortcut-centric paradigm and maintain the shortcut index from the shortcut perspective. Specifically, when the weight of e is updated, instead of identifying the affected vertices in $\mathsf{DCH_{vcs}}$, we identify the affected shortcuts whose weights would be changed due to the weight update of e. For these affected shortcuts, we correct their weights based on the new graph after the weight update of e. In this way, we can totally avoid the unnecessary computation involving in $\mathsf{DCH_{vcs}}$. However, to make our idea practically applicable, the following issues need to be addressed: (1) how to efficiently identify the affected shortcuts, and (2) how to efficiently correct the weights of these identified affected shortcuts. In the following, we will address these two issues.

4.4.2 Weight-Updated Shortcuts Bounded Algorithms

For a given road network G, we first prove that the topological structure of its shortcut index G' keeps the same when the weight of an edge in G is changed, which is shown in the following lemma:

Lemma 6. *Given a road network G and its corresponding shortcut index G', after updating the weight of an edge e to k on G, denoted by $G_{\oplus(e,k)}$, let $G'_{\oplus(e,k)}$ be the corresponding shortcut index of $G_{\oplus(e,k)}$, then $V(G') = V(G'_{\oplus(e,k)})$ and $E(G') = E(G'_{\oplus(e,k)})$.*

According to Lemma 6, G' and $G'_{\oplus(e,k)}$ shares the same topological structure when the weight of an edge e in G is updated. Therefore, to efficiently maintain the shortcut index G', we only need to concentrate on updating the weights of shortcuts whose weights in G' and $G'_{\oplus(e,k)}$ are different. However, since the weight of a shortcut depends on the weights of other shortcuts in the shortcut index, it's hard to identify and update the weights of these shortcuts directly. To address this problem, we first define:

Definition 8. ***(Shortcut Index Topological Structure Satisfaction)*** *Given the shortcut index G' of G, for a graph G'', we call G'' satisfies the* shortcut index topological structure *of G if $V(G'') = V(G')$ and $E(G'') = E(G')$.*

Definition 9. ***(Supporting Shortcut Pair)*** *Given a shortcut index G', for a shortcut $s = (u, v)$ in G', the shortcuts (u, w) and (v, w) are called a supporting shortcut pair of s if $(u, w) \in E(G')$, $(v, w) \in E(G')$ and $\gamma(w) < \gamma(u)$ and $\gamma(w) < \gamma(v)$ and w is called a supporting vertex of s.*

Definition 10. ***(SS-Graph)*** *Given a shortcut index G', the SS-Graph (Shortcut Supporting Graph) G^* of G' is a directed graph and it contains two kinds of vertices: (1) Shortcut type vertices V_s, each such vertex v_s corresponds to a* *shortcut s in G'. (2) Supporting relation type vertices V_r, each such vertex v_r corresponds to a supporting relation instance between a shortcut s and one of its supporting shortcut pair s_1 and s_2. For each supporting relation type vertex v_r, we connects three directed edges $<v_r, v_s>$, $<v_{s_1}, v_r>$ and $<v_{s_2}, v_r>$ in G^*, where v_s, v_{s_1} and v_{s_2} are the corresponding shortcut type vertices of shortcuts s, s_1 and s_2 regarding v_r, respectively.*

If there is a directed edge $<u, v>$ in G^*, we say u is an in-neighbor of v and v is an out-neighbor of u. For each vertex $v \in V(G^*)$, we use $N^-(v, G^*)$ and $N^+(v, G^*)$ to denote the set of its in-neighbors and out-neighbors in G^*, respectively.

Given a road network G and a shortcut index G' satisfying the shortcut index topological structure G, G^* is the SS-Graph of G', for a shortcut s in G' and its corresponding shortcut type vertex v_s in G^*, we use $\phi(v_s, G)$ to denote the weight of s in G if s is in G (if s is not in G, $\phi(v_s, G)$ is ∞). Similarly, we use $\phi(v_s, G')$ to denote the weight of s in G'. Based on G^*, we define the following property:

Definition 11. ***(Minimum Weight Property)*** *Given a road network G, a shortcut index G' satisfying the topological structure of G and the SS-Graph G^* of G', for a shortcut type vertex v_s in G^*, let $v_{r_1}, v_{r_2}, \cdots, v_{r_n}$ be the in-neighbors of v_s and $v_{s_{11}}$, $v_{s_{12}}$, $v_{s_{21}}$, $v_{s_{22}}$, $\cdots$, $v_{s_{n1}}, v_{s_{n2}}$ be the in-neighbors of $v_{r_1}, v_{r_2}, \cdots, v_{r_n}$ in G^*, respectively, $\phi(v_s, G')$ satisfies the minimum weight property if $\phi(v_s, G') = \mathsf{min}\{\phi(v_s, G), \phi(v_{s_{11}}, G') + \phi(v_{s_{12}}, G'), \phi(v_{s_{21}}, G') + \phi(v_{s_{22}}, G'), \cdots, \phi(v_{s_{n1}}, G') + \phi(v_{s_{n2}}, G')\}$.*

Following Definiton 11, we have the following lemma:

Lemma 7. *Given a road network G and a shortcut index G' satisfying the topological structure of G, G^* is the SS-Graph of G', G' is the shortcut index of*

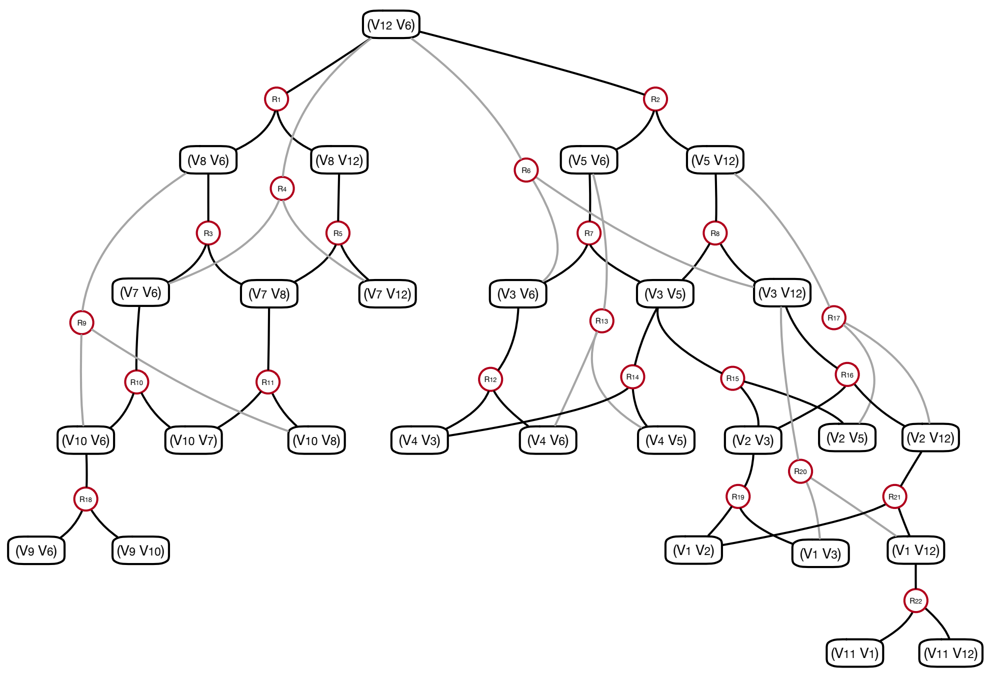


Figure 4.5: SS-Graph G^*

G if and only if $\phi(v_s, G')$ for all the shortcut type vertices v_s in G^* satisfy the minimum weight property.

Example 25. *In Fig. 4.5, we give a SS-Graph example G^* of shortcut graph G' Fig. 4.2. For each shortcut in shortcut graph, we have a corresponding vertex in Fig. 4.5. And we use 22 relation type vertices, $R_1, R_2, \ldots, R_{22}$, to support relation instance. G' satisfy the minimum weight property of G^*. For example, for shortcut (v_6, v_8), there are 3 relation type vertices R_9, R_{10} and R_{11}. We have $\phi((v_8, v_6), G') = min\{\phi((v_8, v_6), G), \phi((v_{10}, v_6), G') + \phi((v_{10}, v_8), G'), \phi((v_7, v_6), G') + \phi((v_7, v_8), G')\} = 4$.*

As shown in Lemma 6, when the weight of an edge in G is updated, $G'_{\oplus(e,k)}$ and G' shares the same topological structure, which means G' satisfies the topological structure of G. Meanwhile, the SS-Graph of $G'_{\oplus(e,k)}$ and G' are the same since the SS-Graph depends on the structure of the corresponding shortcut index alone. Therefore, according to Lemma 7, the problem of updating the weights of shortcuts whose weights in G' and $G'_{\oplus(e,k)}$ are different is equivalent to the following problem:

Definition 12. ***(CH Index Update)*** *Given a road network G, its shortcut index G' and the SS-Graph G^* of G', we aim to adjust $\phi(v_s, G')$ for all shortcut type vertices in G^* to make them satisfy the minimum weight property when $\phi(v_e, G)$ in G^* is updated to k, where e is the edge in G with weight update and v_e is the corresponding shortcut type vertex of e in G^*.*

Based on Definiton 11, we have the following lemma:

Lemma 8. *Given a road network G, its shortcut index G' and the SS-Graph G^* of G', when $\phi(v_e, G)$ is updated, $\phi(v_s, G')$ for the shortcut type vertices unreachable from v_e in G^* satisfy the minimum weight property.*

Therefore, when the weight of an edge e is updated, we do not need to consider the shortcut type vertices unreachable from v_e in G^*. However, all the other shortcut type vertices may violate the minimum weight property. The remaining problem is how to identify these shortcut type vertices precisely and adjust their $\phi(v_s, G')$ efficiently. Below, we introduce a shortcut weight update propagation mechanism on G^* to achieve this goal.

Shortcut Weight Propagation Mechanism on G^*. According to Definiton 11, for a shortcut type vertex v_s, $\phi(v_s, G')$ may violate the minimum weight property if and only if $\phi(v_s, G)$ or one of the sum of $\phi(v_{s_{i1}}, G')$ and $\phi(v_{s_{i2}}, G')$ are changed, where $v_{s_{i1}}$ and $v_{s_{i2}}$ follows the definition in Definiton 11. When $\phi(v_s, G')$ is changed, let v_r be one of the out-neighbors of v_s and v'_s be the out-neighbor of v_r, the change of $\phi(v_s, G')$ may cause the shortcut type vertices v'_s to violate the minimum weight property consequently and we need to adjust $\phi(v'_s, G')$ further. Therefore, our shortcut weight propagation mechanism works as follows: for a shortcut type vertex v_s whose $\phi(v_s, G')$ may be changed, we first determine whether $\phi(v_s, G')$ needs to be adjusted based on the the minimum weight property. If $\phi(v_s, G')$ is changed, we notify the out-neighbors v_s' of v_s's out-neighbors as the candidate shortcut type vertices that $\phi(v_s', G')$ may need to be further adjusted. The shortcut weight propagation mechanism is illustrated in Fig. 4.5. Following this shortcut propagation mechanism, we can iteratively adjust $\phi(v_s, G')$ in a depth-first manner until all the shortcut type vertices satisfy the minimum weight property.

Example 26. *Given graph G in Fig. 4.1, CH graph G' in Fig. 4.2 and SS-graph G^* in Fig. 4.5, we maintain CH graph for updating processing $\phi((v_9, v_{10}), G)$ changing to* 1*. For satisfying the Minimum weight property, we update CH graph by $\phi((v_9, , v_{10}), G') = min\{\phi((v_9, v_{10}), G)\} = 1$, $\phi((v_{10}, v_6), G') = min\{(\phi(v_9, v_{10}), G') + \phi((v_9, v_6), G')\} = 3$ and $\phi((v_6, v_7), G') = min\{(\phi(v_7, v_{10}), G') + (\phi(v_6, v_{10}), G')\} = 5$.*

Shortcut weight propagation mechanism achieves the goal of identifying and adjusting $\phi(v_s, G')$ not satisfying the minimum property. However, using the weight propagation mechanism alone may cause incorrect shortcut weight propagation and lots of futile shortcut weight propagation will be introduced. On the other hand, based on the Definiton 11, for v_s, if $\phi(v_{s_{11}}, G'), \phi(v_{s_{12}}, G'), \cdots,$ $\phi(v_{s_{n1}}, G'), \phi(v_{s_{n2}}, G')$ have been correctly adjusted before processing v_s, then $\phi(v_s, G'')$ can be adjusted correctly when processing v_s and the incorrect shortcut weight propagation problem is avoided. Therefore, we should follow a shortcut type vertex processing order to use the shortcut weight propagation mechanism such that when processing a certain shortcut type vertex v_s, $v_{s_{11}}, v_{s_{12}}, \cdots,$ $v_{s_{n1}}, v_{s_{n2}}$ have been processed before that. Inspired by this, we define the shortcut type vertex priority based on the vertex total order γ as follows:

Definition 13. ***(Shortcut Type Vertex Priority)*** *Given the SS-Graph G^* of a shortcut index G', let v_s and v_s' be two shortcut type vertex in G^* and their corresponding shortcuts in G' are $s = (u, v)$ and $s' = (u', v')$, respectively. Without loss of generality, assume that $\gamma(u) < \gamma(v)$ and $\gamma(u') < \gamma(v')$. We define v_s has a higher priority than v_s' if*

- $\gamma(u) < \gamma(u')$, *or*
- $\gamma(u) = \gamma(u')$ *and* $\gamma(v) < \gamma(v')$

Obviously, following the above shortcut type vertex priority, we can guarantee that $v_{s_{11}}, v_{s_{12}}, \cdots,$ $v_{s_{n1}}, v_{s_{n2}}$ will be processed before processing v_s. With the shortcut weight propagation mechanism and shortcut type vertex priority, we are ready to introduce our algorithm to maintain the shortcut index. In the following, we discuss the edge weight decrease case and edge weight increase case separately for their subtle differences. We first discuss the weight decrease case, which can be implemented following the weight update propagation mechanism directly.

