Efficient query processing on large graphs

Zhu, Diwen

2014

Zhu, D. (2014). Efficient query processing on large graphs. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/61838>

<https://doi.org/10.32657/10356/61838>

NANYANG
TECHNOLOGICAL
UNIVERSITY

EFFICIENT QUERY PROCESSING ON LARGE GRAPHS

ZHU DIWEN

SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

2014

EFFICIENT QUERY PROCESSING ON LARGE GRAPHS

ZHU DIWEN

School of Computer Engineering

A thesis submitted to the Nanyang Technological University

in partial fulfilment of the requirement for the degree of

Doctor of Philosophy

2014

Abstract

Graphs provide a natural way to represent real-world objects and their relationships, and hence they are utilized to model concepts and data in various domains, such as social networks, the Web, biological informatics and spatial databases. In the big data era, most graphs are extremely large and growing rapidly in size, which brings challenges to processing graph queries efficiently. In this thesis, we investigate three important problems in graph query processing, namely, the point-to-point shortest path and distance queries on road networks, the single source shortest path and distance queries on disk-resident graphs, and the point-to-point reachability queries on dynamic graphs. These problems are not only interesting in themselves when the graph size becomes large, but also fundamental building blocks of numerous real-world applications.

In particular, given two locations s and t in a road network, a point-to-point distance query returns the minimum network distance from s to t , while a point-to-point shortest path query computes the actual route that achieves the minimum distance. To solve this problem, we propose *Arterial Hierarchy (AH)*, an index structure that narrows the gap between theory and practice. On the theoretical side, we show that, based on a realistic characteristic of real-world road networks, AH answers any distance query in constant time. In addition, any shortest path query can be answered with k more steps, where k is the number of vertices on the shortest path. On the practical side, AH outperforms the state-of-the-art methods in terms of query time and its space and pre-computation overheads are moderate.

Then, given a vertex s in a graph G , a single source shortest distance query from s asks for the distance from s to every other vertex in G , while a single source shortest path query retrieves the shortest path from s to any other vertex. To handle these two types of queries on disk-resident graphs, we propose *Highways-on-disk (HoD)*, a disk-based index that supports both shortest path and distance queries on directed and weighted graphs. Different from existing methods, HoD does not assume the graph fits into memory, does not limit the graph type as undirected graph, and does not restrict that the edges have unit weights or even integer weights. Although being more general and flexible, HoD is still able to outperform the state-of-the-art methods in terms of query processing time by up to two orders of magnitude. In addition, HoD has a smaller space consumption and pre-computation time in most cases.

Finally, given two vertices s and t in G , a reachability query from s to t asks whether there exists a path from s to t in G . To solve the problem on dynamic graphs, we first

propose the *Total Order Labeling (TOL)* framework, which summarizes three most advanced methods [22, 51, 94] for reachability queries on static graphs. Then we investigate novel algorithms to handle updates of TOL, regarding both insertions and deletions on dynamic graphs. To our best knowledge, this is the first practically efficient and scalable approach that handles the point-to-point reachability queries on dynamic graphs. Finally we propose a new reachability index *Butterfly*, which offers reduced preprocessing, space, and query costs than any existing indices under TOL [22, 51, 94]. We experimentally evaluate TOL using a large variety of benchmark datasets with up to twenty million vertices, and we demonstrate the superiority of TOL against alternative solutions for static and dynamic graphs.

Acknowledgments

I would like to express my sincere gratitude to my advisor Dr. Xiaokui Xiao for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Contents

Abstract	i
Acknowledgments	iii
List of figures	viii
List of Tables	x
1 Introduction	1
1.1 Problems and Motivations	1
1.2 Approaches and Contributions	5
1.3 Thesis Organization	7
2 Literature Survey	9
2.1 Dijkstra’s Algorithm	9
2.2 Single Source Shortest Path and Distance Queries	10
2.2.1 External Memory Dijkstra’s Algorithm	11
2.2.2 VC-Index	12
2.3 Point-to-Point Shortest Path and Distance Queries	13
2.3.1 Contraction Hierarchies	14
2.3.2 Transit Node Routing	16
2.3.3 Spatially Induced Linkage Cognizance	17
2.3.4 Path-Coherent Pairs Decomposition	19
2.4 Point-to-Point Reachability Queries	20
2.4.1 Pruned Depth-first Search	20
2.4.2 Transitive Closure Retrieval	21
2.4.3 2-Hop Label Matching	22
2.5 Summary	24

3	Point-to-Point Shortest Path and Distance Queries	26
3.1	Related Work	28
3.2	Problem and Assumptions	29
3.3	A First-Cut Solution	32
3.3.1	Index Construction	32
3.3.2	Query Processing	34
3.3.3	Complexity Analysis	36
3.3.4	Correctness Proof	38
3.4	Arterial Hierarchy	39
3.4.1	Overview	40
3.4.2	Index Construction	41
3.4.3	Query Processing	46
3.4.4	Node Ranking and Selection	47
3.4.5	Space and Time Complexities	48
3.5	Experiments	50
3.5.1	Datasets and Queries	50
3.5.2	Efficiency for Distance Queries	51
3.5.3	Efficiency for Shortest Path Queries	52
3.5.4	Space and Preprocessing Costs	53
3.6	Summary	54
4	Single Source Shortest Path and Distance Queries	57
4.1	Related Work	59
4.2	Problem Definition	60
4.3	Solution Overview	61
4.3.1	Shortcut Construction	61
4.3.2	Query Processing	63
4.4	Index Construction	65
4.4.1	Node Removal and Shortcut Generation	66
4.4.2	Selecting Nodes for Removal	68
4.4.3	Generation of Baseline Edges	69

4.4.4	Termination Condition	70
4.4.5	Index File Organization	70
4.4.6	Cost Analysis	71
4.5	Algorithm for SSD Queries	72
4.5.1	Forward Search	72
4.5.2	Core Search	73
4.5.3	Backward Search	73
4.5.4	Correctness and Complexities	73
4.6	Extension for SSSP Queries	74
4.7	Experiments	75
4.7.1	Datasets	75
4.7.2	Results on Undirected Graphs	76
4.7.3	Results on Directed Graphs	78
4.8	Summary	78
5	Point-to-Point Reachability Query	79
5.1	Preliminaries	80
5.2	Related Work	82
5.3	Total Order Labeling	82
5.4	Incremental Updates	85
5.4.1	Insertion Algorithm	86
5.4.2	Deletion Algorithm	93
5.5	Iterative Label Reduction	95
5.6	Construction of Initial $\mathcal{L}$	96
5.6.1	Deciding Vertex Level	96
5.6.2	Labeling Algorithm	98
5.7	Experiments	100
5.8	Summary	104

6	Conclusions and Future Work	106
6.1	Conclusion	106
6.2	Future Directions	107
6.2.1	Efficient Updates on Dynamic Graphs	108
6.2.2	Parallel Processing on Distributed Graphs	109
6.3	Summary	111
A	Proofs	112
A.1	Unique Shortest Paths via Weight Perturbation	112
A.2	Proofs of Lemmas 3.1 and 3.2	115
A.3	Proofs of Lemmas 3.3 and 3.4	117
A.4	Proof of Theorem 3.1	127
A.5	Analysis of HoD on Deterministic Scale-free Networks	131
A.6	Proof of Theorem 4.1	134
	Appendix - List of Publications	139
	References	140

List of Figures

1.1	Road network G .	5
1.2	Node hierarchy H .	5
1.3	Graph G .	6
1.4	Augmented Graph G^* .	6
2.1	SSSP query from vertex a by EM-Dij.	11
2.2	Road Network.	15
2.3	Contraction Hierarchies.	15
2.4	Transit Node Routing.	16
2.5	SILC.	16
2.6	PCPD.	18
2.7	G with weights.	18
2.8	Interval labeling of GRAIL.	20
2.9	Chain-form of G .	21
2.10	Transitive Closure Compression.	21
2.11	2-Hop Labeling.	23
3.1	Road network G .	30
3.2	Node hierarchy H .	30
3.3	Arterial dimensions of real road networks.	32
3.4	Strips and bisectors of a region with 4×4 cells.	33
3.5	Illustration of the proximity constraint.	36
3.6	Reconstruction of shortest paths.	38
3.7	Graph G_1 .	43
3.8	Reduce graph G'_1 .	43

3.9	Efficiency of distance queries vs. n .	52
3.10	Efficiency of shortest path queries vs. n .	53
3.11	Space Overhead and Preprocessing Time vs. n .	54
3.12	Efficiency of distance queries vs. query set.	55
3.13	Efficiency of shortest path queries vs. query set.	56
4.1	Graph reduction and shortcut construction.	61
4.2	Node removal and shortcut generation.	69
4.3	Forward graph G_f and backward graph G_b .	72
5.1	DAG G .	82
5.2	Average insertion time on dynamic graphs.	102
5.3	Total query time on dynamic graphs.	103
5.4	Average deletion time on dynamic graphs.	103
5.5	Index sizes on static graphs.	104
5.6	Preprocessing time on static graphs.	104
5.7	Total query time on static graphs.	104
A.1	B_1 is completely in the $(\alpha + 6) \times (\alpha + 6)$ area.	116
A.2	B_2 is not completely in the $(\alpha + 6) \times (\alpha + 6)$ area.	116
A.3	The <i>SlidingWindow</i> Algorithm.	116
A.4	$P = \langle v_1, v_2, \dots, v_6 \rangle$ in R_i .	117
A.5	Construction of the deterministic scale-free network.	132
A.6	HoD on the deterministic scale-free network.	138

List of Tables

3.1	Asymptotic performance of the state of the art.	48
3.2	Dataset Characteristics	51
4.1	Datasets.	75
4.2	Preprocessing time (in minutes).	76
4.3	Space Consumption (in GB).	76
4.4	Average running time for SSD queries (in seconds).	77
4.5	Estimated time for <i>closeness</i> computation (in hours).	77
4.6	Performance of HoD on directed graphs.	78
5.1	Table of notations.	81
5.2	Two TOL indices for the DAG G in Figure 5.1.	84
5.3	Datasets ($M = 10^6$).	102
5.4	Performance of label reduction ($K = 10^3$).	105

Chapter 1

Introduction

Graphs provide a natural way to represent real-world objects and their relationships, and hence they are utilized to model concepts and data in various domains, such as social networks, the Web, biological informatics and spatial databases. In the big data era, most graphs are extremely large and growing rapidly in size. For instance, the social network of Facebook is reported to have 1.19 billion active users (vertices) in 2013, and there is an 18 percent increase year over year. In addition, there are 4.5 billion “likes” (edges) generated daily as of May 2013, which is a 67 percent increase from August 2012. The rapid increase in the size of such graphs brings challenges to processing graph queries efficiently. In this thesis, we investigate three important problems in query processing, which are not only interesting in themselves when the graph size becomes large, but also fundamental building blocks of numerous real-world applications, and we propose efficient methods to solve these problems. In the remainder of this chapter, we present the problems and solutions, and the introduced concepts will be explained in later chapters with more details.

1.1 Problems and Motivations

The first problem we studied is the point-to-point shortest path (*PPSP*) and distance (*PPD*) queries. Let G be a graph, we assign each edge e in G a weight $w(e)$. Then given two vertices (nodes) s and t , a PPSP query from s to t asks for a sequence of edges $e_1, e_2, \dots, e_k$ that form a path from s to t , such that $\sum_{i=1}^k w(e_i)$ is minimized. On the other hand, a distance query from s to t asks only for the value of $\sum_{i=1}^k w(e_i)$ instead of the actual shortest path. These two types of queries find important applications in map, navigation, and location-based services. To illustrate, consider that a user of a map service is looking for a nearby Italian restaurant for dinner. In response to the user’s query, the service provider can first retrieve the list of Italian restaurants in the region close to the user’s current location u . After that, the network distance from u to each

restaurant is computed (using a distance query), and those distances are returned to the user along with the list of restaurants. Then, if the user chooses a preferred restaurant r from the list, the service provider can employ a shortest path query to provide the user with driving directions from u to r . Thus, we focus on processing PPSP and PPD queries on road networks.

The classic solution for the shortest path and distance queries is Dijkstra’s algorithm [32]. It traverses the road network nodes in ascending order of their distances from s ; once it reaches t during the traversal, it can compute the distance from s to t and can retrieve the shortest path based on the information recorded before t is visited. With proper data structures, Dijkstra’s algorithm runs in $O(n \log n + m)$ time for any shortest path or distance query, where n (resp. m) is the number of nodes (resp. edges) in the road network. Albeit simple and elegant, Dijkstra’s algorithm is inefficient for sizable road networks, as it requires traversing all network nodes that are closer to s than t , which incurs a significant overhead when s and t are far apart. A plethora of techniques [9, 15, 17, 28, 34, 38, 42–44, 47, 54–56, 67, 71, 74, 77, 78, 86] have been proposed to improve over Dijkstra’s algorithm in terms of either practical efficiency or asymptotic bounds. Existing methods that focus on practical performance mostly rely on heuristics, and hence, their asymptotic bounds are unattractive in general. For instance, the best heuristic approach by Geisberger et al. [38] answers shortest path or distance queries in at most a few milliseconds even on road networks with millions of nodes, but its space and time complexities are both $O(n^2)$, *i.e.*, its asymptotic performance is even worse than that of Dijkstra’s algorithm. On the other hand, the solutions that offer favorable query time complexities often entail prohibitive preprocessing cost or space overhead, rendering them only applicable for small datasets. For example, the state-of-the-art approaches by Samet et al. [74] and Abraham et al. [9] provide superior bounds on query time, but they require pre-computing the shortest path between *any* pair of nodes, which is impractical for the large road networks commonly used in modern map applications. As such, we are motivated to propose an approach that can narrow the gap between theory and practice for the problem of processing PPSP and PPD queries on road networks.

The second problem we studied is the single source shortest path (*SSSP*) and distance (*SSD*) queries. Given a graph G and a source vertex s , the SSSP query from s asks the shortest paths from s to any other vertex in G , while the SSD query requests the distances only. These two types of queries are fundamental building blocks for numerous graph algorithms [64], and they find important applications in graph analysis [23], especially in the computation of graph measures [11, 13, 33, 85]. For example, the estimation of *closeness* measures [33] on a graph G requires performing SSD queries from a large number of vertices in G , while the approximation of *betweenness* measures [13] requires executing numerous SSSP queries.

The single source shortest path and distance queries can also be answered by Dijkstra’s algorithm, and a plethora of techniques have been proposed to improve over Dijkstra’s algorithm for higher query efficiency [28, 30, 31, 35, 36, 40, 41, 45, 57, 61, 70, 70, 83, 84, 88, 89], however, most of the existing techniques assume that the given graph fits in the main memory (for pre-computation and/or query processing), which renders them inapplicable for the massive disk-resident large graphs commonly used in web and social applications. There are a few methods [59, 62, 63, 65, 66] that address this issue by incorporating Dijkstra’s algorithm with I/O-efficient data structures, but the performance of those methods are shown to be insufficient for practical applications [23]. The main reason is that, when Dijkstra’s algorithm traverses the graph, the order in which it visits nodes can be drastically different from the order in which the nodes are arranged on the disk. This leads to a significant number of random disk accesses, which results in poor query performance. In contrast to the aforementioned techniques, Cheng et al. [23] propose the first practically efficient index (named *VC-Index*) for SSSP and SSD queries on disk-resident graphs. The basic idea of VC-Index is to pre-compute a number of *reduced* versions of the input graph G . Each reduced graph contains some relatively important nodes in G , as well as the distances between some pairs of those nodes. During query processing, VC-Index scans a selected subset of reduced graphs, and then derives query results based on the pre-computed distances. Compared with those methods based on Dijkstra’s algorithm [59, 62, 63, 65, 66], VC-Index is more efficient as it only performs sequential reads on disk-resident data. However, all these disk-based solutions for SSD and SSSP queries require that the input graph is undirected, which renders them inapplicable for any application built upon directed graphs. This is rather restrictive as numerous important types of graphs (*e.g.*, web graphs, social graphs) are directed in nature. Furthermore, even when the input graph is undirected, the query efficiency of the existing solutions is less than satisfactory. In particular, our experiments (in Section 4.7) show that VC-Index, albeit being the state of the art, requires tens of seconds to answer a single SSD query on a graph with less than 100 million edges, and needs more than two days to estimate the closeness measures on the same graph. To address the deficiency of existing work, we propose to support both SSSP and SSD queries on directed and weighted disk-resident graphs.

The last problem we studied is the point-to-point reachability (*PPR*) query. Given a directed graph G and two vertices s and t , a PPR query from s to t asks whether there exists a path from s to t in G . PPR queries are a fundamental operation on graphs and have numerous important applications, such as query processing in XML document, association detecting in Semantic Web graphs, graph pattern analysis and friend suggestion in social networks, connection identifying in road networks, recursive query answering in knowledge base, and program workflow analysis. Devising index

structures for PPR queries is non-trivial, as it requires a careful balancing act between pre-computation cost, index size, and query processing overhead. In particular, if we pre-compute and store the reachability results for all pairs of vertices, then we can process any PPR query in $O(1)$ time but suffer prohibitive costs of preprocessing and space. On the other hand, if we omit indexing and process PPR queries directly on G using depth-first search (DFS) or breadth-first search (BFS), then we minimize space and pre-computation overhead, but fail to ensure query efficiency on large graphs.

Previous work [10, 18–22, 24–26, 29, 46, 48, 50, 53, 73, 79, 80, 82, 90–92, 94–96] has proposed numerous indexing techniques to efficiently support PPR queries without significant space and pre-computation overheads. Most techniques, however, assume that the input graph G is *static*, which makes them inapplicable for the *dynamic* graphs commonly encountered in practice. For example, the social graph of Twitter is constantly changing, with thousands of new users added per day; the Semantic Web is frequently updated with new concepts and relations; even road networks are subject to changes due to road closures and constructions. There exist a few techniques [18, 29, 46, 50, 73, 80, 96] that are designed for dynamic graphs, but as we discuss in Sections 5.2 and 5.7, none of those techniques can scale to sizable graphs without significant loss of efficiency. Specifically, the methods in [18, 29, 46, 50, 73, 80] incur prohibitive preprocessing costs on graphs with more than one million vertices. Meanwhile, the approach in [96] can handle million-vertex graphs, but it offers a query performance that is *generally no better than a simple BFS approach*, as shown in our experiments. As a result, we are motivated to develop an efficient method for answering PPR queries on large dynamic graphs.

The aforementioned problems are closely related. Firstly, a solution for the single source shortest path and distance queries is also a solution for the point-to-point shortest path and distance queries, since the single source queries are the super-queries of point-to-point queries. Secondly, a point-to-point shortest path (resp. distance) querying approach can be executed $n - 1$ times (n is the number of vertices in the graph) to answer a single source shortest path (resp. distance) query from s , by calling the approach from s to any other vertex in the graph repeatedly. Finally, any technique for the point-to-point shortest path or distance query can also answer the point-to-point reachability query, because the existence of the shortest path is equivalent to the existence of a path from the source vertex to the target vertex. However, when considering efficiency and scalability, approaches for different problems have huge differences in methodology, thus making each of the problems worth studying. Details of the state-of-the-art techniques for each problem are presented in Chapter 2.

 Figure 1.1: Road network G .

 Figure 1.2: Node hierarchy H .

1.2 Approaches and Contributions

For the point-to-point shortest path and distance queries on road networks, we propose *Arterial Hierarchy (AH)*, an index structure that narrows the gap between theory and practice in answering shortest path and distance queries on road networks. On the theoretical side, we show that, under a realistic assumption, AH answers any distance query in $O(\log \alpha)$ time, where $\alpha = d_{\max}/d_{\min}$, and $d_{\max}$ (resp. $d_{\min}$) is the largest (resp. smallest) L_∞ distance between any two nodes in the road network. In addition, any shortest path query can be answered in $O(k + \log \alpha)$ time, where k is the number of nodes on the shortest path. On the practical side, we experimentally evaluate AH on a large set of real road networks with up to twenty million nodes, and we demonstrate that (i) AH outperforms the state of the art in terms of query time, and (ii) its space and pre-computation overheads are moderate.

In a nutshell, AH organizes the nodes in the road network into a hierarchy, based on which it pre-computes auxiliary information to facilitate query processing. For instance, given the road network G in Figure 1.1, AH constructs a three-level hierarchy H (illustrated in Figure 1.2), where each level consists of a disjoint subset of the nodes in G . Note that H contains all edges in G , as well as two auxiliary edges, $\langle v_9, v_{10} \rangle$ and $\langle v_{10}, v_{11} \rangle$, each of which has a length that equals the distance between the two nodes that it connects. These auxiliary edges are referred as *shortcuts*, and they can be exploited to considerably reduce the numbers of nodes and edges that need to be traversed during a shortest path or distance query.

For example, given a distance query between v_1 and v_{10} (in G), AH would perform two alternating traversals (in H) starting from v_1 and v_{10} , respectively, and *it would always avoid traveling from a higher-level node to a lower-level node*. In particular, the traversal starting from v_1 can only reach two nodes, v_{10} and v_{11} , since (i) v_{11} is the only node adjacent to v_1 , and (ii) from v_{11} , AH would only traverse to v_{10} (since v_{10} is the only neighbor of v_{11} that is not at a lower level than v_{11}). Similarly, the traversal starting from v_{10} would only reach v_{11} . Once the two traversals terminate, the distance between v_1 and v_{10} is calculated by summing up the weights of $\langle v_1, v_{11} \rangle$ and $\langle v_{11}, v_{10} \rangle$.

 Figure 1.3: Graph G .

 Figure 1.4: Augmented Graph G^* .

In general, AH answers any distance query with two traversals of the node hierarchy, such that each traversal only moves up from low-level nodes to high-level nodes, but not vice versa. We show that, for real road networks, the node hierarchy contains $O(\log \alpha)$ levels, where $\alpha = d_{max}/d_{min}$. Furthermore, each traversal performed by AH visits only a constant number of nodes and edges in any level of the hierarchy. As a consequence, the total number of nodes and edges visited by AH is $O(\log \alpha)$, which results in an $O(\log \alpha)$ time complexity for any distance query. In addition, once the distance between two nodes s and t is computed, AH can derive the actual shortest path from s to t in $O(k)$ time, where k is the number of nodes on the shortest path. These time complexities of AH rely on an assumption on road networks (to be clarified in Section 3.2). We provide detailed discussion on the assumption, and we demonstrate its applicability on practical road networks with extensive experiments on a large collection of real datasets. These experimental findings not only form a basis for our theoretical claims but also shed light on the characteristics of real road networks, which paves the path for future research on shortest path and distance queries on road networks.

For the single source shortest path and distance queries on disk-resident graphs, we propose *Highways-on-Disk (HoD)*, a disk-based index that supports both SSSP and SSD queries on directed and weighted graphs. The key idea of HoD is to augment the input graph with a set of auxiliary edges (referred to as *shortcuts* [75]), and exploit them during query processing to reduce I/O and computation costs. For example, Figure 1.3a illustrates a graph G , and Figure 1.4 shows an augmented graph G^* constructed from G . G^* contains three shortcuts: $\langle v_8, v_9 \rangle$, $\langle v_9, v_7 \rangle$, and $\langle v_9, v_{10} \rangle$. Each shortcut has the same length with the shortest path connecting the endpoints of the shortcut. For example, the length of $\langle v_8, v_9 \rangle$ equals 2, which is identical to the length of the shortest path from v_8 to v_9 . Intuitively, the shortcuts in G^* enable HoD to efficiently traverse from one node to another (in a manner similar to how highways facilitate traversal between distant locations). For instance, if we are to traverse from v_1 to v_{10} in G^* , we may follow the path $\langle v_1, v_9, v_{10} \rangle$, which consists of only three nodes; in contrast, a traversal from v_1 to v_{10} in G would require visiting five nodes: v_1 , v_9 , v_6 , v_7 , and v_{10} .

In general, when HoD answers an SSD or SSSP query, it often traverses the augmented graph via shortcuts (instead of the original edges in G). We show that, with proper shortcut construction and index organization, the query algorithm of HoD always traverses nodes in the same order as they are arranged in the index file. Consequently, HoD can answer any SSD or SSSP query with a linear scan of the index file, and its CPU cost is linear to the number of edges in the augmented graph. We experimentally evaluate HoD on a variety of real-world graphs with up to 100 million nodes and 3 billion edges, and we demonstrate that HoD significantly outperforms VC-Index in terms of query efficiency. In particular, the query time of HoD is smaller than that of VC-Index by up to two orders of magnitude. Furthermore, HoD requires a smaller space and pre-computation time than VC-Index in most cases.

For the point-to-point reachability queries on dynamic graphs, we first introduce the *Total Order Labeling (TOL)* framework, which summarizes three most advanced methods [22, 51, 94] for reachability queries on static graphs. TOL has two important properties: (i) every reachability index under TOL uniquely corresponds to a total order of vertices in the input graph, and (ii) the total order solely decides the index's performances in terms of preprocessing, space, and queries. Given these properties, we investigate algorithms that enable us to insert or delete a vertex in a TOL index without changing the order of the other vertices, i.e., without significantly degrading the performance of the index. This results in general algorithms for handling insertions and deletions on indices under TOL. In particular, our insertion algorithm is *optimal* in that it leads to the minimum index size after insertion.

Interestingly, we observe that our update algorithms can be utilized to reduce the space consumptions and query costs of a TOL index, by adjusting the total order pertinent to the index. This leads to a general approach for improving any index under TOL, including the state-of-the-art techniques [22, 51, 94]. The effectiveness of our adjusting approach shows that the total orders of the techniques in [22, 51, 94] leave much room for enhancement, which motivates us to devise new methods for deriving improved total orders for TOL indices. As a result, we present a new reachability index, *Butterfly*, which offers reduced preprocessing, space, and query costs than any existing indices under TOL [22, 51, 94]. We experimentally evaluate TOL using a large variety of benchmark datasets with up to twenty million vertices, and we demonstrate the superiority of TOL against alternative solutions for static and dynamic graphs.

1.3 Thesis Organization

In chapter 2, we will conduct a survey to efficient query processing on large graphs. Firstly, we will introduce Dijkstra's algorithm and its bidirectional variant, which are the

classic solutions to shortest path and distance queries. After that, we survey the existing methods for PPSP and PPD queries on road networks, and reveal how they employ the characteristics of road networks to achieve better performance in query processing. Then, we present two methods that can handle SSSP and SSD queries on disk-resident graphs, and point out their deficiencies and limitations. Finally, we classify existing techniques for the PPR query into three categories, by comparing the pros and cons, we demonstrate which category possesses the potential to handle large graphs.

In chapter 3, we will present the Arterial Hierarchy (AH). The aforementioned time complexities of AH rely on an assumption on road networks, so we will first provide detailed discussion on the assumption, and demonstrate its applicability on practical road networks with extensive experiments on a large collection of real world road networks. Then, we describe the construction algorithm and the query processing algorithm, and analyse their time and space complexities. Finally, we compare AH with the state-of-the-art methods in theoretical bounds, and also show the practical efficiency of AH by extensive empirical results.

In chapter 4, we will introduce Highways-on-disk (HoD). We first describe the preprocessing of HoD that iteratively reduces the residual graph till it fits into memory while generates shortcuts for query processing. Then we present the three steps in our query processing algorithm, and we show that the entire process requests only one linear scan of our pre-computed index. Finally, we experimentally evaluate HoD on a variety of real-world large graphs with up to 100 million nodes and 3 billion edges, and we demonstrate that HoD significantly outperforms the state-of-the-art in terms of query efficiency by up to two orders of magnitude.

In chapter 5, we will demonstrate the Total Order Labeling (TOL) framework, and present several important properties of TOL. Then based on the properties, we investigate general algorithms for handling insertions and deletions on indices under TOL. After that, we present the general approach for improving an index under TOL, and the new reachability index, Butterfly. Finally, we experimentally evaluate TOL using a large variety of benchmark datasets with up to twenty million vertices, and we demonstrate the superiority of TOL against alternative solutions for static and dynamic graphs.

In chapter 6, we will conclude the thesis and propose some promising future directions on the topic of efficient query processing on large graphs.

Chapter 2

Literature Survey

In this chapter, we first introduce the *Dijkstra's algorithm* and its bidirectional variant. In particular, Dijkstra's algorithm is the classic solution for single source shortest path and distance queries, while the bidirectional variant is for point-to-point shortest path and distance queries. To address the deficiency of Dijkstra's algorithm (resp. the bidirectional variant) on large graphs, we then survey the advanced methods for single source (resp. point-to-point) shortest path and distance queries. In particular, for single source shortest path and distance queries, we introduce two external memory approaches that can handle disk-resident graphs, while for point-to-point shortest path and distance queries, we demonstrate four state-of-the-art techniques and reveal how they exploit the characteristics of road networks to improve the efficiency. Finally, we present three categories of techniques for point-to-point reachability queries on static graphs, which are the preliminary for dynamic approaches, and summarize the approaches surveyed in this chapter.

2.1 Dijkstra's Algorithm

Given a vertex s , Dijkstra's algorithm computes the shortest paths from s to other vertices in ascending order of their distances from s . In particular, at each iteration, Dijkstra's algorithm first selects the closest unvisited vertex v (*i.e.*, the distance from s to v , denoted as $dist(s, v)$ is the smallest among all the unvisited vertices), and breaks ties arbitrarily. Then for each out-going edge from v , say (v, u) , the distance from s to u is updated by $\min(dist(s, u), dist(s, v) + w(v, u))$, where $w(v, u)$ is the weight of edge (v, u) . This operation is called a relaxation from v to u , and we say the relaxation is successful if $dist(s, u)$ is reduced after the relaxation. After relaxations from v are accomplished, v is marked as visited, and the distance from s to v is computed. Meanwhile, the latest successful relaxations recorded at each vertex form a shortest path tree, which can be utilized to construct the shortest paths from s to other vertices. After all the vertices

are visited, the algorithm returns answers to SSSP and SSD queries starting from s , and it also can be used to answer PPSP and PPD queries from s to t by terminating the algorithm when t is visited. But the deficiency of utilizing Dijkstra's algorithm in answer PPSP and PPD queries is obvious, since all the vertices within $dist(s, t)$ distance to s must be visited.

To address this issue, a variant of the Dijkstra's algorithm, called the bidirectional Dijkstra's algorithm [69] is widely used to answer PPSP and PPD queries. It invokes two instances of Dijkstra's algorithm from s and t simultaneously, and the two traversals terminate when they meet at a vertex $u \in V$. Let V_1 (resp. V_2) be the set of vertices visited by the traversal that starts from s (resp. t). It can be verified that the shortest path between s and t must either pass through u , or go across two adjacent vertices $v_1 \in V_1$ and $v_2 \in V_2$. Therefore, $dist(s, t)$ should equal the smallest value among $dist(s, u) + dist(u, t)$ and $dist(s, v_1) + dist(v_1, v_2) + dist(v_2, t)$, for any two adjacent vertices $v_1 \in V_1$ and $v_2 \in V_2$. Once $dist(s, t)$ is decided, the shortest path between s and t can be retrieved from the shortest path trees constructed during the traversals from s and t .

Although the bidirectional variant runs in the same time complexity of $O(n \log n + m)$ as Dijkstra's algorithm under a priority queue implementation, it is usually more efficient than Dijkstra's algorithm in practice for PPSP and PPD queries. This is because, intuitively, each of the two graph traversals invoked by the bidirectional algorithm visit the vertices that are within roughly $dist(s, t)/2$ distance to s or t . The number of such vertices is often smaller than the number of vertices that are within $dist(s, t)$ distance to s , *i.e.*, the vertices that need to be traversed by Dijkstra's algorithm. However, the number is still enormous when s and t are far apart in a large road network.

2.2 Single Source Shortest Path and Distance Queries

Previous research for SSSP and SSD queries focus on improving Dijkstra's algorithm for higher query efficiency, but most of the existing techniques assume that the given graph fits in the main memory (for pre-computation and/or query processing), which renders them inapplicable for the massive disk-resident graphs commonly used in web and social applications. There are a few methods [59, 62, 63, 65, 66] that address this issue by incorporating Dijkstra's algorithm with I/O-efficient data structures, but the performance of those methods are shown to be insufficient for practical applications [23]. The main reason is that, when Dijkstra's algorithm traverses the graph, the order in which it visits nodes can be drastically different from the order in which the nodes are arranged on the disk. This leads to a significant number of random disk accesses, which results in poor query performance.

In contrast to the aforementioned techniques, Cheng et al. [23] propose the first practically efficient index (named *VC-Index*) for SSD and SSSP queries on disk-resident

Iteration	2		3		4		5	
Period	begin	end	begin	end	begin	end	begin	end
Vertex v	—	b	c	c	d	d	e	e
Delayed Relaxations	$a \rightarrow b$	$a \rightarrow b$	$b \rightarrow d$	$b \rightarrow d$	$b \rightarrow d$	$b \rightarrow d$	$b \rightarrow d$	$b \rightarrow d$
	$a \rightarrow c$	$a \rightarrow e$		$c \rightarrow e$	$c \rightarrow e$	$c \rightarrow e$	$c \rightarrow e$	$e \rightarrow e$
	$a \rightarrow d$	$a \rightarrow d$				$d \rightarrow e$	$d \rightarrow e$	$d \rightarrow e$
		$b \rightarrow d$						

 Figure 2.1: SSSP query from vertex a by EM-Dij.

graphs. The basic idea of VC-Index is to pre-compute a number of *reduced* versions of the input graph G . Each reduced graph contains some relatively important nodes in G , as well as the distances between some pairs of those nodes. During query processing, VC-Index scans a selected subset of reduced graphs, and then derives query results based on the pre-computed distances. Compared with those methods based on Dijkstra's algorithm [59, 62, 63, 65, 66], VC-Index is more efficient as it only performs sequential reads on disk-resident data. We hence briefly introduce the external memory Dijkstra's algorithm and VC-Index in the following.

2.2.1 External Memory Dijkstra's Algorithm

In the worst case, Dijkstra's algorithm has to conduct $|V| - 1$ accesses to disk, given that in each iteration, the adjacent list of v must be fetched to relax from v . To address this issue, *External Memory Dijkstra's Algorithm (EM-Dij)* aims to relax a group of vertices together in each iteration, so that the corresponding adjacent list retrievals can be done in one disk access, and this is done by delaying the relaxations from the current iteration to upcoming iterations if the delaying condition is satisfied. To illustrate, suppose some relaxations have been delayed from previous iterations, and now we are at the beginning of the current iteration. The question is, what are the relaxations that must be done before we select the closest unvisited vertex v in this iteration? The answer is that, a relaxation from u to w must be accomplished if $\text{dist}(s, u) + w(u, w) < \text{dist}(s, v)$, this is because, such relaxation may cause w becomes the closest unvisited vertex instead of v if $w \neq v$, or otherwise (*i.e.*, $w = v$), reduces the distance from s to v to $\text{dist}(s, u) + w(u, v)$. On the opposite, a relaxation from u to w with $\text{dist}(s, u) + w(u, w) \geq \text{dist}(s, v)$, denoted as *delay condition*, can be delayed to the current iteration or even further.

In a nutshell, at each iteration of EM-Dij, it first executes all the previously delayed relaxations that cannot be delayed any more, *i.e.*, violate the delaying condition. Then, selects the closest unvisited vertex v and mark v as visited. After that, instead of relaxing v , EM-Dij delays all the relaxations from v to the next iteration, and it terminates when all the vertices are visited. To illustrate, for the graph in Figure 2.7, we show the

processing of SSSP query from a by EM-Dij in Figure 2.1. In the first iteration, relaxing a is delayed directed to the next iteration. Then in the second iteration, in the beginning, all the distances to unvisited vertices from a are infinite, so there is no closest vertex to select, thus all the previously delayed relaxations are executed, and then, b is selected given the distance from a to b is 1. In the end relaxation from b is delayed to next iteration. Continue to the third iteration, in the beginning, the closest vertex is c , and $\text{dist}(s, c) = 1$, so the delay condition for the relaxation from b to d still holds, which delays the relaxation further to the fourth iteration together with the relaxation from c . Similarly, relaxation from d also delays in the fourth iteration. And in the last iteration, all the delayed relaxations are executed. As such, EM-Dij executes all the relaxations in two batches, hence only scans the graph twice from the disk.

Noticed that in the end of each iteration, EM-Dij has to load the adjacent list of v and delayed them to the next iteration. If such action is materialized, it will wipe off the effect from the combination of relaxations. Instead, EM-Dij partition the out-going edges from each vertex v into k categories based on the weight. Specifically, the i -th category consists of edges with weight from 2^{i-1} to $2^i - 1$. Then assign each vertex v a bit-vector which indicates the non-empty categories of v , which is utilized to approximate the relaxations from v in the end of each iteration. For instance, still regarding the above example, in the end of the first iteration, instead of delaying the relaxations from a , EM-Dij delays 2 categories of relaxations from a to the next iteration, namely $\langle a, 1 \rangle$ and $\langle a, 2 \rangle$. Then for the i -th category of relaxations, the delay condition is modified as $\text{dist}(s, u) + 2^{i-1} < \text{dist}(s, v)$, since 2^{i-1} is the lower bound of the weights in this category. Given a graph with integer weights from 1 to K , EM-Dij requires $O(|V| \cdot \log K)$ bits to record the bit-vectors for edge categories, and they assume the memory is large enough to store the bit-vectors. Thus, the delaying operation at the end of each iteration does not incur any access to the disk.

2.2.2 VC-Index

VC-index applies the concept of *vertex cover* (VC) to construct a tree-structured index. The construction of the index is based on the observation that all vertices in a given graph are within 1 hop of the VC. By employing the property of VC, VC-index shows that, once the distance from the source vertex s to the vertices in a VC C of G is known, the distance from s to each remaining vertex in G can be obtained at a small cost, *i.e.*, 1 hop extension. Still using the graph in Figure 2.7 as example, without considering the directions of edges in G (VC-Index assumes the input graph is undirected), the set of vertices $C = \{a, c, d\}$ is a vertex cover of G . To answer the SSD query from e , VC-Index first obtain the distances from e to vertices in C , *i.e.*, $\text{dist}(e, a) = 3$, $\text{dist}(e, c) = 3$, $\text{dist}(e, d) = 1$. Then it further extend the distances to the vertices that are not in the

C , i.e., $\text{dist}(e, b) = \min(\text{dist}(e, d) + w(d, b), \text{dist}(e, a) + w(a, b)) = 3$. As such, with two steps of computation, VC-Index obtains the distances from the source vertex to all the other vertices to answer the SSD query.

However, it is inefficient to compute the distance from s to the vertices in C during the query processing. Moreover, it is also impractical to pre-compute all the distances in the preprocessing and retrieve them in the query processing, because the storage size is prohibitively large. Hence, VC-index compresses the distance information in G step by step until it becomes affordable to do direct distance computation, e.g., bidirectional Dijkstra’s algorithm, with limited memory and short response time. To this end, VC-index formulates the concept of distance graph, which is smaller than G but fully retains the distance information between the vertices in C . After obtaining the distance graph D_0 of G , VC-index recursively constructs a distance graph D_1 from D_0 , and so on until a distance graph D_i that is small enough for in-memory distance computation is obtained. In this way, as long as s is in D_i , the distance from s to all the vertices in G can be obtained by unfolding the distance information in these distance graphs, which form paths in VC-index. For the SSSP query from s , these paths in VC-index can be further reconstructed to obtain the original paths in G .

2.3 Point-to-Point Shortest Path and Distance Queries

Over the past two decades, a plethora of techniques have been proposed to address the deficiency of Dijkstra’s algorithm and its variants by exploiting the characteristics (e.g., planarity) of road networks [9, 15–17, 28, 34, 38, 42–44, 47, 54–56, 69, 71, 74–78, 81]. In particular, the state-of-the-art approaches can be classified into two categories. Algorithms in the first category [74, 76–78] take advantage of the fact that shortest paths in road networks are often *spatially coherent*. To illustrate the concept of *spatial coherence*, let us consider four locations s , s' , t , and t' in a road network. If s is close to s' and t is close to t' , and then the shortest path from s to t is likely to share vertices with the shortest path from s' to t' . Such spatial coherence of shortest paths makes it possible to compress all shortest paths in a road network in a concise format, and the compressed paths can be used to answer queries efficiently. Representative spatial-coherence-based algorithms include *Spatially Induced Linkage Cognizance (SILC)* [74, 76] and *Path-Coherent Pairs Decomposition (PCPD)* [78].

Methods in the second category [15, 16, 38, 71, 75, 81], on the other hand, are built upon the observation that certain vertices in a road network are more important for shortest path queries. For example, a vertex that represents the entrance of a highway tends to be accessed much more frequently (in shortest path queries) than a vertex that corresponds to a road junction in a countryside. This observation motivates various

approaches [15–17, 38, 71, 75, 81] that (i) order the vertices in a road network in terms of their importance, and (ii) pre-compute the shortest paths among the important vertices to accelerate query processing. Among those approaches, *Contraction Hierarchies (CH)* [38] and *Transit Node Routing (TNR)* [15, 16, 81] are shown to be the most efficient. In the following, we demonstrate the representative methods respectively for both categories with more details and examples.

2.3.1 Contraction Hierarchies

Contraction Hierarchies (CH) [38] is a graph indexing technique that imposes a total order on the vertices in G according to their relative importance. It pre-computes the distances among various vertices based on the total order, and it utilizes the pre-computed distances to accelerate shortest path and distance queries. To explain the preprocessing step of CH, let us consider the road network G in Figure 2.2 that contains eight vertices $v_1, v_2, \dots, v_8$ and nine edges. In particular, the lengths of the edges (v_2, v_8) and (v_6, v_8) equal 2, while the lengths of the other edges are 1.

Without loss of generality, assume that CH imposes a total order $v_1 < v_2 < \dots < v_8$ on the vertices in G . The preprocessing step of CH examines the vertices following the total order. For each vertex v_i , CH first inspects the neighbors of v_i (*i.e.*, the vertices adjacent to v_i in G), and checks whether there exist two neighbors v_j and v_k , such that the shortest path from v_j to v_k passes through v_i . For any such v_j and v_k , CH inserts in G an artificial edge c (referred to as a *shortcut*) that connects v_j to v_k , such that $w(c) = \text{dist}(v_j, v_k)$. The shortcut is tagged with v_i to indicate that it is created when v_i is processed. (The tags of shortcuts are crucial for shortest path queries, as will be clarified shortly.) Notice that, with the shortcut added, v_j becomes a neighbor of v_k , and vice versa. Once all neighbors of v_i are examined, v_i is removed from G . This process is referred to as the *contraction* of v_i [38]. After all vertices are contracted, CH terminates the preprocessing step, and the shortcuts that have been created during the contraction process are added to the original road network.

For example, given the road network in Figure 2.2, CH first inspects the vertex v_1 . v_1 has only two neighbors v_3 and v_8 , and the shortest path between v_3 and v_8 goes by v_1 . Therefore, when contracting v_1 , CH connects v_3 and v_8 with a shortcut c_1 , such that $w(c_1) = \text{dist}(v_3, v_8) = 2$, as illustrated in Figure 2.3. After v_1 is removed from G , CH proceeds to examine v_2 . Notice that v_2 has only two neighbors v_3 and v_8 . Given that v_1 has been deleted, the shortest path between v_3 and v_8 consists of the newly constructed shortcut c_1 , which does not pass through v_2 . Hence, v_2 is removed without introducing any new shortcut into G . After that, the contraction process is performed on the other vertices in turn, which leads to two additional shortcuts. In particular, the contraction of v_5 (resp. v_6) results in a shortcut c_2 (resp. c_3) that connects v_7 to v_6 (resp. v_8), and

Figure 2.2: Road Network.

Figure 2.3: Contraction Hierarchies.

$w(c_2) = 2$ (resp. $w(c_3) = 4$). Figure 2.3 illustrates the road network produced by the preprocessing step of CH.

Given the road network augmented with shortcuts, CH answers a distance query between any two vertices $s, t \in V$ using the bidirectional Dijkstra's algorithm with some minor modifications. In particular, when traversing the vertices in G in ascending order of their distances to s (or t), CH considers only those edges and shortcuts that connect a visited vertex v to an unvisited vertex v' whose rank is higher than v , i.e., $v < v'$. For example, if we are to find the distance between v_3 and v_7 in Figure 2.3, the traversal starting from v_3 will visit v_8 but not v_1 or v_2 , since $v_1 < v_2 < v_3 < v_8$. Similarly, the traversal from v_7 will visit v_8 but not any other vertex. After the two traversals meet at v_8 , both of them terminate since there does not exist an edge or shortcut that connects a visited vertex (i.e., v_3 , v_7 , or v_8) to an unvisited vertex with a higher rank. Therefore, CH returns the distance between v_3 and v_7 as $\text{dist}(v_3, v_8) + \text{dist}(v_7, v_8) = w(c_1) + w(c_3) = 6$. In general, the two traversals may not stop immediately after they meet at a vertex; there exist a few conditions that a traversal should fulfill before it can terminate (see [38] for details). But intuitively, CH is more efficient than the bidirectional Dijkstra's algorithm, as it avoids visiting the vertices with lower ranks in the total order.

The aforementioned algorithm can also be used to compute the shortest path from s to t in the *augmented* road network. The resulting path, however, may contain several shortcuts, and hence, it cannot be returned to the user unless it is transformed to a normal path in G . For this purpose, CH examines the tag associated with each shortcut in the path. For a shortcut c that connects two vertices v_j and v_k , if its tag indicates that c is created during the contraction of v_i , and then CH removes c from the path, and replaces it with two edges (v_j, v_i) and (v_i, v_k) . For example, the shortcut c_1 in Figure 2.3 is adjacent to two vertices v_3 and v_8 , and it is constructed when v_1 is contracted. Therefore, if c_1 appears in the shortest path, CH substitutes it with two edges (v_3, v_1) and (v_1, v_8) . Since both (v_3, v_1) and (v_1, v_8) are edges in the original graph, the transformation of c_1 is accomplished. In general, if (v_j, v_i) (resp. (v_i, v_k)) is not an edge in E , and then it can be verified that (v_j, v_i) (resp. (v_i, v_k)) must be a shortcut that has been constructed in the preprocessing step. In that case, CH will recursively replace the shortcut with smaller segments in a similar manner. When all edges remaining in the shortest path are edges in E , CH returns the path as the final result.

Figure 2.4: Transit Node Routing.

Figure 2.5: SILC.

Note that the efficiency of CH is determined by the total order on the vertices. An inferior ordering can lead to $O(n^2)$ shortcuts, which in turn results in an $O(n^2 \log n)$ time complexity for shortest path and distance queries. Existing work [38] on CH has suggested several heuristic approaches for deriving a favorable ordering based on the distribution of the vertices and edges in G .

2.3.2 Transit Node Routing

Transit Node Routing (TNR) [15] is an indexing method that imposes a grid on the road network. It pre-computes the shortest paths from within each grid cell C to a set of vertices that are deemed important for C (those vertices are referred to as the *access nodes* for C). In what follows, we elaborate TNR using the example in Figure 2.4, which shows a grid imposed on the road network in Figure 2.2.

For each cell C in the grid, let us define the *inner shell* (resp. *outer shell*) of C as the boundary of the 5×5 (resp. 9×9) square centered at C . For instance, the dashed-line (resp. dotted-line) square on the left of Figure 2.4 illustrates the inner (resp. outer) shells of the cell that contains v_1 . A set A of vertices in V is a set of *access nodes* for a grid cell C , if and only if it satisfies the following conditions. First, each vertex in A is an endpoint of an edge that intersects the inner shell of C . Second, for any shortest path from a vertex in C to a vertex that lies beyond the outer shell of C , the path must pass through at least one vertex in A , *i.e.*, the vertices in A “cover” all shortest paths from the interior of C to the exterior of its outer shell. For example, in Figure 2.4, $\{v_3, v_8\}$ (resp. $\{v_5\}$) is a set of access nodes for the cell C_1 (resp. C_2).

Given the access nodes of all grid cells, TNR pre-computes two sets of distance information: (i) the distance from each vertex v to each access node of the cell that contains v , and (ii) the distance between any two access nodes of any two different cells. For instance, given the grid in Figure 2.4, TNR pre-computes the distances from v_1 (resp. v_7) to the access nodes of C_1 (resp. C_2), namely, $\text{dist}(v_1, v_3)$, $\text{dist}(v_1, v_8)$, and $\text{dist}(v_7, v_5)$.

In addition, TNR also computes the pairwise distances among the access nodes of C_1 and C_2 , *i.e.*, $dist(v_3, v_5)$ and $dist(v_8, v_5)$.

With the pre-computed distances, TNR can efficiently derive the distance between any two vertices $s, t \in V$, as long as t lies beyond the outer shell of the cell that contains s . For example, suppose that we are to compute the distance between v_1 and v_7 in Figure 2.4. Since v_1 is contained in the cell C_1 , and since v_7 lies in the exterior of C_1 's outer shell, the shortest path from v_1 to v_7 must pass through an access node of C_1 , *i.e.*, the path must go by either v_3 or v_8 . By the same rationale, the shortest path should also pass through v_5 , which is the only access node of the cell C_2 that encloses v_7 . Therefore, the distance between v_1 and v_7 should equal the smaller one of $dist(v_1, v_3) + dist(v_3, v_5) + dist(v_5, v_7)$ and $dist(v_1, v_8) + dist(v_8, v_5) + dist(v_5, v_7)$, both of which can be derived using the pre-computed distances.

In general, given any two cells C_s and C_t such that they are not contained in each other's outer shells, the distance between any vertex s in C_s and any vertex t in C_t can be computed as

$$dist(s, t) = \min_{v_s \in A_s, v_t \in A_t} dist(s, v_s) + dist(v_s, v_t) + dist(v_t, t), \quad (2.1)$$

where A_s and A_t denote the sets of access nodes for C_s and C_t , respectively. On the other hand, if C_t lies inside the outer shell of C_s , and then TNR cannot derive $dist(s, t)$ based on the pre-computed distances. In that case, we need to resort to other techniques (*e.g.*, CH or the bidirectional Dijkstra's algorithm) to compute $dist(s, t)$.

Interestingly, the aforementioned algorithm (for distance queries) can also be adopted to compute the shortest path from s to t . Specifically, we first identify the neighbor v of s that minimizes $dist(s, v) + dist(v, t)$, where $dist(v, t)$ is derived by Equation 2.1. It can be verified that v should lie on the shortest path from s to t . After that, we examine the neighbors of v , and pinpoint the neighbor v' that minimizes $dist(v, v') + dist(v', t)$, so on and so forth. With this traversal approach, we can efficiently compute the part of the shortest path that lies outside the outer shell of C_t . After that, we can start a similar traversal from t to derive the remaining part of the path. In general, TNR can derive the shortest path between s and t using the pre-computed distances, as long as the outer shells of C_s and C_t do not intersect. Otherwise, an alternative method is required for computing the shortest path.

2.3.3 Spatially Induced Linkage Cognizance

Spatially Induced Linkage Cognizance (SILC) [74, 76] is a technique that (i) pre-computes the all-pairs shortest paths in the road network, and (ii) stores the shortest paths in a concise form for efficient query processing. Consider for example the road network G

Figure 2.6: PCPD.

 Figure 2.7: G with weights.

in Figure 2.2. Given the shortest paths between all pairs of vertices in G , SILC first inspects those shortest paths that share the same starting point. For example, let us consider the paths from the vertex v_8 in Figure 2.2 to the other vertices. Observe that each of the shortest paths goes by a neighbor of v_8 . In particular, the paths from v_8 to v_4, v_5, v_6, v_7 pass through v_6 , while the paths from v_8 to v_1 and v_3 are via v_1 . Let us partition the vertices in $V \setminus \{v_8\}$ into equivalence classes, such that for any vertex in the same equivalence class, its shortest path to v_8 passes through the same neighbor of v_8 . Figure 2.5 illustrates the partition of $V \setminus \{v_8\}$, highlighting the vertices in the same partition using a colored or shaded region.

For each vertex $v \in V$, SILC generates a partition of $V \setminus \{v\}$ in the aforementioned manner, and it associates each equivalence class EC with the neighbor of v that lies on the shortest paths from v to the vertices in EC . Given any two vertices $s, t \in V$, SILC computes the shortest path between s and t as follows. It first inspects s , and examines the partition of $V \setminus \{s\}$ to identify the equivalence class EC that contains t . Let v be the neighbor of s that corresponds to EC . According to the property of EC , the shortest path from s to t must pass by v . By inspecting the partition $V \setminus \{v\}$, SILC can identify the neighbor of v that lies on the shortest path from v to t . With an iterative application of this traversal method, the complete shortest path from s to t can be obtained.

The above algorithm requires materializing the partition of $V \setminus \{v\}$ corresponding to each vertex v . To this end, a straightforward approach is to enumerate the elements of each equivalence class in each partition, which, however, leads to a prohibitive $O(n^2)$ space overhead. To address this issue, Samet et al. [74, 76] propose a concise representation of the partitions, based on the observation that *vertices in the same equivalence class are usually located in the same spatial region*. For example, in the partition of $V \setminus \{v_8\}$ in Figure 2.5, the vertices in each equivalence class are close to each other, and the three equivalence classes can be covered using three disjoint squares, respectively. Therefore, instead of recording the element of each equivalence class, we may store the square representations of the equivalence classes, so as to save space. In general, for each vertex $v \in V$, the corresponding partition of $V \setminus \{v\}$ can be represented using $O(\sqrt{n})$

disjoint squares [74]. As a consequence, storing the partitions for all vertices in V incurs only an $O(n\sqrt{n})$ space complexity. Furthermore, searching a square region in a partition can be done in $O(\log n)$ time [74]. Hence, SILC can answer any shortest path query in $O(k \log n)$ time, where k is the number of edges in the shortest path. On the other hand, for any distance query between two vertices $s, t \in V$, SILC needs to first compute the shortest path from s and t , and then return the sum of the lengths of the edges in the path.

2.3.4 Path-Coherent Pairs Decomposition

Path-Coherent Pairs Decomposition (PCPD) is a technique similar to SILC, in the sense that it also requires pre-computing and compressing all shortest paths among the vertices in the road network. Specifically, PCPD employs a concise representation of shortest paths called *path-coherent pairs*. A path-coherent pair is a triplet (X, Y, ψ) , where X and Y are two disjoint square regions, and ψ is either a vertex in V or an edge in E , such that ψ lies on the shortest path from any vertex in X to any vertex in Y . For instance, Figure 2.6 illustrates a path-coherent pair (X, Y, v_8) on the road network in Figure 2.2. Observe that any shortest path from within X to within Y must pass through v_8 . We say that a path-coherent pair (X, Y, ψ) *covers* two vertices $v_a, v_b \in V$, if and only if $v_a \in X$ and $v_b \in Y$.

Given a road network, PCPD pre-computes a set S_{pcp} of path-coherent pairs, such that any two vertices $v_1, v_2 \in V$ are covered by a *unique* path-coherent pair $(X, Y, \psi) \in S_{pcp}$. With S_{pcp} , the shortest path between any two vertices $s, t \in V$ can be computed as follows. First, we retrieve the unique path-coherent pair (X_1, Y_1, ψ_1) in S_{pcp} that covers s and t . Assume without loss of generality that ψ_1 is a vertex in V . By the properties of path-coherent pairs, ψ_1 should lie on the shortest path from s to t . Therefore, we can decompose the shortest path between s and t into two components: the shortest path from s to ψ_1 and the shortest path from ψ_1 to t . After that, we can inspect S_{pcp} to identify a vertex or edge that lies on each of two path components. This enables us to further decompose each component into two smaller parts. By applying the above procedure recursively, we can compute the shortest path from s to t with $O(k)$ lookups in S_{pcp} , where k is number of vertices in the shortest path.

Sankaranarayanan *et al.* [78] show that each lookup in S_{pcp} can be performed in $O(\log |S_{pcp}|)$ time. Furthermore, under some simplifying assumption about the road network, $|S_{pcp}| = O(n)$ holds. This indicates that PCPD has an $O(n)$ space complexity, and its time complexity for shortest path queries is $O(k \log n)$. For any distance query between two vertices $s, t \in V$, PCPD first computes the shortest path between s and t , and then returns the length of the path.

Figure 2.8: Interval labeling of GRAIL.

2.4 Point-to-Point Reachability Queries

Let G be a directed graph, and given two vertices s and t , we say s can reach t if and only if there exists at least one path from s to t in G , and a reachability query from s to t is just to ask whether s can reach t in G . Based on the definition of reachability, we say a subset S of V is a *strongly connected component (SCC)*, if (i) any two vertices in S can reach each other, and (ii) S is *maximum*, i.e., adding any other vertex into S makes it violates condition (i). And it can be verified that for any graph G , the vertex set V can be partitioned into disjoint SCCs.

Motivated by the characteristics of SCCs in G , approaches for PPR queries always construct a concentrated graph $\mathcal{G}$ from G by (i) collapsing each SCC S of G into a single vertex s in $\mathcal{G}$, denoted as $\theta(S) = s$ and (ii) adding a directed edge (u, v) in $\mathcal{G}$ if there exist an edge from any vertex in S_u to any vertex in S_v , where S_u (resp. S_v) is the corresponding SCC of u (v resp.) in G , i.e., $\theta(S_u) = u, \theta(S_v) = v$. It can be verified that $\mathcal{G}$ is a *directed acyclic graph (DAG)* and the computation of $\mathcal{G}$ can be accomplished in linear time [87]. With the concentrated graph $\mathcal{G}$, to answer a PPR query from s to t in G , we first check whether s and t are within the same SCC, if so, we return s can reach t because vertices in the same SCC can reach each other; otherwise, let S (T resp.) be the SCC which contains s (t resp.), we conduct a PPR query from $\theta(S)$ to $\theta(T)$ in $\mathcal{G}$ to answer the query from s to t in G . As such, existing methods for PPR queries always assume the input graph is a DAG, and they can be coarsely classified into three categories based on their query schemes: pruned depth-first search, transitive closure retrieval, and 2-hop label matching. In the sequel, we review the algorithms in each category respectively.

2.4.1 Pruned Depth-first Search

Given two vertices s and t , a naive approach to answer a PPR query from s to t is to start a *depth-first search (DFS)* from s and check whether t can be visited during the

 Figure 2.9: Chain-form of G .

v	$TC(v)$	$I(v)$	$C(v)$
a	$\{c, d, e, f, g\}$	$\{[c, g]\}$	$\{\beta_2, \gamma_1\}$
b	$\{d, e, g\}$	$\{[d, e], [g, g]\}$	$\{\alpha_2\}$
c	$\{e, f\}$	$\{[e, f]\}$	$\emptyset$
d	$\{e, g\}$	$\{[e, e], [g, g]\}$	$\{\beta_2\}$
e	$\emptyset$	$\emptyset$	$\emptyset$
f	$\emptyset$	$\emptyset$	$\emptyset$
g	$\emptyset$	$\emptyset$	$\emptyset$

Figure 2.10: Transitive Closure Compression.

DFS, because the DFS tree of s contains all the vertices that s can reach in the graph. However the cost of performing a full DFS for each PPR query is unacceptable, since the number of vertices in the DFS tree of s can be enormous on a large graph. To address the deficiency, algorithms [20,90,95] in this category pre-compute some auxiliary information to efficiently prune search branches during the DFS in query processing.

The state-of-the-art algorithm in this category is *GRAIL* [95], which generates interval labeling by random DFS traversals on the graph in the preprocessing phase. Specifically, during a DFS traversal on the graph, a pre-post label is recorded at each vertex, which indicate the pre-traversal-order and the post-traversal-order respectively. To illustrate, in Figure 2.8b, the intervals marked near the vertices are the pre-post labels generated by a DFS traversal on G which follows the alphabetical order in visiting untraversed children. Then, *GRAIL* transforms the pre-post labels into min-post labels, which changes the *pre*-label of a vertex into the minimum *pre*-label of its descendants, as shown in Figure 2.8c. And the transformed min-post labels has the property that given two vertices u and v , u cannot reach v if the min-post label of u does not contain the min-post label of v , *e.g.*, the vertex c cannot reach the vertex g , because the interval $[1, 3]$ of c does not contain the interval $[4, 4]$ of g . However, the min-post labeling introduces false positive to the judgement (*e.g.*, the interval $[1, 7]$ of b contains the interval $[2, 2]$ of f , but b cannot reach f), and as a consequence, the min-post labeling can only be utilized to prune the vertices that cannot reach t , but not to answer reachability queries directly in the querying phase. Therefore, being a heuristic approach with no guarantee on pruned search space of DFS in the querying phase, *GRAIL* faces critical query time issue on large graphs, so as the other methods in this category.

2.4.2 Transitive Closure Retrieval

The transitive closure of a vertex v is defined as the set of vertices that v can reach in G , denoted as $TC(v)$. Methods [10, 21, 22, 25, 48, 53, 82, 82, 91, 92] in this category first

compress the transitive closures of each vertex in the preprocessing phase, and then in querying phase, to answer a reachability query from s to t , they decompress the transitive closure of s and then check whether it contains t . Representative methods include chain compression [21, 48], interval compression [10], dual-labeling [92], path-tree [53] and bit-vector compression [91]. Taking interval compression as example, for vertex b in the graph shown in Figure 2.8a, the transitive closure $TC(b)$ is $\{d, e, g\}$. So in the preprocessing phase, it compresses the $TC(b)$ into a set of intervals, *i.e.*, $I(b) = \{[d, e], [g, g]\}$ as shown in Figure 2.10. Then, in the querying phase, to answer a PPR query from vertex b to vertex f , we first decompress $I(b)$ into $TC(b)$, and then by judging that vertex f is not in $TC(b)$, the algorithm returns that vertex b cannot reach vertex f . Alternative example comes from the chain compression [48], and the important property of the chain is, if vertex v can reach vertex u on chain α , it indicates that v can reach any other subsequent vertex of u on chain α . Still refer to the graph in Figure 2.8a, the algorithm first partition vertices in G into three disjoint chains, namely α, β, γ , as shown in Figure 2.9, and the chain-form label is marked at each vertex. Subsequently, the algorithm compresses the transitive closure of each vertex v into the chain-form, denoted as $C(v)$, as shown in Figure 2.10. Noticed that the chain-form of v itself can be omitted from $C(v)$. As such, these methods often have fast query processing on moderate sized graphs, and several independent studies [53, 91, 95] have demonstrated that interval compression and path-tree are the best in terms of query processing time. However, they barely can be scaled to handle large graphs because of the huge space consumption in storing the transitive closures for every vertex in the graph, even in a compressed manner. In a machine with 8GB memory space, methods in this category can only handle graphs with around 1 million vertices, and what was worse, the space consumption increases significantly with the size of the graph, which renders them inapplicable for large graphs. Although we can store the transitive closures on disk, since they are not changing during the querying process. The prohibitively expensive disk-access cost will wipe off their advantage in query processing time. Thus, methods in this category are generally inefficient for large graphs.

2.4.3 2-Hop Label Matching

Algorithms [18, 19, 22, 24–26, 79, 94] in this category pre-compute for each vertex v two labels, *out-label* $L_{out}(v)$ and *in-label* $L_{in}(v)$, where $L_{out}(v)$ contains some vertices that v can reach, while $L_{in}(v)$ consists of some vertices that can reach v . These labels cover the reachability relations among vertices in G , so that given two vertices s and t , the PPR query from s to t is answered by checking whether there exists a vertex that appears both in the out-label of s and the in-label of t . Intuitively, if such common vertex exists, say v , by the above definition of out-label and in-label, we know s can reach t via the

Uncovered		First Iteration		Second Iteration	
$\langle a, c \rangle$	$\langle a, d \rangle$	$\langle a, c \rangle$	$\langle a, d \rangle$	$\langle \emptyset, \emptyset \rangle$	$\langle a, d \rangle$
$\langle a, e \rangle$	$\langle a, f \rangle$	$\langle a, e \rangle$	$\langle a, f \rangle$	$\langle a, e \rangle$	$\langle a, f \rangle$
$\langle a, g \rangle$	$\langle b, d \rangle$	$\langle a, g \rangle$	$\langle b, d \rangle$	$\langle a, g \rangle$	$\langle b, d \rangle$
$\langle b, e \rangle$	$\langle b, g \rangle$	$\langle b, e \rangle$	$\langle b, g \rangle$	$\langle b, e \rangle$	$\langle b, g \rangle$
$\langle c, e \rangle$	$\langle c, f \rangle$	$\langle c, e \rangle$	$\langle c, f \rangle$	$\langle \emptyset, \emptyset \rangle$	$\langle \emptyset, \emptyset \rangle$
$\langle d, e \rangle$	$\langle d, g \rangle$	$\langle d, e \rangle$	$\langle d, g \rangle$	$\langle d, e \rangle$	$\langle d, g \rangle$

v	$L_{out}(v)$	$L_{in}(v)$
a	$\{c, d\}$	$\{\emptyset\}$
b	$\{d\}$	$\{\emptyset\}$
c	$\{c\}$	$\{c\}$
d	$\{d\}$	$\{d\}$
e	$\{\emptyset\}$	$\{c, d\}$
f	$\{\emptyset\}$	$\{c\}$
g	$\{\emptyset\}$	$\{d\}$

Figure 2.11: 2-Hop Labeling.

connection of v . Otherwise, it indicates that s cannot reach t due to the non-existence of bridging vertex.

To illustrate the querying process, consider the graph in Figure 2.8a, the labels generated by the 2-hop method [26], which will be presented later, is shown in Figure 2.11. To answer the PPR query from vertex b to vertex e , we find that vertex d appears in both the out-label of vertex b and in-label of vertex e , hence we know vertex b can reach vertex e . On the other hand, the answer for the PPR query from vertex b to vertex f is unreachable, since $L_{out}(b) \cap L_{in}(f) = \emptyset$. Given the above query scheme, a reasonable implementation is to, first sort the vertices in each out/in label, and then conduct two sequential scans on the out-label of the source vertex s and in-label of the target vertex t simultaneously, once a matching vertex is detected, we return s can reach t immediately; otherwise, we answer s cannot reach t in the end. Noticed that we can perform the sorting operations in the preprocessing phase instead, since the labels will never be modified during the querying phase. So in the worst case, we need to scan $|L_{out}(s)| + |L_{in}(t)|$ vertices to answer the PPR query from s to t under the implementation, which means for random queries, the amortized complexity for the querying process is $\frac{|L|}{|V|}$, where $|L|$ is the labeling size, which equals the total number of vertices contained in all the out/in labels.

The above analysis gave an intuition on the importance of the labeling size (*i.e.*, $|L|$), as it determines the query performance given that $|V|$ is a fixed number for a certain graph. And primitively, the labeling size presents the space consumption since it equals the total number of vertices we need to store for the entire labeling, which makes the size more crucial for large graphs. Although approaches in this category aim to find labeling with smaller size, the problem of minimizing the labeling size is NP-hard [26], which renders most of the approaches turn to approximation or heuristic approaches.

Cohen. *et al.* [26] formalized the problem of minimizing the label size into a covering problem and proposed a greedy algorithm to compute the labeling. In particular, in the beginning, each reachable pairs of vertices in G is marked as uncovered. Then, at each

iteration, the algorithm finds the vertex that can cover most number of uncovered pairs with the lowest increment to $|L|$, and then updates corresponding labels. Finally, the algorithm terminates when all the reachable pairs are covered. Still taking the graph in Figure 2.8a as example, in the beginning, there are 12 uncovered pairs shown in Figure 2.11. Then in the first iteration, vertex d is selected as the crucial vertex, since 9 pairs can be covered by inserting d into the out-labels of vertices a, b, c and in-labels of vertices c, e, g respectively. The cover-cost ratio of vertex d is $9/6 = 1.5$, which is the highest. Next, in the second iteration, with only 3 pairs $\langle a, c \rangle, \langle c, e \rangle, \langle c, f \rangle$ uncovered, c is selected, and c is inserted into $L_{out}(a), L_{out}(c), L_{in}(c), L_{in}(e), L_{in}(f)$ respectively to cover the remaining 3 pairs, after which, the algorithm terminates, and the labeling shown in Figure 2.11 is generated.

However, the above greedy algorithm lacks efficiency on large graphs, because, i) the entire transitive closures for all vertices must be materialized in the beginning of the algorithm, which incurs huge cost in space consumption; ii) in each iteration, the cover-cost ratio must be computed for each remaining vertices, which makes the preprocessing time prohibitively long for large graphs. Even with significant reduction of the preprocessing cost by following work [19, 52, 79], these approaches can only handle graphs with far few than a million vertices. Nevertheless, compared with the above transitive closure retrieval category, approaches in this category have much smaller space consumption, while their amortized query time complexity is determined by the average labeling size, which is not as sensitive as the pruned depth-first search approaches in the ever-increasing size of large graphs, thus, possess the potential to handle large graphs.