Edge Weight Decrease Case.

Our algorithm to handle an edge weight decrease, $\mathsf{DCH_{scs}}$-WDec, is shown in Algorithm 8. $\mathsf{DCH_{scs}}$-WDec uses a priority queue Q to store the shortcut type vertices with $\phi(v_s, G')$ changed and the processing priority of shortcut type vertices in Q follow Definiton 13. The priority queue is initialised as $\emptyset$ (line 1). When the weight of an edge e (v_e denotes the corresponding shortcut type vertex of e in G^*) in G is decreased to k, if $\phi(v_e, G') < k$, which means $\phi(v_e, G')$ does not satisfy the minimum weight property, it adjusts $\phi(v_e, G')$ to k and pushes v_e into Q (line 4). After that, it iteratively notify other shortcut type vertices that may violate the minimum weight property to adjust $\phi(v_s, G')$ following the

Algorithm 8 $\mathsf{DCH_{scs}\text{-}WDec}$ (G^*, v_e, k)

```
1: PriorityQueue $Q \leftarrow \emptyset$; $\phi(v_e, G) \leftarrow k$;
2: if $\phi(v_e, G') > k$ then
3:     $\phi(v_e, G') \leftarrow k$;
4:     $Q$.push($v_e$);
5: while $Q \neq \emptyset$ do
6:     $v_s \leftarrow Q$.pop();
7:     for each $v_r \in N^+(v_s)$ do
8:         $v'_s \leftarrow$ the other in-neighbor of $v_r$ except $v_s$;
9:         $v''_s \leftarrow N^+(v_r)$;
10:        if $\phi(v''_s, G') > \phi(v_s, G') + \phi(v'_s, G')$ then
11:            $\phi(v''_s, G') \leftarrow \phi(v_s, G') + \phi(v'_s, G')$;
12:            if $v''_s \notin Q$ then
13:                $Q$.push($v''_s$);
```

shortcut weight propagation mechanism (line 5-13). Specifically, $\mathsf{DCH_{scs}\text{-}WDec}$ first pops out the shortcut type vertex v_s from Q (line 6) and iterates the out-neighbors of v_s (line 7), and for each out-neighbor v_r of v_s, it retrieves the other in-neighbor v'_s of v_r (line 8) and the unique out-neighbor v''_s of v_r (line 9). If $\phi(v''_s, G') > \phi(v_s, G') + \phi(v'_s, G')$ (line 10), which means $\phi(v''_s, G')$ does not satisfy the minimum weight property, it updates $\phi(v''_s, G')$ to $\phi(v_s, G') + \phi(v'_s, G')$ (line 11) and pushes v''_s into Q (line 12-13). $\mathsf{DCH_{scs}\text{-}WDec}$ terminates when Q is empty (line 5).

Example 27. *Given graph G in Fig. 4.1 , CH graph G' in Fig. 4.2 and SS-graph G^* in Fig. 4.5, we maintain CH graph for updating processing $\phi((v_4, v_6), G)$ changing to 1. We use Algorithm 8 to maintain CH graph. At first, we change $\phi((v_4, v_6), G)$ to 1. Because $\phi((v_4, v_6), G') > 1$, we assign it to 1 and push it to Q. Then we check all relation type vertex of $N^+(v_{(v_4,v_6)})$. For R_{12}, $\phi((v_3, v_6), G')$ is belong to $N^+(R_{12})$. With $\phi((v_3, v_6), G') = 4 < \phi((v_4, v_6), G') + \phi((v_4, v_3), G')$, we assign $\phi((v_3, v_6), G')$ with 3 and push it to Q. For R_{13} and $\phi((v_5, v_6), G') \in N^+(R_{13}$, because $\phi((v_4, v_6), G') + \phi((v_4, v_5), G') = 2 < \phi((v_5, v_6), G')$. We also assign $\phi((v_3, v_6), G')$ with 2 and push it to Q. Next, we pop $v_{(v_3,v_6)}$ which is* *the top element of Q. For R_6 and R_7 $\in N^+(v_{(v_3,v_6)})$, we find $\phi((v_3,v_6),G') + \phi((v_3,v_5),G') = 5 > \phi((v_5,v_6),G')$ and $\phi((v_3,v_6),G') + \phi((v_3,v_{12}),G') = 6 > \phi((v_6,v_{12}),G')$. So we do not update any weight. After that, we pop the top element (v_5,v_6) from Q. For R_2, because of $\phi((v_5,v_6),G') + \phi((v_5,v_{12}),G') = 4 < \phi((v_6,v_{12}),G')$. We assign $\phi((v_6,v_{12}),G')$ with 4 and push it to Q. Finally, we pop $(v_6,v_{12}$ from Q. There is not relation type vertex in $N^+(v_{(v_6,v_{12})})$. We finish algorithm and obtain new CH graph in Fig. 4.3.*

Theorem 8. *Give a road network G with the weight of an edge e in G decreased to k, Algorithm 8 computes $G'_{\oplus(e,k)}$ correctly.*

Proof. According to Lemma 6, G' and $G'_{\oplus(e,k)}$ shares the same topological structure and Algorithm 8 does not change the topological structure of G'. Therefore, the topological structure of $G'_{\oplus(e,k)}$ obtained by Algorithm 8 is correct. Now, we prove that Algorithm 8 correctly computes the shortcut weight of $G'_{\oplus(e,k)}$. Following Lemma 7, to prove Algorithm 8 computes the shortcut weight of $G'_{\oplus(e,k)}$ correctly, we only need to prove that all the satisfy the minimum property when Algorithm 8 terminates. We prove this from two aspects: (1) for all the shortcuts with different weights in G' and $G'_{\oplus(e,k)}$, their corresponding shortcut type vertices in G^* are explored in Algorithm 8. (2) for all the explored shortcut type vertices, their $\phi(v_s,G')$ are correctly adjusted to satisfy the minimum weight property. Based on above analysis, the theorem holds. □

Performance Guarantees. Now, we arrived at the first maintain result of this chapter:

Theorem 9. *Give a road network G with the weight of an edge e in G decreased to k, Algorithm 8 computes $G'_{\oplus(e,k)}$ in $O(\Delta \cdot (\log \Delta + \mathsf{deg}'_{\mathsf{max}}))$.*

Proof. In Algorithm 8, we use heuristic algorithm to update Contraction Hierarchies. Every shortcut s which are pushed in Q if and of only if s need to be updated. Because shortcut can only be pushed in Q once in this algorithm. The amount of shortcuts be pushed and popped in Q is Δ. The time complexity of priority queue is $\Delta \cdot \log \Delta$. For shortcut which is popped from Q, we check all neighbors in G^*. The time complexity of checking neighbors is $\Delta \cdot \mathsf{deg}'_{\mathsf{max}}$. Hence the total time complexity is $O(\Delta \cdot (\log \Delta + \mathsf{deg}'_{\mathsf{max}}))$.

□

To compute $G'_{\oplus(e,k)}$, at least the shortcut type vertices v_s that $\phi(v_s, G')$ and $\phi(v_s, G'_{\oplus(e,k)})$ are different, namely the shortcut type vertices in Δ, have to be explored. Meanwhile, as shown in Theorem 9, Algorithm 8 only explores the vertices in Δ and their neighbors to obtain $G'_{\oplus(e,k)}$. Therefore, Algorithm 8 is efficient regarding computing $G'_{\oplus(e,k)}$.

Edge Weight Increase Case.

In this section, we present our algorithm for the edge weight increase case. We first present an intuitive but not well bounded algorithm. Then, we optimize it and reduce its running time complexity to the same as that of Algorithm 8.

A 2-hop neighbors explored Algorithm. Following the shortcut weight propagation mechanism, we can directly obtain an algorithm, $\mathsf{DCH}_{\mathsf{scs}}$-$\mathsf{WInc}$, for edge weight increasing case in a similar framework as $\mathsf{DCH}_{\mathsf{scs}}$-$\mathsf{WDec}$, which is shown in Algorithm 9.

$\mathsf{DCH}_{\mathsf{scs}}$-$\mathsf{WInc}$ also uses a priority queue Q to store the shortcut type vertices with $\phi(v_s, G')$ changed and initialises it as $\emptyset$ (line 1). After updating the weight of edge e with k (v_e denotes the corresponding shortcut type vertex of e in G^*) (line 1), it checks whether $\phi(v_e, G')$ satisfying the minimum weight property. If $\phi(v_e, G')$ is bigger than the minimum weight ϕ computed by procedure $\mathsf{minWeight}$, $\mathsf{DCH}_{\mathsf{scs}}$-$\mathsf{WInc}$ updates $\phi(v_e, G')$ with ϕ and pushes it into Q (line 4-5). Then, it iteratively propagates the shortcut weight and further notifies

Algorithm 9 DCH$_{\text{scs}}$-WInc (G^*, v_e, k)

```
1: PriorityQueue $Q \leftarrow \emptyset$; $\phi(v_e, G) \leftarrow k$;
2: $\phi \leftarrow$ minWeight($G^*, v_e$);
3: if $\phi(v_e, G') > \phi$ then
4:     $\phi(v_e, G') \leftarrow \phi$;
5:     $Q$.push($v_e$);
6: while $Q \neq \emptyset$ do
7:     $v_s \leftarrow Q$.pop();
8:     for each $v_r \in N^+(v_s)$ do
9:         $v'_s \leftarrow N^+(v_r)$;
10:        $\phi \leftarrow$ minWeight($G^*, v'_s$);
11:        if $\phi(v'_s, G') > \phi$ then
12:            $\phi(v'_s, G') \leftarrow \phi$;
13:            $Q$.push($v'_s$);
14: procedureminWeight$G^*$, $v_s$
15: $\phi \leftarrow \phi(v_s, G)$;
16: for each $v_r \in N^-(v_s)$ do
17:     let $v_{s_1}$ and $v_{s_2}$ be the two in-neighbors of $v_r$;
18:     if $\phi > \phi(v_{s_1}, G') + \phi(v_{s_2}, G')$ then
19:         $\phi \leftarrow \phi(v_{s_1}, G') + \phi(v_{s_2}, G')$;
20: return $\phi$;
```

the shortcut type vertices through its out-neighbor similarly as DCH$_{\text{scs}}$-WDec. DCH$_{\text{scs}}$-WInc terminates when Q is empty (line 6).

Procedure minWeight computes $\min\{\phi(v_s, G), \phi(v_{s_{11}}, G')+\phi(v_{s_{12}}, G'), \phi(v_{s_{21}}, G')+\phi(v_{s_{22}}, G'), \cdots, \phi(v_{s_{n1}}, G')+\phi(v_{s_{n2}}, G')\}$ for the given shortcut type vertex v_s following Definiton 11. It first retrieves $\phi(v_s, G)$ as the minimum weight ϕ (line 15). Then, for each in-neighbor v_r of v_s (line 16), it computes the sum of $\phi(v_{s_1}, G')$ and $\phi(v_{s_2}, G')$, where v_{s_1} and v_{s_2} are the two in-neighbors of v_r. If the sum is less than the ϕ, ϕ is updated with the sum (line 18-19). minWeight returns the minimum weight in line 20.

Algorithm 9 is intuitive for edge weight increase case. However, for an edge weight update, the complexity of Algorithm 9 cannot be bounded by the 1-hop neighbors of Δ, where Δ represents the shortcut type vertices whose weight are changed after the weight update of e. The reasons are as follows. In Algorithm 9, two types of shortcut type vertices are pushed in the priority queue Q, namely:

- Type-1: The set of shortcut type vertices whose weights in G' and $G'_{\oplus(e,k)}$ are different, i.e., the vertices in Δ.
- Type-2: The set of shortcut type vertices that are (1) out-neighbors of Type-1 vertices; and (2) not Type-1 vertices.

For every shortcut type vertex v_s in Q, v_s is processed by procedure minWeight. As a result, for each Type-2 shortcut type vertex, its in-neighbors are explored because we do not know whether v_s is a Type-2 vertex before invoking minWeight. Since a Type-2 shortcut type vertex is a neighbor of a Type-1 shortcut type vertex, the 2-hop neighbors of some Type-1 vertices need to be explored, which results in the explored shortcut type vertices in Algorithm 9 can not be bounded by the 1-hop neighbors of Δ. In the following, we aim to optimize Algorithm 9 and bound the explored shortcut type vertices by the 1-hop neighbors of Δ.

A 1-hop neighbors explored Algorithm. As discussed above, the reason that Algorithm 9 has to explore the 2-hop neighbors of Δ is that Algorithm 9 cannot determine whether $\phi(v_s, G')$ will be changed directly after the weight change of e and the procedure minWeight is utilized to achieve this goal. Based on this reason, if we can determine whether $\phi(v_s, G')$ will be changed in $O(1)$, then we can avoid exploring the neighbors of Type-2 shortcut type vertices and the explored shortcut type vertices can be bounded by the 1-hop neighbors of Δ.

Following this idea, we design an elegant approach in which we can determine whether $\phi(v_s, G')$ will be changed after the weight update of e in $O(1)$ time by introducing a shortcut weight counter for each shortcut type vertex. Specifically, for each shortcut type vertex v_s, we use $c_\phi(v_s)$ to records the number of supporting shortcut pairs of v_s with $\phi(v_{s_1}, G') + \phi(v_{s_2}, G')$ equals to $\phi(v_s, G')$, where v_{s_1}

Algorithm 10 $\mathsf{DCH_{scs}\text{-}WInc^{+}}$ (G^*, v_s, k)

```
1: PriorityQueue $Q \leftarrow \emptyset$;
2: if $\phi(v_s, G') = \phi(v_s, G)$ then
3:     $c_\phi(v_s) \leftarrow c_\phi(v_s) - 1$;
4: $\phi(v_s, G) \leftarrow k$;
5: if $c_\phi(v_s) < 1$ then
6:     updateWeight($G^*, v_s$);
7:     $Q$.push($v_s$);
8: while $Q \neq \emptyset$ do
9:     $v_s \leftarrow Q$.pop();
10:    for each $v_r \in N^+(v_s)$ do
11:        $v'_s \leftarrow N^+(v_r)$;
12:        if $c_\phi(v'_s) < 1$ then
13:            updateWeight($G^*, v'_s$);
14:            $Q$.push($v'_s$);
15: procedure updateWeight$G^*$, $v_s$
16: for each $v_r \in N^+(v_s)$ do
17:     $v_{s_1}, v_{s_2} \leftarrow N^-(v_r)$; $v_{s_3} \leftarrow N^+(v_r)$;
18:     if $\phi(v_{s_3}, G') = \phi(v_{s_1}, G') + \phi(v_{s_2}, G')$ then
19:         $c_\phi(v_{s_3}) \leftarrow c_\phi(v_{s_3}) - 1$;
20: $\phi \leftarrow \phi(v_s, G)$; $c_\phi(v_s) \leftarrow 1$;
21: for each $v_r \in N^-(v_s)$ do
22:     let $v_{s_1}$ and $v_{s_2}$ be the two in-neighbors of $v_r$;
23:     if $\phi > \phi(v_{s_1}, G') + \phi(v_{s_2}, G')$ then
24:         $\phi \leftarrow \phi(v_{s_1}, G') + \phi(v_{s_2}, G')$; $c_\phi(v_s) \leftarrow 1$;
25:     else
26:         if $\phi = \phi(v_{s_1}, G') + \phi(v_{s_2}, G')$ then
27:             $c_\phi(v_s) \leftarrow c_\phi + 1$;
28: $\phi(v_s, G') \leftarrow \phi$;
```

and v_{s_2} represents an arbitrary supporting shortcut pairs of v_s (If $\phi(v_s, G)$ equals to $\phi(v_s, G')$, we also count it in $c_\phi(v_s)$).Based on Definiton 11, For a shortcut type vertex v_s, if we can compute $\phi(v_{s_1}, G'_{\oplus(e,k)})$, $\phi(v_{s_2}, G'_{\oplus(e,k)})$ and $c_\phi(v_s)$ correctly based on their definitions. Then, if $c_\phi(v_s)$ is still not less than 1, based on Definiton 11, We know that $\phi(v_s, G')$ will not be changed after the weight change of e. In this way, we can determine whether $\phi(v_s, G')$ will be changed after the weight update of e in $O(1)$ time. Our new approach is shown in Algorithm 10.

The framework of Algorithm 10 is similar to that of Algorithm 9. It utilizes a priority queue Q to store the shortcut type vertices that need to be processed (line 1) and iteratively pops the shortcut type vertex v_s out from Q (line 9), and propagates the change of $\phi(v_s, G')$ to its out-neighbors and further pushes the shortcut type vertices whose $\phi(v'_s, G')$ are updated into Q (line 10-14). The process terminates when the priority queue is empty (line 8). Different from Algorithm 9, in Algorithm 10, we introduce $c_\phi(v_s)$ and a new procedure updateWeight. For each shortcut type vertex v_s whose weight may be changed, we first check whether $c_\phi(v_s)$ is less than 1 (line 5 and line 12). If $c_\phi(v_s)$ is less than 1, we invoke updateWeight to compute $\phi(v_s, G'_{\oplus(e,k)})$ and update $c_\phi(v_s)$ regarding the updated $\phi(v_s, G'_{\oplus(e,k)})$ of v_s (line 6 and line 13).

Procedure updateWeight is used to update the weight of v_s in $G'_{\oplus(e,k)}$ and maintain the $c_\phi(v_s)$ and $c_\phi(v_{s_3})$ where v_{s_3} is the shortcut type vertex supporting v_s. Specifically, for a shortcut type vertex v_s, it first iterates the out-neighbors of v_s (line 16) and for each out-neighbor v_r of v_s, it retrieves the two in-neighbors v_{s_1} and v_{s_2} of v_s and the out-neighbor v_{s_3} of v_r (line 17). If $\phi(v_{s_3}, G') = \phi(v_{s_1}, G') + \phi(v_{s_2}, G')$, $c_\phi(v_{s_3})$ is decreased by 1 (line 18-19). Then, updateWeight updates $\phi(v_s, G'_{\oplus(e,k)})$ and computes $c_\phi(v_s)$ based on $\phi(v_s, G'_{\oplus(e,k)})$ by iterating the in-neighbors of v_s and computing the minimum weight based on Definiton 11 (line 21-27).

Example 28. *Given graph G in Fig. 4.1 , CH graph G' in Fig. 4.2 and SS-graph with $c_\phi(v_s)$ G^* in Fig. 4.7, we maintain CH graph for updating processing $\phi((v_{10}, v_7), G)$ changing to 2. We use Algorithm 10 to maintain CH graph. At first, we initialize Q. Because $\phi((v_{10}, v_7), G') = \phi((v_{10}, v_7), G)$, we assign $c_\phi((v_{10}, v_7))$ to 0. Then we run updateWeight procedure for (v_{10}, v_7). We check all relation type vertex of $N^+((v_{10}, v_7))$. For R_{10}, because $\phi((v_7, v_6), G')$ $= \phi((v_{10}, v_7), G') + \phi((v_{10}, v_6), G')$, we assign $c_\phi((v_7, v_6))$ to 0. For R_{11}, because $\phi((v_7, v_8), G') = \phi((v_{10}, v_7), G') + \phi((v_{10}, v_8), G')$, we assign $c_\phi((v_7, v_8))$ to 1. We*

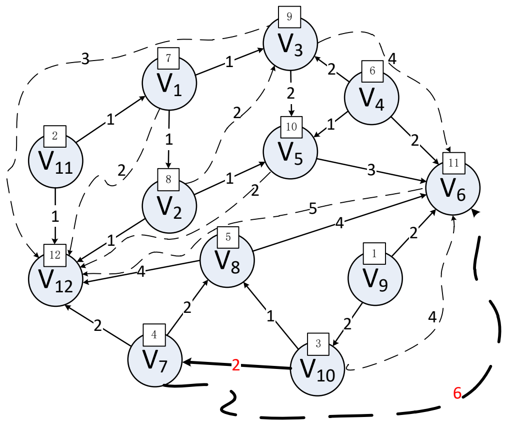


Figure 4.6: new CH graph after (v_7, v_{10})'s weight increasing

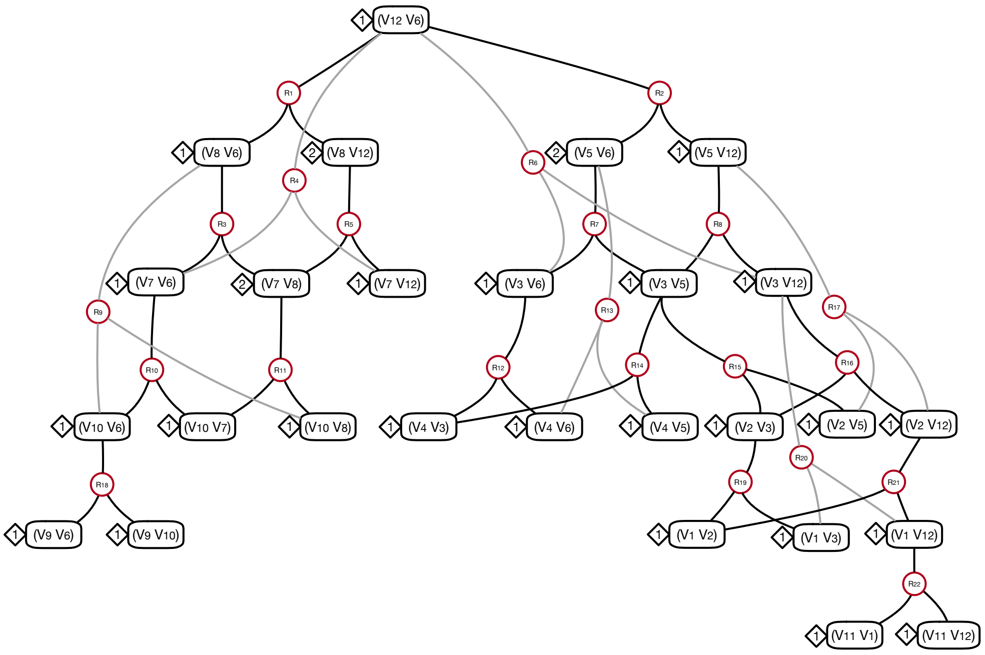


Figure 4.7: SS-Graph with $c_\phi(v_s)$ G^*

update $\phi((v_{10}, v_7), G')$ by $min\{\phi((v_{10}, v_7), G)\}$ to 2. After updateWeight procedure, we push $v_{(v_{10},v_7)}$ to Q. Next, we check relation type vertex of $N^+(v_{(v_{10},v_7)})$. For R_{10}, because $c_\phi((v_7, v_6)) = 0$, we run updateWieght procedure for $v_{(v_7,v_6)}$. For relation type vertex R_3 and R_4, we find $\phi((v_8, v_6), G') < \phi((v_7, v_8), G') + \phi((v_7, v_6), G')$ and $\phi((v_8, v_{12}), G') < \phi((v_7, v_{12}), G') + \phi((v_7, v_6), G')$. We do not reassign any value of v_ϕ. After this procedure, we push $v_{(v_7,v_6)}$ to Q. Finally, we pop $v_{(v_7,v_6)}$ from Q. And we find there is no more shortcut vertex v_s whose $c_\phi(v_s)$ is less than 0. We finish the loop because of empty Q. We finish algorithm and obtain new CH graph in Fig. 4.6.

Theorem 10. *Give a road network G with the weight of an edge e in G increased to k, Algorithm 10 computes $G'_{\oplus(e,k)}$ correctly.*

Proof. This theorem can be proved similarly as Theorem 8. □

Performance Guarantees. We arrived at the second maintain result of this chapter:

Theorem 11. *Give a road network G with the weight of an edge e in G increased to k, Algorithm 10 computes $G'_{\oplus(e,k)}$ in $O(\Delta \cdot (\log \Delta + \mathsf{deg}'_{\mathsf{max}}))$.*

Proof. This theorem can be proved similarly as Theorem 9. □

4.4.3 Algorithms without Materialized SS-Graph

In the above discussion, we always assume that the SS-Graph has been materialized and all the algorithms are designed based on the materialized SS-Graph. However, the space consumption of SS-Graph could be very large for big road networks. In this section, we remove this assumption and propose efficient algorithms without materialized SS-Graph.