2.5 Summary

In this chapter, we first introduce the Dijkstra's algorithm and its bidirectional variant, which are the classic solutions for shortest path and distance queries. After that, we survey the advanced techniques for SSSP and SSD queries (resp. PPSP and PPD queries), which address the deficiency of Dijkstra's algorithm (resp. the bidirectional variant) on large graphs. Finally, we present three categories of reachability indices on static graphs, which are the preliminary for dynamic approaches.

For SSSP and SSD queries, we present EM-Dij and VC-Index which are approaches that can handle disk-resident graphs. In particular, EM-Dij aims to reduce the massive disk accesses incurred by Dijkstra's algorithm from delaying certain relaxations to the upcoming iterations, so that in each scan of disk, more relaxations are executed. Unfortunately, the performance of EM-Dij is severely affected by the edge weight distribution of the graph. On the other hand, VC-Index applies the concept of vertex cover to construct a tree-structured index on the input graph, and trades-off the index space consumption

with the query time efficiency to determine a rational height of the tree structure, so that the SSSP and SSD queries can be answered efficiently. However the nature of the vertex cover concept renders VC-Index only applicable for undirected graphs.

For PPSP and PPD queries on road networks, the state-of-the-art methods employ the characteristics of the road networks to achieve better querying efficiency. Specifically, the spatial-coherence-based methods rely on the fact that shortest paths with nearby sources and destinations share common sub-path in the middle. On the other hand, the vertex-importance-based methods are built on the observation that some vertices are more important for shortest paths on road networks, *e.g.*, vertices representing bridges and highway exits. Thus, for each category, we demonstrate two representative methods with examples that reveal the spirits of these methods in handling large road networks.

For the PPR query, we summarize the existing methods for static graphs into three categories: pruned depth-first search, transitive closure retrieval and bidirectional label matching. In particular, pruned depth-first search approaches prune the DFS conducted from the source vertex by precomputed labels, so that the reachability to the target vertex can be determined rapidly, but the online search essence inevitably faces critical query efficiency problem on large graphs. In contrast, transitive closure retrieval methods compress the transitive closures of each vertex in the preprocessing phase, so that in answering the PPR query, they decompress the transitive closure of the source vertex, and return whether the target vertex is contained. Albeit simple and fast in query answering, methods in this category can barely scale to large graphs, due to the cost of materializing the transitive closures for all the vertices is prohibitively expensive on large graphs. Different from above techniques, bidirectional label matching approaches assign each vertex a pair of labels: out-label and in-label, which capture partial reachability of the vertex bidirectionally. As such, the PPR query is answered by matching the out-label of the source vertex and the in-label of the target vertex. Methods in this category often have moderate labeling space and their query complexity does not increase significantly with the size of the graph, hence, possess the potential to handle large graphs.

All the aforementioned approaches require pre-processing the input graph which brings additional space and time consumptions, but the cost pays off when the number of queries to be processed is large, as is often the case for graph applications. Thus, our solutions to the three querying problems also follow the same infrastructure.

Chapter 3

Point-to-Point Shortest Path and Distance Queries

Given two locations s and t in a road network, a *point-to-point distance (PPD)* query returns the network distance from s to t , while a *point-to-point shortest path (PPSP)* query computes the actual route that achieves the minimum distance. These two types of queries find important applications in map, navigation, and location-based services. To illustrate, consider that a user of a map service is looking for a nearby Italian restaurant for dinner. In response to the user’s query, the service provider can first retrieve the list of Italian restaurants in the region close to the user’s current location u . After that, the network distance from u to each restaurant is computed (using a PPD query), and those distances are returned to the user along with the list of restaurants. Then, if the user chooses a preferred restaurant r from the list, the service provider can employ a PPSP query to provide the user with driving directions from u to r .

The classic solution for PPSP and PPD queries is Dijkstra’s algorithm [32]. It traverses the road network nodes in ascending order of their distances from s ; once it reaches t during the traversal, it can compute the distance from s to t and can retrieve the shortest path based on the information recorded before t is visited. With proper data structures, Dijkstra’s algorithm runs in $O(n \log n + m)$ time for any PPSP or PPD query, where n (resp. m) is the number of nodes (resp. edges) in the road network. Albeit simple and elegant, Dijkstra’s algorithm is inefficient for sizable road networks, as it requires traversing all network nodes that are closer to s than t , which incurs a significant overhead when s and t are far apart.

A plethora of techniques [9, 15, 17, 28, 34, 38, 42–44, 47, 54–56, 67, 71, 74, 77, 78, 86] have been proposed to improve over Dijkstra’s algorithm in terms of either practical efficiency or asymptotic bounds (details are shown in Section 3.1). The existing methods that focus on practical performance mostly rely on heuristics, and hence, their asymptotic bounds are unattractive in general. For instance, the best heuristic approach by Geisberger

et al. [38] answers PPSP or PPD queries in at most a few milliseconds even on road networks with millions of nodes, but its space and time complexities are both $O(n^2)$, i.e., its asymptotic performance is even worse than that of Dijkstras algorithm. On the other hand, the solutions that offer favorable query time complexities often entail prohibitive preprocessing cost or space overhead, rendering them only applicable for small datasets. For example, the state-of-the-art approaches by Samet et al. [2008] and Abraham et al. [2010] provide superior bounds on query time, but they require pre-computing the shortest path between *any* pair of nodes, which is impractical for the large road networks commonly used in modern map applications.

To narrow the gap between theory and practice in answering PPSP and PPD queries on road networks, this chapter presents the *Arterial Hierarchy (AH)* index structure. On the theoretical side, we show that, under a realistic assumption, AH answers any distance query in $\tilde{O}(\log \alpha)$ time¹, where $\alpha = d_{max}/d_{min}$, and d_{max} (resp. d_{min}) is the largest (resp. smallest) L_∞ distance between any two nodes in the road network. In addition, any shortest path query can be answered in $\tilde{O}(k + \log \alpha)$ time, where k is the number of nodes on the shortest path. On the practical side, we experimentally evaluate AH on a large set of real road networks with up to twenty million nodes, and we demonstrate that (i) AH outperforms the state of the art in terms of query time, and (ii) its space and pre-computation overheads are moderate.

In a nutshell, AH organizes the nodes in the road network into a hierarchy, based on which it pre-computes auxiliary information to facilitate query processing. For instance, given the road network G with bidirectional edges in Figure 3.1, AH constructs a three-level hierarchy H (illustrated in Figure 3.2), where each level consists of a disjoint subset of the nodes in G . Note that H contains all edges in G , as well as two auxiliary edges, $\langle v_9, v_{10} \rangle$ and $\langle v_{10}, v_{11} \rangle$, each of which has a length that equals the distance between the two nodes that it connects. These auxiliary edges are referred as *shortcuts*, and they can be exploited to considerably reduce the numbers of nodes and edges that need to be traversed during a shortest path or distance query.

For example, given a distance query between v_1 and v_{10} (in G), AH would perform two alternating traversals (in H) starting from v_1 and v_{10} , respectively, and *it would always avoid traveling from a higher-level node to a lower-level node*. In particular, the traversal starting from v_1 can only reach two nodes, v_{10} and v_{11} , since (i) v_{11} is the only node adjacent to v_1 , and (ii) from v_{11} , AH would only traverse to v_{10} (since v_{10} is the only neighbor of v_{11} that is not at a lower level than v_{11}). Similarly, the traversal starting from v_{10} would only reach v_{11} . Once the two traversals terminate, the distance between v_1 and v_{10} is calculated by summing up the weights of $\langle v_1, v_{11} \rangle$ and $\langle v_{11}, v_{10} \rangle$.

¹ $\tilde{O}$ is a variant of O notation that ignores logarithmic factors. E.g., $f(n) = \tilde{O}(g(n))$ is shorthand for $f(n) = O(g(n) \log^k g(n))$ for some constant k .

In general, AH answers any distance query with two traversals of the node hierarchy, such that each traversal only moves up from low-level nodes to high-level nodes, but not vice versa. We show that, for real road networks, the node hierarchy contains $O(\log \alpha)$ levels, where $\alpha = d_{max}/d_{min}$. Furthermore, each traversal performed by AH visits only a constant number of nodes and edges in any level of the hierarchy. As a consequence, the total number of nodes and edges visited by AH is $O(\log \alpha)$, which results in an $\tilde{O}(\log \alpha)$ time complexity for any distance query. In addition, once the distance between two nodes s and t is computed, AH can derive the actual shortest path from s to t in $O(k)$ time, where k is the number of nodes on the shortest path.

The aforementioned time complexities of AH rely on an assumption on road networks (to be clarified in Section 3.2). We provide detailed discussion on the assumption, and we demonstrate its applicability on practical road networks with extensive experiments on a large collection of real datasets. These experimental findings not only form a basis for our theoretical claims but also shed light on the characteristics of real road networks, which paves the path for future research on shortest path and distance queries.

The rest of this chapter is organized as follows. Section 3.1 reviews related work of PPSP and PPD queries on road networks. Section 3.2 formally defines the problem and demonstrates an assumption on road networks. Section 3.3 presents a first-cut solution which is the miniature of AH. Section 3.4 presents Arterial Hierarchy. Section 3.5 shows the experimental results of AH on various real-life road networks. Section 3.6 summarizes this chapter.

3.1 Related Work

Numerous techniques (*e.g.*, [9, 15, 17, 28, 34, 38, 42–44, 47, 54–56, 67, 71, 74, 77, 78, 86]) have been proposed for processing shortest path and distance queries on road networks. Many of these techniques focus on practical performance, and they are mostly heuristic-based. For example, ALT [42] pre-computes the road network distances from each node to a fixed set of nodes (referred to as *landmarks*), and then utilizes those pre-computed distances to reduce the search space of each query. Hiti [55] partitions the road network into vertex-disjoint subgraphs, and then pre-computes the shortest paths that connect different subgraphs to facilitate query processing. We refer the reader to [93] for a survey of the existing heuristic-based techniques.

In addition, there also exists a large number of worst-case efficient algorithms for shortest path and distance queries (see [34, 44, 56, 67, 74, 78] and the references therein). Most of these algorithms assume that the road network is a planar graph with non-negative weights, while some recent work [9, 74, 78] adopts more subtle assumptions on the road network to derive tighter bounds on space and time complexities. Compared

with the state of the art, our method offers superior query efficiency while incurring moderate costs of space and pre-computation (See details in Section 3.4.5).

The work most related to ours is by Bast et al. [15], Abraham et al. [9], and Geisberger et al. [38]. Bast et al. [15] observe that, in practice, there often exist a small set S of nodes in the road network (referred to as *transit nodes*), such that any shortest path connecting two distant locations must pass through at least one node in S . Based on this observation, Bast et al. propose a heuristic solution for answering shortest path and distance queries. However, the proposed solution is shown to be flawed in that it may return incorrect query results [93]. Our notion of arterial dimension is motivated by Bast et al.’s observation, but our definition of arterial edges is considerably different from Bast et al.’s formulation of transit nodes.

Abraham et al. [9] introduce a theoretical abstraction of Bast et al.’s observation, based on which they propose several worst-case efficient algorithms for shortest path and distance queries. The proposed algorithms adopt an assumption that is similar in spirit to our Assumption 3.1, but is more elegant in a theoretical sense. Nevertheless, the assumption adopted by Abraham et al. has not been tested on any real road networks, while our Assumption 3.1 is backed by empirical evidence from real datasets, as shown in Section 3.2. Furthermore, Abraham et al.’s algorithms require pre-computing the shortest path between any pair of nodes in the road network, which renders them inapplicable even for moderate-size datasets.

Geisberger et al. [38] propose a road network index called the *Contraction Hierarchies*, which (i) heuristically imposes a total order on the road network nodes and (ii) constructs shortcuts from low-rank nodes to high-rank nodes to enable efficient query processing. Our AH method is inspired by CH, and it outperforms CH in terms of both asymptotic and practical performance, as will be shown in Section 3.5.

3.2 Problem and Assumptions

Let G be a road network. We assume that G is a directed, degree-bounded, and connected graph with a node set V and an edge set E , such that (i) $|V| = n$, (ii) each node in V locates in a two-dimensional space, and (iii) each edge $e \in E$ is associated with a positive length $l(e)$. For any path P in G , we define its length $l(P)$ as the total length of the edges in P .

We study two types of queries on G , namely, shortest path queries and distance queries. Given an ordered pair of nodes $(s, t) \in V \times V$, a shortest path query asks for a sequence of edges $e_1, e_2, \dots, e_k$ that form a path from s to t , such that $\sum_{i=1}^k l(e_i)$ is minimized. On the other hand, a distance query from s to t asks only for the value of $\sum_{i=1}^k l(e_i)$ instead of the actual shortest path. For convenience, we define the *distance* from s to t as $dist(s, t) = \sum_{i=1}^k l(e_i)$.

 Figure 3.1: Road network G .

 Figure 3.2: Node hierarchy H .

Our solution for shortest path and distance queries is developed based on an observation on the properties of real road networks, as explained in the following.

Observation. Assume that we impose a square grid R on G . Let B be a region containing 4×4 cells in the grid. We define the left-most (resp. right-most) column of cells in B as the *west strip* (resp. *east strip*) of B , and we refer to the vertical line that evenly divides B as the *vertical bisector* of B . We also define B 's *north strip*, *south strip*, and *horizontal bisector* in a similar manner. For example, Figure 3.4 illustrates (i) a square grid imposed on the road network in Figure 3.1, (ii) a region B covering 4×4 grid cells, and (iii) the strips and bisectors of B .

We observe that, in practice, the shortest paths between the west and east strips of B can often be covered by a small set S_{we} of road network edges intersecting B 's vertical bisector. That is, given any two points in B 's west and east strips, respectively, the shortest path between the two points should pass through at least one edge in S_{we} . For instance, suppose that B covers the area of a state. In that case, any shortest path P between the west and east strips of B corresponds to a route that connects the west and east ends of the state. Intuitively, P would have to pass through some major intra-state highways. Therefore, if S_{we} contains the road network edges on the intra-state highways that intersect B 's vertical bisector, and then S_{we} should cover any aforementioned shortest path P . Furthermore, the cardinality of S_{we} should be small, as there should exist only a handful of major highways in the state that go across the vertical bisector. Similar statements can be made even when B corresponds to a larger region (*e.g.*, a continent) or a smaller one (*e.g.*, a city). In addition, we also observe that all shortest paths between the north and south strips of B can be covered by a few edges intersecting B 's horizontal bisector.

The above observations are similar in spirit to those made in previous work [9, 15, 78], which all illustrate that there exists a small set of *important* road network edges or nodes that cover all shortest paths connecting distant regions (see Section 3.1 for a survey of related work). In what follows, we will formalize our observations and provide empirical evidence, so as to form a basis for further discussions in Sections 3.3 and 3.4.

Formalization. Given a region B of 4×4 grid cells, we say that a road network path P is a *local path* in B , if at most one edge in P intersects the boundary of B . For instance, in Figure 3.4, the paths $\langle v_9, v_5, v_8 \rangle$ and $\langle v_{11}, v_7, v_4 \rangle$ are both local paths in B . A local path in B is the *shortest*, if it is shorter than any other local path in B with the same endpoints. For simplicity, we assume that there do not exist two local paths in B that share the same endpoints and have the same length – This assumption can be enforced by adding a small perturbation to each edge in G , as shown in Appendix A.1.

We are interested in the local shortest paths between opposite strips of B , and a set of edges on B 's bisectors that cover all such paths, as defined in the following.

Definition 3.1 (Spanning Paths & Arterial Edges). *A local shortest path P in B is a **spanning path** of B , if (i) the two endpoints of P are on different sides of a bisector of B (denoted as l_b), and (ii) neither of the endpoints is contained in a grid cell adjacent to the bisector l_b . Any edge on P that intersects l_b is an **arterial edge** of B .*

By Definition 3.1, the path $P = \langle v_9, v_6, v_{10}, v_8 \rangle$ in Figure 3.4 is a spanning path of B , since (i) P is a local shortest path of B , (ii) v_9 and v_8 are on different sides of B 's vertical bisector, and (iii) neither v_9 nor v_8 is in a grid cell adjacent to the bisector. Accordingly, the edge $\langle v_6, v_{10} \rangle$ is an arterial edge of B , as it is the only edge in P that intersects B 's vertical bisector. Likewise, $\langle v_{11}, v_7, v_4 \rangle$ is also a spanning path of B , and $\langle v_{11}, v_7 \rangle$ is an arterial edge of B .

As explained previously, the number of arterial edges in a (4×4) -cell region B tends to be small in practice, since there usually exist only a few major connections between opposite strips of B . We formalize this observation as follows.

Assumption 3.1 (Arterial Dimension). *For any square grid on G and any region B with 4×4 grid cells, the number of arterial edges of B is at most a constant λ , referred to as the **arterial dimension** of G .*

To demonstrate the applicability of Assumption 3.1, we conduct an experiment on eight real datasets that represent various parts of the road network in the United States (see Section 3.5 for details). The weight of each edge in the data equals the time required to travel between the two endpoints of the edge. On each dataset, we impose a $2^r \times 2^r$ square grid ($r \in [3, 17]$), and compute the number of arterial edges for each (4×4) -cell region (ignoring the regions that are empty). After that, we compute the maximum number of arterial edges for a region, as well as the mean, 90% quantile, and 99% quantile. Figure 3.3 plots the results as functions of the grid resolution r . Regardless of the grid resolution and the dataset size, the maximum number of arterial edges for a (4×4) -cell region is at most 97, and is below 60 in most cases. Furthermore, the 90% and 99% quantiles are at most 60, while the mean is never above 22. This indicates that

Figure 3.3: Arterial dimensions of real road networks.

practical road networks have fairly small arterial dimensions. Notice that the assumption of arterial dimension only holds for road networks, since it captures the spatial coherence of spanning paths in any (4×4) -cell region. Thus, for general graphs with no spatial property of nodes, arterial dimension cannot be defined. In Sections 3.3 and 3.4, we will exploit this fact to construct efficient indices for shortest path and distance queries for road networks.

3.3 A First-Cut Solution

This section presents *FC* (first-cut), an index structure designed for road networks with small arterial dimensions. *FC* is worst-case efficient for distance queries, and its space consumption is modest; nevertheless, *FC* is unsuitable for large road networks as it incurs significant pre-processing cost. The reasons that we introduce *FC* are (i) it is a conceptually simple method that demonstrates the key idea of our proposal, and (ii) with a few modifications and optimizations, *FC* can be turned into a scalable method that handles both distance and shortest path queries (see Section 3.4).

3.3.1 Index Construction

Given a road network G , *FC* first assigns a *level* to each node in G , such that nodes with higher levels tend to be more *important*. After that, *FC* organizes the nodes into

Figure 3.4: Strips and bisectors of a region with 4×4 cells.

a hierarchy based on their levels, and it adds auxiliary edges between various nodes to facilitate query processing. In the following, we will elaborate how the node levels are decided and how the auxiliary edges are created.

Deciding Node Levels. First, FC imposes on G a (4×4) -cell square grid that tightly covers all nodes in G . After that, FC recursively splits each grid cell into 2×2 smaller cells, until each cell contains at most one node in G . This results in a sequence of square grids with increasing resolutions. Let h be the number of grids thus constructed. We use R_i to denote the grid with $2^{h+2-i} \times 2^{h+2-i}$ cells, i.e., R_h is the (4×4) -cell grid that FC first constructed, and R_1 is the grid with the highest resolution.

Let d_{max} (resp. d_{min}) be the largest (resp. smallest) L_∞ distance between any two nodes in G . It can be verified that $h \leq \log_2(d_{max}/d_{min}) - 1$. We note that h is always a small number for practical road networks: Even if d_{max} is as large as the length of the Equator ($\approx 4 \times 10^7$ meters) and d_{min} is as small as 1 meter, the value of h is no more than 26.

Given each R_i ($i \in [1, h]$), FC computes the arterial edges in any (4×4) -cell region in R_i . Let A_i be the set of arterial edges obtained from R_i . For any edge in A_i , we define it as a *level- i edge* if it does not appear in $A_{i+1}, \dots, A_h$. If an edge does not appear in any A_i , and then we refer to it as a *level-0 edge*. In other words, an edge has a higher level if it is an arterial edge for a larger region. Similarly, we also define the level of each node v in G : we say that v is a *level- i node* if it is adjacent to some edge at level i but not any edge at level $i+1, \dots, h$. Intuitively, a higher-level node tends to be more important for shortest path and distance queries.

Creation of Shortcuts. Once the node levels are decided, FC organizes the nodes in G into a hierarchy H of $h+1$ levels $L_0, L_1, \dots, L_h$, such that all level- i ($i \in [0, h]$) nodes are contained in L_i . For example, Figure 3.2 illustrates a 3-level hierarchy of the nodes in Figure 3.1.

The hierarchy H retains all edges in G . In addition, FC inserts into H some auxiliary edges, referred to as *shortcuts*. For any two nodes v_s and v_t , FC creates a shortcut c from v_s to v_t , if the shortest path from v_s to v_t only passes through nodes whose levels are

lower than both v_s 's and v_t 's. Furthermore, the length of c equals to the distance from v_s to v_t , *i.e.*, $l(c) = \text{dist}(v_s, v_t)$. For instance, consider the nodes v_6, v_8, v_9, v_{10} in Figure 3.2, whose levels are 0, 1, 1, and 2, respectively. There is a shortcut from v_9 to v_{10} , since the shortest path from v_9 to v_{10} only goes through v_6 , and the level of v_6 is lower than those of v_9 and v_{10} . On the other hand, there is no shortcut from v_8 to v_9 , since the shortest path from v_8 to v_9 passes through v_{10} , whose level is higher than both v_8 's and v_9 's.

The shortcuts inserted into H enable us to avoid visiting unimportant nodes when processing distance queries. For example, given the shortcut c from v_9 to v_{10} in Figure 3.2, we can determine that $\text{dist}(v_9, v_{10}) = l(c)$, without having to compute the actual shortest path from v_9 to v_{10} . In general, for any two nodes v_s and v_t in H , there exists a path from v_s to v_t that bypasses unimportant nodes with shortcuts, as will be explained in Section 3.3.2. For simplicity, we will use the term “edge” to refer to either an original edge or a shortcut in H , unless otherwise specified.

3.3.2 Query Processing

Consider a query q that asks for the distance from a node s in G to another node t . Given the node hierarchy H , FC answers q with two concurrent traversals of H that start from s and t , respectively. Each traversal is performed using a *constrained* version of Dijkstra's algorithm [32], as explained in the following.

Traversal Algorithm. The traversal from s maintains a hash table T_s and a priority queue Q_s . The hash table T_s maps each node v in G to a value $\kappa_s(v)$, which equals the length of the shortest path from s to v that has been found so far. Initially, we have $\kappa_s(s) = 0$ and $\kappa(v) = +\infty$ for any other node v .

Meanwhile, each entry in the priority queue Q_s corresponds to a certain node v' in G , and the key of the entry equals $\kappa_s(v')$. In the beginning of the traversal, Q_s contains only one entry, which corresponds to s . Subsequently, FC iteratively extracts (from Q_s) the node u with the smallest key. For each u extracted, FC inspects every edge $\langle u, v \rangle$ in H that starts from u , and it checks whether v satisfies certain constraints. (We will clarify these constraints shortly). If v violates any of the constraints, it would be ignored; otherwise, FC would further check whether $\kappa_s(u) + l(\langle u, v \rangle) < \kappa_s(v)$, *i.e.*, whether the path from s to v via u is shorter than all known paths from s to v . If the inequality holds, and then FC sets $\kappa_s(v) = \kappa_s(u) + l(\langle u, v \rangle)$ and inserts v into Q_s (if v has not been inserted before).

The traversal from t also maintains a hash table T_t and a priority queue Q_t . It is performed in a manner similar to the traversal from s , with one notable difference: Whenever FC extracts a node u from T_t , it only inspects the edges $\langle v, u \rangle$ that *points to* u . In other words, the traversal from t focuses on paths that end at t .

FC conducts the above two traversals in a round-robin fashion, *i.e.*, it extracts nodes from the two priority queues Q_s and Q_t in turns. To determine when the traversals can be terminated, FC maintains a variable θ that records the length of the shortest path from s to t that is seen so far. Initially, $\theta = +\infty$. After that, for each node u extracted from either priority queue, FC retrieves its key $\kappa_s(u)$ in the hash table T_s , as well as its key $\kappa_t(u)$ in T_t . Recall that $\kappa_s(u)$ (resp. $\kappa_t(u)$) records the length of the shortest path from s to u (resp. from u to t) found so far. Therefore, the shortest path from s to t should be no longer than $\kappa_s(u) + \kappa_t(u)$. Based on this, if $\kappa_s(u) + \kappa_t(u) < \theta$, and then FC would update θ and set it to $\kappa_s(u) + \kappa_t(u)$.

Whenever θ is no more than the smallest key value in Q_s , we know that $\text{dist}(s, u) \geq \theta$ for any node u remaining in Q_s , which indicates that u cannot be on the shortest path from s to t . In that case, FC would terminate the traversal from s . Similarly, the traversal from t is stopped if θ is no more than any key values in Q_t . When both traversals are terminated, FC returns θ as the answer to the distance query.

Constraints on Node Traversals. As mentioned above, whenever FC extracts a node u from a priority queue (either Q_s or Q_t), it inspects the neighbors of u , and it processes only those neighbors v that satisfy certain constraints. Specifically, there are two constraints on v :

- (i) *Level Constraint:* v should not be at a level lower than u 's.
- (ii) *Proximity Constraint:* Let i be the level of v ($i \in [0, h - 1]$). If u is extracted from Q_s (resp. Q_t), and then v and s (resp. t) should be covered in the same (3×3) -cell region in R_{i+1} . (Recall that R_i is a square grid with $2^{h+2-i} \times 2^{h+2-i}$ cells.)

Both of the above constraints are intended to improve the efficiency of FC. In particular, the level constraint helps FC bypass unimportant nodes during query processing. For example, consider that we use FC to compute the distance from v_8 to v_{11} in Figure 3.2. As explained previously, FC would invoke two traversals starting from v_8 and v_{11} , respectively. Since v_{11} is at level 2, the traversal from v_{11} would only visit the neighbors of v_{11} that are at levels no lower than 2. As a consequence, v_{10} is visited (since its level equals 2), while v_1 and v_9 are bypassed. After that, the traversal would visit any neighbor of v_{10} whose level is no lower than that of v_{10} . Since none of the neighbors of v_{10} fulfills this requirement, the traversal terminates. Similarly, the traversal from v_8 would first visit two of v_8 's neighbors, v_7 and v_{10} , ignoring the remaining neighbor v_3 , since v_3 's level is lower than that of v_8 . After that, the traversal visits only v_{11} and terminates, as all other remaining nodes violate the level constraint. In summary, the two traversals by FC visit only four nodes: v_7 , v_8 , v_{10} , and v_{11} .

Meanwhile, the proximity constraint ensures that FC only searches a small number of grid cells in each level of the node hierarchy H . For example, suppose that we are

Figure 3.5: Illustration of the proximity constraint.

given the node hierarchy in Figure 3.5, and we use FC to compute the distance from v_1 to v_6 . Among the two traversals invoked by FC, the one starting from v_1 would first visit v_2 , and then v_3 and v_7 . The node v_7 has a neighbor v_8 , which is at the same level as v_7 , *i.e.*, v_8 satisfies the level constraint. However, v_8 would still be ignored by FC, as it violates the proximity constraint. In particular, v_8 is a level-1 node, but there does not exist any (3×3) -cell region in R_2 that can cover both v_1 and v_8 . In contrast, the node v_4 , which is a neighbor of v_3 , would be visited by FC as it satisfies both the level and proximity constraints. Specifically, the level of v_4 equals 2, which is no less than that of v_3 ; furthermore, v_4 and v_1 are contained in the same (3×3) -cell region in R_3 . Note that, although FC ignores v_8 , the correctness of the query result is not affected, since v_8 is not on the shortest path from v_1 to v_6 .

In general, the proximity constraint guarantees that in each level i of the node hierarchy, FC only traverses the nodes contained in two (5×5) -cell regions, which are centered at the source s and destination t of the query, respectively. In particular, the region centered at s (resp. t) is the union of all (3×3) -cell regions that cover s (resp. t). This, when combined with the level constraint, ensures that FC is worst-case efficient in terms of query time, as will be shown in Section 3.3.3.

3.3.3 Complexity Analysis

In this section, we will prove that FC takes $O(hn)$ space, and it answers any distance query in $O(h^2)$ time, where h is the maximum level in the node hierarchy H , and n is the number of nodes in the road network G . In addition, we will discuss the pre-computation time of FC.

Query Time. As explained in Section 3.3.2, FC answers any distance query by two traversals on the node hierarchy H , starting from the source s and destination t of the query, respectively. Due to the level and proximity constraints, each traversal of FC visits any level of H at most once; in addition, for the i -th level ($i \in [0, h]$), each traversal only examines the nodes in a (5×5) -cell region in the grid R_{i+1} . A natural question is: How many level- i nodes are there in the (5×5) -cell region? The following lemma provides an answer.

Lemma 3.1. *Any $(\alpha \times \alpha)$ -cell region in R_i contains $O(\alpha^2 \lambda)$ level- i nodes in H , where λ is the arterial dimension of G .*

Proof. The proofs for all lemmas and theorems in this thesis can be found in Appendix A.2. $\square$

To explain the rationale behind Lemma 3.1, recall that each level- i node in H is adjacent to an arterial edge in a (4×4) -cell region in R_i . Furthermore, each (4×4) -cell region in R_i has at most λ arterial edges. For any $(\alpha \times \alpha)$ -cell region in R_i , it can overlap with $O(\alpha^2)$ regions of 4×4 cells, and hence, it contains $O(\alpha^2 \lambda)$ level- i nodes.

Observe that any (5×5) -cell region in R_{i+1} corresponds to a (10×10) -cell region in R_i . By Lemma 3.1, this region contains $O(\lambda)$ level- i nodes in H . In other words, the number of level- i nodes visited by each traversal of FC is $O(\lambda)$. Given that H has $h + 1$ levels, the total number of nodes traversed by FC is $O(h\lambda)$.

Next, we will show that each node in H has $O(h\lambda)$ edges that satisfy the level constraint. (The edges that violate the constraint can be removed from H beforehand, as they would never be traversed by FC for any query). Consider any level- i node u , and any node v whose level is at least i . By the way that H is constructed, there is a shortcut connecting u to v , if and only if the shortest path between u and v only goes through nodes at levels lower than i . Intuitively, this indicates that u and v should not be too far apart from each other; otherwise, the shortest path between u and v in G would be a path that connects two distant locations, in which case the path might contain some highly important node at a level higher than i , due to which there would not be any shortcut between u and v . More formally, we have the following lemma:

Lemma 3.2. *Let P be a shortest path in G , such that no (3×3) -cell region in R_i ($i \in [1, h]$) can cover all nodes in P simultaneously. Then, P must contain an arterial edge of some (4×4) -cell region in R_i .*

By Lemma 3.2, u and v must be covered in the same (3×3) -cell region in R_{i+1} ; otherwise, the shortest path between u and v must pass through a level- $(i+1)$ node, for which there cannot exist any shortcut between u and v . This implies that v must be in the (5×5) -cell region in R_{i+1} that is centered at u . By Lemma 3.1, this region contains $O(\lambda)$ level- i nodes in H . With a similar analysis, it can be shown that the region also covers $O(\lambda)$ nodes at any level higher than i . Therefore, the total number of edges adjacent to u is $O(h\lambda)$.

In summary, FC answers any distance query with two constrained Dijkstra search, each of which traverses $O(h\lambda)$ nodes and $O(h^2 \lambda^2)$ edges. As such, the time complexity of each traversal equals $O(h\lambda \log(h\lambda) + h^2 \lambda^2)$. Given that the arterial dimension λ of the road network G is a constant, the overall time complexity of FC is $O(h^2)$.

Figure 3.6: Reconstruction of shortest paths.

Space Complexity. Recall that the node hierarchy contains $h + 1$ levels, each of which contains $O(n)$ nodes. In addition, each node in H has $O(h\lambda)$ edges. Therefore, the space consumption of FC is $O(hn)$ when λ is constant.

Preprocessing Cost. The pre-computation of FC consists of two steps: First, we identify the arterial edges in any (4×4) -cell region in any grid R_i ($i \in [1, h]$); After that, we decide the level of each node and we connect pairs of nodes with shortcuts. The identification of arterial edges requires computing the shortest paths in all (4×4) -cell regions in all R_i , which incurs considerable overhead, especially when the granularity of the grid is low. Similarly, the construction of shortcuts is time consuming as it requires deriving a larger number of shortest paths (between nodes that are potentially far apart). Such significant preprocessing cost renders FC only applicable for small road networks. In Section 3.4, we will address this issue and present a modified and scalable version of FC.

3.3.4 Correctness Proof

Let $P = \langle v_1, v_2, \dots, v_k \rangle$ be a shortest path in G . Let P' be a path from v_1 to v_k on the node hierarchy H , such that FC reports $l(P')$ as the distance from v_1 to v_k . We will prove the correctness of FC's result by showing that $l(P') = l(P)$. In particular, we will show that both $l(P') \geq l(P)$ and $l(P') \leq l(P)$ hold.

Proving $l(P') \geq l(P)$. Recall that every shortcut on H corresponds to a path in G . Therefore, if we replace each shortcut in P' with the corresponding path, we can transform P' into a path P'' , such that (i) P'' does not contain any shortcut, (ii) P'' connects v_1 to v_k , and (iii) $l(P') = l(P'')$. On the other hand, we have $l(P'') \geq l(P)$, since P is the shortest path from v_1 to v_k in G . Therefore, $l(P') = l(P'') \geq l(P)$.

Proving $l(P') \leq l(P)$. Assume for simplicity that P contains a node v_j ($j \in [1, k]$) whose level is higher than that of any other node on P . (Our analysis can be easily extended to the case when the highest-level node on P is not unique.) Let $P_1 = \langle v_1, v_2, \dots, v_j \rangle$ and $P_2 = \langle v_j, v_{j+1}, \dots, v_k \rangle$. In the following, we will show that the node hierarchy H contains a path P'_1 from v_1 to v_j that has the same length with P_1 . Furthermore, we will prove that the sequence of nodes on P'_1 satisfies both the level and proximity constraints, *i.e.*, P'_1 can be identified by FC with a traversal starting from v_1 . In a similar manner, it can be shown that H contains a path P'_2 from v_j to v_k , such that $l(P'_2) = l(P_2)$, and that

P'_2 can be found by FC with a constrained Dijkstra search starting from v_k . This would lead to

$$l(P') \leq l(P'_1) + l(P'_2) = l(P_1) + l(P_2) = l(P).$$

Consider the path $P_1 = \langle v_1, v_2, \dots, v_j \rangle$. Suppose that we remove from P_1 any node v_i ($i \in [1, j]$) that has a smaller level than some node v_a ($a < i$) preceding it. Let $S = \langle v'_1, v'_2, \dots, v'_b \rangle$ be the sequence of nodes remaining on P_1 . We have $v'_1 = v_1$ (since no node precedes v_1), and $v'_b = v_j$ (since v_j is the highest-level node in P). For instance, if P_1 contains six nodes $v_1, v_2, v_3, v_4, v_5, v_6$ at levels 1, 0, 2, 2, 1, 3, respectively, and then $S = \langle v_1, v_3, v_4, v_6 \rangle$.

By the way that S is constructed, for any v'_i ($i \in [1, b-1]$), the shortest path from v'_i to v'_{i+1} contains only nodes whose levels are smaller than those of v'_i and v'_{i+1} . As such, the node hierarchy H would contain a shortcut from v'_i to v'_{i+1} , and the length of the shortcut equals $\text{dist}(v'_i, v'_{i+1})$. This indicates that H contains a path $P'_1 = \langle v'_1, v'_2, \dots, v'_b \rangle$ that connects v_1 to v_k , such that $l(P'_1) = l(P)$. Furthermore, P'_1 satisfies the level constraint, since the level of v'_i ($i \in [1, b-1]$) is no larger than that of v'_{i+1} .

Assume to the contrary that P'_1 does not satisfy the proximity constraint. Then, there should exist a node v'_a ($a \in [1, b-1]$) on P'_1 , such that (i) v'_a is at level i ($i \in [0, h]$), but (ii) no (3×3) -cell region in R_{i+1} covers both v'_1 and v'_{a+1} . Then, by Lemma 3.2, the shortest path from v'_1 to v'_a must contain an arterial edge e of a (4×4) -cell region in R_{i+1} , since none of the (3×3) -cell regions in R_{i+1} covers both v'_1 and v'_a . In that case, each endpoint of e has a level at least $i + 1$. In other words, on the shortest path from v'_1 to v'_a , there exists some node whose level is higher than that of v'_a (recall that v'_a is at level i). This contradicts the assumption that v'_a has a level no lower than any node preceding it on P_1 .

In summary, the node hierarchy H contains a path P'_1 from v_1 to v_j , such that P'_1 has the same length with P_1 and satisfies both the level and proximity constraints. Therefore, FC can correctly identify the distance from v_1 to v_j with a traversal starting from v_1 . Similarly, we can show that FC can correctly compute the distance from v_j to v_k with a traversal starting from v_k . This proves the correctness of the query processing algorithm of FC.

3.4 Arterial Hierarchy

This section presents *Arterial Hierarchy (AH)*, a scalable indexing method built upon the FC approach introduced in Section 3.3. Compared with FC, AH has the same space complexity, a similar time complexity for distance queries, but significantly smaller pre-computation cost. In addition, AH also supports shortest path queries in a worst-case efficient manner.

3.4.1 Overview

The main structure of AH is a node hierarchy H^* that resembles FC's node hierarchy H . In particular, both H^* and H have $h + 1$ levels, and both of their i -th levels ($i \in [1, h]$) are associated with a square grid R_i of $2^{h+2-i} \times 2^{h+2-i}$ cells. However, AH and FC differ substantially in the ways that they decide node levels, construct shortcuts, and process queries.

Differences in Node Levels. To compute the level of each node, FC first imposes each R_i on the road network G , and then computes the arterial edges in each (4×4) -cell region in R_i , after which FC decides the node levels based on the arterial edges. As discussed in Section 3.3.3, the derivation of arterial edges could incur significant overheads, since each (4×4) -cell region in a coarse grid may cover a large number of nodes and edges in G .

In contrast, AH computes node levels with an incremental algorithm that substantially improves efficiency. Given G , it first imposes the grid R_1 on G . Based on R_1 , it identifies a set of unimportant nodes in G , and it assigns them to level 0 of the node hierarchy H^* . Then, it removes a subset of the unimportant nodes from G , and constructs shortcuts between the remaining nodes. This results in a *reduced* graph G_1 that is considerably smaller than G . After that, AH recursively reduces G_1 into smaller graphs $G_2, G_3, \dots, G_h$, during which it assigns nodes to higher levels of H^* . For the reduction from G_i to G_{i+1} , AH needs to impose the grid R_{i+1} on G_i and compute the shortest paths in each (4×4) -cell region. However, this computation is inexpensive since (i) G_i has a much smaller size than G , and hence, (ii) each (4×4) -cell region in R_{i+1} contains only a small number of nodes and edges in G_i .

Differences in Shortcuts. FC creates only necessary shortcuts to ensure correct results for distance queries under the level and proximity constraints. In contrast, the shortcuts constructed by AH are not only for processing distance queries under the level and proximity constraints, but also for computing the actual shortest path between any two given nodes. Specifically, every shortcut $\langle v_a, v_c \rangle$ in AH's node hierarchy H^* is associated with a node v_b , such that (i) both $\langle v_a, v_b \rangle$ and $\langle v_b, v_c \rangle$ are edges in H^* , and (ii) the length of $\langle v_a, v_c \rangle$ equals the lengths of $\langle v_a, v_b \rangle$ and $\langle v_b, v_c \rangle$ combined. In other words, $\langle v_a, v_c \rangle$ can be transformed into a two-hop shortest path $\langle v_a, v_b, v_c \rangle$. As such, given any path P' in H^* , we can transform P' into a path in G , by recursively replacing each shortcut in P' with its corresponding two-hop path.

For example, Figure 3.6 illustrates a shortest path $\langle v_1, v_2, \dots, v_6 \rangle$ in G , as well as three shortcuts $\langle v_1, v_4 \rangle$, $\langle v_2, v_4 \rangle$, and $\langle v_4, v_6 \rangle$. The shortcut $\langle v_1, v_4 \rangle$ is associated with the node v_2 , since v_1 is directly connected with v_2 and v_2 is directly connected with v_4 . Similarly, $\langle v_2, v_4 \rangle$ and $\langle v_4, v_6 \rangle$ are associated with v_3 and v_5 , respectively. Now suppose

that, given a distance query from v_1 to v_6 , AH identifies $P' = \langle v_1, v_4, v_6 \rangle$ as the shortest path from v_1 to v_6 in H^* . To derive the actual shortest path from v_1 to v_6 in G , AH first replaces the shortcut $\langle v_1, v_4 \rangle$ in P' with a two-hop path $\langle v_1, v_2, v_4 \rangle$, since $\langle v_1, v_4 \rangle$ is associated with v_2 . This transforms P' into another path $\langle v_1, v_2, v_4, v_6 \rangle$. After that, we can replace $\langle v_2, v_4 \rangle$ with $\langle v_2, v_3, v_4 \rangle$, and substitute $\langle v_4, v_6 \rangle$ with $\langle v_4, v_5, v_6 \rangle$. As such, we obtain the shortest path $\langle v_1, v_2, \dots, v_6 \rangle$ from v_1 to v_6 in G .

In general, given any shortest path query from a node s to another node t , AH first computes the shortest path P' from s to t in H^* , and then it converts P' into the corresponding path P in the original road network. The conversion from P' to P takes only $O(k)$ time, where k is the number of edges in P . This is because (i) for any shortcut in H^* , we can identify its corresponding two-hop path in $O(1)$ time, and (ii) converting P' to P requires only $O(k)$ replacements of shortcuts.

Differences in Query Processing. Besides the aforementioned shortcuts (for reconstructing shortest paths), the node hierarchy H^* of AH also contains some extra shortcuts that can be leveraged for higher query efficiency. As a consequence, AH's query processing algorithm is slightly more sophisticated than FC's, as will be elaborated in Section 3.4.3.

3.4.2 Index Construction

Similar to the case of FC, AH constructs its node hierarchy H^* in two steps: it first assigns each node in G to a level in H^* , and then it constructs shortcuts in H^* for query processing.

Deciding Node Levels. Given the road network G , AH first imposes on G the grid R_1 , where each cell contains at most one node. After that, AH identifies all (4×4) -cell regions in R_1 that cover at least one node in G . For each of the (4×4) -cell region identified, AH computes the arterial edges of the region in $O(1)$ time, and it marks each endpoint of an arterial edge as a *level-1 core*. At the same time, AH assigns all unmarked nodes to level 0 of the node hierarchy H^* since, intuitively, those nodes are less important than the level-1 cores. After that, if any (4×4) -cell region B contains a local shortest path P from a level-1 core u to another level-1 core v , such that P only goes through unmarked nodes, and then AH inserts into G a shortcut $\langle u, v \rangle$ with the same length as P . We say that $\langle u, v \rangle$ is a shortcut generated from B , and we use G_1 to denote the modified version of G with all shortcuts added. Overall, the computation of level-1 cores and the construction of G_1 take only $O(n)$ time, since the number of non-empty (4×4) -cell regions in R_1 is $O(n)$, and each of those regions contains $O(1)$ nodes and edges in G (recall that G is degree-bounded).

For example, given the road network G and the grid R_1 in Figure 3.4, assume that AH identifies 5 level-1 cores: $v_7, v_8, v_9, v_{10}, v_{11}$. After adding shortcuts, G is transformed

into the graph G_1 in Figure 3.7. There exists a shortcut $\langle v_9, v_{10} \rangle$ in G_0 since (i) both v_9 and v_{10} are level-1 cores, and (ii) in the (4×4) -cell region B illustrated in Figure 3.4, the local shortest path between v_9 and v_{10} goes only through v_6 , which is unmarked. In general, the shortcuts in G_1 ensure that the level-1 cores form a connected graph even if we remove all unmarked nodes from G_0 .

Given G_1 , AH selects a subset of the level-1 cores in G_1 that are deemed more important than the others. The selected nodes are marked as the *level-2 cores*, while the remaining level-1 cores are assigned to level 1 of H^* . After that, AH converts G_1 into a smaller graph G_2 that retains all level-2 cores. This procedure is applied in a recursive manner: In the i -th recursion ($i \in [1, h - 1]$), AH picks level- $(i+1)$ cores from the level- i cores in G_i , and then assigns the un-picked ones to level i of H^* , after which it transforms G_i into a smaller graph G_{i+1} .

A natural question is: Given G_i , how should AH select the level- $(i+1)$ cores from the level- i cores? One straightforward solution is to construct a subgraph of G_i that contains only the level- i cores, and then compute the arterial edges in the subgraph to identify the more important nodes as level- $(i+1)$ cores. For example, given G_1 in Figure 3.7, we can first construct a subgraph of G_1 that contains only the five level-1 cores (*i.e.*, $v_7, v_8, v_9, v_{10}, v_{11}$) and the edges connecting them (*i.e.*, the five edges on the loop $\langle v_7, v_8, v_{10}, v_9, v_{11}, v_7 \rangle$). After that, we impose the grid R_2 on the subgraph, compute the arterial edges, and then mark the endpoints of the arterial edges as level-2 cores. While this approach is intuitive, we find that (i) the resulting node hierarchy does not guarantee query correctness under the level and proximity constraints, and (ii) without the level and proximity constraints, it is difficult to achieve favorable asymptotic bounds on query time. To address this issue, we adopt a more careful approach to choose the level- $(i+1)$ cores without affecting the applicability of the level and proximity constraints in query processing. Specifically, our approach utilizes the concept of *border nodes*:

Definition 3.2 (Border Nodes). *Let B be a (4×4) -cell region in R_i ($i \in [1, h]$). A node v in G is a **border node** of B , if (i) v is not contained in the 2×2 cells centered at B , and (ii) v is an endpoint of an edge in G that intersects the boundary of the east, west, south, or north strip of B .*

For example, in Figure 3.4, v_1, v_2, v_9, v_{11} are all border nodes of the (4×4) -cell region B , since each of them is an endpoint of an edge that intersects the boundary of B 's west strip, and none of them is contained in the 2×2 cells centered at B . Similarly, v_3, v_4, v_7, v_8 are also border nodes of B . On the other hand, v_6 and v_{10} are not border nodes of B , since they are not adjacent to any edge that intersects the boundaries of B 's four strips.

To select level- $(i+1)$ cores from G_i , we first reduce G_i by removing any node in G_i that is neither a level- i core nor a border node of any (4×4) -cell region in R_{i+1} . We use

 Figure 3.7: Graph G_1 .

 Figure 3.8: Reduce graph G'_1 .

G'_i to denote the reduced graph thus obtained. For instance, given G_1 in Figure 3.7, we would remove v_5 and v_6 , since none of them is a level-1 core or a border node in R_2 . Figure 3.8 illustrates the reduced graph G'_1 , with the border nodes in R_2 highlighted.

Given G'_i , we impose R_{i+1} on G'_i and inspect each (4×4) -cell region in R_{i+1} that contains at least one node. For each such region B , we compute every spanning path of B (see Definition 3.1) that satisfies two conditions:

- (i) *Border Condition*: The two endpoints of the path are border nodes of B , while the other nodes are all level- i cores.
- (ii) *Coverage Condition*: Every shortcut on the path is generated from a region completely covered by B .

For example, in the (4×4) -cell region in Figure 3.8, the spanning path $\langle v_2, v_9, v_{10}, v_8, v_3 \rangle$ satisfies both the border and coverage conditions, since (i) both v_2 and v_3 are border nodes, and (ii) the only shortcut on the path, $\langle v_9, v_{10} \rangle$, is generated from the region B in Figure 3.4, which is contained in the current (4×4) -cell region.

For each spanning path P that fulfills the border and coverage conditions, if it connects the west and east (resp. north and south) strips of B , we identify the edge² in P that intersects B 's vertical bisector (resp. horizontal bisector) as a *pseudo-arterial edge* of B . Observe that each pseudo-arterial edge of B corresponds to a path in G that contains an arterial edge of B . Intuitively, this indicates the importance of pseudo-arterial edges in the reduced graph G'_i . Accordingly, we mark the two endpoints of every pseudo-arterial edge as level- $(i+1)$ cores, and we assign all unmarked level- i cores to the i -th level of the node hierarchy H^* . After that, for any local shortest path in a (4×4) -cell region B' , if (i) the two endpoints of the path are either level- $(i+1)$ cores or border nodes of B' , and (ii) other than its endpoints, the path does not go through any level- $(i+1)$ core, and then we insert into G'_i a shortcut between u and v with the same length as the local shortest

²If multiple edges or shortcuts in P intersect the bisector, we choose an arbitrary one among them as the pseudo-arterial edge.

path. Once all such shortcuts are added, we define the resulting graph as G'_{i+1} , and use it to recursively compute higher-level nodes in H^* .

It remains to show that we can efficiently derive the pseudo-arterial edges and construct shortcuts in G'_i . Let B be a (4×4) -cell region in R_{i+1} , and u be a border node of B . Suppose that we invoke Dijkstra's algorithm to start a traversal of G'_i from u ; for each node visited, we follow the outgoing edges of the node, ignoring any edge that violates the border condition or coverage condition. Once the traversal terminates, we can obtain the spanning paths of B starting from u , as well as the pseudo-arterial edges on those edges. Similarly, with a traversal from u that follows only the incoming edges of each node, we can compute the desired spanning paths of B ending at u , along with the pseudo-arterial edges therein. By repeating this process on all border nodes of B , we can derive the set of all pseudo-arterial edges in B . With the same traversal algorithm, we can construct all shortcuts in B using two traversals from each border node of B .

Creation of Shortcuts. After the level of each node is decided, AH adds shortcuts in the node hierarchy H^* to facilitate query processing. The construction of shortcuts requires as input a strict total order on the nodes in the same level of H^* . We will elaborate our ordering approach in Section 3.4.4, but in general, any strict total order can be used without affecting the space and time complexities of AH. For our discussion that follows, it suffices to know that less important nodes tend to precede more important nodes in our strict total order. For convenience, we define a rank for each node in G , such that a node u ranks lower than another node v , if (i) v is at a higher level than u , or (ii) u and v have the same level, but u precedes v in the strict total order.

AH constructs shortcuts in H^* in an incremental manner similar to the algorithm for deciding node levels. In particular, it first inspects G , and inserts into H^* a set of shortcuts that concern level-0 nodes. After that, it reduces G to a smaller graph G_1^* . Subsequently, it recursively reduces G_i^* into another graph G_{i+1}^* ($i \in [1, h-1]$), during which it constructs shortcuts that concern nodes at the i -th level of H^* . In the following, we will elaborate the reduction from G_i^* to G_{i+1}^* ($i \in [0, h-1]$), assuming $G_0^* = G$. For convenience, we define the *level* of every edge in G_0^* as -1 .

Given G_i^* , AH first imposes the grid R_{i+1} on G_i^* . For each node $u \in G_i^*$, AH inspects the (5×5) -cell region C centered at u , as well as the subgraph of G_i^* that consists of any level- $(i-1)$ edge overlapping with C . Then, AH computes two *shortest path trees* (SPT) of the subgraph, as defined in the following:

Definition 3.3 (Shortest Path Trees (SPT)). *Let G be a graph, and T be a directed spanning tree of G rooted at a node u . T is a **forward SPT** of G , if T contains the shortest path from u to any node in G . On the other hand, if T contains the shortest path from any node in G to u , and then T is a **backward SPT** of G .*

Let T_f (resp. T_b) be the forward (resp. backward) SPT of the aforementioned subgraph that is rooted at u . Observe that T_f (resp. T_b) can be computed by one traversal of the subgraph using Dijkstra's algorithm. Let v be any node in T_f , such that u ranks lower than v but higher than any ancestor of v in T_f . For any such v , AH generates a shortcut $\langle u, v \rangle$ with a length equal to the distance from u to v in T_f . In addition, AH associates $\langle u, v \rangle$ with a node w on the path from u to v , such that w ranks higher than any node on the path except u and v . This is to indicate that, when answering shortest path queries, AH can replace $\langle u, v \rangle$ with a two-hop path $\langle u, w, v \rangle$. (Our algorithm guarantees that such a two-hop path always exists.) Similarly, for any node v in T_b , AH creates a shortcut $\langle v, u \rangle$, if u 's rank is lower than v 's but higher than those of v 's ancestors in T_b . Furthermore, the shortcut is associated with the node w' that ranks the highest among v 's ancestors except u . We refer to the shortcuts constructed above as *level- i edges*³. Intuitively, these shortcuts connect each level- i node u directly to its nearby higher-rank nodes. By following these shortcuts during query processing, AH can avoid visiting less important nodes, which helps improve efficiency.

Besides the level- i edges, AH creates a shortcut from u and to a node v in T_f if (i) u and v are both at level i or above, and (ii) all ancestors of v except u are below level i . Likewise, if T_b contains a node v with a level at least i , such that u is the only ancestor of v at level i or higher, and then AH generates a shortcut from v to u . These shortcuts are to ensure that G_i^* would remain connected when we reduce G_i^* by removing some nodes below level i , as will be clarified shortly.

When $i > 0$ (i.e., G_i^* is produced from a previous reduction step), AH also generates some extra shortcuts (referred to as *elevating edges*), in a manner slightly different from the construction of level- i edges. First, AH inspects each node u in G_i^* at a level lower than i , and it examines the (5×5) -cell region C in R_i that is centered at u . Then, AH constructs a subgraph of G_i^* that comprises of all level- i edges covered by C , *as well as all edges that connect u with any node at level i or above*. After that, AH computes the subgraph's forward and backward SPTs rooted at u . Let P_f be any path in the forward SPT that connects u to a node outside of C , and let v be the first node on P_f at level i or above (our algorithm ensures that such v always exists). AH constructs a shortcut from $\langle u, v \rangle$, and associates it with the node that immediately follows u on P_f , *if u is below level $i - 1$* . On the other hand, if u is at level $i - 1$, and then the shortcut is associated with the first node on P_f that ranks higher than u . This shortcut is constructed to enable AH to efficiently traverse from u to the i -th level of H^* . Similarly, if the backward SPT contains a path P_b that links u with a node located beyond C , AH creates a shortcut $\langle v, u \rangle$, where v is the node closet to u on P_b among those at level i or above. If u is below level $i - 1$, the shortcut is associated with the node that immediately precedes u

³If multiple shortcuts are constructed from one node u to another node v , AH retains only the shortest one.

on P_b ; otherwise, it is associated with the node that is closest to u on P_b among those with higher ranks than u .

Once all level- i edges and elevating edges are created, they are inserted into both G_i^* and H^* . After that, AH reduces G_i^* by retaining only (i) the border nodes in R_{i+2} and (ii) nodes at level i or above. The resulting graph is defined as G_{i+1}^* and is fed into the next reduction step.

3.4.3 Query Processing

The query processing algorithm of AH is similar to that of FC. In particular, for distance query from a node s to another node t , AH also answers the query with two traversals of the node hierarchy H^* starting from s and t , respectively. As with the case of FC, each traversal of AH is performed with a constrained version of Dijkstra's algorithm. However, the constraints adopted by AH are slightly different: It adopts the proximity constraint (see Section 3.3.2) and a *rank constraint* as follows:

- *Rank Constraint*: When the traversal from s (resp. t) visits a node u , it ignores any neighbor of u that ranks lower than u .

Intuitively, the rank constraint is a refined version of the level constraint, in that it takes into account not only the levels of nodes but also the strict total order defined on each level of H^* . It leads to higher query efficiency as it helps AH bypass a larger number of relative unimportant nodes during query processing.

In addition, AH also exploits the elevating edges in H^* (see Section 3.4.2) to reduce query cost, based on the following lemma:

Lemma 3.3. *For any two nodes $u, v \in G$, if no (3×3) -cell region in R_i ($i \in [1, h]$) can cover u and v simultaneously, and then the shortest path from u to v must go through a node at level i or above.*

Let R_j ($j \in [1, h]$) be the coarsest grid where no (3×3) -cell region contains both s and t . By Lemma 3.3, the shortest path from s to t should pass through at least one node with a level at least j . This indicates that AH's traversal from s would meet its traversal from t at level j or above. Therefore, if s is a border node in R_j (in which case s has elevating edges to level j), and then when we start the traversal from s , we can follow the elevating edges of s to move directly to level j , ignoring any edge that connects s to a node at a level lower than j . After that, we can continue the traversal from level j under the rank and proximity constraints.

More generally, for any level- i ($i < j$) node v visited in the traversal from s , if v is a border node in R_j , and then we move along the elevating edges of v to level j or above, omitting any other edges of v . On the other hand, if v is a border node in $R_{j'}$ instead

of R_j ($j' < j$), and then we follow the elevating edges v to level j' or higher, *i.e.*, we traverse as close to level j as possible. Meanwhile, if v does not have any elevating edges or v is at a level at least j , and then we traverse the edges of v that satisfies the rank and proximity constraints. The same strategy is used when AH traverses from t . This traversal strategy reduces query time, since it enables AH to avoid visiting the low levels of node hierarchy H^* .

So far we have only discussed distance queries. For any shortest path query from s to t , AH first treats it as a distance query and computes the shortest path P' from s to t in H^* . After that, AH recursively replaces each shortcut in P' with its corresponding two-hop path, which converts P' to the actual shortest path from s to t in G , as explained in Section 3.4.1.

3.4.4 Node Ranking and Selection

As mentioned, the shortcut construction algorithm of AH assumes that there is a strict total order on the nodes in the same level. While any strict total order can be used without affecting the asymptotic bounds of AH, we have found a heuristic ordering approach that leads to high practical performance. Specifically, for nodes in the 0-th level of the node hierarchy H^* , we adopt a random order; for nodes in the i -th level ($i \in [1, h]$) of H^* , we derive their ordering based on information from the preprocessing procedure of AH. To explain, recall that AH decides node levels by recursively applying a reduction procedure on the road network G . During the i -th reduction step ($i \in [1, h - 1]$), AH examines a graph that contains level- $(i-1)$ cores; It identifies a set S_i of pseudo-arterial edges in the graph, marks the endpoints of those edges as level- i cores, and then assigns all unmarked level- $(i-1)$ cores to the $(i-1)$ -th level of H^* .

We observe that the edges in S_i are connected to some extent, and there are some level- i cores that serve as hub nodes for the connections (*i.e.*, they are adjacent to a sizable number of edges in S_i). Intuitively, those hub nodes are more important than the rest of the level- i cores. Motivated by this, we order the level- i cores using a *vertex cover* approach: we inspect the graph formed by the edges in S_i , and we compute a vertex cover of the graph using the linear-time $O(\log n)$ -approximation algorithm [27]. The output of the algorithm is a sequence ξ of nodes in the graph, such that the i -th node v in ξ is adjacent to the largest number of edges that are disjoint from the first $i - 1$ nodes. Based on ξ , we order the level- i cores as follows: The i -th node in ξ is given the i -th highest rank, and the level- i cores not in ξ are given the lowest ranks arbitrarily.

Interestingly, we find that if a level- i core does not appear in ξ , and then we can downgrade it to a level- $(i-1)$ core without affecting correctness or asymptotic performance of AH. Such downgrading reduces the number of high-level nodes in the node hierarchy H^* , which in turn improves query efficiency, since the high levels of H^* are frequently traversed during query processing. Our implementation of AH adopts this downgrading approach to improve query performance.

Table 3.1: Asymptotic performance of the state of the art.

Reference	Space	Preprocessing	PPD Query	PPSP Query
[67]*	$O(n)$	$O(n \log n)$	$O(n^{0.5+\epsilon})$	$O(k + n^{0.5+\epsilon})$
	$O(S)$	$\tilde{O}(S)$	$\tilde{O}(n/\sqrt{S})$	$\tilde{O}(k + n/\sqrt{S})$
[9]	$O(n \log n \log D)$	$O(n^2 \log n)$	$O(\log^2 n \log^2 D)$	$O(k + \log^2 n \log^2 D)$
	$O(n \log n \log D)$	$O(n^2 \log n)$	$O(\log n \log D)$	N/A
[74]	$O(n\sqrt{n})$	$O(n^2 \log n)$	$O(k \log n)$	$O(k \log n)$
AH	$O(hn)$	$O(hn^2)$	$O(h \log h)$	$O(k + h \log h)$

* [67] assumes the input graphs are planar graphs.

3.4.5 Space and Time Complexities

To establish the space and time complexities of AH, we first introduce a lemma that quantifies the densities of nodes in each level of AH's node hierarchy H^* :

Lemma 3.4. *Any $(\alpha \times \alpha)$ -cell region in R_i contains $O(\alpha^2 \lambda^2)$ nodes whose level in H^* are no lower than i , where λ is the arterial dimension of G .*

Our proof of Lemma 3.4 is similar to that of Lemma 3.1. We first show that any $(\alpha \times \alpha)$ -cell region B in R_i contains the endpoints of $O(\alpha^2 \lambda)$ arterial edges in G . After that, we prove that there exists a one-to-many mapping from the arterial edges in G to the nodes in B with levels at least i , such that each edge is mapped to $O(\lambda)$ nodes. Based on this, we show that any $(\alpha \times \alpha)$ -cell region in R_i contains only $O(\alpha^2 \lambda^2)$ nodes at level i or above.

Space Overhead. Given Lemma 3.4, we can prove that each node in H^* has $O(h\lambda^2)$ elevating edges and $O(\lambda^2)$ non-elevating edges. This is because, by the preprocessing algorithm of AH, there is an elevating edge from a node u to a level- i node v , only if u and v are contained in the same (4×4) -cell region in R_i . By Lemma 3.4, there exist $O(\lambda^2)$ such level- i nodes. Since H^* contains only $h + 1$ levels, the total number of elevating edges adjacent to u is $O(h\lambda^2)$. Similarly, we can prove that each node in H^* has $O(\lambda^2)$ non-elevating edges. Therefore, the space overhead of AH is $O(hn\lambda^2)$, which reduces to $O(hn)$ when λ is a constant.

Query Time. AH answers any distance query with two traversals of H^* , starting from the source s and destination t of the query, respectively. Due to the proximity constraint, in the i -th level of H^* ($i \in [0, h]$), each traversal of AH only visits the nodes in a (5×5) -cell region in R_{i+1} . By Lemma 3.4, such a region only contains $O(\lambda^2)$ nodes at level i . Hence, the total number of nodes traversed by AH is $O(h\lambda^2)$. Furthermore, for each node v visited during a traversal, AH either follows the elevating edges of v to a certain level of H^* , or moves along the non-elevating edges of v that satisfy the rank and proximity

constraints. As previously discussed, v has $O(\lambda^2)$ elevating edges to each level of H^* , and has $O(\lambda^2)$ non-elevating edges. Therefore, the total number of edges visited by AH is $O(h\lambda^4)$. Since each traversal is performed using Dijkstra's algorithm, its overall time complexity is $O(h\lambda^2 \log(h\lambda^2) + h\lambda^4)$. Consequently, when λ is a constant, the time complexity of AH for a distance query is $O(h \log h)$.

To answer a shortest path query from s to t , AH first processes its corresponding distance query to retrieve the shortest path P' from s to t in H^* , and then it transforms P' into the actual shortest path P from s to t in G . The transformation from P' to P takes $O(k)$ time, where k is the number of edges in P . Therefore, AH requires $O(k + h \log h)$ time to answer a shortest path query.

Preprocessing Cost. The preprocessing algorithm of AH consists of three steps: (i) assigning nodes to each level of H^* , (ii) deriving the strict total order on nodes at the same level, and (iii) creating shortcuts in H^* . When assigning nodes to the i -th level of H^* ($i \in [0, h-1]$), AH inspects each non-empty (4×4) -cell region in R_{i+1} , and constructs a subgraph that consists of the level- i cores and border nodes in the region. For each node in the subgraph, AH needs to apply Dijkstra's algorithm to traverse the subgraph a constant number of times. Given Lemma 3.4 and the fact that each level- i core or border node (i) has $O(\lambda^2)$ edges and (ii) is contained in a constant number of (4×4) -cell region in R_{i+1} , it can be proved that AH requires $O(n^2 \lambda^2)$ time to assign nodes to level i of H^* . Meanwhile, AH takes only $O(n)$ time to derive the strict total order at level i of H^* , since the derivation is based on a linear time algorithm for vertex cover.

To construct shortcuts at the i -th level of H^* , AH needs to inspect a graph G_i^* reduced from G . For each node u in G_i^* , AH examines the (5×5) -cell region in R_{i+1} that is centered at u , and it creates shortcuts for u by traversing the nodes in the region at level i or above. Based on Lemma 3.4, it can be proved that the total cost of generating shortcuts for u is $O(\lambda^2)$. As such, the time required to create shortcuts at level i of H^* is $O(n\lambda^2)$.

Summing up the above analysis, we have the following theorem:

Theorem 3.1. *Given a road network with a constant arterial dimension, AH takes $O(hn^2)$ time to construct an index that requires $O(hn)$ space. With the index, AH answers any distance query in $(h \log h)$ time and any shortest path query in $O(k + h \log h)$ time, where k is the number of edges in the shortest path.*

Almost every method for distance and shortest path queries trades off between space overhead and query performance. In particular, one can pre-compute the pair-wise distance matrix with $O(n^2)$ space overhead, so that each distance query can be answered in $O(1)$ time. On the other hand, the Dijkstra's algorithm does not require any additional space consumption, but its query time complexity is $O(n^2)$. Neither solution is practical

on large road networks, since both the $O(n^2)$ space consumption and the $O(n^2)$ query time are prohibitive on large road networks with millions of nodes. Interestingly, AH achieves near optimal in both aspects. As we shown in Section 3.3.1, h is no more than 26 in real-world road networks. Hence, the $O(hn)$ space consumption of AH is in the same order of the road network size. This makes AH applicable in real-life applications, since even the largest existing road network can easily fit into the main memory. Meanwhile, the query complexity of AH is independent to n , which renders the consistent query performance of AH on large road networks. As will be shown in Section 3.5.2, the average query time only grows to 10 times longer, when the road network size becomes 500 times larger.

Table 3.1 compares the performance bounds of AH with several most recent algorithms. In the table, S is a user-defined parameter in the range of $[n \log \log n, n^2]$. $D = l_{max}/l_{min}$, where l_{max} (resp. l_{min}) is the largest (resp. smallest) road network distance between two nodes. k is the number of edges in the shortest path between the source and destination of the query. h is as defined in Section 3.3.1.

3.5 Experiments

This section experimentally compares our AH method with three techniques: (i) Dijkstra’s algorithm [32], (ii) *Spatially Induced Linkage Cognizance (SILC)* [74], one of the most advanced worst-case efficient indices for shortest path and distance queries, and (iii) *Contraction Hierarchies (CH)* [38], a heuristic approach that offers the highest overall efficiency in shortest path and distance queries while incurring minimal costs of space and pre-computation, as shown in a recent experimental study [93] of the state of the art. We implement AH and Dijkstra’s algorithm using C++, and we obtain the C++ implementations of SILC and CH from [1, 2]. All experiments are conducted on a 64-bit windows machine with an Intel Xeon 2.8GHz CPU and 32GB RAM.

3.5.1 Datasets and Queries

We use ten publicly available datasets [3], each of which corresponds to a part of the road network in the US. Table 4.1 shows the number of nodes and edges in the data. For each edge in the datasets, its weight quantifies the time required to traverse the road segment that is represented by the edge.

Following previous work [93], we generate ten sets of queries $Q_1, Q_2, \dots, Q_{10}$ on each dataset as follows. We first estimate the maximum network distance l_{max} between two nodes in the road network. After that, we insert 10000 pairs of nodes (s, t) into Q_i ($i \in [1, 10]$) as queries, such that the distance between s and t is in $[2^{i-11} \cdot l_{max}, 2^{i-10} \cdot l_{max})$. In other words, the network distance between any pair of nodes in Q_i is larger than that in Q_{i-1} .

Table 3.2: Dataset Characteristics

Name	Corresponding Region	Number of Nodes	Number of Edges
DE	Delaware	48,812	120,489
NH	New Hampshire	115,055	264,218
ME	Maine	187,315	422,998
CO	Colorado	435,666	1,057,066
FL	Florida	1,070,376	2,712,798
CA	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

3.5.2 Efficiency for Distance Queries

Our first set of experiments focus on distance queries. Figure 3.12a shows the average running time of each technique when answering the distance queries in Q_i ($i \in [1, 10]$) on the DE road network (which contains 48,812 nodes). Observe that AH consistently outperforms all competitors including CH, the state-of-the-art heuristic approach. In particular, on query sets Q_8 , Q_9 , and Q_{10} (where each query concerns two distant locations), AH's running time is lower than that of CH and SILC by more than 50%. CH performs slightly worse than SILC on $Q_1, Q_2, \dots, Q_6$, but it is evidently superior to SILC on Q_8, Q_9 , and Q_{10} . Dijkstra's algorithm incurs the highest computation overhead on all query sets.