Space consumption of SS-Graph G^*. We first analyze the space consumption of SS-Graph, which is shown in the following lemma:

Algorithm 11 $\mathsf{DCH_{scs}\text{-}WDec}$ $(G', (u,v), k)$

```
1: PriorityQueue $Q \leftarrow \emptyset$; $\phi((u,v),G) \leftarrow k$;
2: if $\phi((u,v),G') > k$ then
3:     $\phi((u,v),G') \leftarrow k$;
4:     $Q$.push($(u,v)$);
5: while $Q \neq \emptyset$ do
6:     $(u,v) \leftarrow Q$.pop();                    // w.l.o.g., assume $\gamma(u) > \gamma(v)$
7:     for each $w \in N^+(u)$ and $w \neq v$ do
8:         if $\gamma(w) < \gamma(v)$ then
9:             if $\phi((w,v),G') > \phi((u,w),G') + \phi((u,v),G')$ then
10:                $\phi((w,v),G') \leftarrow \phi((u,w),G') + \phi((u,v),G')$;
11:                $Q$.push($(w,v)$);
12:        else
13:            if $\phi((v,w),G') > \phi((u,w),G') + \phi((u,v),G')$ then
14:                $\phi((v,w),G') \leftarrow \phi((w,u),G') + \phi((u,v),G')$;
15:                $Q$.push($(v,w)$);
```

dataset	*NY*	*COL*	*FLA*	*CAL*	*E-US*	*W-US*	*C-US*	*US*
G' size(MB)	9.22	9.44	23.97	44.1	84.4	142	358	590
G^* size(MB)	146	121	209	562	1390	2085	12379	18199

Table 4.1: Space consumption of G' vs G^*

Lemma 9. *Given the SS-Graph G^* of a shortcut index G', the space of G^* is $O(E(G') + \sum_{v \in V(G')}(|N^+(v)| \cdot (|N^+(v) - 1|)/2))$.*

Proof. V^* contains two types vertices, V_{sc} and V_I. For each shortcut $e' \in E'$, we create a corresponding vertices $v^* \in V_{sc}$. The number of $|V_{sc}| = |V'|$. For each vertex v which is under contraction operation, we create a immediate vertex for each pair neighbours x and y with shortcut $(u,x) \in E'$ and $(u,y) \in E'$. It means for contraction operation of v, $(N^+(v))(N^+(v) - 1)/2$ immediate vertices are created. So $|V_I| = \sigma_{v \in V'}((N^+(v))(N^+(v) - 1)/2)$. In edge set E^*, there are only two kinds of edges, from V_{sc} to V_I and V_I to V_{sc}. For each immediate vertex $v \in V_I$, $|N^-(v)| = 2$ and $N^+(v)| = 1$. So the number of E^* is $3 \times V_I$. The space consumption is $|V^*|+|E^*|=|E'|+4\times V_I=|E'|+\sigma_{v\in V'}((N^+(v))(N^+(v)-1)/2$. □

We also compare the space consumption of G' and G^* regarding the datasets used in our experiments and show the results in Table 4.1. As shown in Table 4.1, G^* consumes much more space than G'. For example, . Obviously, the large space consumption of G^* will limit the scalability of our algorithm to handle big road networks. In this section, we propose efficient algorithms to handle the edge weight update without introducing any extra memory consumption. To avoid tedious presentation, we focus on the weight decrease case in this section and the proposed techniques can be easily extended for weight increase case.

Revisiting Algorithm 8, for a shortcut type vertex v_s, we only leverage G^* to retrieve the out-neighbor of v_s; for a supporting relation type vertex v_r, we leverage G^* to retrieve the in-neighbors and out-neighbor of v_r. Without loose of generality, assume v_s represents the shortcut (u, v) in G'. Essentially, these operations in Algorithm 8 regarding v_s equals to retrieve the supporting shortcuts that (u, v) supports and the supporting shortcut pairs that supports (u, v). According to Definiton 9, we can retrieve these shortcuts for (u, v) based on G' directly, which means we can achieve the same goal of Algorithm 8 on G' without G^*.

Algorithm. Following the above idea, our new algorithm is shown in Algorithm 11. Algorithm 11 shares the same framework with Algorithm 8. The differences locate in line 7-15. In Algorithm 11, for a shortcut (u, v), to retrieve all the shortcuts that (u, v) supports, we iterate all the out-neighbors w of u and the shortcut $(v, w)/(w, v)$ are the shortcuts that (u, v) supports based on Definiton 9 (line 7). For the shortcut $(v, w)/(w, v)$, another shortcut in the shortcut supporting pair involving (u, v) is $(u, w)/(w, u)$. Therefore, we retrieve these shortcuts in this way (line 9-10 and line 13-14). Obviously, Algorithm 11 updates the weight of shortcuts in $G'_{\oplus(e,k)}$ correctly based on the correctness of Algorithm 8. Besides, we have the following theorem regarding the space consumption of Algorithm 11:

Theorem 12. *The memory consumption of Algorithm 11 is $O(|G| + |G'|)$.*

Example 29. *Given graph G in Fig. 4.1 and CH graph G' in Fig. 4.2 , we maintain CH graph for updating processing $\phi((v_4, v_6), G)$ changing to 1. We use Algorithm 11 to maintain CH graph. At first, we change $\phi((v_4, v_6), G)$ to 1. Because $\phi((v_4, v_6), G') > 1$, we assign it to 1 and push it to Q. Then we check all relation type vertex of $N^+(v_4)$. For v_3, because of $\gamma(v_3) < \gamma(v_6)$ and $\phi((v_3, v_6), G') = 4 < \phi((v_4, v_6), G') + \phi((v_3, v_4), G')$, we assign $\phi((v_3, v_6), G')$ with 3 and push it to Q. For v_5, because of $\gamma(v_5) < \gamma(v_6)$ $\phi((v_4, v_6), G') + \phi((v_4, v_5), G') = 2 < \phi((v_5, v_6), G')$. We also assign $\phi((v_3, v_6), G')$ with 2 and push it to Q. Next, we pop (v_3, v_6) which is the top element of Q. For v_5 and v_{12}, we find $\phi((v_3, v_6), G') + \phi((v_3, v_5), G') = 5 > \phi((v_5, v_6), G')$ and $\phi((v_3, v_6), G') + \phi((v_3, v_{12}), G') = 6 > \phi((v_6, v_{12}), G')$. So we do not update any weight. After that, we pop the top element (v_5, v_6) from Q. For v_{12}, because $\phi((v_5, v_6), G') + \phi((v_5, v_{12}), G') = 4 < \phi((v_6, v_{12}), G')$. We assign $\phi((v_6, v_{12}), G')$ with 4 and push it to Q. Finally, we pop $(v_6, v_{12}$ from Q. There is not neighbor in $N^+(v_{(v_6, v_{12})})$. We finish algorithm and obtain new CH graph in Fig. 4.3*

4.5 Performance Studies

In this section, we compare our algorithm with the current state-of-the-art methods. All experiments are conducted on a machine with an Intel Xeon 2.8GHz CPU (10 cores) and 256 GB main memory running Linux (Red Hat Linux 4.4.7, 64bit).

Datasets. We use ten publicly available datasets from DIMACS. These data are from road network in the US. Table 4.2 provides the data details. The edge weight of graphs is time consumption.

Algorithms. We implement and compare 2 algorithms:

Name	**Corresponding Region**	Number of Nodes	Number of Edges
NY	New York City	264,346	733,846
COL	Colorado	435,666	1,057,066
FLA	Florida	1,070,376	2,712,798
CAL	California and Nevada	1,890,815	4,657,742
E-US	Eastern US	3,598,623	8,778,114
W-US	Western US	6,262,104	15,248,146
C-US	Central US	14,081,816	34,292,496
US	United States	23,947,347	58,333,344

Table 4.2: Datasets used in Experiments

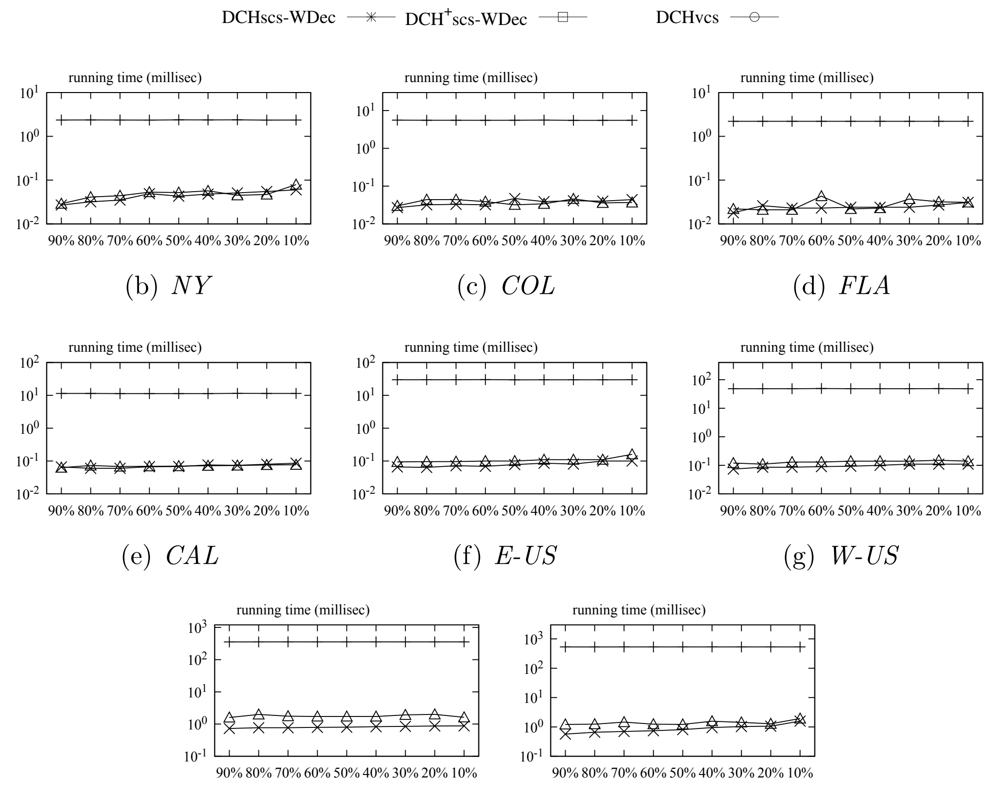
- DCH$_{vcs}$: DCH$_{vcs}$ is vertex-centric for streaming weight update.
- DCH$_{scs}$-WDec: DCH$_{scs}$-WDec is our algorithm to handle an edge weight decrease.
- DCH$^{+}_{scs}$-WDec: DCH$^{+}_{scs}$-WDec is our algorithm to handle an edge weight decrease without materialised SS-Graph.
- DCH$_{scs}$-WInc: DCH$_{scs}$-WInc is our algorithm to handle an edge weight increase.
- DCH$^{+}_{scs}$-WInc: DCH$^{+}_{scs}$-WInc is our algorithm to handle an edge weight increase without materialised SS-Graph.

Exp-1: Efficiency of Maintaining with Weight Decreasing In the second experiment, we test the index updating efficiency of CH Index by varying decreasing weight value of edges. For each dataset, we also generate 9 groups from 0.9 to 0.1 of query of weight decreasing processing set as follows: for group $0.i$, we random choose 1000 roads in the graph. For each queries of a road (x, y, z) whose ending vertices are x and y with weight z. We construct a $((x, y), z \times (0.i))$. We also record the average updating processing time for the 1000 queries in the corresponding processing set for all three algorithms.

The experimental results when increasing the road weight from 0.9 to 0.1 are shown in Fig. 4.8(a)-(h). We make the following observations. First, when the extent of declince be from 0.9 to 0.1, the updating processing time of DCH$_{vcs}$ is mostly stale. This is because when weight of a edge reduce, the DCH$_{vcs}$ need to check all shortcut which may be influenced by weight changing edge. Whatever the extent is large or small, the potential shortcuts which need to be checked are the same. Therefore, the time consumption for DCH$_{vcs}$ is stale. For DCH$_{scs}$-WDec and DCH$^{+}_{scs}$-WDec, time cost tends to increase moderately. This means when extent of decline is low, more weight of shortcut need to check and update for DCH$_{scs}$-WDec and DCH$^{+}_{scs}$-WDec. Especially, for Fig. 4.9(a), times cost with extent of decline 0.9 is 2 times more than 0.1. Secondly, DCH$_{scs}$-WDec and DCH$^{+}_{scs}$-WDec achieve a speedup of over 2 order of magnitude with DCH$_{vcs}$ for all extent of decline in 8 datasets. Especially, in *US*, DCH$_{scs}$-WDec and DCH$^{+}_{scs}$-WDec are fater than DCH$_{vcs}$ of 3 order. Third, efficiency of DCH$_{scs}$-WDec and DCH$^{+}_{scs}$-WDec are similar for *NY*, *COL*, *FLA* and *CAL*. For other 4 graphs, DCH$^{+}_{scs}$-WDec is under 2 times faster than DCH$_{scs}$-WDec. In brief, DCH$_{scs}$-WDec and DCH$^{+}_{scs}$-WDec have the same time complexity. The efficiency in experiment are similar.

Exp-2: Efficiency of Changing Maintain with Road Weight Increasing. In this experiment, we test the index updating efficiency of the algorithms by varying increasing weight value of edges. For each dataset, we generate 9 groups from 2 to 10 of weight increasing processing set as follows: For group i, we random choose 1000 roads in the graph. For each queries of a road (x, y, z) whose ending vertices are x and y with weight z. We construct weight updates $((x, y), z \times i)$ which means updating weight of (x, y) to $z \times i$. For three algorithm DCH$_{vcs}$,DCH$_{scs}$-WInc and DCH$^{+}_{scs}$-WInc, we record the average updating processing time for the 1000 queries in the corresponding processing set.

The experimental results when varing the groth rate of road weight from 2.0 to 10.0 are shown in Fig. 4.9(a)-(h). We make the following observations. First,



(b) *NY* (c) *COL* (d) *FLA*

(e) *CAL* (f) *E-US* (g) *W-US*

(h) *C-US* (i) *US*

Figure 4.8: Updating Processing Time (Varying extent of decline)

when the increasing rate grows up, the updating processing time of DCH$_{vcs}$, DCH$_{scs}$-WInc and DCH$^{+}_{scs}$-WInc are mostly stale. Second, DCH$_{scs}$-WInc and DCH$^{+}_{scs}$-WInc are 3-4 times quicker than DCH$_{vcs}$. Comparing with experiment 1, DCH$_{vcs}$ is slower 10 times with weight increasing than decreasing. This is because there are more seed vertcies in weight increasing processing. Third, DCH$_{scs}$-WInc and DCH$^{+}_{scs}$-WInc also have the similar time consumption.

4.6 Chapter Summary

Computing shortest path distance on dynamic road network is an important problem. The state-of-the-art solution, vertex-centric algorithm, has obvious

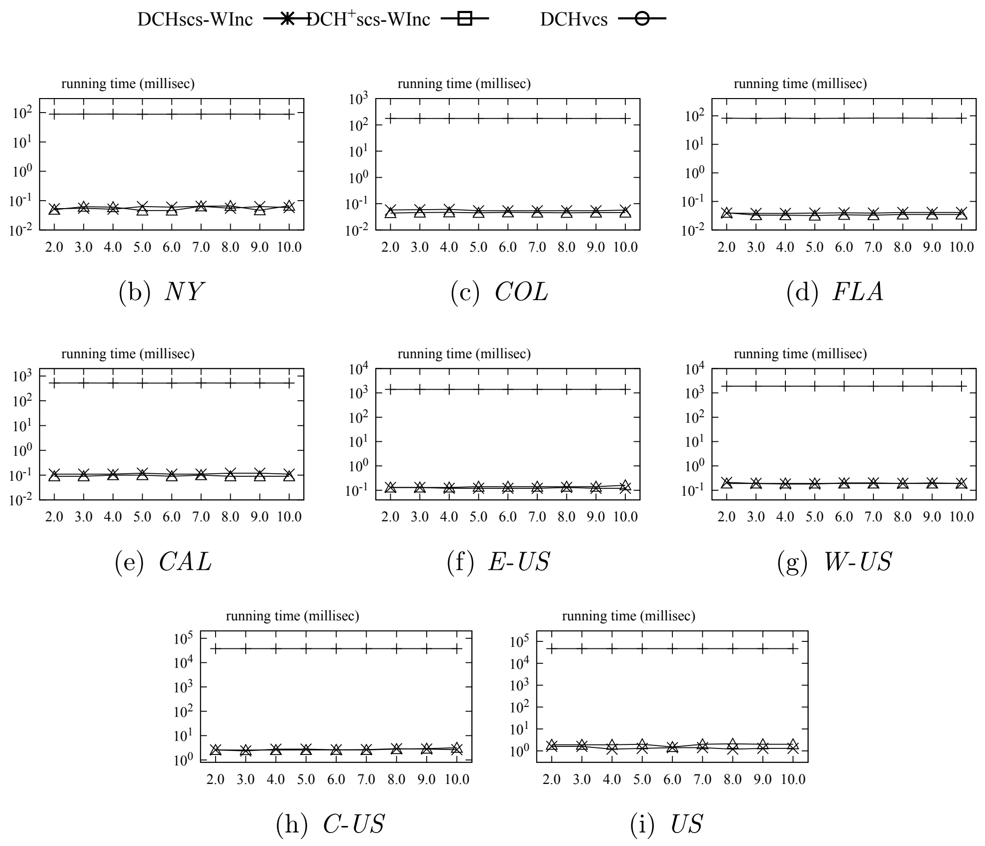


Figure 4.9: Updating Processing Time (Varying increasing rate)

drawback. Our shortcut-centric algorithm avoid abundant vertex and shortcut computing. We propose a shortcut-centric algorithm to detect affected shortcut by SS-graph. We also give a space saving algorithm which has similar time consumption. Our shortcut-centric algorithm has better theoretical bound and experiment performance.

Chapter 5

Path Skyline on Bicriteria Network

5.1 Overview

In this chapter, we introduce our proposed search based solution of skyline path on bicriteria road network by adopting an ordered path exploring strategy. This work is published in [46]. The rest of this chapter is organized as follow. Section 5.2 introduces the preliminaries and definition of the problem. Section 5.3 illustrates art-of-the-state algorithm of path skyline query. Section 5.4 proposes our idea of how to accelerate path skyline query algorithm. Section 5.5 practically evaluates PSQ and PSQ$^+$ algorithm and shows experimental results. Section 5.6 summarizes this chapter.

5.2 Preliminaries

A bicriteria network is a graph $G = (V, E)$ which is described in Section 1.4. Beside, two positive cost $\phi_1(u, v)$ and $\phi_2(u, v)$ are associated with each edge $(u, v) \in E$. For bicriteria network, we have new path and cost of path definition.

Definition 14. ***(Path and Cost of a Path)*** *Given a bicriteria network G,* *a path from node s to node t, denoted by p_{st}, is a sequence of nodes $(s = v_1, v_2, \cdots v_k = t)$ where $(v_i, v_{i+1}) \in E$ for each $1 \leq i < k$. The cost of a path $p_{st} = (s = v_1, v_2, \cdots v_k = t)$ w.r.t the l-th cost is $\phi_l(p_{st}) = \sum_{i=1}^{k-1} \phi_l(v_i, v_{i+1})$, where $l \in \{1, 2\}$.*

Definition 15. ***(Path domination)*** *Given two path p^1_{st} and p^2_{st} in G, p^1_{st} dominates p^2_{st}, denoted as $p^1_{st} \prec p^2_{st}$, iff*

- $\phi_1(p^1_{st}) < \phi_1(p^2_{st})$, *or*
- $\phi_1(p^1_{st}) = \phi_1(p^2_{st}) \wedge \phi_2(p^1_{st}) < \phi_2(p^2_{st})$

Definition 16. ***(Skyline Path)*** *Given a source s and a destination t in V, p_{st} is a skyline path w.r.t s and t if there exists no other path p'_{st} which dominates p_{st}.*

Definition 17. ***(Path Skyline Query)*** *Given a bicriteria network G, a source s and a destination t in V, the path skyline query w.r.t s and t, denoted by $\mathcal{PSQ}(s, t)$, identifies all the skyline paths from s to t in G.*

Problem Statement. In this chapter, we study the path skyline query problem and aim to devise an efficient algorithm to find $\mathcal{PSQ}(s, t)$ for a source s and a destination t in G.

5.3 Existing Solution

Among the algorithms for path skyline query in the literature, PSQ [29] is a representative algorithm in the labelling methods and is a building block for the two phases algorithms. Therefore, we focus on improving the efficiency of PSQ. PSQ is shown in Algorithm 12.

Given a source s and a destination t in G, PSQ enumerates all the skyline paths from s to t. To achieve this goal, for each node v which is possible on the

Algorithm 12 PSQ (Graph G, node s, node t)

```
1: for each $v \in V(G)$ do
2:     $v$.Skyline $\leftarrow \emptyset$;
3: PriorityQueue $Q \leftarrow \emptyset$;
4: $Q$.push($(s, 0, 0)$);
5: while $Q \neq \emptyset$ do
6:     $(u, c_1, c_2) \leftarrow Q$.pop();
7:     for each $v \in N(u)$ do
8:         $\phi_1(p_{su\to v}) \leftarrow c_1 + \phi_1(u, v)$; $\phi_2(p_{su\to v}) \leftarrow c_2 + \phi_2(u, v)$;
9:         if $p_{su\to v}$ is not dominated by any path in $v$.Skyline then
10:            insert $p_{su\to v}$ into $v$.Skyline;   // line 9-11 maintains the skyline paths
               for $v$
11:            verify the dominant relations and eliminate all dominated paths in
               $v$.Skyline;
12:        if $v$.Skyline is changed and $v \neq t$ then
13:            $Q$.push($(v, \phi_1(p_{su\to v}), \phi_2(p_{su\to v}))$);
14: output the skyline paths from source $s$ to $t$ by backtracking from $t$.Skyline.
```

skyline paths from s to t, PSQ uses v.Skyline to record the skyline paths from s to v (line 1-2). Starting from s, PSQ traverses the nodes of G to enumerate the skyline paths in an iterative manner. It maintains a priority queue Q (line 3) and each element (u, c_1, c_2) in Q represents an explored candidate skyline path p_{su} with the first cost as c_1 and the second cost as c_2. The priority function of Q is the lexicographic order of the two costs for the explored paths in Q. It first pushes $(s, 0, 0)$ into Q (line 4) as s is the source node. Then, it pops an element out from Q each time (line 6). For the popped element (u, c_1, c_2), it iterates each neighbor $v \in N(u)$ and conducts the edge relaxation for edge (u, v) (line 7-12). It first computes the first and the second cost of path $p_{su\to v}$ (line 8). Here, $p_{su\to v}$ denotes the path from s to v through the edge (u, v). PSQ checks whether $p_{su\to v}$ is a candidate skyline path w.r.t s and v (line 9). If $p_{su\to v}$ is a candidate skyline path, PSQ inserts $p_{su\to v}$ into v.Skyline and maintains the explored candidate skyline paths from s to v by eliminating all the dominated paths in v.Skyline (line 10-11). If v.Skyline changes due to $p_{su\to v}$, element $(v, \phi_1(p_{su\to v}), \phi_2(p_{su\to v}))$

is pushed into Q (line 12-13), which means $p_{su \to v}$ possibly leads to new candidate skyline paths for other nodes. The procedure terminates when Q is empty (line 5). At last, PSQ outputs the skyline paths from s to t by backtracking from t.Skyline (line 14).

Drawback of PSQ. In order to compute the skyline paths from s to t, PSQ intends to record the skyline paths w.r.t s and v for the node v which is possible on the skyline paths from s to t. To achieve this goal, PSQ computes the skyline paths w.r.t s and v gradually as follows: it keeps all the paths from s to v which are the skyline paths w.r.t s and v hitherto as a candidate set in v.Skyline. When a new path p_{sv} is explored (line 8), PSQ enlarges v.Skyline by inserting p_{sv} into it (line 9), verifies the dominant relations among the paths in v.Skyline and eliminates all the dominated paths in it (line 10). The issue of this approach is that PSQ only needs to record the skyline paths from s to v, but those paths that are not skyline paths w.r.t. s and v are also inserted and eliminated from v.Skyline repetitiously during the processing. As shown in Algorithm 12, maintaining v.Skyline is a basic procedure in PSQ and is performed for every new explored path in each iteration. As a result, lots of fruitless paths are maintained in PSQ, which makes it inefficient in terms of handling the path skyline query.