Figure 3.12b shows the query processing time of each method on NH, which is about 2 times the size of DE. Again, AH is consistently more efficient than the other three techniques, especially on query sets Q_8 , Q_9 , and Q_{10} . CH suppresses SILC in most query sets, which contrasts the case on DE where CH only dominates SILC on Q_8 , Q_9 , and Q_{10} . This indicates that SILC does not scale as well as CH. Dijkstra's algorithm is still the least efficient one among the four techniques. Similar results are shown in Figures 3.12c and 3.12d.

Figures 3.12e - 3.12j show the running time of AH, CH, and Dijkstra's algorithm on the largest six datasets. (SILC is omitted since its preprocessing and space overheads on those these datasets are prohibitive, as will be shown in Section 3.5.4) The relative performance of AH, CH, and Dijkstra's algorithm remain the same as in Figures 3.12a - 3.12d, with AH (resp. Dijkstra's algorithm) being the most (resp. least) efficient method by far.

Figure 3.9a shows how the running time of AH, CH, SILC and Dijkstra's algorithm varies with n on Q_1 . The running time of Dijkstra's algorithm is $O(n \log n)$, and thus, it

 Figure 3.9: Efficiency of distance queries vs. n .

increases significantly with n . In contrast, the running time of AH increases quite slowly, as well as CH and SILC, because they all have sub-linear query time complexities (as shown in Section 3.4.5). In particular, the running time of AH only grows to 10 times longer, when the road network becomes 500 times larger.

Figure 3.9b - 3.9d show the running time of AH, CH, and SILC on Q_4 , Q_7 and Q_{10} respectively. (We omit their adjacent querysets, since similar results are derived from adjacent querysets. Dijkstra's algorithm is also omitted, since its running time is prohibitive on distant queries.) The increase of AH, CH, and SILC with n remain the same tendency as in Figure 3.9a. But the gap between AH and other methods becomes larger when the query distance increases. This indicates that the query performance of AH is more consistent than the state-of-the-arts over close and distant queries.

3.5.3 Efficiency for Shortest Path Queries

Figure 3.13 shows the average computation time of each technique when answering the shortest path queries in Q_i ($i \in [1, 10]$) on all ten datasets. Regardless of the dataset, AH significantly outperforms the other three techniques. SILC is superior to CH on DE, but the performance of the two methods becomes comparable on the larger datasets. Dijkstra's algorithm is the least efficient one in all cases.

The running time of AH is higher for shortest path queries than distance queries. This is because, when answering a shortest path query from a source s to a destination t , AH first (i) computes the distance from s to t , and then (ii) derives the shortest path based on the result of the distance query. As a consequence, any shortest path query incurs a strictly higher overhead than a distance query with the same source and destination. Similarly, CH also incurs a higher cost for shortest path queries than distance queries.

In contrast, the running time of SILC (resp. Dijkstra's algorithm) is identical in Figures 3.12 and 3.13. The reason is that, SILC (resp. Dijkstra's algorithm) answers any distance query by first deriving the shortest path P from the source to the destination,

 Figure 3.10: Efficiency of shortest path queries vs. n .

and then returning the length of P . Computing the length of P incurs only negligible overhead, which explains why the costs of distance queries are the same as that of the shortest path queries.

Similar to the distance queries, Figure 3.10 shows how the shortest path querying time of AH, CH, SILC and Dijkstra's algorithm varies with n on Q_1 , Q_4 , Q_7 , and Q_{10} respectively. The relative performance remains the same as Figure 3.9, and thus, AH is still the most consistent and efficient solution over close and distant shortest path queries.

3.5.4 Space and Preprocessing Costs

In the last sets of experiments, we evaluate the space and pre-computation overheads of AH, SILC, and CH. (We omit Dijkstra's algorithm as it does not require building an index on the road network.) Figure 3.11a illustrates the index space required by AH, SILC, and CH on each dataset. Although SILC is worst-case efficient, its space overhead is extremely high, and it increases super-linearly with the number of nodes n in the road network. In particular, for all datasets with more than 500,000 nodes, the index of SILC is more than 32GB in size, *i.e.*, it cannot fit in the main memory of our machine. For this reason, we omit SILC from the experimental on those datasets. Meanwhile, the space consumption of AH is moderate, and it increases linearly with n (refer to the blue line which represents a linear function of n). This is consistent with our analysis in Section 3.4.5 that AH incurs a linear space complexity. Lastly, CH is the most space-economic method: it requires no more than 2GB of space even for the largest dataset.

Figure 3.11b shows the time required by AH, SILC, and CH to construct indices on our datasets. Observe that SILC has a pre-computation cost super-linear to n , and it requires more than one week to preprocess even the relatively small dataset CO (which contains 435,666 nodes). In contrast, the preprocessing time of AH exhibits a linear increase with n (refer to the blue line which represents a linear function of n), even

 Figure 3.11: Space Overhead and Preprocessing Time vs. n .

though AH's index construction algorithm has an $O(hn^2)$ time complexity. Furthermore, the pre-computation cost of AH is fairly small: it only requires around three hours to preprocess the US road network with 23 million nodes. On the other hand, the pre-computation time of CH is minimum and is below 40 minutes for all datasets.

3.6 Summary

This chapter presents Arterial Hierarchy(AH), a worst-case efficient index structure for shortest path and distance queries on road networks. Under a practical assumption about the road network, AH offers superior query time complexities in both shortest path and distance queries, and its space and preprocessing time complexities are comparable to the best existing worst-case efficient methods. With extensive experiments on real datasets, we show that AH also provides excellent query efficiency in practice, and it even outperforms CH (*i.e.*, the state-of-the-art heuristic method) in terms of query time. Furthermore, the space consumption and preprocessing cost of AH are fairly small: It takes only around three hours to preprocess a continent-scale road network with 23 million nodes, and the resulting index structure is no more than 32GB in size.

Figure 3.12: Efficiency of distance queries vs. query set.

Figure 3.13: Efficiency of shortest path queries vs. query set.

Chapter 4

Single Source Shortest Path and Distance Queries

Given a graph G , a *single source distance (SSD)* query from a node $v \in G$ asks for the distance from v to any other node in G . Meanwhile, a *single source shortest path (SSSP)* query retrieves the shortest path from v to any other node. These two types of queries are fundamental building blocks for numerous graph algorithms [64], and they find important applications in graph analysis [23], especially in the computation of graph measures [11, 13, 33, 85]. For example, the estimation of *closeness* measures [33] on a graph G requires performing SSD queries from a large number of nodes in G , while the approximation of *betweenness* measures [13] requires executing numerous SSSP queries.

The classic solution for SSD and SSSP queries is Dijkstra’s algorithm [32]. Given an SSD or SSSP query from a node s , Dijkstra’s algorithm traverses the graph starting from s , such that the nodes in G are visited in ascending order of their distances from s . Once a node v is visited, the algorithm returns the distance from s to v based on the information maintained during the traversal; the shortest path from s to v can also be efficiently derived if required.

A plethora of techniques have been proposed to improve over Dijkstra’s algorithm for higher query efficiency (see [28, 84] for surveys). Although those techniques all require pre-processing the given graph (which incurs extra overhead compared with Dijkstra’s algorithm), the pre-computation pays off when the number of queries to be processed is large, as is often the case in graph analysis. Nevertheless, most of the existing techniques assume that the given graph fits in the main memory (for pre-computation and/or query processing), which renders them inapplicable for the massive disk-resident graphs commonly used in web and social applications. There are a few methods [59, 62, 63, 65, 66] that address this issue by incorporating Dijkstra’s algorithm with I/O-efficient data structures, but the performance of those methods are shown to be insufficient for practical applications [23]. The main reason is that, when Dijkstra’s algorithm traverses the graph,

the order in which it visits nodes can be drastically different from the order in which the nodes are arranged on the disk. This leads to a significant number of random disk accesses, which results in poor query performance.

In contrast to the aforementioned techniques, Cheng et al. [23] propose the first practically efficient index (named *VC-Index*) for the SSD query on disk-resident graphs. The basic idea of VC-Index is to pre-compute a number of *reduced* versions of the input graph G . Each reduced graph contains some relatively important nodes in G , as well as the distances between some pairs of those nodes. During query processing, VC-Index scans a selected subset of reduced graphs, and then derives query results based on the pre-computed distances. Compared with those methods based on Dijkstra’s algorithm [59, 62, 63, 65, 66], VC-Index is more efficient as it only performs sequential reads on disk-resident data.

However, all existing disk-based solutions for SSD and SSSP queries [59, 62, 63, 65, 66] require that the input graph is undirected, which renders them inapplicable for any application built upon directed graphs. This is rather restrictive as numerous important types of graphs (e.g., road networks, web graphs, social graphs) are directed in nature. Furthermore, even when the input graph is undirected, the query efficiency of the existing solutions is less than satisfactory. In particular, our experiments (in Section 5.7) show that VC-Index, albeit being the state of the art, requires tens of seconds to answer a single SSD query on a graph with less than 100 million edges, and needs more than two days to estimate the closeness measures on the same graph.

To address the deficiency of existing work, this chapter presents *Highways-on-Disk (HoD)*, a disk-resident index that supports both SSD and SSSP queries on directed and weighted graphs. The key idea of HoD is to augment the input graph with a set of auxiliary edges (referred to as *shortcuts* [75]), and exploit them during query processing to reduce I/O and computation costs. For example, Figure 4.1a illustrates a graph G , and Figure 4.1e shows an augmented graph G^* constructed from G . G^* contains three shortcuts: $\langle v_8, v_9 \rangle$, $\langle v_9, v_7 \rangle$, and $\langle v_9, v_{10} \rangle$. Each shortcut has the same length with the shortest path connecting the endpoints of the shortcut. For example, the length of $\langle v_8, v_9 \rangle$ equals 2, which is identical to the length of the shortest path from v_8 to v_9 . Intuitively, the shortcuts in G^* enable HoD to efficiently traverse from one node to another (in a manner similar to how highways facilitate traversal between distant locations). For instance, if we are to traverse from v_1 to v_{10} in G^* , we may follow the path $\langle v_1, v_9, v_{10} \rangle$, which consists of only three nodes; in contrast, a traversal from v_1 to v_{10} in G would require visiting five nodes: v_1, v_9, v_6, v_7 , and v_{10} .

In general, when HoD answers an SSD or SSSP query, it often traverses the augmented graph via shortcuts (instead of the original edges in G). We show that, with proper shortcut construction and index organization, the query algorithm of HoD always traverses nodes in the same order as they are arranged in the index file. Consequently, HoD can

answer any SSD or SSSP query with a linear scan of the index file, and its CPU cost is linear to the number of edges in the augmented graph. We experimentally evaluate HoD on a variety of real-world graphs with up to 100 million nodes and 3 billion edges, and we demonstrate that HoD significantly outperforms VC-Index in terms of query efficiency. In particular, the query time of HoD is smaller than that of VC-Index by up to two orders of magnitude. Furthermore, HoD requires a smaller space and pre-computation time than VC-Index in most cases.

The rest of this chapter is organized as follows. Section 4.1 briefly reviews the related work. Section 4.2 formally defines SSSP and SSD queries on graphs. Section 4.3 gives an overview on the preprocessing phase and querying phase of HoD. Section 4.4 demonstrates the index-construction method in detail. Section 4.5 presents the algorithms for the SSD query. Section 4.6 extends the algorithm for the SSSP query. Section 4.7 shows experimental results on real-life large graphs, including a Web graph with 3 billion edges. Section 4.8 summarizes this chapter.

4.1 Related Work

The existing techniques for I/O-efficient SSD and SSSP queries include VC-Index [23] and a few methods that adopt Dijkstra’s algorithm [59, 62, 63, 65, 66]. All of those techniques are exclusively designed for undirected graphs, and they incur significant query overheads, as is shown in our experiments. In contrast, HoD supports both directed and undirected graphs, and it offers high query efficiency along with small costs of pre-computation and space.

Other than the aforementioned work, there exists a large body of literature on in-memory algorithms for shortest path and distance queries (see [28, 84] for surveys). The majority of those algorithms focus on two types of queries: (i) *point-to-point shortest path (PPSP)* queries, which ask for the shortest path from one node to another, and (ii) *point-to-point distance (PPD)* queries, which ask for the length of the shortest path between two given nodes. These two types of queries are closely related to SSD and SSSP queries, in the sense that any SSD (resp. SSSP) query can be answered using the results of n PPD (resp. PPSP) queries, where n is the number of nodes in the graph. Therefore, it is possible to adopt a solution for PPD (resp. PPSP) queries to handle SSD (resp. SSSP) queries. Such an adoption, however, incurs significant overheads, especially when n is large. For example, the state-of-the-art solution [7] for PPD queries requires 266ns to answer a PPD query on the USRN dataset in Section 4.7. (Note: the solution is not I/O efficient and it requires 25.4GB memory to handle USRN.) If we use this solution to answer an SSD query on USRN, and then we need to execute 24.5 million PPD queries, which takes roughly $266\text{ns} \times 24.5 \times 10^6 = 6.52\text{s}$. In contrast, HoD requires only 1.8s to process an SSD query on USRN, using only 1GB memory.

Furthermore, almost all existing solutions for PPD and PPSP queries require that the dataset fits in the main memory during pre-computation and/or query processing. This renders them inapplicable for the massive disk-resident graphs considered in this chapter. The only exception that we are aware of is a concurrent work by Fu et al. [37], who propose an I/O-efficient method called *IS-Label*. HoD and IS-Label's preprocessing algorithms are similar in spirit, but their index structures and query algorithms are drastically different, as they are designed for different types of queries. In particular, IS-Label focuses on PPD and PPSP queries, and does not efficiently support SSD or SSSP queries.

Finally, we note that previous work [28, 38, 75] has exploited the idea of augmenting graphs with shortcuts to accelerate PPD and PPSP queries. Our adoption of shortcuts is inspired by previous work [28, 38, 75], but the adoption is highly non-trivial due to the facts that (i) we address I/O efficiency under memory-constrained environments, while previous work [28, 38, 75] focus on main memory algorithms; (ii) we tackle SSD and SSSP queries instead of PPD and PPSP queries; (iii) we consider general graphs, while previous work [28, 38, 75] only deals with road networks (which are largely planar and degree-bounded).

4.2 Problem Definition

Let G be a weighted and directed graph with a set V of nodes and a set E of edges. Each edge e in E is associated with a positive weight $l(e)$, which is referred to as the *length* of e . A path P in G is a sequence of nodes $\langle v_1, v_2, \dots, v_k \rangle$, such that $\langle v_i, v_{i+1} \rangle$ ($i \in [1, k-1]$) is a directed edge in G . The length of P is defined as the sum of the length of each edge on P . We use $l(e)$ and $l(P)$ to denote the length of an edge e and a path P , respectively.

For any two nodes s and t in G , we define the *distance* from s to t , denoted as $dist(s, t)$, as the length of the shortest path from s to t . Given a *source node* s in G , a *single-source distance (SSD)* query asks for the distance from s to any other node in G . Meanwhile, a *single-source shortest path (SSSP)* query from s retrieves not only the distance from s to any other node v , but also the *predecessor* of v , *i.e.*, the node that immediately precedes v in the shortest path from s to v . Note that, given the predecessor of each node, we can easily reconstruct the shortest path from s to any node v by backtracking from v following the predecessors. One may also consider an alternative formulation of SSD (resp. SSSP) query that, given only a *destination node* t , asks for the distance (resp. shortest path) from any other node to t . For simplicity, we will focus on SSD and SSSP queries from a source node s , but our solution can be easily extended to handle queries under the alternative formulation.

Let M be the size of the main memory available, and B be the size of a disk block, both measured in the number of words. We assume that $B \leq |V| \leq M \leq |E|$, *i.e.*,

Figure 4.1: Graph reduction and shortcut construction.

the main memory can accommodate all nodes but not all edges in G . This is a realistic assumption since modern machines (even the commodity ones) have gigabytes of main memory, which is sufficient to store the node set of a graph with up to a few billion nodes. On the other hand, the number of edges in a real graph is often over an order of magnitude larger than the number of nodes, due to which E can be enormous in size and does not fit in the main memory.

Our objective is to devise an index structure on G that answers any SSD or SSSP query with small I/O and CPU costs, such that the index requires at most M main memory in pre-computation and query processing. In what follows, we will first focus on SSD queries in Sections 4.3-4.5, and will extend our solution for SSSP queries in Section 4.6.

4.3 Solution Overview

The main structure of HoD is a graph G^* that augments the input graph G with shortcuts. In this section, we present the overall idea of how the shortcuts in G^* are constructed and how they can be utilized for query processing, so as to form a basis for the detailed discussions in subsequent sections.

4.3.1 Shortcut Construction

In a nut shell, HoD constructs shortcuts with an iterative procedure, which takes as input a copy of the graph G (denoted as G_0). In the i -th ($i \geq 1$) iteration of the procedure,

HoD first *reduces* G_{i-1} by removing a selected set of *less important* nodes in G_{i-1} , and then, it constructs shortcuts in the reduced graph to ensure that the distance between any two remaining nodes is not affected by the node removal. The resulting graph (with shortcuts added) is denoted as G_i , and it is fed as the input of the $(i+1)$ -th iteration of procedure. This procedure terminates only when the reduced graph G_i is *sufficiently small*. All shortcuts created during the procedure are inserted into the original graph G , leading to an augmented graph G^* that would be used by HoD for query processing. We illustrate the iterative procedure with an example as follows.

Example 4.1. Assume that the input to the iterative procedure is the graph G_0 in Figure 4.1a. Further assume that the reduced graph is sufficiently small if it contains at most two nodes and two edges. In the first iteration of the procedure, HoD inspects G_0 and identifies v_1 , v_2 , and v_3 as less important nodes. To explain, observe that the node v_1 in G_0 does not have any incoming edge, while v_2 and v_3 have no outgoing edges. As a consequence, v_1 , v_2 , and v_3 do not lie on the shortest path between any two other nodes. That is, even if we remove v_1 , v_2 , and v_3 from G_0 , the distance between any two remaining nodes is not affected. Intuitively, this indicates that v_1 , v_2 , v_3 are of little importance for SSD queries. Therefore, HoD eliminates v_1 , v_2 , and v_3 from G_0 , which results in the reduced graph G_1 in Figure 4.1b.

In the second iteration, HoD selects v_4 , v_5 , and v_6 as the less important nodes in G_1 , and removes them from G_1 . The removal of v_4 changes the distance from v_8 to v_9 to $+\infty$, since $\langle v_8, v_4, v_9 \rangle$ is the only path (in G_1) that starts at v_8 and ends at v_9 . To mitigate this change, HoD inserts into G_1 a shortcut $\langle v_8, v_9 \rangle$ that has the same length with $\langle v_8, v_4, v_9 \rangle$, as illustrated in Figure 4.1c. As such, the distance between any two nodes in G_1 remains unchanged after v_4 is removed. Similarly, when HoD eliminates v_6 , it constructs a shortcut $\langle v_9, v_7 \rangle$ with a length 2 to reconnect the two neighbors of v_6 . Meanwhile, v_5 is removed without creating any shortcut, since deleting v_5 does not change the distance between its two neighbors. Figure 4.1d illustrates the resulting reduced graph G_2 .

To explain why HoD chooses to remove v_4 , v_5 , and v_6 from G_1 , observe that each of those nodes has only two neighbors. For any of such nodes, even if the removal of the node changes the distance between its neighbors, HoD only needs to construct one shortcut to reconnect its neighbors. In other words, the number of shortcuts required is minimum, which helps reduce the space consumption of HoD. In contrast, if HoD chooses to remove v_9 from G_1 (which has a larger number of neighbors than v_4 , v_5 , and v_6), and then much more shortcuts would need to be constructed.

Finally, in the third iteration, HoD removes v_7 and v_8 from G_2 as they are considered unimportant. The removal of v_7 leads to a new shortcut $\langle v_9, v_{10} \rangle$ with a length 3, since $\langle v_9, v_7, v_{10} \rangle$ is the only path connecting v_9 to v_{10} , and the length of the path equals 3. On

the other hand, v_8 is directly eliminated as it is not on the shortest path between its only two neighbors v_9 and v_{10} . Figure 4.1e shows the reduced graph G_3 after the removal of v_7 and v_8 .

Assume that the reduced graph G_3 is considered sufficiently small by HoD. Then, the iterative procedure of HoD would terminate. The three shortcuts created during the procedure (*i.e.*, $\langle v_8, v_9 \rangle$, $\langle v_9, v_7 \rangle$, and $\langle v_9, v_{10} \rangle$) are added into the original graph G , which leads to the augmented graph G^* in Figure 4.1f. $\square$

The above discussion leaves several issues open, *i.e.*, (i) the specific criterion for identifying less important nodes in the reduced graph, (ii) the detailed algorithm for generating shortcuts after node removal, and (iii) the exact termination condition of the reduction procedure. We will clarify these issues in Section 4.4 by presenting the detailed preprocessing algorithm of HoD. For the discussions in the rest of this section, it suffices to know that when HoD terminates the reduction procedure, the reduced graph must fit in the main memory. We use G_c to denote this memory-resident reduced graph, and we refer to it as the *core graph*. (Note that G_c is a subgraph of the augmented graph G^* .) In addition, we define the *rank* $r(v)$ of each node v in G as follows:

- (i) If v is removed in the i -th iteration of the iterative procedure, and then $r(v) = i$;
- (ii) If v is not removed in any iteration (*i.e.*, v is retained in the core graph G_c), and then $r(v) = 1 + \max_{v \notin G_c} r(v)$, *i.e.*, $r(v)$ is larger than the maximum rank of any node not in G_c .

For instance, in Example 4.1, the ranks of v_1 , v_2 , and v_3 equal 1, since they are removed from G in the first iteration of the reduction procedure. Similarly, $r(v_4) = r(v_5) = r(v_6) = 2$, and $r(v_7) = r(v_8) = 3$. The ranks of v_9 and v_{10} equal 4, since they are in the core graph G_c . The ranks of the nodes are utilized in the query processing algorithms of HoD, as will be illustrated shortly. Unless otherwise specified, we use the term *edge* to refer to both a shortcut and an original edge in G^* .

4.3.2 Query Processing

Given an SSD query from a node s , HoD answers the query with two traversals of the augmented graph G^* . The first traversal starts from s , and it follows only the *outgoing* edges of each node, ignoring any edge *whose starting point ranks higher than the ending point*. For instance, if HoD traverses from the node v_9 in Figure 4.1f, it would ignore the outgoing edge $\langle v_9, v_7 \rangle$, since $r(v_9) = 4 > r(v_7) = 3$. As such, the first traversal of HoD never moves from a high-rank node to a low-rank node, and it terminates only when no higher-rank nodes can be reached. For each node v visited, HoD maintains the distance from s to v along the paths that have been seen during the traversal, denoted as $\text{dist}(s, v)$.

Let V' be the set of nodes that are not in the core graph of G^* . The second traversal of HoD is performed as a linear scan of the nodes in V' , *in descending order of their ranks*. For each node $v' \in V'$ scanned, HoD inspects each *incoming* edge e of v' , and then checks the starting point u of the edge. For any such u , HoD calculates $\text{dist}(s, u) + l(e)$ as an upperbound of the distance from s to v' . (Our solution guarantees that u should have been visited by HoD before v' .) Once all incoming edges of v' are inspected, HoD derives the distance from s to v' based on the upperbounds, and then it moves on to the next node in V' . This process terminates when all nodes in V' are examined.

We illustrate the above query algorithm of HoD with an example.

Example 4.2. Consider an SSD query from node v_1 in Figure 4.1a. Given the augmented graph G^* in Figure 4.1f, HoD first traverses G^* starting from v_1 , following the outgoing edges whose ending points rank higher than the starting points. Since v_1 has only one outgoing edge $\langle v_1, v_9 \rangle$, and since v_9 ranks higher than v_1 , HoD moves from v_1 to v_9 . v_9 has three outgoing edges: $\langle v_9, v_6 \rangle$, $\langle v_9, v_7 \rangle$, and $\langle v_9, v_{10} \rangle$. Among them, only $\langle v_9, v_{10} \rangle$ has an ending point that ranks higher than the starting point. Therefore, HoD moves from v_9 to v_{10} . v_{10} has outgoing edges to three unvisited nodes, *i.e.*, v_3 , v_5 , and v_8 . Nevertheless, all of those nodes rank lower than v_{10} , and hence, they are ignored. As none of the remaining nodes can be reached without violating the constraints on node ranks, the first traversal of HoD ends. Based on the edges visited, HoD calculates $\text{dist}(v_1, v_9) = 1$ and $\text{dist}(v_1, v_{10}) = 4$.

The second traversal of HoD examines the nodes *not* in the core graph in descending order of their ranks, *i.e.*, it first examines v_7 and v_8 (whose ranks equal 3), followed by v_4 , v_5 , and v_6 (whose ranks equal 2), and finally v_2 and v_3 (whose ranks equal 1), ignoring v_1 (as it is the source node of the query). v_7 has two incoming edges, $\langle v_6, v_7 \rangle$ and $\langle v_9, v_7 \rangle$. Among v_6 and v_9 , only v_9 has been visited by HoD. Therefore, HoD calculates $\text{dist}(v_1, v_7) = \text{dist}(v_1, v_9) + l(\langle v_9, v_7 \rangle) = 3$. Similarly, after inspecting v_8 's only incoming edge $\langle v_{10}, v_8 \rangle$, HoD computes $\text{dist}(v_1, v_8) = \text{dist}(v_1, v_{10}) + l(\langle v_{10}, v_8 \rangle) = 5$. The remaining nodes are processed in the same manner, resulting in

$$\begin{aligned} \text{dist}(v_1, v_4) &= \text{dist}(v_1, v_8) + l(\langle v_8, v_4 \rangle) &= 6 \\ \text{dist}(v_1, v_5) &= \text{dist}(v_1, v_{10}) + l(\langle v_{10}, v_5 \rangle) &= 5 \\ \text{dist}(v_1, v_6) &= \text{dist}(v_1, v_9) + l(\langle v_9, v_6 \rangle) &= 2 \\ \text{dist}(v_1, v_2) &= \text{dist}(v_1, v_4) + l(\langle v_4, v_2 \rangle) &= 7. \end{aligned}$$

Observe that all the above distances computed from G^* are identical with those from the original graph in Figure 4.1a. □

The query algorithm of HoD has an interesting property: the first traversal of the algorithm always visits nodes in ascending order of their ranks (as it never follows an edge that connects a high-rank node to low-rank node), while the second phase always visits nodes in descending rank order. Intuitively, if we maintain two copies of the augmented graph, such that the first (resp. second) copy stores nodes in ascending (resp. descending) order of their ranks, and then HoD can answer any SSD query with a linear scan of the two copies. This leads to high query efficiency as it avoids random disk accesses. In Section 4.4, we will elaborate how such two copies of the augmented graph can be constructed.

4.4 Index Construction

As discussed in Section 4.3.1, the preprocessing algorithm of HoD takes as input a copy G_0 of the graph G , and it iteratively reduces G_0 into smaller graphs $G_1, G_2, \dots$, during which it creates shortcuts to augment G . More specifically, the $(i+1)$ -th ($i \geq 0$) iteration of the algorithm has four steps:

- (i) Select a set R_i of nodes to be removed from G_i .
- (ii) For each node $v \in R_i$, construct shortcuts in G_i to ensure that the removal of v does not change the distance between any two remaining nodes.
- (iii) Remove the nodes in R_i from G_i to obtain a further reduced graph G_{i+1} . Store information about the removed nodes in the index file of HoD.
- (iv) Pass the G_{i+1} to the $(i+2)$ -th iteration as input.

In the following, we first elaborate Steps 2 and 3, and then clarify Step 1. After that, we will discuss the termination condition of the preprocessing algorithm, as well as its space and time complexities.

For ease of exposition, we represent each edge $e = \langle u, v \rangle$ as a triplet $\langle u, v, l(e) \rangle$ or $\langle v, u, -l(e) \rangle$, where $l(e)$ is the length of e . For example, the edge $\langle v_9, v_7 \rangle$ in Figure 4.1a can be represented as either $\langle v_9, v_7, 2 \rangle$ or $\langle v_7, v_9, -2 \rangle$. That is, a negative length in the triplet indicates that the second node in the triplet is the starting point of the edge. In addition, we assume that the input graph G is stored on the disk as adjacency lists, such that (i) for any two nodes v_i and v_j , the adjacency list of v_i precedes that of v_j if $i < j$, and (ii) each edge $\langle v_i, v_j \rangle$ with length l is stored twice: once in the adjacency list of v_i (as a triplet $\langle v_i, v_j, l \rangle$), and another in the adjacency list of v_j (as a triplet $\langle v_j, v_i, -l \rangle$).

4.4.1 Node Removal and Shortcut Generation

Let v^* be a node to be removed from G_i . We define an *outgoing neighbor* of v^* as a node u to which v^* has an outgoing edge. Similarly, an *incoming neighbor* of v^* is a node w from which v^* has an incoming edge. We have the following observation:

Observation 4.1. For any two nodes v_j and v_k in G_i , the distance from v_j to v_k changes after v^* is removed, if and only if the shortest path from v_j to v_k contains a sub-path $\langle u, v^*, w \rangle$, such that u (resp. w) is an incoming (resp. outgoing) neighbor of v^* . $\square$

By Observation 4.1, we can preserve the distance between any two nodes in G_i after removing v^* , as long as we ensure that the distance between any incoming neighbor and any outgoing neighbor of v^* remains unchanged. This can be achieved by connecting the incoming and outgoing neighbors of v^* with shortcuts, as demonstrated in Section 4.3.1. Towards this end, a straightforward approach is to generate a shortcut $\langle u, w \rangle$ for any incoming neighbor u and any outgoing neighbor w . The shortcuts thus generated, however, are often *redundant*. For example, consider the graph G_i in Figure 4.2a. Suppose that we are to remove v_2 , which has an incoming neighbor v_1 and an outgoing neighbor v_3 . If we construct a shortcut from v_1 to v_3 , it is useless since (i) v_1 already has an outgoing edge to v_3 , and (ii) the edge $\langle v_1, v_3 \rangle$ is even shorter than the path from v_1 to v_3 via v_2 . As another example, assume that v_4 in Figure 4.2a is also to be removed. v_4 has an incoming neighbor v_1 and an outgoing neighbor v_5 , but the path $\langle v_1, v_4, v_5 \rangle$ is no shorter than another path from v_1 to v_5 , *i.e.*, $\langle v_1, v_3, v_5 \rangle$, which does not go through v_4 . As a consequence, even if we remove v_4 from G_i , the distance from v_1 to v_5 is still retained, and hence, it is unnecessary to insert a shortcut from v_1 to v_5 .

In general, for any incoming neighbor u and outgoing neighbor w of v^* , a shortcut from u to w is unnecessary if there is a path P from u to w , such that (i) P does not go through v^* , and (ii) P is no longer than $\langle u, v^*, w \rangle$. To check whether such a path P exists, one may apply Dijkstra's algorithm to traverse G_i from u (or w), ignoring v^* during the traversal. However, when G_i does not fit in main memory (as is often the case in the pre-computation process of HoD), this approach incurs significant overhead, due to the inefficiency of Dijkstra's algorithm for disk-resident graphs. To address this issue, we adopt a heuristic approach that is not as effective (in avoiding redundant shortcuts) but much more efficient. Specifically, for each v^* in the node set R_i to be removed from G_i , we generate a *candidate edge* $e_c = \langle u, w \rangle$ from each incoming neighbor u of v^* to each outgoing neighbor w of v^* , setting the length of the shortcut to $l(\langle u, v^*, w \rangle)$. For any such candidate edge e_c , we insert it into a temporary file T as two triplets: $\langle u, w, l(e_c) \rangle$ and $\langle w, u, -l(e_c) \rangle$.

In addition to the candidate edges, we also insert two additional groups of edges (referred to as *baseline edges*) into the temporary file T as triplets. The first group

consists of any edge e in G_i connecting two nodes not in R_i , *i.e.*, the two endpoints of e are not to be removed. The second group is generated as follows: for each node v not in R_i , we select v 's certain incoming neighbor u' and outgoing neighbor w' , and we construct a baseline edge $\langle u', w' \rangle$, setting its length to $l(\langle u', v, w' \rangle)$.

The purpose of inserting a baseline edge e into the temporary file T is to help eliminate any redundant candidate edge that (i) shares the same endpoints with e but (ii) is not shorter than e . Towards this end, once all baseline edges are added into T , we sort the triplets in T using a standard algorithm for external sort, such that a triplet $t_1 = \langle v_a, v_b, l_1 \rangle$ precedes another triplet $t_2 = \langle v_a, v_b, l_2 \rangle$, if any of the following conditions hold:

- (i) $a < \alpha$, or $a = \alpha$ but $b < \beta$.
- (ii) $a = \alpha$, $b = \beta$, and $l_1 > 0 > l_2$. That is, any outgoing edge of a node precedes its incoming edges.
- (iii) $a = \alpha$, $b = \beta$, $l_1 \cdot l_2 > 0$ (*i.e.*, t_1 and t_2 are both incoming edges or both outgoing edges), and $|l_1| < |l_2|$. That is, t_1 and t_2 share the same starting and ending points, but t_1 is shorter than t_2 .
- (iv) $a = \alpha$, $b = \beta$, $l_1 \cdot l_2 > 0$, $|l_1| = |l_2|$, and t_1 is a baseline edge while t_2 is a candidate edge.

Once T is sorted, the outgoing (resp. incoming) edges with the same endpoints are grouped together, and the first edge in each group should have the smallest length within the group. If the first edge e in a group is a candidate edge, and then we retain e as it is shorter than any other baseline or candidate edges that we have generated. On the other hand, if e is a baseline edge, and then the distance between the endpoints of e must not be affected by the removal of any nodes in R_i . In that case, all candidate edges in the group can be omitted. With one linear scan of the sorted T and the adjacency lists of G_i , we can remove the information of any node in R_i , and merge the retained candidate edges into the adjacency lists of the remaining nodes. We illustrate the above algorithm with an example.

Example 4.3. Suppose that, given the graph G_i in Figure 4.2a, we are to remove a node set $R_i = \{v_2, v_4\}$ from G_i . v_2 has only one incoming neighbor v_1 and one outgoing neighbor v_3 , and $l(\langle v_1, v_2, v_3 \rangle) = 2$. Accordingly, HoD generates a candidate edge $\langle v_1, v_3 \rangle$ by inserting into the temporary file T two triplets, $\langle v_1, v_3, 2 \rangle$ and $\langle v_3, v_1, -2 \rangle$. Similarly, for v_4 , HoD creates a candidate edge $\langle v_1, v_5 \rangle$, which is represented as two triplets in T : $\langle v_1, v_5, 2 \rangle$ and $\langle v_5, v_1, -2 \rangle$.

Meanwhile, the edge $\langle v_1, v_3 \rangle$ in G_i is selected as a baseline edge and is inserted into T , since neither v_1 nor v_3 is in R_i . In addition, HoD also generates a baseline edge

$\langle v_1, v_5 \rangle$ from the neighbors of v_3 . This is because that (i) v_1, v_3, v_5 are not in R_i , (ii) v_1 is an incoming neighbor of v_3 , and (iii) v_5 is an outgoing neighbor of v_3 . Figure 4.2c illustrates the temporary file T after all candidate and baseline edges are inserted, with some triplets omitted for simplicity. Figure 4.2d shows the file T after it is sorted. The baseline edge $\langle v_1, v_3, 1 \rangle$ precedes the candidate edge $\langle v_1, v_3, 2 \rangle$, which indicates that we do not need to add a shortcut from v_1 to v_3 . Similarly, the baseline edge $\langle v_3, v_1, -1 \rangle$ precedes the candidate edge $\langle v_3, v_1, -2 \rangle$, in which case the latter is omitted. Overall, each of the candidate edges in T is preceded by a baseline edge, and hence, no shortcut will be created. Consequently, HoD removes from G_i the adjacency lists of v_2 and v_4 , as well as all edges to and from v_2 and v_4 in any other adjacency lists. This results in the reduced graph illustrated in Figure 4.2b. $\square$

In summary, HoD decides whether a candidate edge e should be retained, by comparing it with all edges in G_i as well as some two-hop paths in G_i . This heuristic approach may retain unnecessary candidate edges, but it does not affect the correctness of SSD queries. To understand this, recall that each candidate edge $e = \langle u, w \rangle$ has the same length with a certain path $\langle u, v^*, w \rangle$ that *exists* in G_i , where v^* is the node whose removal leads to the creation of e . In other words, the length of e is at least larger than or equal to the distance from u to w . Adding such an edge into G_i would not decrease the distance between any two nodes in G_i , and hence, retaining e does not change the results of any SSD queries.

The above discussions assume that HoD has selected a set R_i of nodes to be removed from G_i , and has decided which baseline edges are to be generated from the neighbors of the nodes not in R_i . We will clarify these two issues in Section 4.4.2 and 4.4.3.

4.4.2 Selecting Nodes for Removal

Consider any node v in G_i . Intuitively, if the removal of v requires us to insert a large number of shortcuts into G_i , and then v may lie on the shortest paths between many pair of nodes, in which case v should be considered important. Let B_{in} and B_{out} be the set of incoming and outgoing neighbors of v , respectively. The maximum number of shortcuts induced by v 's removal is:

$$s(v) = |B_{in}| \cdot |B_{out} \setminus B_{in}| + |B_{out}| \cdot |B_{in} \setminus B_{out}|. \quad (4.1)$$

We refer to $s(v)$ as the *score* of v in G_i , and we consider v *unimportant* if $s(v)$ is no more than the median score in G_i . (For practical efficiency, we use an approximated value of the median score computed from a sample set of the nodes.)

Ideally, we would like to remove all unimportant nodes from G_i , but this is not always feasible. To explain, consider that we are given the reduced graph G_1 in Figure 4.1b,

Figure 4.2: Node removal and shortcut generation.

and we aim to eliminate both v_6 and v_7 . v_6 has only one incoming neighbor v_9 and one outgoing neighbor v_7 , and hence, HoD creates one candidate edge $\langle v_9, v_7 \rangle$, setting its length to 2 (*i.e.*, the length of the path $\langle v_9, v_6, v_7 \rangle$). Similarly, for v_7 , HoD generates a candidate edge $\langle v_6, v_{10} \rangle$. These two candidate edges are intended to preserve the distance between any two nodes in G_i after v_6 and v_7 are removed. However, none of the two candidate edges is valid if both v_6 and v_7 are eliminated. In particular, $\langle v_9, v_7 \rangle$ points from v_9 to v_7 , *i.e.*, it connects v_9 to a node that no longer exists. To avoid the above error, whenever HoD chooses to delete a node v from G_i , it will retain all neighbors of v in G_i , even if some neighbor might be unimportant. For example, in Figure 4.1b, if HoD decides to remove v_6 , and then it will prevent v_7 from being removed at the same time, and vice versa.

4.4.3 Generation of Baseline Edges

As mentioned in Section 4.4.1, a baseline edge generated by the preprocessing algorithm of HoD is either (i) an edge in G_i whose endpoints are not to be removed, or (ii) an artificial edge $\langle u, w \rangle$ that corresponds to certain two-hop path $\langle u, v, w \rangle$ in G_i , such that none of u, v, w is to be removed. The construction of baseline edges from two-hop paths is worth discussing. First, given that there exists an enormous number of two-hop paths in G_i , it is prohibitive to convert each two-hop path into a baseline edge. Therefore, we only select a subset of the two-hop paths in G_i for baseline edge generation. In particular, the total number of two-hop paths selected is set to $c \cdot \sum_{v \in R_i} s(v)$, where c is a small constant, $s(v)$ is as defined in Equation 4.1, and $\sum_{v \in R_i} s(v)$ is the total number of candidate edges induced by the removal of nodes in R_i . In other words, we require that the number of

baseline edges generated from two-hop paths is at most c times the number of candidate edges. In our implementation of HoD, we set $c = 5$.

Those $c \cdot \sum_{v \in R_i} s(v)$ baseline edges are generated as follows. First, we randomly choose an edge in G_i , and arbitrarily select an endpoint v of the edge that is not in R_i . (Note that such an endpoint always exists.) After that, from the incoming (resp. outgoing) neighbors of v , we randomly select a node u (resp. w), and we generate a baseline edge $\langle u, w \rangle$, setting its length to $l(\langle u, v, w \rangle)$. As such, if a node v is adjacent to a large number of edges, and then it has a high chance of being selected to produce baseline edges. This is intuitive since such a node v tends to lie on the shortest paths between many pairs of nodes, and hence, the baseline edges generated from v may be more effective in eliminating redundant shortcuts.

4.4.4 Termination Condition

As mentioned in Section 4.3, HoD requires that the core graph G_c fits in the main memory, where G_c is the reduced graph obtained in the last iteration of HoD's preprocessing algorithm. Accordingly, we do not allow the pre-computation procedure of HoD to terminate before the reduced graph G_i has a size no more than M . In addition, even after G_i fits in the main memory, we will still continue the preprocessing procedure, until the size of G_i is reduced by less than 5% in an iteration of the processing algorithm. This is intended to reduce the size of the core graph G_c to improve query efficiency, as will be explained in Section 4.5.

4.4.5 Index File Organization

Once the preprocessing procedure completes, the core graph G_c is written to the disk in the form of adjacency lists. Meanwhile, the adjacency list of each node not in G_c is separated into two parts that are stored in two different files, F_f and F_b . These two files are created at the beginning of HoD's preprocessing algorithm, and they are initially empty. Whenever a node v is removed from the reduced graph G_i , we inspect the adjacency list of v in G_i , and we append all of v 's outgoing (resp. incoming) edges to F_f (resp. F_b). Upon termination of the preprocessing procedure, we reverse the order of nodes in F_b , but retain the order of nodes in F_f . We refer to the graph represented by F_f as the *forward graph*, denoted as G_f . Meanwhile, we refer to the graph represented by F_b as the *backward graph*, denoted as G_b . When combined, G_c , G_f , and G_b form the augmented graph that is used by HoD for query processing. For example, for the augmented graph in Figure 4.1f, its core graph is as illustrated in Figure 4.1e, while its forward and backward graphs are as shown in Figure 4.3. For ease of exposition, we will abuse notation and use G_f (resp. G_b) to refer to both G_f (resp. G_b) and its underlying file structure F_f (resp. F_b).

G_f and G_b have two interesting properties. First, all nodes in G_f (resp. G_b) are stored in ascending (resp. descending) order of their ranks. To explain this, recall that any node v removed in the i -th iteration of the preprocessing algorithm has a rank $r(v) = i$. Consequently, if a node u is stored in G_f before another node w , and then $r(u) \leq r(w)$. As for G_b , since we reverse the order of all edges in G_b upon termination of the preprocessing procedure, we have $r(u) \geq r(w)$ for any node u that precedes another node w in G_b . Second, for any node v , its edges in G_f and G_b only connect it to the nodes whose ranks are strictly higher than v . This is because, by the time v is removed from the reduced graph, all nodes that rank lower than v must have been eliminated from the reduced graph, and hence, any edge in v 's adjacency list only links v to the nodes whose rank is at least $r(v)$. Meanwhile, any neighbor u of v in the reduced graph should have a rank higher than $r(v)$. Otherwise, we have $r(u) = r(v)$, which, by the definition of node ranks, indicates that u and v are removed in the same iteration of the preprocessing algorithm. This is impossible as the pre-computation procedure of HoD never eliminates two adjacent nodes in the same iteration, as explained in Section 4.4.1. In Section 4.5, we will show how HoD exploits the above two properties of G_f and G_b to efficiently process SSD queries.

4.4.6 Cost Analysis

The preprocessing algorithm of HoD requires $O(n)$ main memory, where n is the number of nodes in G . This is because (i) when removing a node v from the reduced graph, HoD needs to record the neighbors of v and exclude them from the node removal process, and (ii) v may have $O(n)$ neighbors. Other parts of the preprocessing algorithm do not have a significant memory requirement.

Each iteration of the preprocessing algorithm consists of three parts, namely, node removal with shortcut generation, baseline edge generation and the sorting process. Apparently, the sorting process dominates the other two, since all the triplets generated are sorted together by the sorting process. In the worst case when the input graph G is a complete graph, there are $O(n^2)$ triplets generated in each iteration, and $O(n)$ iterations could be incurred, leading to prohibitive I/O and CPU costs, which are $O(\frac{n^3}{M})$ and $O(\frac{n^3}{B} \log_{M/B} \frac{n^2}{B})$ respectively. Fortunately, real-world graphs are seldom complete graphs, and they tend to contain a large number of nodes with small degrees (e.g., scale-free networks). In that case, each iteration of HoD's preprocessing procedure would only generate a moderate number of edge triplets, leading to a relatively small overhead. An example on scale-free networks can be found in Appendix A.5.

Lastly, the space consumption of HoD's index is $O(n^2)$ since, in the worst case, HoD may construct a shortcut from each node v to every node that ranks higher than v . This space complexity is unfavorable, but it is comparable to the space complexity of VC-Index [23]. In addition, as shown in our experiments, the space requirement of HoD in practice is significantly smaller than the worst-case bound.

 Figure 4.3: Forward graph G_f and backward graph G_b .

4.5 Algorithm for SSD Queries

Given an SSD query from a node s that is not in the core graph G_c , HoD processes the query in three steps:

- (i) **Forward Search:** HoD traverses the forward graph G_f starting from s , and for each node v visited, computes the distance from s to v in G_f .
- (ii) **Core Search:** HoD reads the core graph G_c into the main memory, and continues the forward search by following the outgoing edges of each node in G_c .
- (iii) **Backward Search:** HoD linearly scans the backward graph G_b to derive the exact distance from s to any node not in G_c .

On the other hand, if s is in G_c , and then HoD would answer the query with a core search followed by a backward search, skipping the forward search. In what follows, we will present the details of three searches performed by HoD. For convenience, we define an *index* value $\theta(v)$ for each node *not* in the core graph G_c , such that $\theta(v) = i$ only if the i -th adjacency list in G_f belongs to v . By the way G_f is constructed (see Section 4.4.5), for any two nodes u and v with $\theta(u) < \theta(v)$, the rank of u is no larger than the rank of v .

4.5.1 Forward Search

The forward search of HoD maintains a hash table H_f and a min-heap Q_f . In particular, H_f maps each node v to a key $\kappa_f(v)$, which equals the length of the shortest path from s to v that is seen so far. Initially, $\kappa_f(s) = 0$, and $\kappa_f(v) = +\infty$ for any node $v \neq s$. On the other hand, each entry in Q_f corresponds to a node v , and the key of the entry equals $\theta(v)$, *i.e.*, the index of v . As will become evident shortly, Q_f ensures that the forward search visits nodes in ascending order of their indices, and hence, it scans the file structure of G_f only once, without the need to backtrack to any disk block that it has visited before.

HoD starts the forward search by inspecting each edge $e = \langle s, v \rangle$ adjacent to s in G_f , and then inserting v into H_f with a key $\kappa(v) = l(e)$. (Note that G_f contains only outgoing edges.) In addition, HoD also inserts v into Q_f . After that, HoD iteratively removes the top entry u in Q_f , and processes u as follows: for each edge $e = \langle u, v \rangle$ adjacent to u , if $\kappa_f(v) = +\infty$ in the hash table H_f , HoD sets $\kappa_f(v) = \kappa_f(u) + l(e)$ and inserts v into Q_f ; otherwise, HoD sets $\kappa_f(v) = \min\{\kappa_f(v), \kappa_f(u) + l(e)\}$. When Q_f becomes empty, HoD terminates the forward search, and retains the hash table H_f for the second step of the algorithm, *i.e.*, the core search.

4.5.2 Core Search

The core search of HoD is a continuation of the forward search, and it inherits the hash table H_f created during the forward search. In addition to H_f , HoD creates a min-heap Q_c , such that Q_c stores entries of the form $\langle v, \kappa_f(v) \rangle$, where v is a node and $\kappa_f(v)$ is the key of v in H_f . For any node u with $\kappa_f(u) \neq +\infty$ (*i.e.*, u is visited by the forward search), HoD inserts u into Q_c .

Given H_f and Q_c , HoD performs the core search in iterations. In each iteration, HoD extracts the top entry v from Q_c , and examines each outgoing edge e of v . For every such edge, HoD inspects its ending point w , and sets $\kappa_f(w) = \min\{\kappa_f(w), \kappa_f(v) + l(e)\}$. Then, HoD adds w into Q_c if w is currently not in Q_c . This iterative procedure is repeated until Q_c becomes empty. After that, the hash table H_f is sent to the last step (*i.e.*, the backward search) for further processing.

4.5.3 Backward Search

Given the hash table H_f obtained from the core search, the reversed search of HoD is performed as a sequential scan of the backward graph G_b , which stores nodes in descending order of their index values. For each node v visited during the sequential scan, HoD checks each edge $e = \langle u, v \rangle$ adjacent to v . (Note that G_b contains only incoming edges). If $\kappa_f(u) \neq +\infty$ and $\kappa_f(u) + l(e) < \kappa_f(v)$, and then HoD sets $\kappa_f(v) = \kappa_f(u) + l(e)$. Once all nodes in G_b are scanned, HoD terminates the backward search and, for each node v , returns $\kappa_f(v)$ as the distance from s to v .

One interesting fact about the backward search is that it does not require a heap to decide the order in which the nodes are visited. This leads to a much smaller CPU cost compared with Dijkstra's algorithm, as it avoids all expensive heap operations.

4.5.4 Correctness and Complexities

Compared with Dijkstra's algorithm, the main difference of HoD's query algorithm is that it visits nodes in a pre-defined order based on their ranks. The correctness of this

approach is ensured by the shortcuts constructed by the preprocessing algorithm of HoD. In particular, for any two nodes s and t in G , it can be proved that the augmented graph G^* always contains a path P from s to t , such that (i) P 's length equals the distance from s to t in G , and (ii) P can be identified by HoD with a forward search from s , followed by a core search and a backward search. More formally, we have the following theorem.

Theorem 4.1. *Given a source node s , the SSD query algorithm of HoD returns $\text{dist}(s, v)$ for each node $v \in G$.*

The proof of Theorem 4.1 is included in Appendix A.6.

The query algorithm of HoD requires $O(n + m_c)$ main memory, where m_c is the size of the core graph. This is due to the fact that (i) the forward, core, and back searches of HoD all maintain a hash table that takes $O(n)$ space, and (ii) the core search requires reading the core graph G_c into the memory. The time complexity of the algorithm is $O(n \log n + m')$, where m' is the total number of edges in G_c , G_f , and G_b . The reason is that, when processing an SSD query, HoD may need to scan G_f , G_c , and G_b once, and it may need to put $O(n)$ nodes into a min-heap. Finally, the I/O costs of the algorithm is $O((n + m')/B)$, since it requires at most one scan of G_f , G_c , and G_b .

4.6 Extension for SSSP Queries

Given a source node s , an SSSP query differs from an SSD query only in that it asks for not only (i) the distance from s to any other node v , but also (ii) the predecessor of v , *i.e.*, the node that immediately precedes v on the shortest path from s to v . To extend HoD for SSSP queries, we associate each edge $\langle u, w \rangle$ in the augment graph G^* with a node v , such that v immediately precedes w on the shortest path from u to w in G . For example, given the augmented graph in Figure 4.1f, we would associate the edge $\langle v_9, v_7 \rangle$ with v_6 , since (i) the shortest path from v_9 to v_7 in G is $\langle v_9, v_6, v_7 \rangle$, and (ii) v_6 immediately precedes v_7 in the path.

With the above extension, HoD processes any SSSP query from a node s using the algorithm for SSD query with one modification: Whenever HoD traverses an edge $\langle u, w \rangle$ and finds that $\text{dist}(s, u) + l(\langle u, w \rangle) < \text{dist}(s, w)$, HoD would not only update $\text{dist}(s, w)$ but also record the node associated with $\langle u, w \rangle$. That is, for each node w visited, HoD keeps track of the predecessor of w in the shortest path from s to w that have been seen so far. As such, when the SSD query algorithm terminates, HoD can immediately return $\text{dist}(s, w)$ as well as the predecessor of w .

Finally, we clarify how the preprocessing algorithm of HoD can be extended to derive the node associated with each edge. First, for each edge e in the original graph, HoD

Table 4.1: Datasets.

Name	V	E	Weighted?	Directed?	Size
USRN	24.9M	28.9M	yes	no	1.3GB
FB	58.8M	92.2M	no	no	4.4GB
u-BTC	16.3M	95.7M	no	no	4.6GB
u-UKWeb	6.9M	56.5M	yes	no	2.7GB
BTC	16.3M	99.4M	no	yes	2.3GB
Meme	53.6M	117.9M	no	yes	2.8GB
UKWeb	104M	3708M	no	yes	89GB

associates e with the starting point of e . After that, whenever HoD generates a candidate edge $\langle u, w \rangle$ during the removal of a node v , HoD would associate $\langle u, w \rangle$ with the node that is associated with the edge $\langle v, w \rangle$. For example, in Figure 4.1d, when HoD removes v_7 and creates a candidate edge $\langle v_9, v_{10} \rangle$, it associates the edge with v_7 , which is the node associated with $\langle v_7, v_{10} \rangle$.

4.7 Experiments

This section experimentally compares HoD with three methods: (i) VC-Index [23], the state-of-the-art approach for SSD and SSSP queries on disk-resident graphs; (ii) EM-BFS [12], an I/O efficient method for breadth first search; and (iii) EM-Dijk [63], an I/O efficient version of Dijkstra’s algorithm. We include EM-BFS since, on unweighted graphs, any SSD query can be answered using breadth first search, which is generally more efficient than Dijkstra’s algorithm. We obtain the C++ implementations of VC-Index, EM-BFS, and EM-Dijk from their inventors, and we implement HoD with C++. As the implementation of VC-Index only supports SSD queries, we will focus on SSD queries instead of SSSP queries. All of our experiments are conducted on a linux machine with a 2.4GHz CPU and 32GB memory¹.

4.7.1 Datasets

We use five real graph datasets as follows: (i) USRN [4], which represents the road network in the US; (ii) FB [39], a subgraph of the Facebook friendship graph; (iii) BTC [5], a semantic graph; (iv) Meme [60] and UKWeb [6], which are two web graphs. Among them, only USRN and FB are undirected. Since VC-Index, EM-BFS, and EM-Dijk are all designed for undirected graphs only, we are in need of more undirected datasets for experiments. For this purpose, we transform BTC and UKWeb into undirected graphs, using the same approach as in previous work (see [23] for details). After that, for each

¹We use the “ulimit -v” command to set up the memory limit, e.g., if the memory limit is 1GB (i.e., 1048576KB), then we set “ulimit -v 1048576” before running the code.

Table 4.2: Preprocessing time (in minutes).

Method	USRN	FB	u-BTC	u-UKWeb
HoD	4.0	22.4	34.4	105.5
VC-Index	20.3	281.8	78.1	768.2

Table 4.3: Space Consumption (in GB).

Method	USRN	FB	u-BTC	u-UKWeb
HoD	2.5	5.1	3.8	3.3
VC-Index	4.3	8.3	1.2	14.0

undirected (resp. directed) graph G , we compute its largest connected component (resp. weakly connected component) C , and we use C for experiments. Table 4.1 illustrates the details of the largest component obtained from each graph. In particular, u-BTC and u-UKWeb are obtained from the undirected versions of BTC and UKWeb, respectively.

Remark. The previous experimental study on VC-Index [23] uses USRN, u-BTC, and u-UKWeb instead of their largest connected components (CC) for experiments. We do not follow this approach as it leads to less meaningful results. To explain, consider a massive undirected graph G where each CC is small enough to fit in the main-memory. On such a graph, even if a disk-resident method can efficiently answer SSD queries, it does not necessarily mean that it is more scalable than a main-memory algorithm. In particular, one can easily answer an SSD query from any node s in G , by first reading into memory the CC that contains s , and then running a main-memory SSD algorithm on the CC. In general, given any graph G , one can use an I/O efficient algorithm [68] to pre-compute the (weakly) connected components in G , and then handle SSD queries on each component separately.

4.7.2 Results on Undirected Graphs

In the first sets of experiments, we evaluate the performance of each method on four undirected graphs: USRN, FB, u-BTC, and u-UKWeb. For HoD, EM-BFS, and EM-Dijk, we limit the amount of memory available to them to 1GB, which is smaller than the sizes of all datasets. For VC-Index, we test it with 2GB memory as it cannot handle any of our datasets under a smaller memory size.

Table 4.2 shows the preprocessing time of HoD and VC-Index on the four graphs. (EM-BFS and EM-Dijk are omitted as they do not require any pre-computation.) In all cases, HoD incurs a significantly smaller overhead than VC-Index does. In particular, on FB, the preprocessing time of HoD is more than ten times smaller than that of VC-Index. Table 4.3 compares the space consumptions of HoD and VC-Index. Except for the case of u-BTC, the space required by VC-Index is consistently larger than that by HoD.

Table 4.4: Average running time for SSD queries (in seconds).

Method	USRN	FB	u-BTC	u-UKWeb
HoD	1.8	3.2	1.6	1.4
VC-Index	27.2	94.9	10.1	70.0
EM-BFS	—	465.3	395.4	—
EM-Dijk	430.7	1597.4	844.1	553.8

Table 4.5: Estimated time for *closeness* computation (in hours).

Method	USRN	FB	u-BTC	u-UKWeb
HoD	0.9	2.0	1.3	2.4
VC-Index	13.2	51.8	6.1	43.1
EM-BFS	—	231.1	182.2	—
EM-Dijk	203.2	793.3	389.0	240.0

To evaluate the query efficiency of each method, we generate 100 SSD queries for each dataset, such that the source node of each query is randomly selected. Table 4.4 shows the average running time of each approach in answering an SSD query. The query time of HoD is at least an order of magnitude smaller than that of VC-Index. Meanwhile, VC-Index always outperforms EM-BFS, which is consistent with the experimental results reported in previous work [23]. We omit EM-BFS on USRN and u-UKWeb, since those two graphs are weighted, for which EM-BFS cannot be used to answer SSD queries. Finally, EM-Dijk incurs a larger query overhead than all other methods.

In the next experiment, we demonstrate an application of HoD for efficient graph analysis. In particular, we consider the task of approximating the *closeness* for all nodes in a graph G , using the algorithm by Eppstein and Wang [33]. The algorithm requires executing $k = \ln n / \epsilon^2$ SSD queries from randomly selected source nodes, where n is the number of nodes in G and ϵ is a parameter that controls the approximation error. Following previous work [23], we set $\epsilon = 0.1$.

Table 4.5 shows an estimation of the time required by each method to complete the approximation task. Specifically, we estimate the total processing time of each method as (i) its query time in Table 4.4 multiplied by k , plus (ii) its preprocessing time (if any). Observe that both EM-BFS and EM-Dijk incur prohibitive overheads – they require more than a week to finish the approximation task. In contrast, HoD takes at most 2.4 hours to complete the task, despite that it pays an initial cost for pre-computation. Meanwhile, VC-Index is significantly outperformed by HoD, and it needs around two days to accomplish the task on FB and u-UKWeb.

Table 4.6: Performance of HoD on directed graphs.

Dataset	Preprocessing	Index Size	SSD Query Time
BTC	11.4 minutes	2.1 GB	2.6 sec
Meme	1.2 minutes	2.3 GB	1.8 sec
UKWeb	18.4 hours	72.6 GB	53.7 sec

4.7.3 Results on Directed Graphs

Our last experiments consider the three directed graphs: BTC, Meme, and UKWeb. As none of the existing I/O methods [23, 59, 62, 63, 65, 66] support directed graphs, we focus on HoD. In particular, we run HoD on BTC and Meme with 1GB memory, and on UKWeb with 24GB memory, as the enormous size of UKWeb leads to a higher memory requirement. Table 4.6 shows the preprocessing and space overheads of HoD, as well as its average query time in answering 100 randomly generated SSD queries on each dataset. On BTC and Meme, HoD only incurs small pre-computation costs and moderate space consumptions. On UKWeb, HoD entails considerably higher preprocessing, space, and query overheads, but they are still reasonable given that UKWeb contains 30 times more edges than BTC and Meme do. To the best of our knowledge, this is the first result in the literature that demonstrates practical support for SSD queries on a billion-edge graph.

4.8 Summary

This chapter presents HoD, a practically efficient index structure for distance queries on massive disk-resident graphs. In particular, HoD supports both directed and undirected graphs, and it efficiently handles *single-source shortest path (SSSP) queries* and *single-source distance (SSD) queries* under memory-constrained environments. This contrasts the existing methods, which either (i) require that the dataset fits in the main memory during pre-computation and/or query processing, or (ii) support only undirected graphs. With extensive experiments on a variety of real-world graphs, we demonstrate that HoD significantly outperforms the state of the art in terms of query efficiency, space consumption, and pre-computation costs. For future work, we plan to investigate how HoD can be extended to (i) support point-to-point shortest path and distance queries and (ii) handle dynamic graphs that change with time.

Chapter 5

Point-to-Point Reachability Query

Given a directed graph G and two vertices s and t in G , a *point-to-point reachability (PPR)* query asks whether there exists a path from s to t in G . PPR queries are a fundamental operation on graphs and have numerous important applications, such as query processing in XML document, association detecting in Semantic Web graphs, graph pattern analysis and friend suggestion in social networks, connection identifying in road networks, recursive query answering in knowledge base, and program workflow analysis. Devising index structures for PPR queries is non-trivial, as it requires a careful balancing act between pre-computation cost, index size, and query processing overhead. In particular, if we pre-compute and store the reachability results for all pairs of vertices, then we can process any PPR query in $O(1)$ time but suffer prohibitive costs of preprocessing and space. On the other hand, if we omit indexing and process PPR queries directly on G using depth-first search (DFS) or breadth-first search (BFS), then we minimize space and pre-computation overhead, but fail to ensure query efficiency on large graphs.

Previous work [10,18–22,24–26,29,46,48,50,53,73,79,80,82,90–92,94–96] has proposed numerous indexing techniques to efficiently support PPR queries without significant space and pre-computation overheads. Most techniques, however, assume that the input graph G is *static*, which makes them inapplicable for the *dynamic* graphs commonly encountered in practice. For example, the social graph of Twitter is constantly changing, with thousands of new users added per day; the Semantic Web is frequently updated with new concepts and relations; even road networks are subject to changes due to road closures and constructions. There exist a few techniques [18,29,46,50,73,80,96] that are designed for dynamic graphs, but as we discuss in Sections 5.2 and 5.7, none of those techniques can scale to sizable graphs without significant loss of efficiency. Specifically, the methods in [18,29,46,50,73,80] incur prohibitive preprocessing costs on graphs with more than one million vertices. Meanwhile, the approach in [96] can handle million-vertex graphs, but it offers a query performance that is *generally no better than a simple BFS approach*, as shown in our experiments.

In summary, no existing method is able to effectively handle PPR queries on *large dynamic graphs*. Motivated by this, this chapter presents a comprehensive study on scalable reachability indices that support updates. We first introduce a *total order labeling (TOL)* framework, which summarizes three most advanced methods [22, 51, 94] for PPR queries on static graphs. TOL has two important properties: (i) every reachability index under TOL uniquely corresponds to a *total order* of vertices in the input graph, and (ii) the total order *solely* decides the index’s performances in terms of preprocessing, space, and queries. Given these properties, we investigate algorithms that enable us to insert or delete a vertex in a TOL index without changing the order of the other vertices, i.e., without significantly degrading the performance of the index. This results in general algorithms for handling insertions and deletions on indices under TOL. In particular, our insertion algorithm is *optimal* in that it leads to the minimum index size after insertion.

Interestingly, we observe that our update algorithms can be utilized to reduce the space consumptions and query costs of a TOL index, by adjusting the total order pertinent to the index. This leads to a general approach for improving any index under TOL, including the state-of-the-art techniques [22, 51, 94]. The effectiveness of our adjusting approach shows that the total orders of the techniques in [22, 51, 94] leave much room for enhancement, which motivates us to devise new methods for deriving improved total orders for TOL indices. As a result, we present a new reachability index, *Butterfly*, which offers reduced preprocessing, space, and query costs than any existing indices under TOL [22, 51, 94]. We experimentally evaluate TOL using a large variety of benchmark datasets with up to twenty million vertices, and we demonstrate the superiority of TOL against alternative solutions for static and dynamic graphs.

The rest of this chapter is organized as follows. Section 5.1 introduces the preliminaries for PPR queries. Section 5.2 reviews the existing work. Section 5.3 formally defines the TOL framework. Section 5.4 presents the update algorithms of TOL for both insertions and deletions. Section 5.5 demonstrates an iterative label reduction algorithm which utilizes the update algorithms to reduce the label size of existing TOL instantiations. Section 5.6 presents Butterfly, the new reachability index under the TOL framework. Section 5.7 shows extensive experimental results of TOL on a variety of benchmark datasets. Section 5.8 summarizes this chapter.

5.1 Preliminaries

Let $G = (V, E)$ be a directed graph with a set V of vertices and a set E of edges. For any two vertices s and t in V , we say that s can *reach* t (denoted as $s \rightarrow t$), iff there exists a directed path in G that starts from s and ends at t . Given s and t , a reachability query returns **TRUE** if $s \rightarrow t$, and **FALSE** otherwise. We refer to s and t as the *source vertex* and *terminal vertex* of the query, respectively.

Notation	Description
$G = (V, E)$	a DAG with a vertex set V and an edge set E
$o(v)$	the topological rank of a vertex v (see Section 5.1)
$N_{in}(v)$	the set of v 's in-neighbors
$N_{out}(v)$	the set of v 's out-neighbors
$l(v)$	the level of v (see Section 5.3)
$\mathcal{L}$	a total order labeling of G
$L_{in}(v)$	the in-label set of vertex v (see Definition 5.1)
$L_{out}(v)$	the out-label set of vertex v (see Definition 5.1)
$I_{in}(v)$	the inverted index for v 's in-labels (see Equation 5.3)
$I_{out}(v)$	the inverted index for v 's out-labels (see Equation 5.4)

Table 5.1: Table of notations.

If $s \rightarrow t$ and $t \rightarrow s$ both holds, then s and t are *strongly connected*. Accordingly, a *strongly connected component (SCC)* of G is defined as a maximal subset of V where any two vertices are strongly connected. Observe that a vertex u can reach another vertex v , iff either of the following conditions holds: (i) u and v belong to the same SCC, or (ii) there is a path that starts from the SCC containing u to the SCC containing v . Given this observation, there exists a simple method that reduces G into a smaller graph G^* to improve the efficiency of reachability queries:

- (i) Compute all SCCs of G . (This can be done in $O(|V| + |E|)$ time [87].
- (ii) Map each SCC C to a vertex $f(C)$. For any two SCCs C and C' , if G contains an edge that starts at a vertex in C and ends at a vertex in C' , then construct an directed edge from $f(C)$ to $f(C')$. Denote the resulting graph as G^* .
- (iii) Given a reachability query from s to t on G , we first retrieve the SSC S (resp. T) of G that contains s (resp. t). Then, we return **TRUE** for the query, if and only if (i) S and T are the same or (ii) $f(S)$ can reach $f(T)$ in G^* .

In the remainder of the chapter, we assume that G has been preprocessed with the above reduction method, i.e., G does not contain any strongly connected component with more than one vertex. (The same assumption is made in existing work [8, 19, 22, 24, 25, 49, 51, 52, 79, 94, 95].) Under this assumption, G should be a *directed acyclic graph (DAG)*. In addition, there exists a total order o on V , such that for any two vertices u and v in G , if $u \rightarrow v$ then $o(u) < o(v)$ (but not necessarily vice versa). Such a total order can be derived in linear time using a DFS on G [22]. We refer to o as a *topological order*, and $o(u)$ as the *topological rank* of u . For ease of reference, Table 5.1 lists the notations that will be frequently used in this chapter.

Figure 5.1: DAG G .

5.2 Related Work

Other than the static methods introduced in Section 2.4, there also exist a few studies [18, 29, 46, 50, 72, 73, 80] on reachability indices for dynamic graphs. In particular, [29, 46, 72, 73] propose algorithms for incrementally maintaining transitive closures on dynamic graphs. Those algorithms, however, cannot scale to graphs with more than a few thousand vertices, as shown by Krommidias et al. [58]. There also exist two methods [18, 80] that extend Cohen et al.’s 2-hop labeling approach [26] to handle updates. Nevertheless, the method in [80] is restricted to XML graphs, while [18] is unable to handle any of the million-node graphs used in our experiments. In addition, [50] presents an algorithm for performing updates on a reachability index, but the index itself incurs tremendous preprocessing costs on large graphs. Very recently, Yildirim et al. propose *Dagger* [96], an extension of GRAIL [95] that supports dynamic graphs. As we show in Section 5.7, however, *Dagger*’s query performance is up to 10^8 times worse than the solution in this chapter, and is generally no better than a simple BFS approach.