Fig. 5.1 shows the percentage of time used for the maintenance of v.Skyline in PSQ upon six real networks evaluated in our performance studies. We generate 1000 path skyline queries randomly for each real network and compare the average time used for the maintenance of v.Skyline and the average time used for other operations in PSQ for each query on the certain real network. As shown in Fig. 5.1, the time used for the maintenance of v.Skyline occupies most of the total time for the query processing in PSQ. For example, on BAY, maintaining v.Skyline occupies 47% of the total processing time and the percentage is 74% on

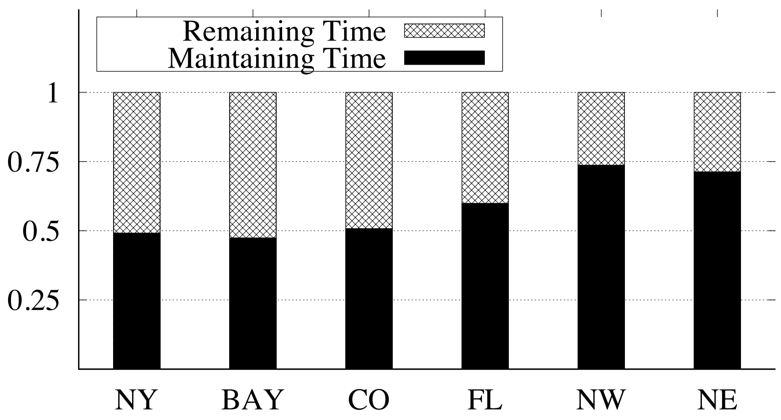


Figure 5.1: Percentage of time used for the maintenance of v.Skyline in PSQ

NW. The experimental results are consistent with our aforementioned analysis.

2P is a 2 phases algorithm. In phase 1, 2P find all support shortest path. A shortest path $p = (x, y)$ is supported, if and only if, for some λ_1 and λ_2, $\lambda_1 \cdot x + \lambda_2 \cdot y$ is minimum in all shortest path. When all supported shortest paths are found, we can prune paths by the following rule. For arbitrary two support shortest path $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, let $x_1 < x_2$ and $y_1 > y_2$, any path $p = (x, y)$ is dominated if $x > x_2$ and $y > y_1$. In phase 2, 2P finds skyline shortest paths during all the existing shortest paths.

5.4 Our New Approach

In order to address the drawback of PSQ, we propose a new algorithm, PSQ$^+$, for the path skyline query. Our algorithm can avoid the problem of fruitless candidate skyline paths maintenance in PSQ and guarantees that each path inserted into v.Skyline is exactly a skyline path from s to v. Before presenting our approach, we first introduce the following two lemmas:

Lemma 10. *Let p_{sv} be a skyline path from s to v in G, then $\phi_2(p_{sv})$ is minimum among all the second costs of the paths from s to v with the first cost as $\phi_1(p_{sv})$.*

Proof. We can prove this lemma by contradiction. Assume that there exists another path p'_{sv} such that $\phi_1(p'_{sv}) = \phi_1(p_{sv})$ and $\phi_2(p'_{sv}) < \phi_1(p_{sv})$. According to Definiton 15, we can derive $p'_{sv} \prec p_{sv}$, which contradicts with the given condition that p_{sv} is a skyline path. Thus, the lemma holds. □

Lemma 11. *Let p^1_{sv} and p^2_{sv} be two skyline paths from s to v in G, for a path p^3_{sv}, assume that $\phi_1(p^1_{sv}) < \phi_1(p^2_{sv}) < \phi_1(p^3_{sv})$ and there exists no path with the first cost in between $\phi_1(p^2_{sv})$ and $\phi_1(p^3_{sv})$, if $\phi_2(p^3_{sv}) < \phi_2(p^2_{sv})$, then $p^1_{sv} \not\prec p^3_{sv}$ and $p^2_{sv} \not\prec p^3_{sv}$; otherwise, $p^2_{sv} \prec p^3_{sv}$.*

Proof. The dominance relation between p^2_{sv} and p^3_{sv} can be proved directly based on Definiton 15. Then, we focus on the dominance relation between p^1_{sv} and p^3_{sv}. Since p^1_{sv} and p^2_{sv} are two skyline paths and $\phi_1(p^1_{sv}) < \phi_1(p^2_{sv})$, according to Definiton 15, we have $\phi_2(p^2_{sv}) < \phi_2(p^1_{sv})$. And if $\phi_2(p^3_{sv}) < \phi_2(p^2_{sv})$, we have $\phi_2(p^3_{sv}) < \phi_2(p^1_{sv})$. Then, we can derive that $p^1_{sv} \not\prec p^3_{sv}$ according to Definiton 15. Thus, the lemma holds. □

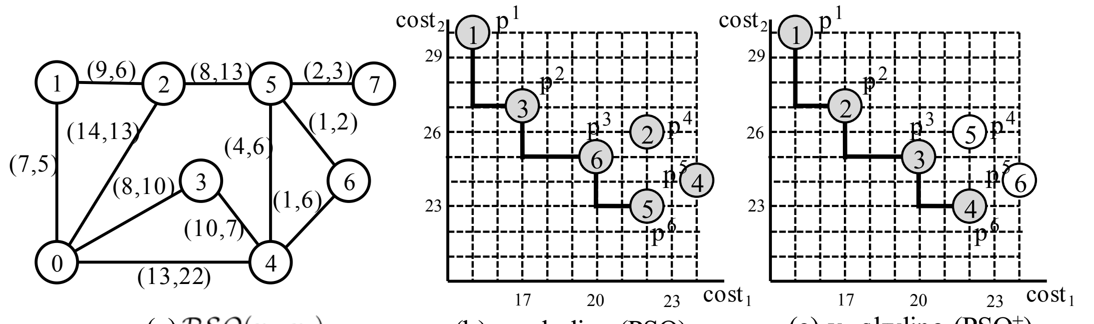
According to Lemma 10, for a specific the first cost value c_1, all the paths from s to v with the first cost value as c_1 but without the minimum value of the second cost among these paths are not skyline paths. According to Lemma 11, given two skyline paths p^1_{sv} and p^2_{sv}, we can determine whether a path p^3_{sv} is a skyline path by p^2_{sv} alone if these paths are given in the increasing order based their first costs. Combining these two considerations together, we can derive that if we handle the paths from s to v in the lexicographic order of their two costs, we can determine whether a path p_{sv} is a skyline path w.r.t s and v based on the previous skyline path w.r.t s and v that has been determined. In other words, we can determine whether a path is a skyline path immediately when we handle it. As a result, we can avoid the problem of fruitless candidate skyline paths maintenance existing in Algorithm 12. The following problem is how we can handle the paths from s to v in the lexicographic order of their two costs.

Algorithm 13 PSQ$^+$ (Graph G, node s, node t, SkylinePath $\mathcal{P}$)

```
1: for each $v \in V(G)$ do
2:     $v$.Skyline $\leftarrow \emptyset$;
3: PriorityQueue $Q \leftarrow \emptyset$;
4: $Q$.push($(s, s, 0, 0)$);
5: while $Q \neq \emptyset$ do
6:     $(u, v, c_1, c_2) \leftarrow Q$.pop();
7:     if $P_{su\to v}$ is not dominated by the last path in $v$.Skyline then
8:         insert $P_{su\to v}$ into $v$.Skyline;
9:         if $v \neq t$ then
10:            for each $w \in N(v)$ do
11:                $\phi_1(P_{sv\to w}) \leftarrow c_1 + \phi_1(v, w)$;
12:                $\phi_2(P_{sv\to w}) \leftarrow c_2 + \phi_2(v, w)$;
13:                $Q$.push($(v, w, \phi_1(P_{sv\to w}), \phi_2(P_{sv\to w}))$);
14: output the skyline paths from source $s$ to $t$ by backtracking from $t$.Skyline.
```

We can achieve this by postponing the process of v.Skyline from edge relaxation on edge (u, v) to edge (v, w). Following this idea, our improved skyline paths enumeration algorithm, PSQ$^+$, is shown in Algorithm 13.

Algorithm Design. Procedure PSQ$^+$ follows a similar framework as PSQ. v.Skyline records all the skyline paths from s to v and is initialized as $\emptyset$ (line 1-2). PSQ$^+$ maintains a priority queue Q (line 3) and each element (u, v, c_1, c_2) in Q represents a path $p_{su\to v}$ with the first cost as c_1 and the second cost as c_2. The priority function is the lexicographic order of the two costs for the explored paths in Q. It first pushes $(s, s, 0, 0)$ into Q (line 4) as s is the starting node. Then, it pops an element out from Q each time (line 6). For each popped element (u, v, c_1, c_2), PSQ$^+$ checks whether $p_{su\to v}$ is dominated by the last path in v.Skyline (line 7). If $p_{su\to v}$ is a skyline path, PSQ$^+$ inserts $p_{su\to v}$ into v.Skyline (line 8) and conducts the edge relaxation for each edge (v, w) where $w \in N(v)$ (line 10-13). For an edge (v, w), it first computes the first and the second cost of path $p_{sv\to w}$ (line 11-12) and then pushes the element $(v, w, \phi_1(p_{sv\to w}), \phi_2(p_{sv\to w}))$ into Q (line 13). The procedure terminates when Q is empty (line 5). At last,



(a) $\mathcal{PSQ}(v_0, v_7)$ (b) v_5.skyline (PSQ) (c) v_5.skyline (PSQ$^+$)

Figure 5.2: PSQ and PSQ$^+$

PSQ$^+$ outputs the skyline paths from s to t by backtracking from t.Skyline (line 14). Since PSQ$^+$ can avoid the fruitless candidate skyline paths maintenance problem, it does not have explicit maintenance procedure for v.Skyline compared with PSQ.

Example 30. *Consider the graph G in Fig. 5.2 (a) and assume a query $\mathcal{PSQ}(v_0, v_7)$ on G. Fig. 5.2 (b) and (c) show the skyline paths maintained in v_5.*Skyline *during the query processing for $\mathcal{PSQ}(v_0, v_7)$ in* PSQ *and* PSQ$^+$*, respectively. In Fig. 5.2 (b), x-axis represents the first cost ϕ_1 of a path, y-axis represents the second cost ϕ_2 of a path and each point represents a skyline path with its first and second costs. For example, the first and second cost of the path p^1 are 15 and 30, respectively. In Fig. 5.2(b), we also show the order in which* PSQ *maintains v_5.*Skyline *by the number shown in the point.* PSQ *processes the paths in the order $p^1, p^4, p^2, p^5, p^6, p^3$. As shown in Fig. 5.2 (b), when processing p^4, since only p^1 is in v_5.*Skyline, PSQ *considers p^4 as a skyline path and inserts it into v_5.*Skyline. PSQ *processes p^5 in the similar way. When processing p^6,* PSQ *finds that p^4 and p^5 are dominated by p^6, then, it inserts p^6 into v_5.*Skyline *and removes p^4 and p^5 from it. In Fig. 5.2 (b), although p^4 and p^5 are not skyline paths,* PSQ *inserts them into v_5.*Skyline *during the processing. On the other hand,* PSQ$^+$ *processes the paths in the order $p^1, p^2, p^3, p^6, p^4, p^5$, which is shown in Fig. 5.2 (c). For*

PSQ$^+$, *when processing* p^4 *and* p^5, p^6 *has been in* v_5.Skyline. *Thus,* p^4 *and* p^5 *are not inserted into* v_5.Skyline. *As shown in Fig. 5.2,* PSQ$^+$ *can avoid the fruitless candidate skyline paths maintenance problem in* PSQ.

Lemma 12. *Given a path skyline query* $\mathcal{PSQ}(s,t)$, *for an arbitrary node* v, *Algorithm 13 explores the paths from* s *to* v *in line 7 in lexicographic order of their two costs.*

Proof. We can prove this by contradiction. Without loss of generality, assume that p^1_{sv} and p^2_{sv} are two paths from s to v with $\phi_1(p^1_{sv}) < \phi_1(p^2_{sv})$ but Algorithm 13 explores p^2_{sv} earlier than p^1_{sv}. Then we have two cases: (1) when Algorithm 13 explores p^2_{sv}, p^1_{sv} is in Q. In this case, exploring p^2_{sv} earlier than p^1_{sv} contradicts with the property of priority queue Q. (2) when Algorithm 13 explores p^2_{sv}, p^1_{sv} is not in Q. Let p_{su} be a subpath of p^1_{sv} that is in Q when Algorithm 13 explores p^2_{sv}. According to $\phi_1(p^1_{sv}) < \phi_1(p^2_{sv})$, we have $\phi_1(p_{su}) < \phi_1(p^2_{sv})$. Based on the property of priority queue, Algorithm 13 explores p_{su} before p^2_{sv} and extends the paths through p_{su} until p^1_{sv} in Q. Thus, it is impossible that p^1_{sv} is not in Q when Algorithm 13 explores p^2_{sv}. Combining these two cases together, Algorithm 13 explores p^1_{sv} before p^2_{sv}. Similarly, we can prove the case that $\phi_1(p^1_{sv}) = \phi_1(p^2_{sv})$ and $\phi_2(p^1_{sv}) < \phi_2(p^2_{sv})$. Thus, Algorithm 13 explores the paths from s to v in line 7 in lexicographic order of their two costs and the lemma holds. □

Theorem 13. *Given a path skyline query* $\mathcal{PSQ}(s,t)$, *for an arbitrary node* v, *Algorithm 13 records the skyline paths from* s *to* v *in* v.Skyline.

Proof. Without loss of generality, let p_{sv} is an arbitrary path in line 7. If p_{sv} is inserted into v.Skyline in line 8, then we have p_{sv} is not dominated by the skyline path that previously inserted into v.Skyline. According to Lemma 11, p_{sv} is not dominated by any other paths in v.Skyline. Besides, based on Lemma 12, Algorithm 13 explores the paths from s to v in line 7 in lexicographic order. Then, the paths explored after p_{sv} cannot dominate p_{sv}. Thus, Algorithm 13 stores the skyline paths from s to v in v.Skyline and the theorem holds. □

Corollary 1. *Given a path skyline query $\mathcal{PSQ}(s,t)$, for an arbitrary node v, the paths that are not skyline paths from s to v are not maintained in Algorithm 13.*

Proof. This corollary can be proved directly based on Theorem 13 and the procedure of Algorithm 13. □

Theorem 14. *Given a path skyline query $\mathcal{PSQ}(s,t)$, Algorithm 13 answers $\mathcal{PSQ}(s,t)$ correctly.*

Proof. Based on Theorem 13, we can derive that t.Skyline stores the skyline paths from s to t. Besides, base on Definiton 16, if there is a skyline path p_{st} from s to t, then any subpath p_{sv} of p_{st} is also a skyline path w.r.t s and v, we can prove similarly as Theorem 13 that all these subpaths are inserted in v.Skyline. Since we conduct the edge relaxation for every edge (v,w) where $w \in N(v)$ in line 10, then, all the skyline paths w.r.t s to t extending from p_{sv} will be explored in Algorithm 13, which means all the skyline paths from s to t are stored in t.Skyline when Algorithm 13 terminates. In line 14, Algorithm 13 retrieves all the skyline paths by backtracking t.Skyline, thus, Algorithm 13 computes all the skyline paths from s to t correctly and the theorem holds. □

5.5 Performance Studies

In this section, we compare our proposed algorithm with other path skyline query algorithms. All experiments are conducted on a machine with an Intel Xeon 3.4GHz CPU (8 cores) and 32 GB main memory running Linux (Red Hat Linux 4.4.7, 64bit).

Name	**Corresponding Region**	Number of Nodes	Number of Edges
NY	New York City	264,346	733,846
BAY	San Francisco Bay Area	321,270	800,172
COL	Colorado	435,666	1,057,066
FLA	Florida	1,070,376	2,712,798
NW	Northwest USA	1,207,945	2,840,208
NE	Northeast USA	1,524,453	3,897,636
CAL	California and Nevada	1,890,815	4,657,742
LKS	Great Lakes	2,758,119	6,885,658

Table 5.1: Datasets used in Experiments

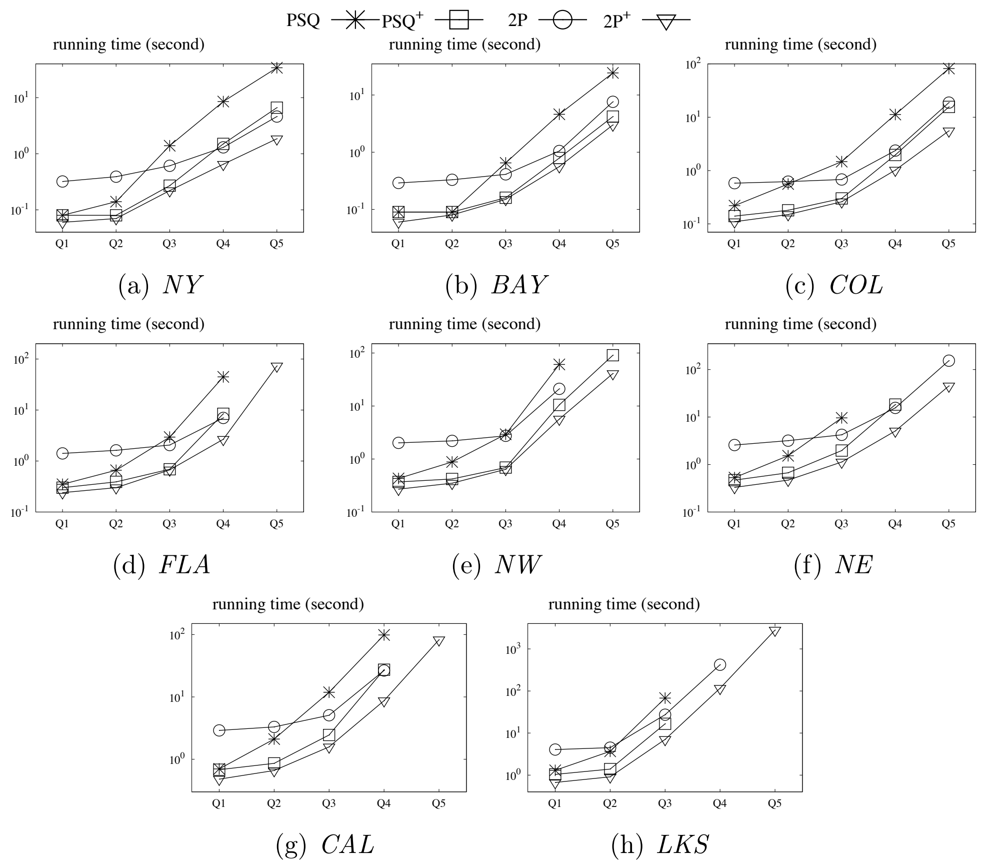
Datasets. We evaluate the algorithms on eight publicly available datasets from DIMACS [1], each of which corresponds to a part of the road network in the US. In each dateset, intersections and endpoints are represented by nodes and the roads connecting these intersections or road endpoints are represented by undirected edges. For each edge (u, v) in the dateset, two positive costs are associated with (u, v), which represent the physical distance and transit time between two nodes in the network, respectively. The details of the datasets are shown in Table 5.1.
Algorithms. We implement and compare the following four algorithms:

- PSQ: Algorithm 12 (Section 5.3)
- PSQ$^+$: Algorithm 13 (Section 5.4)
- 2P: A representative two phase algorithm proposed in [49], which uses PSQ as a subroutine to enumerate the skyline paths.
- 2P$^+$: The two phase algorithm in which PSQ is replaced with PSQ$^+$ to enumerate the skyline paths. Except PSQ and PSQ$^+$, all the remaining parts in 2P and 2P$^+$ are the same.

All algorithms are implemented in C++ and compiled with GNU GCC 4.4.7 using optimization level 3. The time cost of the algorithm is measured as the

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

amount of elapsed wall-clock time during the program's execution. For each test, we set the maximum running time for each test to be 30,000s. If an algorithm cannot finish the query processing in the time limit, we do not show its processing time in the figures.



(a) *NY* (b) *BAY* (c) *COL*

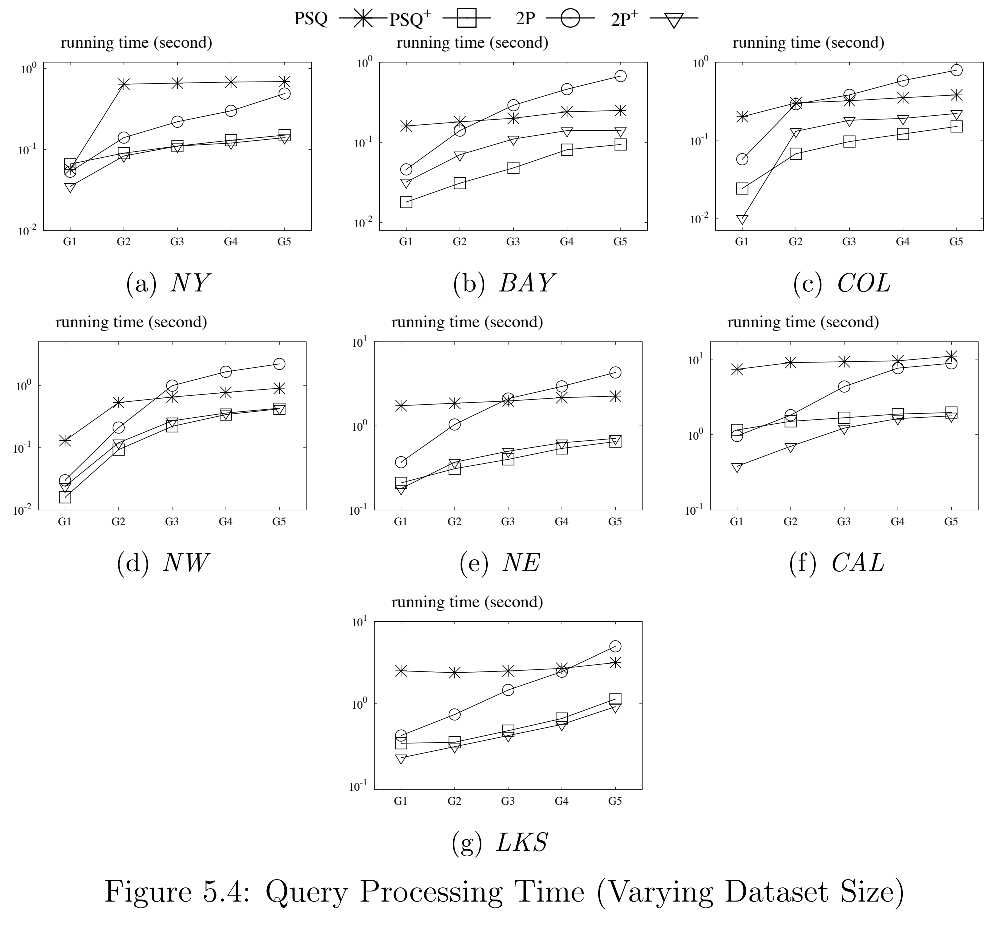
(d) *FLA* (e) *NW* (f) *NE*

(g) *CAL* (h) *LKS*

Figure 5.3: Query Processing time (Varying Query Location)

Exp-1: Efficiency. In this experiment, we evaluate the efficiency of these four algorithms. Since the locations of the source node and the destination node w.r.t the cost criteria influence the efficiency of various algorithm. We generate the queries for the experiments as follows: for each dataset, we generate 5 sets of queries $Q_1, Q_2, ..., Q_5$. To generate these query sets for a specific dataset, we first figure out the maximum distance l_{max} and the minimum distance l_{min} between any two nodes of the network w.r.t two criteria, respectively. For each criterion, let $x = (l_{max}/l_{min})^{1/10}$, and we generate 100 queries (s, t) to form Q_i for each $1 \leq i \leq 5$ such that the distance between s and t w.r.t the specific criterion is in $[x^{i-6} \cdot l_{max}, x^{i-5} \cdot l_{max}]$. In this way, we can ensure that the source node and destination node of these queries in Q_i are located more remotely than these in Q_{i-1} in the dateset w.r.t the specific criterion for all $1 < i \leq 5$. We execute all algorithms on the datasets and computes the average processing time for each query. The results are shown in Fig. 5.3.

Fig. 5.3 shows that: (1) For all test cases, PSQ$^+$ outperforms PSQ and 2P$^+$ outperforms 2P. For example, on *FLA* (Fig. 5.3 (d)), the processing time of PSQ is 44.7s for Q_4 while that of PSQ$^+$ is 8.45s. On the same dataset, the processing time of 2P for Q_4 is 7s while that of 2P$^+$ is 2.67s. On *NW* (Fig. 5.3 (e)), both PSQ and 2P cannot finish the processing of Q_5 in the time limit while PSQ$^+$ and 2P$^+$ can finish the query processing. This is because the operation of maintaining skyline paths in PSQ and 2P is time-consuming while PSQ$^+$ and 2P$^+$ can avoid the problem of fruitless skyline maintenance completely. (2) For all evaluated four algorithms, the query processing time increases as the distance between the source node and destination node in the query increases. For example, on *COL* (Fig. 5.3 (c)), the query processing times of Q_1 for PSQ, PSQ$^+$, 2P and 2P$^+$ are 0.22s, 0.14s, 0.58s, 0.11s, respectively. However, the query processing time of Q_5 for them are 81.8s, 15.6s, 18.8s, 5.55s. This is because as the distance between the source node and destination node increases, all the algorithms have to visit more nodes in the networks and more paths are explored during the query processing. As a result, they have to consume more time to process the query. (3) The performance gap between PSQ and PSQ$^+$ (2P and 2P$^+$) increases as the distance between the source node and destination node in the query increases. For example, on *NW* (Fig. 5.3 (e)), regarding PSQ and PSQ$^+$, the processing time of PSQ for Q_1 is 0.43s and that of PSQ$^+$ is 0.37s, the gap between processing times is 0.06s. For Q_4, the processing time of PSQ is 60.89s and that of PSQ$^+$ is 10.5s, the gap between processing times is 50.39s. Regarding 2P and 2P$^+$, the processing time of 2P for Q_1 is 1.68s and that of 2P$^+$ is 0.26s, the gap between processing times is 1.42s. For Q_4, the processing time of 2P is 21.1s and that of 2P$^+$ is 5.68s, the gap between processing times is 15.42s. This is because with the distance between the source node and destination node in the query increases, more paths are explored, as a result, the time saved by avoiding the fruitless skyline maintenance increases as well. Therefore, the performance gap increases with the query distance increases.