5.3 Total Order Labeling

This section presents *total order labeling (TOL)*, a reachability indexing framework that can be instantiated into various 2-hop labeling indices. The instantiation of TOL requires two input parameters, namely, a DAG $G = (V, E)$ and a *strict* total order l on V . We refer to l as a *level order*. For any vertex v , we define $l(v) \in [1, |V|]$ as the rank of v in l , and refer to $l(v)$ as the *level* of v . In addition, we say that v has a *higher* (resp. *lower*) level than another vertex u , if $l(v) < l(u)$ (resp. $l(v) > l(u)$).

Definition of TOL Indices. Given G and l , TOL uniquely defines a 2-hop labeling index $\mathcal{L}$ on G as follows:

Definition 5.1 (Total Order Labeling $\mathcal{L}$). *Given a DAG $G = (V, E)$ and a level order l , a **TOL index** $\mathcal{L}$ is a 2-hop labeling index where each vertex is associated with an in-label set $L_{in}(v)$ and an out-label set $L_{out}(v)$, such that $L_{in}(v)$ (resp. $L_{out}(v)$) contains every vertex u satisfying all of the following constraints:*

- *Reachability Constraint:* $u \rightarrow v$ (resp. $v \rightarrow u$);
- *Level Constraint:* $l(u) < l(v)$;
- *Path Constraint:* No simple path from u to v (resp. from v to u) in G contains a vertex w with $l(w) < l(u)$.

We illustrate Definition 5.1 with the following example.

Example 5.1. Given the DAG G shown in Figure 5.1, Table 5.2 shows two TOL indices on G (i.e., $\mathcal{L}_1$ and $\mathcal{L}_2$) with level orders l_1 and l_2 , respectively. Consider vertex g in G . Its in-label set in $\mathcal{L}_1$ contains a since (i) a can reach g , (ii) a has a higher level than g , and (iii) G contains only one simple path from a to g , and the path does not contain any vertex with a higher level than a . In addition, g 's in-label set does not contain any vertex other than a , since the path from any other vertex to g must pass through a , whereas a has the smallest level in l_1 , i.e., adding any other vertex to $L_{in}(g)$ violates the Path Constraint in Definition 5.1.

In contrast, g 's in-label set in $\mathcal{L}_2$ is $\emptyset$. This is because g has the smallest level in l_2 , due to which we cannot add any vertex into g 's in-label set without violating the Level Constraint in Definition 5.1. In general, the label sets in $\mathcal{L}_1$ are drastically different from their counterparts in $\mathcal{L}_2$, due to the differences between l_1 and l_2 . $\square$

As demonstrated in Example 5.1, the level order l used to instantiate TOL has a profound effect on resulting reachability index. Therefore, if we are to obtain a TOL index with high efficiency, it is essential that we choose an appropriate level order l . In Sections 5.4 and 5.6, we will discuss how a good level order can be derived and incrementally maintained for dynamic graphs. For convenience, we define the size of a TOL index $\mathcal{L}$ as the total size of the label sets in $\mathcal{L}$, i.e.,

$$|\mathcal{L}| = \sum_{v \in V} (|L_{in}(v)| + |L_{out}(v)|).$$

Query Algorithm. Given a reachability query from a vertex s to another vertex t , a TOL index $\mathcal{L}$ processes the query in the same way as other 2-hop labeling methods do. In particular, $\mathcal{L}$ first retrieves the out-label sets of s and the in-label set of t , and then computes a *witness set* as follows:

$$W(s, t) = (L_{out}(s) \cup \{s\}) \cap (L_{in}(t) \cup \{t\}). \quad (5.1)$$

If the witness set is empty, then $\mathcal{L}$ returns **FALSE** for the query; otherwise, $\mathcal{L}$ returns **TRUE**. The following lemma shows the correctness of the above query processing approach.

	TOL index $\mathcal{L}_1$			TOL index $\mathcal{L}_2$		
v	$l_1(v)$	$L_{in}(v)$	$L_{out}(v)$	$l_2(v)$	$L_{in}(v)$	$L_{out}(v)$
a	1	$\emptyset$	$\emptyset$	7	$\emptyset$	$\{b,c,d,f,g,h\}$
b	2	$\{a\}$	$\emptyset$	4	$\emptyset$	$\{c,f\}$
c	3	$\{a,b\}$	$\emptyset$	2	$\{g\}$	$\emptyset$
d	4	$\{a\}$	$\{c\}$	5	$\emptyset$	$\{c,f\}$
e	5	$\emptyset$	$\{a\}$	8	$\emptyset$	$\{a,b,c,d,f,g,h\}$
f	6	$\{a,b,d\}$	$\{c\}$	3	$\emptyset$	$\{c\}$
g	7	$\{a\}$	$\{c\}$	1	$\emptyset$	$\emptyset$
h	8	$\{a\}$	$\{b\}$	6	$\emptyset$	$\{b,c,f\}$

 Table 5.2: Two TOL indices for the DAG G in Figure 5.1.

Lemma 5.1. *Given any two vertices s and t , we have $W(s,t) \neq \emptyset$ iff $s \rightarrow t$ in G .*

Proof. We first prove that $W(s,t) \neq \emptyset$ implies $s \rightarrow t$. Observe that, when $W(s,t) \neq \emptyset$, at least one of the following three cases must hold: (i) $t \in L_{out}(s)$, (ii) $s \in L_{in}(t)$, and (iii) $L_{in}(t) \cap L_{out}(s) \neq \emptyset$. By Definition 5.1, both $t \in L_{out}(s)$ and $s \in L_{in}(t)$ indicate that $s \rightarrow t$. Meanwhile, if $L_{out}(s) \cap L_{in}(t) \neq \emptyset$, then for any $u \in L_{out}(s) \cap L_{in}(t)$, we have $s \rightarrow u$ and $u \rightarrow t$, which leads to $s \rightarrow t$.

Next, we show that if $s \rightarrow t$, then $W(s,t) \neq \emptyset$. Consider the set of all simple paths from s to t in G . Let v be the vertex with the highest level among all vertices on those paths. We differentiate three cases: (i) $v = t$, (ii) $v = s$, and (iii) $v \neq s$ and $v \neq t$. If $v = t$, then by Definition 5.1, we have $t \in L_{out}(s)$, in which case $W(s,t)$ contains t , i.e., $W(s,t) \neq \emptyset$. Meanwhile, if $v = s$, then $s \in L_{in}(t)$ holds, which leads to $W(s,t) \supseteq \{s\} \neq \emptyset$. Finally, if $v \neq s$ and $v \neq t$, then v must appear in both $L_{out}(s)$ and $L_{in}(t)$, in which case $W(s,t) \supseteq \{v\} \neq \emptyset$. $\square$

Interesting, the label sets in any TOL index $\mathcal{L}$ are *minimal*, in the sense that no label can be removed without affecting the correctness of TOL's query processing algorithm:

Lemma 5.2. *Let $\mathcal{L}$ be a TOL index on G with a level ordering l . For any vertex u , if we remove a vertex v_1 from $L_{out}(u)$, then $W(u, v_1) = \emptyset$ but $u \rightarrow v_1$. Meanwhile, if we remove a vertex v_2 from $L_{in}(u)$, then $W(v_2, u) = \emptyset$ but $v_2 \rightarrow u$.*

Proof. Consider vertex v_1 . Given that $v_1 \in L_{out}(u)$, we have $u \rightarrow v_1$ and $l(u) < l(v_1)$ by Definition 5.1. Since $l(u) < l(v_1)$, we have $u \notin L_{in}(v_1)$ by the Level Constraint in Definition 5.1.

Assume on the contrary that, after v_1 is removed from $L_{out}(u)$, $W(u, v_1) \neq \emptyset$. Then, by Equation 5.1 and $u \notin L_{in}(v_1)$, there must exist a vertex $w \in L_{out}(u) \cap L_{in}(v_1)$. In that

case, G must contain a simple path from u to v_1 via w , and w must have a higher level than both u and v_1 . This contradicts the Path Constraint in Definition 5.1 since $v_1 \in L_{out}(u)$ initially holds. The case for vertex v_2 can be established in a similar manner. $\square$

Existing Instantiations of TOL. By Definition 5.1, TOL defines a family of 2-hop labeling approaches that satisfy the Reachability, Level, and Path Constraints. This index family does not include all existing 2-hop labeling methods (as many of them violate the aforementioned constraints), but it captures the three most advanced 2-hop labeling techniques, i.e., TF-Label [22], DL [51], and PLL [94]. In particular, TF-Label utilizes a topological order o of the vertices in G (see Section 5.1 for the definition of o). It constructs indices in way such that (i) a label set of a vertex v only contains vertices u with $o(v) < o(u)$, and (ii) the label sets are minimal. It can be shown the TF-Label corresponds to a TOL index that uses o as the level order (with ties broken arbitrarily when multiple vertices have the same rank in o).

Meanwhile, DL sorts the vertices in G in descending order of their degrees, and it follows the sorted order to inspect vertices in G and constructs label sets accordingly. Specifically, each time it examines a vertex v , it uses two constrained BFSs on G to identify a number of vertices that (i) are connected to v and (ii) rank lower than v in the sorted order; then, it adds v into the label sets of those vertices. It can be proved that DL is equivalent to a TOL index where the level order ranks vertices in descending order of their degrees. Finally, as PPL is shown to be equivalent to DL [51], it is also an instantiation of TOL.

It is noteworthy that, if we modified the vertex order in DL, and use the modified order to construct a reachability index based on DL’s preprocessing algorithm, then the resulting index is equivalent to a TOL index adopting the same modified order. In other words, any TOL index can be obtained using a modified version of DL’s pre-computation algorithm. Nevertheless, the paper that describes DL [51] does not summarize the Reachability, Level, and Path Constraints (see Definition 5.1) that underpin TOL. The summarization of those constraints is crucial in the context of this chapter, as they are imperative in our analysis on how TOL indices can be incrementally updated (see Section 5.4).

5.4 Incremental Updates

In this section, we study how a TOL index $\mathcal{L}$ can be incrementally updated when a vertex is inserted into or deleted from G . Our objective is twofold:

- (i) $\mathcal{L}$ should remain a TOL index after any update, i.e., it should always satisfy the constraints in Definition 5.1. This is to ensure the correctness of $\mathcal{L}$ ’s query algorithm and the minimality of $\mathcal{L}$ ’s label sets

- (ii) *Inserting or deleting a vertex should not change the level order l on the other vertices.* Intuitively, this reduces the amount of changes required in the label sets of $\mathcal{L}$, and helps retain the performance of $\mathcal{L}$ after the update, since a TOL index's label sets (and thus, its performance) are solely decided by its level order.

5.4.1 Insertion Algorithm

Consider that we insert a new vertex v into G and connect v with some existing vertices in G . Let $G' = (V', E')$ be the graph obtained after the insertion. Following previous work [18, 96], we assume that G' is also a DAG. The case when G' is not a DAG can be handled by incrementally maintaining the strongly connected components in G , as discussed in [96]. Let $\mathcal{L}$ be a TOL index on G with a level order l . As mentioned, our objective is to update $\mathcal{L}$ into a TOL index $\mathcal{L}'$ on G' with a level order l' , such that for any two vertices $u_1, u_2 \in V$, $l(u_1) < l(u_2)$ iff $l'(u_1) < l'(u_2)$.

In a nutshell, our insertion algorithm runs in two steps: In Step 1, it decides the value of $l'(v)$, and then sets $l'(u)$ for any vertex u in G as follows:

$$l'(u) = \begin{cases} l(u) & \text{if } l(u) < l'(v) \\ l(u) + 1 & \text{otherwise} \end{cases} \quad (5.2)$$

Then, in Step 2, it updates the label sets in $\mathcal{L}$ according to l' , which transforms $\mathcal{L}$ into $\mathcal{L}'$. For ease of exposition, we will first elaborate Step 2, assuming that l' has been constructed.

5.4.1.1 Step 2: Updating Label Sets

Given G' , $\mathcal{L}$, and l' , the second step of our insertion algorithm is further divided into two sub-steps. In Step 2.1, we create the label sets $L'_{in}(v)$ and $L'_{out}(v)$ for the new vertex v , and insert v into the label sets of other vertices. Then, in Step 2.2, we further refine the label sets of the vertices in V , so as to convert $\mathcal{L}$ into $\mathcal{L}'$. Throughout the algorithm, for each vertex u , we maintain two *inverted lists* $I_{in}(u)$ and $I_{out}(u)$, such that

$$I_{in}(u) = \{w \mid u \in L_{in}(w)\}, \quad (5.3)$$

$$I_{out}(u) = \{w \mid u \in L_{out}(w)\}. \quad (5.4)$$

In other words, if u appears in the in-label (resp. out-label) set of a vertex w , then w is recorded in the inverted list $I_{in}(u)$ (resp. $I_{out}(u)$). These inverted lists enable us to efficiently identify the label sets that are affected by any vertex u . In addition, they can be easily maintained with respect to changes in the label sets.

Step 2.1. Algorithm 1 shows the pseudo-code for the first sub-step. The algorithm first creates a *candidate set* $C_{in}(v)$ (Lines 1-4), and then refines it into the in-label set $L'_{in}(v)$

Algorithm 1: INSERT-STEP-2.1

input : $G', \mathcal{L}, l'$, and v
output: $L'_{in}(v), L'_{out}(v)$, and a modified version of $\mathcal{L}$

```

1 create two candidate label sets  $C_{in}(v)$  and  $C_{out}(v)$  ;
2 set  $L'_{in}(v) = L'_{out}(v) = C_{in}(v) = C_{out}(v) = \emptyset$  ;
3 for each of  $v$ 's in-neighbors  $u$  do
4    $C_{in}(v) = C_{in}(v) \cup L_{in}(u) \cup \{u\}$  ;
5 for each  $u \in C_{in}(v)$  in ascending order of  $l'(u)$  do
6   if  $L_{out}(u) \cap L'_{in}(v) = \emptyset$  then
7     if  $l'(u) < l'(v)$  then
8       add  $u$  into  $L'_{in}(v)$  ;
9     else
10      add  $v$  into  $L_{out}(u)$  ;
11 for each of  $v$ 's out-neighbors  $u$  do
12    $C_{out}(v) = C_{out}(v) \cup L_{out}(u) \cup \{u\}$  ;
13 Repeat Lines 5-10 with subscripts “in” and “out” exchanged ;
14 return  $L'_{in}(v), L'_{out}(v)$ , and the label sets in  $\mathcal{L}$  ;
    
```

of v (Lines 5-10). In particular, the candidate set $C_{in}(v)$ contains all in-neighbors of v (i.e., the starting vertices of the edges pointing to v), as well as the in-label sets of those in-neighbors (Line 3-4).

By Definition 5.1, $C_{in}(v)$ is a superset of v 's in-labels in $\mathcal{L}'$. To explain, consider a vertex u that is an in-label of v in $\mathcal{L}'$. By the Reachability Constraint, there exists a path P from u to v in G' . Let w be the in-neighbor of v on P . Then, u can reach w . In addition, $l(u) < l(w)$; otherwise, w is a vertex on P with a higher level than u , which violates the Path Constraint. Finally, all paths from u to w should contain only vertices with levels lower than u ; otherwise, u should not be an in-label of v in the first place. All of the above indicates that u is an in-label of w , as it fulfills the three constraints in Definition 5.1. Therefore, $C_{in}(v)$ is superset of v 's in-labels in $\mathcal{L}'$.

To refine $C_{in}(v)$ into $L'_{in}(v)$, Algorithm 1 removes the vertices in $C_{in}(v)$ that violate any of the Level and Path Constraints in Definition 5.1 (Lines 5-10). (The Reachability Constraint is ignored as all vertices in $C_{in}(v)$ can reach v .) Specifically, the algorithm examines the vertices in $C_{in}(v)$ in ascending order of their level values. For each vertex u with $l'(u) < l'(v)$ (i.e., u satisfies the Level Constraint), the algorithm adds u into $L'_{in}(v)$ if $L_{out}(u) \cap L'_{in}(v) = \emptyset$. The rationale is that, if $L_{out}(u) \cap L'_{in}(v) = \emptyset$, then no vertex with a higher level than $l(u)$ can connect u to v , in which case u fulfills the Path Constraint. Meanwhile, if $L_{out}(u) \cap L'_{in}(v) = \emptyset$ but $l'(u) > l'(v)$, then we add v into u 's out-label set $L_{out}(u)$ instead.

Algorithm 2: INSERT-STEP-2.2

input : G' , l' , v , and the output of Algorithm 1
output: A TOL index $\mathcal{L}'$ for G' with a level order l'

```

1 for each vertex  $u \in L'_{in}(v)$  in ascending order of  $l'(u)$  do
2   for each vertex  $w \in L'_{out}(v) \cup \{v\}$  with  $l'(w) > l'(u)$  do
3     if  $L_{out}(u) \cap L_{in}(w) = \emptyset$  then
4        $L_{in}(w) = L_{in}(w) \cup \{u\}$ ;
5     for each vertex  $x \in I_{in}(w)$  do
6       if  $L_{out}(u) \cap L_{in}(x) = \emptyset$  then
7          $L_{in}(x) = L_{in}(x) \cup \{u\}$ ;
8   for each vertex  $x' \in I_{in}(u)$  do
9     for each vertex  $y' \in I_{out}(u)$  do
10      if  $y' \in L_{in}(x')$  then
11        remove  $y'$  from  $L_{in}(x')$ ;
12      if  $x' \in L_{out}(y')$  then
13        remove  $x'$  from  $L_{out}(y')$ ;
14 Repeat Lines 1-14 with subscripts “in” and “out” exchanged ;
15 return the revised label sets ;
    
```

After $L'_{in}(v)$ is created, Algorithm 1 constructs $L'_{out}(v)$ (Lines 11-13) and then terminates. We omit the discussion on $L'_{out}(v)$ as it is similar to the case of $L'_{in}(v)$.

Step 2.2. Given Step 2.1’s output, Step 2.2 of our algorithm proceeds to refine the label sets in $\mathcal{L}$, as shown in Algorithm 2. The rationale is that, since the insertion of v creates new paths among the vertices in V , we may need to adjust the label sets in $\mathcal{L}$ to ensure that $\mathcal{L}$ remains a TOL index. Specifically, Algorithm 2 first inspects the vertices in $L'_{in}(v)$ in ascending order of their level values (Line 1). For each vertex u , the algorithm examines each vertex w in $L'_{out}(v) \cup \{v\}$ with lower levels than u (Line 2). Notice that, for every such pair of vertices u and w , the insertion of v creates a new path from u to w via v . Accordingly, the algorithm adds u into w ’s in-label set $L_{in}(w)$ if $L_{out}(u) \cap L_{in}(w) = \emptyset$, i.e., if inserting u into $L_{in}(w)$ does not violate the Path Constraint in Definition 5.1 (Lines 3-4). Similarly, the algorithm also inserts u into $L_{in}(x)$ for any vertex $x \in I_{in}(w)$, i.e., any vertex x that has w as an in-label (Line 5-7).

After that, the algorithm proceeds to check whether the operations in Lines 2-7 have resulted in unnecessary labels (Lines 8-13). In particular, it examines each pair of vertices $x' \in I_{out}(u)$ and $y' \in I_{in}(u)$, i.e., x' has u as an out-label and y' has u as an in-label. For each pair of x' and y' , the algorithm checks whether y' appears in the out-label of x' ; if so, it removes y' from $L_{out}(x')$, since (i) u has higher level than both x' and y' and (ii) u

connects x' to y' , which leads to a violation of the Path Constraint. Similarly, if x' is in the out-label set of y' , and it is removed.

Once the above nested loop is finished, Algorithm 2 enters another nested-loop, where (i) the outer loop linearly scans each vertex u in $L'_{out}(v)$ in ascending order of level values, and (ii) the inner loop examines each vertex w in $L'_{in}(v) \cup \{v\}$ with $l'(u) < l'(w)$. This nested loop complements the previous nested loop, in that the former processes vertex pairs in $L'_{in}(v)$ and $L'_{out}(v)$ that are ignored by the latter. Finally, Algorithm 2 terminates and returns the revised label sets, which constitute $\mathcal{L}'$.

5.4.1.2 Step 1: Deciding Vertex Level

Our algorithms in Section 5.4.1.1 requires that the level $l'(v)$ of the new vertex v is decided. A straightforward approach is to set $l'(v) = |V| + 1$, i.e., give v the lowest possible level. This leads to relatively small update overheads because, when $l'(v)$ is maximized, we do not need to insert v into the label sets of any other vertex (due to the Level Constraint in Definition 5.1). In terms of space overhead and query efficiency, however, setting $l'(v) = |V| + 1$ could be highly sub-optimal than other choices $l'(v)$. To address this issue, we present an alternative solution that sets $l'(v)$ to a value that minimizes the total size of the label sets. Such a $l'(v)$ is also likely to improve query efficiency, since the cost of a reachability query on a TOL index is linear to the sizes of the source and terminal vertices' label sets.

Let $\mathcal{L}_k$ be the TOL index obtained by inserting v into $\mathcal{L}$ with $l'(v) = k$. To identify the value of k that minimizes $\mathcal{L}$, we examine all possible $k \in [1, |V| + 1]$, but avoid repeatedly using Algorithms 1 and 2 to construct all $\mathcal{L}_k$. Instead, we propose a lightweight approach for deriving

$$\Delta_k = |\mathcal{L}_k| - |\mathcal{L}_{k+1}| \quad (5.5)$$

for any $k \in [1, |V|]$. Once Δ_k are computed, we can easily determine the optimal value of $l'(v)$.

The key observation behind our approach is as follows. When we change $l'(v)$ from k to $k - 1$, the level order of all vertices remain unchanged, except for v and the vertex u with $l(u) = k - 1$ (since the order between u and v would be reversed). As a consequence, the size difference between $\mathcal{L}_k$ and $\mathcal{L}_{k-1}$ only depends on the label sets that concern u and v . Intuitively, tracking the changes in those label sets is much simpler than creating a TOL index from scratch, and hence, deriving Δ_k can be much more efficient than constructing $\mathcal{L}_k$.

Algorithm 3 shows the pseudo-code of our approach. It first sets $l'(v) = |V| + 1$, and applies Algorithm 1 to compute, for v , two label sets $L'_{in}(v)$ and $L'_{out}(v)$ and two inverted lists $I'_{in}(v)$ and $I'_{out}(v)$ (Lines 1-2). The subsequent part of the algorithm consists of $|V|$ iterations (Lines 3-14). In the $(|V| - k + 1)$ -th iteration, the algorithm considers the case

Algorithm 3: INSERT-STEP-1

input : $G = (V, E)$, $\mathcal{L}$, l , and v
output: the value for $l'(v)$ that minimizes $|\mathcal{L}'|$

```

1  set  $l'(v) = |V| + 1$  ;
2  construct  $L'_{in}(v)$ ,  $L'_{out}(v)$ ,  $I'_{in}(v)$ , and  $I'_{out}(v)$  as in Algorithm 1 (without materializing
   any changes to  $\mathcal{L}$ ) ;
3  for  $k = |V|, |V| - 1, \dots, 1$  do
4       $\Delta_k = 0$  ;
5      let  $u$  be the vertex with  $l(u) = k$  ;
6      if  $u \in L'_{in}(v)$  then
7          remove  $u$  from  $L'_{in}(v)$  and add it into  $I'_{out}(v)$  ;
8          for each vertex  $w \in I'_{in}(v)$  such that  $u \in L_{in}(w)$  do
9               $\Delta_k = \Delta_k - 1$  ;
10         for each vertex  $w' \in I_{out}(u)$  such that  $v \notin L_{out}(w)$  do
11              $\Delta_k = \Delta_k + 1$  ;
12             add  $w'$  into  $I'_{out}(v)$  ;
13     repeat Lines 6-12 with subscripts “in” and “out” exchanged ;
14  initialize variables  $\theta_1, \theta_2, \dots, \theta_{|V|+1}$  ;
15   $\theta_{|V|+1} = 0$  ;
16  for  $k = |V|, |V| - 1, \dots, 1$  do
17       $\theta_k = \theta_{k+1} + \Delta_k$  ;
18  return  $\arg \min_k \{\theta_k\}$  ;
    
```

when $l'(v)$ changes from $k + 1$ to k , and evaluates the corresponding changes in the label sets, based on which it derives Δ_k .

Specifically, the algorithm first sets $\Delta_k = 0$ and inspects the vertex u with $l(u) = k$, i.e., the vertex whose level is to be exchanged with v when $l'(v)$ is decreased from $k + 1$ to k . Observe that, if the exchange between u and v leads to changes in some label sets, then $u \in L'_{in}(v) \cup L'_{out}(v)$ should hold. The reason is that, when $u \notin L'_{in}(v) \cup L'_{out}(v)$, either (i) there is no path between u and v or (ii) all paths between u and v contain at least one vertex with higher level than u and v . In either case, switching levels between u and v would not lead to violations of the Reachability, Level, or Path Constraint in any label sets. Therefore, if $u \notin L'_{in}(v) \cup L'_{out}(v)$, then no label set would be affected by swapping u and v 's levels. Based on this analysis, Algorithm 3 sets the final value of Δ_k to 0, whenever $u \notin L'_{in}(v) \cup L'_{out}(v)$ (Lines 4-13).

Now consider that $u \in L'_{in}(v)$. After we exchange u and v 's levels, u should be removed from $L'_{in}(v)$, and v should become an out-label of u . This explains Line 7 in Algorithm 3. Meanwhile, for any vertex $w \in I'_{in}(v)$ (i.e., w has v as an in-label), we check if u is an in-label of w (Line 8). If $u \in L_{in}(w)$, then after the levels of u and v are

swapped, u should be removed from $L_{in}(w)$ due to the Path Constraint, which reduces the size of $L_{in}(w)$ by one. Accordingly, Algorithm 3 decreases Δ_k by 1 for each such vertex w (Line 9). In addition, for any vertex $w' \in I_{out}(u)$ (i.e., u is an out-label for w'), we examine if v is not out-label of w' (Line 10). If $v \notin L_{out}(w')$, then after we swap u and v 's levels, v will become an out-label of w' , i.e., the size of $L_{out}(w')$ is increased by one. Therefore, for each such vertex w' , Algorithm 3 increases Δ_k by 1 and inserts w' into $I'_{out}(v)$ (Lines 11 and 12). It can be verified that, apart from the label sets mentioned above, no other label sets would be affected by the exchange between u and v .

Although the above discussion assumes $u \in L'_{in}(v)$, it can be easily extended to the case when $u \in L'_{out}(v)$ (Line 13). Once all Δ_k are obtained, Algorithm 3 derives $|V| + 1$ variables $\theta_1, \theta_2, \dots, \theta_{|V|+1}$, such that $\theta_{|V|+1} = 0$ and $\theta_k = \theta_{k+1} + \Delta_k$ ($k \in [1, |V|]$). By the definition of Δ_k , the value of k that minimizes θ_k should also minimize $|\mathcal{L}_k|$. Accordingly, Algorithm 3 terminates by returning $\arg \min_k \{\theta_k\}$ (Line 18).

5.4.1.3 Correctness and Complexity

We first show the correctness of our insertion algorithm by proving Lemma 5.3, and then analyze the complexity of our insertion algorithm.

Lemma 5.3. *Given G and a new vertex v , Algorithms 1 and 2 produces a TOL index on G' , and Algorithm 3 computes a level for v that minimizes the label size of $\mathcal{L}'$.*

Proof. We start by proving that the index $\mathcal{L}'$ produced by Algorithm 1 and 2 is a TOL index on G' . In particular, we first show that the creating phase (Algorithm 1 and Lines 1-7 in Algorithm 2) creates labels that are the super sets of corresponding labels in the TOL on G' , and then demonstrate how the reducing phase (Lines 8-13 in Algorithm 2) revises $\mathcal{L}'$ into the TOL index on G' by removing redundant labels.

Since we have shown the correctness of $L'_{in}(v)$ and $L'_{out}(v)$ created by Algorithm 1 in Section 5.4.1.1, in the following, we prove that the pseudo code in Lines 1-7 in Algorithm 2 creates in-label sets which are the super sets of corresponding TOL in-label sets for the other vertices in G' . Given a vertex u , and another vertex x , according to Definition 5.1, the insertion of v causes u becoming an in-label of x , only if all of the following conditions hold: (i) $l'(u) < l'(x)$; (ii) u can reach v in G' and v can reach x in G' ; (iii) no simple path from u to x contains a vertex that has higher level than u . Thus in Line 1, we eliminate vertex u that is not in $L'_{in}(v)$ from consideration, because if u is not in $L'_{in}(v)$, then let z be the highest level vertex among all the paths from u to v , we know $z \neq u$, and z is also on the path from u to any x that can be reached by v , then condition (iii) is violated, so u will never become an in-label of any vertex in G' . On the other hand, let x be the vertex filtered by Line 2, if x has higher level than u , then it should be eliminated as it violates condition (i); otherwise, x is not in $L'_{out}(v) \cup \{v\}$, and let the highest level vertex

among all the paths from v to x be z , then we know $z \in L'_{out}(v)$. If $l'(z) > l'(u)$, then x is examined in the loop when $w = z$; otherwise, we have $l'(z) < l'(u)$, which indicates that there is a path from u to x that contains a vertex (i.e., z) that has higher level than u , violating condition (iii). In addition, Line 3 and Line 6 also guarantee that the newly created labels do not violate condition (iii). Hence, the creating phase only omits those unnecessary cases that will never lead to a label creation, which indicates that the sets of in-labels created in Lines 1-7 are the super sets of corresponding TOL in-label sets on G' .

After adding u into $L_{in}(x)$, some existing labels related to x become redundant. According to Definition 5.1, neither the Level Constraint nor the Reachability Constraint would be affected for an existing label, and the only violation is on the Path Constraint. In particular, if u is also an out-label of a vertex y , then no matter y is an in-label of x or x is an out-label of y , these labels violate the Path Constraint, since there is a path from y to x with u on it, and u has higher level than both y and x . By enumerating x' in Line 8 (v is also included since we add u into $L'_{in}(v)$ in Algorithm 1) and y' in Line 9, the nested loops cover all the labels that may be affected by u , and we reduce the labels in Lines 10-11 and Lines 12-13 accordingly. Hence, the redundant in-labels of vertices that can be reached by v are removed, as well as the redundant out-labels of vertices that can reach v (via u).

After that we repeat the above procedure (Line 14) with "in" and "out" exchanged to update in-labels of vertices that can reach v and out-labels of vertices that v can reach for the complementary updates from the above process. Notice that the code in Line 14 does not conflict with the former process, since the set of vertices that can reach v and the set of vertices that v can reach do not intersect due to G' is a DAG. Summarizing the above discussion, as we create labels that form super sets of corresponding label sets in the TOL on G' , and then remove all the redundant labels in the reducing phase, we proved that the labels constructed by Algorithm 1 and Algorithm 2 is the TOL on G' .

Finally, Algorithm 3 accurately evaluates the size differences between consecutive levels for v , i.e., $\Delta_i, i = 1, \dots, |V|$ (as shown in Section 5.4.1.2), which allows Algorithm 3 to find the level k for v , such that the resulting labeling size is minimized. $\square$

Complexity Analysis. To analyze our algorithm, we consider the complexity step by step. In Step 1, given a vertex v for insertion, Algorithm 3 inspects all the vertices $u \in L'_{in}(v) \cup L'_{out}(v)$, and derives Δ_k by computing all vertices $w \in I_{in}(u) \cup I_{out}(u)$, each with one set operation, i.e., Lines 8 and 10. Let β be the cost of one set operation, then the complexity of Step 1 is bounded by $O(|V|^2\beta)$. In Step 2.1, Algorithm 1 incurs $|C_{in}(v)| + |C_{out}(v)|$ number of set operation in Line 6, therefore, its cost is bounded by $C_2 = O((|C_{in}(v)| + |C_{out}(v)|)\beta)$. In Step 2.2, Algorithm 2 performs the set operations for each pair of vertices in the worse case, resulting a complexity of $O(|V|^2\beta)$. Note that,

Lines 9-14 inspects the vertices in $I_{in}(u) \cap L_{in}(x')$ and $I_{out}(u) \cap L_{out}(y')$, which can be implemented in set operations as well. Hence, the complexity for our entire insertion algorithm is $O(|V|^2\beta)$.

5.4.2 Deletion Algorithm

Next, we discuss our algorithm handling deletion in G . Let $G' = (V', E')$ be the graph obtained by removing a vertex v from G . As with the case of insertion, we assume that G' is a DAG, and we aim to transform $\mathcal{L}$ into a TOL index $\mathcal{L}'$ on G' , such that the level orders of $\mathcal{L}$ and $\mathcal{L}'$ sort the vertices in V' in the same order.

Algorithm 4 presents the pseudo-code of our deletion method. It first removes v from every label set that it appears (Lines 1-4). Then, it refines the label sets in $\mathcal{L}$ to convert it into $\mathcal{L}'$ (Lines 5-22). In particular, it first retrieves the set $B^+(v)$ of all vertices that v can reach, using a BFS from v that follows the outgoing edges of each vertex. Then, it inspects the vertices in $B^+(v)$ in ascending order of their topological ranks (see Section 5.1), and reconstructs the in-label set of each vertex. (Note that the out-label sets of those vertices are not affected by the deletion of v .)

For each vertex $u \in B^+(v)$, the algorithm first creates a candidate $C_{in}(u)$ (Line 8). Then, for each of u 's in-neighbors z such that $z \neq v$, the algorithm inserts the in-labels of z into $C_{in}(u)$ (Lines 9-10). It can be proved that $C_{in}(u)$ is a superset of the in-labels of u in $\mathcal{L}'$. To refine $C_{in}(u)$ into u 's in-label set, the algorithm examines each vertex w in $C_{in}(u)$ in ascending order of $l(w)$ (Line 12). If $l(w) < l(u)$ and $L_{out}(w) \cap L_{in}(u) = \emptyset$, then we identify w as an in-label of u in $\mathcal{L}'$ (Line 14). Subsequently, we remove the labels that become redundant due to the insertion of w into $L_{in}(u)$ (Lines 15-17). In particular, for each vertex s having w as an out-label, if u is also in the out-label set of s , then we remove u from the out-label of s .