(a) *NY* (b) *BAY* (c) *COL*

(d) *NW* (e) *NE* (f) *CAL*

(g) *LKS*

Figure 5.4: Query Processing Time (Varying Dataset Size)

Exp-2: Scalability. In this experiment, we evaluate the scalability of the four algorithms when the size of the network increases. To do this, we first divide each dataset into a 5×5 gird. Then we create five networks using the $1\times1, 2\times2, ..., 5\times5$ grids in the middle of the dataset. These networks are donated as $G_1, G_2, ..., G_5$, respectively. For each $i \in [1, 4]$, it is obvious that G_i is contained in G_{i+1}. For the query set of each dataset, we generate 100 queries randomly from G_1. We execute all the algorithms on these networks and compute the average processing time for each query. The results are shown in Fig. 5.4. Note that G_1 of *FLA* does not contains any nodes and we cannot generate queries for it, thus, the results on *FLA* are not shown in Fig. 5.4.

From Fig. 5.4, we can observe that the average processing time for each query of all the four algorithms increases stably as the size of the network increases. This is because as the size of the network increases, more nodes and paths are explored during the query processing. As a result, the average processing time increases as the size of network increases. In the meantime, for all the test cases, PSQ$^+$ outperforms PSQ and 2P$^+$ outperforms 2P. This is also because PSQ and 2P involve the time-consuming operations for the maintenance of skyline paths while PSQ$^+$ and 2P$^+$ can completely avoid the fruitless skyline maintenance problem.

5.6 Chapter Summary

In this chapter, we study the path skyline query problem in bicriteria networks. In the literature, PSQ is a fundamental algorithm for path skyline query and is also used as a building block for the afterwards proposed algorithms. We investigate the drawbacks in PSQ and propose a new algorithm, PSQ$^+$, to answer the path skyline query. By adopting an ordered path exploring strategy, our algorithm can totally avoid the fruitless path maintenance problem in PSQ. We evaluate our proposed algorithm on real networks and the experimental results demonstrate the efficiency of our proposed algorithm. Besides, the experimental results also demonstrate the significant performance improvement of 2P which uses PSQ as a building block after substituting PSQ^+ for PSQ.

Chapter 6

EPILOGUE

In this thesis, we study efficient shortest distance query processing and indexing on large road network. We propose more efficient algorithms on shortest distance query on the static and dynamic road network, and path skyline query on bicriteria network. For shortest path query on static network, we study the shortest path distance problem on road network. At first, for a weighted undirected graph, we propose an efficient H2H index algorithm which can bound query processing by treewith $O(w)$. Hierarchy-based algorithm needs large search space for long-distance query. Hop-based algorithm results in high computational waste for short-distance query. H2H solves the drawback of both algoriothms. Besides query processing, we also propose an efficient index construction algorithm for H2H index. By using partial labels generated for former vertices to compute new label, we reduce construction time and bound it with worst-case time complexity. For shortest path problem on dynamic road network which is often happening in morning peak, we propose a maintaining algorithm for CH index. Compare to existing vertex-centric algorithm, our shortcut-centric algorithm has better theoretical bound and experiment performance. Vertex-centric algorithm may deal with redundant vertex and shortcut. However, shortcut-centric algorithm is strict bounded by number of affected shortcuts and degree. On bicriteria road network which has two certeria for weight of each road, we propose PSQ^+ which is an advanced path skyline query algorithm. In this algorithm, we adopt an ordered path exploring strategy to avoid the fruitless path maintenance.

In future work, we want to extend our existing techniques to the practical problems like k-th nearest neighbors, optimal location on road networks and betweenness centrality on road networks. These topics have some similar property to our existing solution. For example, they have similar datasets. Road network is a special degree-bounded network. It is not a planar graph but has a similar property. The vertex cut of road network is much smaller than the social network. Hence, in future work, we want to propose more efficient algorithms by the property to accelerate practial and important real-life problem.

BIBLIOGRAPHY

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proc. of ISEA'11*, pages 230–241, 2011.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *Proc. of ESA'12*, pages 24–35, 2012.

[3] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Proc. of ALENEX'14*, pages 147–154, 2014.

[4] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proc. of SIGMOD'13*, pages 349–360, 2013.

[5] T. Akiba, C. Sommer, and K.-i. Kawarabayashi. Shortest-path queries for complex networks: Exploiting low tree-width outside the core. In *Proc. of EDBT'12*, pages 144–155, 2012.

[6] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[7] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[8] J. Arz, D. Luxen, and P. Sanders. Transit node routing reconsidered. In *Proc. of SEA'13*, pages 55–66, 2013.

[9] W.-T. Balke, U. Güntzer, and J. Zheng. Efficient distributed skylining for web information systems. *Advances in Database Technology-EDBT 2004*, pages 573–574, 2004.

[10] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm Engineering - Selected Results and Surveys*, pages 19–80. 2016.

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

[12] M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Proc. of LASTI'00*, pages 88–94, 2000.

[13] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–21, 1993.

[14] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[15] H. L. Bodlaender. Treewidth: Characterizations, applications, and computations. In *Graph-Theoretic Concepts in Computer Science*, pages 1–14, 2006.

[16] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of ICDE*, pages 421–430, 2001.

[17] J. Brumbaugh-Smith and D. Shier. An empirical investigation of some bicriterion shortest path algorithms. *European Journal of Operational Research*, 43(2):216–224, 1989.

[18] C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *Proceedings of SIGMOD*, pages 203–214, 2005.

[19] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *Proceedings of SIGMOD*, pages 503–514, 2006.

[20] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao. The exact distance to destination in undirected world. *The VLDB Journal*, 21(6):869–888, 2012.

[21] J. C. Clímaco and M. Pascoal. Multicriteria path and tree problems: discussion on exact algorithms and applications. *International Transactions in Operational Research*, 19(1-2):63–98, 2012.

[22] J. C. N. Climaco and E. Q. V. Martins. A bicriterion shortest path algorithm. *European Journal of Operational Research*, 11(4):399–404, 1982.

[23] M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *Or Spectrum*, 22(4):425–460, 2000.

[24] V. Gabrel and D. Vanderpooten. Enumeration and interactive selection of efficient paths in a multiple criteria graph for scheduling an earth observing satellite. *European Journal of Operational Research*, 139(3):533–542, 2002.

[25] R. G. Garroppo, S. Giordano, and L. Tavanti. A survey on multi-constrained optimal path computation: Exact and approximate algorithms. *Computer Networks*, 54(17):3081–3107, 2010.

[26] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proc. of WEA'08*, pages 319–333, 2008.

[27] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[28] A. V. Goldberg and C. Harrelson. Computing the shortest path: *A* search meets graph theory. In *Proc. of SODA'05*, pages 156–165, 2005.

[29] P. Hansen. Bicriterion path problems. In *Multiple criteria decision making theory and application*, pages 109–127. 1980.

[30] M. S. Hassan, W. G. Aref, and A. M. Aly. Graph indexing for shortest-path finding over dynamic sub-graphs. In *Proceedings of SIGMOD*, pages 1183–1197, 2016.

[31] R. Hassin. Approximation schemes for the restricted shortest path problem. *Math. Oper. Res.*, 17(1):36–42, 1992.

[32] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *Proceedings of ICDE*, pages 66–66, 2006.

[33] S. Jang and J. Yoo. Processing continuous skyline queries in road networks. In *Computer Science and its Applications*, pages 353–356, 2008.

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

[35] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Trans. Knowl. Data Eng.*, 14(5):1029–1046, 2002.

[36] A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. *Electronic Notes in Discrete Mathematics*, 8:54–57, 2001.

[37] H.-P. Kriegel, M. Renz, and M. Schubert. Route skyline queries: A multi-preference path planning approach. In *Proceedings of ICDE*, pages 261–272, 2010.

[38] Y. Li, L. H. U, M. L. Yiu, and N. M. Kou. An experimental study on hub labeling based shortest path algorithms. *PVLDB*, 11(4):445–457, 2017.

[39] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *Proceedings of ICDE*, pages 502–513, 2005.

[40] E. Q. V. Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

[41] W. Matthew Carlyle and R. Kevin Wood. Near-shortest and k-shortest simple paths. *Networks*, 46(2):98–109, 2005.

[42] J. Mote, I. Murthy, and D. L. Olson. A parametric approach to solving bicriterion shortest path problems. *European Journal of Operational Research*, 53(1):81–92, 1991.

[43] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *Proc. of SODA'12*, pages 209–222, 2012.

[44] M. Müller-Hannemann and K. Weihe. On the cardinality of the pareto set in bicriteria shortest path problems. *Annals of Operations Research*, 147(1):269–286, 2006.

[45] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 709–724, 2018.

[46] D. Ouyang, L. Yuan, F. Zhang, L. Qin, and X. Lin. Towards efficient path skyline computation in bicriteria networks. In *Database Systems for Advanced Applications - 23rd International Conference, DASFAA 2018, Gold Coast, QLD, Australia, May 21-24, 2018, Proceedings, Part I*, pages 239–254, 2018.

[47] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *Proceedings of VLDB*, pages 15–26, 2007.

[48] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *Proceedings of VLDB*, pages 253–264, 2005.

[49] A. Raith and M. Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research*, 36(4):1299–1331, 2009.

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

[51] N. Robertson and P. D. Seymour. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

[52] N. Robertson and P. D. Seymour. Graph minors iii: Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[53] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proc. of SIGMOD'08*, pages 43–54, 2008.

[54] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *Proc. of ESA'05*, pages 568–579, 2005.

[55] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *IEEE Trans. Knowl. Data Eng.*, 22(8):1158–1175, 2010.

[56] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.

[57] A. Sedeno-Noda and A. Raith. A dijkstra-like method computing all extreme supported non-dominated solutions of the biobjective shortest path problem. *Computers & Operations Research*, 57:83–94, 2015.

[58] P. Serafini. Some considerations about computational complexity for multi objective combinatorial problems. In *Recent advances and historical development of vector optimization*, pages 222–232. 1987.

[59] M. Shekelyan, G. Jossé, and M. Schubert. Linear path skylines in multicriteria networks. In *Proceedings of ICDE*, pages 459–470, 2015.

[60] A. J. Skriver. A classification of bicriterion shortest path (bsp) algorithms. *Asia-Pacific Journal of Operational Research*, 17(2):199, 2000.

[61] A. J. Skriver and K. A. Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers & Operations Research*, 27(6):507–524, 2000.

[62] S. Storandt. Route planning for bicycles-exact constrained shortest paths made practical via contraction hierarchy. In *ICAPS*, volume 4, page 46, 2012.

[63] Z. Tarapata. Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms. *International Journal of Applied Mathematics and Computer Science*, 17(2):269–287, 2007.

[64] E. Ulungu, J. Teghem, P. Fortemps, and D. Tuyttens. Mosa method: a tool for solving multiobjective combinatorial optimization problems. *Journal of multicriteria decision analysis*, 8(4):221, 1999.

[65] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Efficient route planning on public transportation networks: A labelling approach. In *Proc. of SIGMOD'15*, pages 967–982, 2015.

[66] F. Wei. Tedi: efficient shortest path query answering on graphs. In *Proc. of SIGMOD'10*, pages 99–110. ACM, 2010.

[67] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke. Reachability and time-based path queries in temporal graphs. In *Proc. of ICDE'16*, pages 145–156, 2016.

[68] 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. *PVLDB*, 5(5):406–417, 2012.

[69] T. Xia and D. Zhang. Refreshing the sky: the compressed skycube with efficient support for frequent updates. In *Proceedings of SIGMOD*, pages 491–502, 2006.

[70] Y. Xiang. Answering exact distance queries on real-world graphs with bounded performance guarantees. *The VLDB Journal*, 23(5):677–695, 2014.

[71] B. Yang, C. Guo, C. S. Jensen, M. Kaul, and S. Shang. Multi-cost optimal route planning under time-varying uncertainty. In *Proceedings of ICDE*, 2014.

[72] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proceedings of VLDB*, pages 241–252, 2005.

[73] D. Zhang, D. Yang, Y. Wang, K.-L. Tan, J. Cao, and H. T. Shen. Distributed shortest path query processing on dynamic road networks. *The VLDB Journal*, 26(3):399–419, 2017.

[74] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu. Probabilistic skyline operator over sliding windows. In *Proceedings of ICDE*, pages 1060–1071, 2009.

[75] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: Towards bridging theory and practice. In *Proc. of SIGMOD'13*, pages 857–868, 2013.