Once all vertices in $B^+(v)$ are processed, the algorithm derives the set $B^-(v)$ of vertices that can reach v , by applying a BFS from v that follows the incoming edges of each vertex. After that, it reconstructs the out-label set of each vertex in $B^-(v)$, in a way similar to the case of $B^+(v)$ (Lines 18-21). Finally, it returns the modified label sets for all vertices except v , i.e., the label sets that form $\mathcal{L}'$.

Correctness Proof. Lemma 5.4 proves the correctness of our deletion algorithm.

Lemma 5.4. *Given G and a vertex v to be deleted, the updated labeling $\mathcal{L}'$ produced by Algorithm 4 is a TOL on G'*

Proof. We first prove that, the candidate set $C_{in}(u)$ generated in Lines 8-10 is a super set of $L'_{in}(u)$, and then we show that the redundant labels caused by creating $L'_{in}(u)$ are removed in Lines 12-19.

Algorithm 4: DELETE

input : $G = (V, E)$, $\mathcal{L}$, and $v \in V$
output: $\mathcal{L}'$

- 1 **for** each $x \in I_{out}(v)$ **do**
- 2 $\perp$ remove v from $L_{out}(x)$;
- 3 **for** each $y \in I_{in}(v)$ **do**
- 4 $\perp$ remove v from $L_{in}(y)$;
- 5 identify the set $B^+(v)$ of vertices that v can reach, using a BFS from v that follows the outgoing edges of each vertex ;
- 6 **for** each $u \in B^+(v)$ in ascending order of $o(u)$ **do**
- 7 let $N_{in}(u)$ be the set of in-neighbors of u ;
- 8 create a candidate set $C_{in}(u) = \emptyset$;
- 9 **for** each $z \in N_{in}(u)$ such that $z \neq v$ **do**
- 10 $\perp$ $C_{in}(u) = C_{in}(u) \cup L_{in}(z) \cup \{z\}$;
- 11 $L_{in}(u) = \emptyset$;
- 12 **for** each $w \in C_{in}(u)$ in ascending order of $l(w)$ **do**
- 13 **if** $l(w) < l(u)$ and $L_{out}(w) \cap L_{in}(u) = \emptyset$ **then**
- 14 $\perp$ add w into $L_{in}(u)$;
- 15 **for** each $s \in I_{out}(w)$ **do**
- 16 **if** $u \in L_{out}(s)$ **then**
- 17 $\perp$ remove u from $L_{out}(s)$;
- 18 identify the set $B^-(v)$ of vertices that can reach v , using a BFS from v that follows the incoming edges of each vertex ;
- 19 **for** each $u \in B^-(v)$ in descending order of $o(u)$ **do**
- 20 let $N_{out}(u)$ be the set of out-neighbors of u ;
- 21 $\perp$ repeat Lines 8-17 with subscripts “in” and “out” exchanged ;
- 22 **return** the label sets of all vertices except v ;

Consider each vertex x which is an updated in-label of u (i.e. $x \in L'_{in}(u)$), by the Reachability Constraint and the Path Constraint in Definition 5.1, we know there is a path from x to u in G' and x is the highest level vertex on the path. Let z be the adjacent vertex of u on the path, we can infer that either $x = z$ or x is also an updated in-label of z (i.e., $L'_{in}(z)$). To see this, suppose $x \neq z$ and x is not in $L'_{in}(z)$, we know the Path Constraint must be violated, since x can reach z and has higher level than z . This means that there is a path P from x to z that contains a higher level vertex than x . Then by extending P in the end of z to u , we form a path P' which connects x to u and contains a higher level vertex than x . This contradicts to the Path Constraint regarding the fact that x is in $L'_{in}(u)$, which shows that x is also in $L'_{in}(z)$ or $x = z$. Hence, we can see that for any vertex x that is in $L'_{in}(u)$, it is either an in-neighbor of x or an

in-label of an x 's in-neighbor. Accordingly, the pseudo code in Lines 8-10 constructs the candidate set $C_{in}(u)$ by combining all the in-neighbors of u as well as the in-labels of these in-neighbors together, and since we compute $L'_{in}(u)$ in the ascending order of $o(u)$, we ensure the in-label sets of the in-neighbors of u are updated before the construction of $C_{in}(u)$, which guarantees that $C_{in}(u)$ is a super set of $L'_{in}(u)$.

Now for each vertex w in $C_{in}(u)$, we add it into $L'_{in}(u)$ if $l(w) < l(u)$ and $L_{out}(w) \cap L'_{in}(u) = \emptyset$ which guarantee the Level Constraint and the Path Constraint respectively. Since the Reachability Constraint is already confirmed in the construction of the candidate set, we know adding w to $L'_{in}(u)$ does not harm the TOL properties. However, some existing labels related to w and u may become redundant. According to Definition 5.1, neither the Level Constraint nor the Reachability Constraint will be affected for an existing label, and the only violation is on the Path Constraint. In particular, if w is also an out-label of a vertex s , then if u is in the out-label of s , we know u violates the Path Constraint, since there is a path from s to u with w on it, and w has higher level than u . On the other hand, the case that s is an in-label of u never happens, since we are adding w into $L'_{in}(u)$ in ascending order of $l(w)$.

Summarizing the above discussion, for each vertex u , we first generate a super set of $L'_{in}(u)$, and then construct the $L'_{in}(u)$ by adding vertices from the candidate set. In particular, for each w that is added into $L'_{in}(u)$, we remove all the redundant labels which are caused by w on the fly. As such, after we create all the in-label sets for vertices in B^+ (Lines 5-17) and out-label sets for vertices in B^- (Lines 18-21), we finally generate a TOL on G' . $\square$

Complexity Analysis. To analyze our algorithm, we consider the complexity step by step. In Step 1, given a vertex v for insertion, Algorithm 3 inspects all the vertices $u \in L'_{in}(v) \cup L'_{out}(v)$, and derives Δ_k by computing all vertices $w \in I_{in}(u) \cup I_{out}(u)$, each with one set operation, i.e., Lines 8 and 10. Let β be the cost of one set operation, then the complexity of Step 1 is bounded by $O(|V|^2\beta)$. In Step 2.1, Algorithm 1 incurs $|C_{in}(v)| + |C_{out}(v)|$ number of set operation in Line 6, therefore, its cost is bounded by $C_2 = O((|C_{in}(v)| + |C_{out}(v)|)\beta)$. In Step 2.2, Algorithm 2 performs the set operations for each pair of vertices in the worse case, resulting a complexity of $O(|V|^2\beta)$. Note that, Lines 9-14 inspects the vertices in $I_{in}(u) \cap L_{in}(x')$ and $I_{out}(u) \cap L_{out}(y')$, which can be implemented in set operations as well. Hence, the complexity for our entire insertion algorithm is $O(|V|^2\beta)$.

5.5 Iterative Label Reduction

The update algorithms in Section 5.4 retains the level order on the vertices in G , which, as mentioned, helps ensure that the performance of $\mathcal{L}$ does not significantly degrade after

updates. As the initial level order l is retained during updates, however, it is essential that l is chosen carefully. Otherwise, if l renders $\mathcal{L}$ inefficient, then this inefficiency is likely to persist even after updates.

A straightforward solution to choose the initial l is to enumerate all possible level orders (i.e., all permutations of vertices in V), then construct a TOL index based on each order, and finally select the one that optimizes performance. However, this approach is far from practical due to the enormous number of possible level orders. As an alternative solution, one may select an initial level order l using some heuristic approach (e.g., using the existing TOL instantiations [22, 51, 94]), and then adjusts the level order l to improve $\mathcal{L}$. Interestingly, our update algorithms can be utilized for such adjustments of l .

Specifically, given $\mathcal{L}$, we can first remove a vertex v using Algorithm 4, and then insert v back using the insertion algorithm in Section 5.4.1. By the properties of the insertion algorithm, when v is re-inserted, its level $l(v)$ is set to a value that minimizes $|\mathcal{L}|$, i.e., the total size of the label sets in $\mathcal{L}$. Therefore, $|\mathcal{L}|$ is likely to decrease (and will never increase) after the *deletion and re-insertion* of v . By repeating this process for each vertex v , we can obtain an improved version of $\mathcal{L}$ with a (much) reduced total size. This decrease in $|\mathcal{L}|$ not only reduces space consumption, but also improves query efficiency (as $\mathcal{L}$ process queries by scanning label sets). In Section 5.7, we experimentally show that this label reduction approach can significantly enhance the performance of existing TOL instantiations [22, 51, 94].

5.6 Construction of Initial $\mathcal{L}$

Although the label reduction algorithm improves the performance of $\mathcal{L}$, we observe from our experiments that it incurs substantial computation costs on large graphs. This motivates us to investigate more efficient methods for choosing a good initial level order l . In the following, we first present a new algorithm for construct an initial l (Section 5.6.1), and then discuss the construction of $\mathcal{L}$ given l (Section 5.6.2).

5.6.1 Deciding Vertex Level

Given a G and a vertex $v \in V$, let $S_{in}(v, G)$ be the set of vertices that can reach v in G , and $S_{out}(v, G)$ be the set of vertices that v can reach in G . Suppose that we set the level of v higher than the level of any vertex in $S_{in}(v, G) \cup S_{out}(v, G)$. Then, in the corresponding TOL index, we need to add v to the in-label sets of the vertices in $S_{in}(v, G)$, as well as the out-label set of the vertices in $S_{out}(v, G)$. In that case, v contributes $|S_{in}(v, G)| + |S_{out}(v, G)|$ labels in $\mathcal{L}$.

On the other hand, if we set the level of v to a lower level than all vertices in $S_{in}(v, G) \cup S_{out}(v, G)$, then we need to (i) add each vertex in $S_{in}(v, G)$ into $L_{in}(v)$, and (ii) add each

vertex in $S_{out}(v, G)$ into $L_{out}(v)$. Furthermore, if v happens to be the only vertex that connects vertices in $S_{in}(v, G)$ to those in $S_{out}(v, G)$, then in the worst case, we have to add every vertex in $S_{out}(v, G)$ to the out-label set of every vertex in $S_{in}(v, G)$. In that scenario, v contributes $|S_{in}(v, G)| \cdot |S_{out}(v, G)| + |S_{in}(v, G)| + |S_{out}(v, G)|$ labels in $\mathcal{L}$.

We define $|S_{in}(v, G)|$ and $|S_{out}(v, G)|$ as the *in-score* and *out-score* of v , respectively. In addition, we define a *score function* f as follows:

$$f(v, G) = \frac{|S_{in}(v, G)| \cdot |S_{out}(v, G)| + |S_{in}(v, G)| + |S_{out}(v, G)|}{|S_{in}(v, G)| + |S_{out}(v, G)|}.$$

In the pathological case when $|S_{in}(v, G)| + |S_{out}(v, G)| = 0$, we define $f(v, G) = 0$. Intuitively, if $f(v, G)$ is large, then v should be given a higher order than the vertices in $S_{in}(v, G) \cup S_{out}(v, G)$, so as to avoid the worst-case space cost of $|S_{in}(v, G)| \cdot |S_{out}(v, G)| + |S_{in}(v, G)| + |S_{out}(v, G)|$.

Based on the above intuition, we can design an algorithm to derive a good level order l as follows. Given G , we first identify the vertex v_1 that maximizes $f(v_1, G)$, and then set $l(v_1) = 1$, i.e., we assign v_1 to the highest level. After that, we remove v_1 from G , and proceed to identify the vertex v_2 that maximizes $f(v_2, G)$ in the modified G , then set $l(v_2) = 2$. We repeat this process until all vertices are removed from G , i.e., until each vertex is given a level.

Although the above algorithm is intuitively, it is difficult to implement efficiently, as (i) the computation of $f(v, G)$ requires us to derive the in-score and out-score of v using BFS (or DFS) on G , and (ii) the in-score and out-score of a vertex need to be re-computed whenever another vertex is removed from G . To address this deficiency, we propose to *approximate* the in-scores and out-scores of the vertices in G .

Let $S_{in}^\top(v)$ and $S_{out}^\top(v)$ be the approximate in-score and out-score of a vertex v , respectively. For each vertex w in G with no in-neighbor (resp. out-neighbor), we set $S_{in}^\top(w) = 0$ (resp. $S_{out}^\top(w) = 0$) to zero; note that this is also the exact in-score (resp. out-score) of w . After that, based on those vertices w , we recursively compute the approximate in-score and out-score of each remaining vertex v as follows:

$$\begin{aligned} S_{in}^\top(v) &= \begin{cases} \sum_{u \in N_{in}(v)} (S_{in}^\top(u) + 1), & \text{if } N_{in}(v) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases} \\ S_{out}^\top(v) &= \begin{cases} \sum_{u \in N_{out}(v)} (S_{out}^\top(u) + 1), & \text{if } N_{out}(v) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

where $N_{in}(v)$ and $N_{out}(v)$ denote the sets of in-neighbors and out-neighbors of v , respectively. It can be verified that $S_{in}^\top(v)$ and $S_{out}^\top(v)$ are upperbounds of v 's in-score and out-score, respectively. As an alternative solution, we also consider using a lowerbound

of v 's in-score (resp. out-score), denoted as $S_{in}^\perp(v)$ (resp. $S_{out}^\perp(v)$), for approximation. In particular, $S_{in}^\perp(v) = 0$ if v has no in-neighbor, and $S_{out}^\perp(v) = 0$ if v has no out-neighbor. For any other vertex, we have

$$\begin{aligned} S_{in}^\perp(v) &= \begin{cases} \sum_{u \in N_{in}(v)} \frac{S_{in}^\perp(u)+1}{|N_{out}(u)|}, & \text{if } N_{in}(v) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases} \\ S_{out}^\perp(v) &= \begin{cases} \sum_{u \in N_{out}(v)} \frac{S_{out}^\perp(u)+1}{|N_{out}(u)|}, & \text{if } N_{out}(v) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

Note that we can compute $S_{in}^\top(v)$ for all vertices v in G , using a linear scan of the vertices. Specifically, we inspect the vertices v in ascending order of their topological ranks $o(v)$ (see Section 5.1). As G is a DAG, the vertices with the smallest ranks must have no in-neighbors, and hence, we have $S_{in}^\top(u) = 0$ for any of those vertices. By the definition of topological ranks, when we inspect any other vertex w , the $S_{in}^\top$ values of w 's in-neighbor must have been computed. Therefore, $S_{in}^\top(w)$ can be easily derived. In summary, we can compute $S_{in}^\top(v)$ for all vertices v in $O(|V| + |E|)$ time. The same algorithm can be easily extended to compute $S_{in}^\perp(v)$. Meanwhile, we can derive $S_{out}^\top(v)$ and $S_{out}^\perp(v)$ for all vertices v , using a linear scan of the vertices in G in descending order of their topological ranks.

5.6.2 Labeling Algorithm

As discussed in Section 5.3, a level order l uniquely decides a TOL index $\mathcal{L}$. In this section, we introduce an algorithm for constructing $\mathcal{L}$ given a level order l computed using the methods in Section 5.6.1. Similar approaches have been proposed in [8, 51] for specific instantiations of TOL (with specific level orderings), but the correctness analysis therein is not immediately applicable under the general TOL framework. Therefore, we present our algorithm and analysis for the sake of completeness.

Algorithm. Algorithm 5 presents our method (referred to as *Butterfly* for constructing $\mathcal{L}$ given a level order l on G . The algorithm first creates a copy of G (referred to as G_1), and then it runs in $|V|$ iterations. In the k -th iteration, it removes from G_k the vertex v with $l(v) = k$, and inserts v into the label sets of other vertices. In particular, we first obtain the set $B^+(v)$ of vertices in G_k that v can reach, using a BFS from v that follows the outgoing edges of each vertex. Then, for each vertex $w \in B^+(v)$, if $L_{out}(v) \cap L_{in}(w) = \emptyset$, then we add v into $L_{in}(w)$. After that, we perform a BFS on G_k from v following the incoming edges of each vertex, to identify the set $B^-(v)$ of vertices that can reach v in G_k . For each $u \in B^-(v)$, we insert v into $L_{out}(u)$ if $L_{out}(u) \cap L_{in}(v) = \emptyset$. At the end of the iteration, we remove v from G_k , and denote the resulting graph as G_{k+1} . After

Algorithm 5: BUTTERFLY

input : G and a level order l
output: a TOL index $\mathcal{L}$

```

1 let  $G_1 = G$  ;
2 for  $k = 1, \dots, |V|$  do
3     let  $v$  be the vertex whose level is  $k$  ;
4     identify the set  $B^+(v)$  of vertices that  $v$  can reach in  $G_k$ , using a BFS from  $v$  that
       follows the outgoing edges of each vertex ;
5     identify the set  $B^-(v)$  of vertices that can reach  $v$  in  $G_k$ , using a BFS from  $v$  that
       follows the incoming edges of each vertex ;
6     for each vertex  $u$  in  $B^+(v)$  do
7         if  $L_{out}(v) \cap L_{in}(u) = \emptyset$  then
8             add  $v$  to  $L_{in}(u)$  ;
9     for each vertex  $u$  in  $B^-(v)$  do
10        if  $L_{out}(u) \cap L_{in}(v) = \emptyset$  then
11            add  $v$  to  $L_{out}(u)$  ;
12    remove  $v$  and  $G_k$  and denote the resulting graph as  $G_{k+1}$  ;
13 return the label sets of all vertices in  $G$  ;
    
```

that, we proceed to the $(k + 1)$ -th iteration. Once all iterations are finished, Algorithm 5 returns the label sets constructed, which form a TOL index $\mathcal{L}$ on G .

Correctness and Complexity. We prove the correctness of Algorithm 5 by the following lemma.

Lemma 5.5. *Given a DAG $G = (V, E)$ and a vertex level, Algorithm 5 outputs a TOL index of G .*

Proof. By contradiction, assume that the output of Algorithm 5 is not a TOL index, i.e., there exist two vertices u and v such that (i) u is in the in-label or out-label of v and (ii) u violates Level Constraint or Path Constraint. We first discuss the case if u is in the in-label of v , i.e., $u \rightarrow v$, and the same discussion can be extended to the alternative case.

If u does not satisfy the Level Constraint, then $l(u) > l(v)$. In the k -iteration of Algorithm 5, let w be the vertex whose level is k , which is the highest level in G_k . Besides we only add w to the labels of the other vertices in G_k , i.e., if w is in a label of a vertex w' , we have $l(w) < l(w')$, which contradicts the assumption.

On the other hand, if u does not fulfill Path Constraint, i.e., there exists a vertex w in the path from u to v or from v to u such that the level of w is higher than the levels of u and v . Consider that there may exist several such paths, we let w be the vertex

with highest level in all these paths. Since the level of w is higher than the levels of u and v , w is removed before the removals of u and v . Due to that (i) u can reach w and w can reach v , and (ii) there does not exist a vertex w' in the path from u to w and from w to v such that level of w' is higher than w , therefore, Algorithm 5 adds w to the out-label of u and the in-label of v . When adding u to v 's in-label, Algorithm 5 inspects the intersection of the out-label of u and the in-label of v , which is not empty. As such, Algorithm 5 avoid adding u to the in-label of v , which contradicts the assumption. $\square$

We now discuss the complexity of Algorithm 5. Let $G_k = (V_k, E_k)$ be the input graph to the k -th iteration, and denote the vertex with highest vertex level in G_k by v . The complexity of forward and backward BFS is bounded by $|E_k|$, and total size of $B^-(v)$ and $B^+(v)$ is bounded by $|V_k|$. Besides, the size of in-label or out-label of any vertex at k -th iteration is at most k since only the vertices whose levels are higher than v are added the current labels. As such, the complexity of each operation, intersection or addition, can be bounded by $O(k)$. In sum, we bound the complexity at k -iteration as $O(|E_k| + k|V_k|)$. Note that, as we remove a vertex from G_k at the end of the k -th iteration, the graph for computing becomes smaller and smaller, that is, the complexity at the k -th iteration decreases significantly when k increases.

5.7 Experiments

This section experimentally evaluates our solution against the state of the art. We implement our algorithms in C++, and we adopt the C++ implementations of all competitors provided by their authors. All of our experiments are conducted on a machine with an Intel Xeon 2.4GHz CPU and 48GB RAM, running Ubuntu 12.4. In each experiment, we measure the performance of each method for 5 times, and we report the average measurement. If a method requires more than 24 hours or more than 48GB RAM to preprocess a dataset D , we omit the method from the experiments on D .

Datasets and Queries. Table 5.3 shows the datasets used in our experiments. Among them, RG5, RG10, RG20, and RG40 are synthetic DAGs generated using the method in experiments in [22], varying the average degree of vertices from 5 to 40, setting the *topological level* to 8 (see [22] for details). The other 7 datasets are the largest DAGs that have been adopted in the literature. In particular, *uniprot22m*, *uniprot100m*, *uniprot150m*, *wiki*, *citeseerx*, *go-uniprot*, and *patents* are from [22, 51], while *GovWild*, *Yago2*, *Twitter*, and *Web-UK* are used in [82].

On each dataset G , we generate a set of 10^6 reachability queries. In particular, we first derive a topological order on the vertices in G . Then, for each query q , we randomly select two vertices from G , and we choose the vertex with lower (resp. higher) topological rank

as the source (resp. terminal) vertex s (resp. t). This method of query generation ensures that none of the queries can be answered by trivially checking whether the terminal vertex has a lower topological rank than the source vertex.

For each experiment on updates on G , we randomly remove 10^4 vertices one by one from G , and measure the average deletion time of each method. After that, we insert the deleted vertices back into G , in reverse order of their removal. During this process, we evaluate the average insertion time of each algorithm.

Experiments on Dynamic Graphs. Our first set of experiments evaluates our solution against existing techniques for dynamic graphs. As mentioned in Section 5.2, there exist a few methods [18, 29, 46, 50, 73, 80, 96] for handle updates on reachability indices. Among them, [29, 46, 73] are shown to be restricted to small graphs with at most a few thousand vertices [58], while [80] only handles XML graphs. We test the remaining methods, and find that only Dagger [96] is able to run on more than one datasets in our experiments. Therefore, we choose Dagger as our competitor. In addition, we evaluate two versions of our solution, namely, *Butterfly-U* (BU) and *Butterfly-L* (BL), such that BU adopts $S_{in}^\top$ and $S_{out}^\top$ as its score functions, and BL adopts $S_{in}^\perp$ and $S_{out}^\perp$ (see Section 5.6).

Figure 5.2 shows the average insertion time of BU, BL, and Dagger. Observe that BU is more efficient than Dagger in almost all cases. In particular, on Twitter, and BU’s insertion time is lower than that of Dagger by four orders of magnitude. Meanwhile, BL is evidently less efficient than BU, although it still outperforms Dagger on the majority of the datasets. (Note that we omit BL on RG40, as it incurs excessive memory consumption on the graph.) The performance gap between BU and BL indicates that the vertex ordering adopted by BU is superior to that by BL.

Figure 5.3 illustrates the total query time (for processing 10^6 queries) of BU, BL, Dagger, as well as a simple baseline approach. In particular, given a reachability query q on a graph G , the baseline approach performs a BFS from the source vertex of q (following the outgoing edges of each vertex), as well as a BFS from the terminal vertex of q (following the incoming edges of each vertex). The two BFSs take turns to traverse the vertices in G , until a common vertex is visited by both BFSs (i.e., when a path from the source vertex to the terminal vertex is found). As shown in Figure 5.3, BU consistently outperforms Dagger and the BFS approach, and BL’s query time is slightly worse than BU’s in general. On the other hand, Dagger is only slightly better than the BFS approach on most datasets, and is more than 700 times slower than the latter on Twitter. This shows that Dagger is *not* a favorable approach for handling updates on dynamic graphs, as it incurs significantly higher update overheads than the BFS approach without providing substantially better query performance. (Note that the BFS approach entails zero update costs as it does not maintain any index.)

Finally, Figure 5.4 shows the average deletion time of each method. BU and BL’s performance is generally comparable to Dagger’s, except on RG40. The slightly inferior

dataset	$ V $	$ E $	avg. deg.
RG5	1.0M	5.0M	5.00
RG10	1.0M	10.0M	10.00
RG20	1.0M	20.0M	20.00
RG40	1.0M	40.0M	40.00
uniprot22m	1.6M	1.6M	1.00
uniprot100m	16.1M	16.1M	1.00
uniprot150m	25.0M	25.0M	1.00
wiki	2.3M	2.3M	1.01
Twitter	16.6M	18.4M	1.10
Yago2	16.1M	25.7M	1.59
Web-UK	20.4M	37.8M	1.85
citeseerx	6.3M	14.8M	2.36
GovWild	8.0M	23.7M	2.95
patent	3.7M	15.7M	4.27
go-uniprot	7.0M	34.8M	4.99

Table 5.3: Datasets ($M = 10^6$).

Figure 5.2: Average insertion time on dynamic graphs.

performance of BU on deletion is justified by its superior efficiency on insertions and queries.

Experiments on Static Graphs. Our second set of experiments compares BU and BL with three state-of-the-art methods for static graphs, namely, *TF-label (TF)* [22], *hierarchical labeling (HL)* [51], *distribution labeling (DL)* [51]. For completeness, we also include Dagger in the experiments. Figure 5.5 illustrates the space consumption of each method. Observe that BU and BL generally outperform both TF and DL. To explain, recall that BU, BL, TF, and DL are all instantiations of the TOL framework. As such, their performance are *solely* decided by the vertex orderings that they adopt. The vertex ordering in DL (resp. TF), however, simply ranks vertices based on their degrees (topological ranks). In contrast, both BU and BL rank vertices based on advanced score functions that take into account the characteristics of the input graphs. As such, the vertex orderings employed by BU and BL are superior to those by TF and DL, and hence,

Figure 5.3: Total query time on dynamic graphs.

Figure 5.4: Average deletion time on dynamic graphs.

lead to smaller index sizes. In particular, on RG10, the space overhead of BU is 5 times smaller than that of DL. Meanwhile, the space cost of HL is always higher than that of DL, which is consistent with the experimental results in [51].

Figure 5.6 illustrates the preprocessing time of each method. The relative performance of BU, BL, TF, HL, and DL are similar to the case of Figure 5.5, since a smaller index size indicates fewer labels in the label sets, and hence, the construction cost for the label sets is generally smaller. We omit HL and DL on RG20 and RG40, since their memory consumptions on those graphs exceed 48GB. On RG10, the preprocessing costs of HL, DL, and TF are at least an order of magnitude higher than that of BU, which indicates that the former are more sensitive to the average degree of the input graph.

Figure 5.7 plots each algorithm’s total query time for 10^6 random queries. Again, BU and BL consistently outperform TF and DL, due to their improved vertex orderings. In particular, on RG10, the query time of BU is 4 (resp. 7) times lower than that of TF (resp. DL). Meanwhile, the query cost of HL is comparable to that of DL.

Experiments on Label Reduction. Our last set of experiments evaluates the label reduction approach presented in Section 5.5. Specifically, we first use DL and TF to construct reachability indices $\mathcal{L}$ on each graph, and then apply our label reduction algorithm on $\mathcal{L}$ to obtain an improved index $\mathcal{L}^*$. Then, we measure the difference $\Delta_{\mathcal{L}}$ between the sizes of $\mathcal{L}^*$ and $\mathcal{L}$, and we divide $\Delta_{\mathcal{L}}$ by the size of $\mathcal{L}$ to derive the ratio of space reduction. Observe that the ratio of space reduction is up to 81.61% and 96.23% for DL and TF, respectively. This demonstrates the effectiveness of our label reduction approach. On the other hand, the label reduction process may incur significant computation overheads on certain graphs. This indicates that one should not overly rely on the label reduction

Figure 5.5: Index sizes on static graphs.

Figure 5.6: Preprocessing time on static graphs.

Figure 5.7: Total query time on static graphs.

approach to improve the performance of a TOL index, but should adopt a good initial vertex order (e.g., the ones adopted by BU and BL). Finally, we note that no result is presented for TF on RG10, since the label reduction process takes excessive time on the graph.

5.8 Summary

This chapter presents a novel study on point-to-point reachability queries on *large dynamic graphs*. We propose general and efficient algorithms for processing vertex insertions and deletions on reachability indices, and we show that our algorithms can also be used to improve the performance of existing techniques for static graphs. In addition, we devise a new algorithm for constructing an efficient reachability index on an input graph

	DL			TF		
	$\Delta_{\mathcal{L}}$ (MB)	$\Delta_{\mathcal{L}}/ \mathcal{L} $	time (s)	$\Delta_{\mathcal{L}}$ (MB)	$\Delta_{\mathcal{L}}/ \mathcal{L} $	time (s)
RG5	98.91	51.94%	986.3	877.22	82.79%	2.9K
RG10	1.63K	81.61%	98.9K	—	—	—
RG20	—	—	—	—	—	—
RG40	—	—	—	—	—	—
uniprot22m	0.00	0.00%	0.1	0.00	0.01%	0.3
uniprot100m	0.71	0.96%	3.6	2.26	3.89%	5.5
uniprot150m	3.14	2.46%	201.1	4.77	10.59%	324.2
wiki	0.00	0.00%	0.1	0.01	0.74%	0.3
Twitter	0.00	0.00%	3.7	0.02	8.41%	6.5
Yago2	0.00	0.00%	18.9	0.00	0.00%	18.8
Web-UK	38.69	21.80%	456.3	820.17	47.02%	12.7K
citeseerx	9.14	13.95%	773.2	192.71	96.23%	1.5K
GovWild	20.69	16.24%	844.1	2.40K	94.62%	59.3K
patent	255.09	48.92%	3.0K	3.69K	93.27%	186.5K
go-uniprot	53.24	26.91%	2.9K	104.61	66.38%	5.4K

Table 5.4: Performance of label reduction ($K = 10^3$).

from scratch. We evaluate our solution on a large set of real graphs, and we demonstrate that our solution not only supports efficient updates on large dynamic graphs, but also provides even better query performance than the state-of-the-art techniques for *static* graphs. To our knowledge, we are the first in the literature to present a reachability index that can efficiently handle updates while offering superior query performance.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

Graph query processing plays an important role in graph-based applications, for example, map, navigation, and location-based services on road networks; data mining, structure analysis on web graphs; relation evaluation, graph pattern analysis on social networks, just name a few. Among the widely used queries on graphs, we figured out that the point-to-point shortest path and distance queries, the single source shortest path and distance queries, and the point-to-point reachability queries are the fundamental building blocks for numerous graph-based applications. Due to the rapidly increasing size of the graphs where these applications are relied on, the efficient processing of the aforementioned queries becomes a significant challenge for computer science and engineering nowadays. In Chapter 2, we conducted a survey on the application field for the above queries, along with the state-of-the-art methods, and then, we investigate three most challenging problems, namely, the point-to-point shortest path (PPSP) and distance (PPD) queries on road networks, the single source shortest path (SSSP) and distance (SSD) queries on disk-resident graphs, and the point-to-point reachability (PPR) queries on dynamic graphs.

For point-to-point shortest path and distance queries on road networks, we propose Arterial Hierarchy (AH), a worst-case efficient index structure for point-to-point shortest path and distance queries. On the theoretical side, under a practical assumption on the road network, AH offers superior query time complexities in both shortest path and distance queries, and its space and preprocessing time complexities are comparable to the best existing worst-case efficient methods. On the practical side, with extensive experiments on realworld road networks, we show that AH also provides excellent query efficiency in practice, and it even outperforms CH (*i.e.*, the state-of-the-art heuristic method) in terms of query time.

For single source shortest path and distance queries on disk-resident graphs, we propose HoD, which supports both directed and undirected graphs for both SSSP and SSD queries under memory-constrained environments. This contrasts the existing methods, which either (i) require that the dataset fits in the main memory during precomputation and/or query processing, or (ii) support only undirected graphs. With extensive experiments on a variety of realworld graphs, we demonstrate that HoD significantly outperforms the state of the art in terms of query efficiency, space consumption, and pre-computation costs.

For the point-to-point reachability query, we introduce the Total Order Labeling (TOL) framework, which summarizes three most advanced methods [22, 51, 94] for reachability queries on static graphs. Then we propose general and efficient algorithms for processing vertex insertions and deletions on TOL, and we show that our algorithms can also be used to improve the performance of existing TOL instantiations. In addition, we devise a new algorithm for constructing an efficient TOL index on an input graph from scratch. We evaluate our solution on a large set of real graphs, and we demonstrate that our solution not only supports efficient updates on large dynamic graphs, but also provides even better query performance than the state-of-the-art techniques for static graphs. To our knowledge, we are the first in the literature to present a reachability index that can efficiently handle updates while offering superior query performance.

6.2 Future Directions

In this section, we discuss two directions for the future work of the problems studied in this thesis. The first direction is to extend our solutions to handle corresponding queries on dynamic graphs efficiently. (For reachability queries, we focus on the efficiency since our solution in Chapter 5 already handles dynamic graphs) This is motivated by the trend of rapid growth in the real-time applications, such as real-time navigation systems, real-time friend suggestion applications and real-time public opinion analysis. Because real-time applications provide up-to-date results, which are more accurate and closer to reality, and thus, fulfill the real-time requirements of today's users. The second direction is to extend our solutions for parallel processing on distributed systems. This is based on the fact that more and more large graphs are stored in distributed systems nowadays, since their volume can easily exceed the memory limit of a single machine. At the same time, parallel processing is also the basic for online applications, since massive worldwide users have to be served simultaneously through the Internet. In the following, we will discuss the two interesting future directions respectively.

6.2.1 Efficient Updates on Dynamic Graphs

The point-to-point shortest path and distance queries on dynamic road networks. In contrast to the static road networks we considered in this thesis, dynamic road networks assign each edge a weight that changes with time, which simulates the real-time traffic condition more accurately. The dynamic weight can be computed by the signals collected from the embedded sensors on the roads. So the first problem is how to answer *PPSP or PPD queries on the dynamic road networks*. The challenge is, almost every efficient existing methods in handling PPSP and PPD queries on road networks, including AH, rely on the pre-computed auxiliary information. However, when handling the PPSP or PPD query on dynamic road networks, the auxiliary information is often out-dated when the query is received. Hence, a decent solution for this problem should be able to update the pre-computed information in a fast and sustainable manner. To extend AH on dynamic road networks, the basic idea is to mark each shortcut with additional labels, which point to the edges in the road network, whose weight change affects the length of the shortcut. Consequently, reducing the number of such additional labels is the key to guarantee both the practical efficiency and the theoretical bound of AH. To address this issue, we will examine the relationship between the shortcut and the original edges in the road network carefully, and modify the shortcut generation process accordingly to work out an efficient solution for PPSP and PPD queries on dynamic graphs in the future.

Furthermore, in dynamic road networks, a previously computed shortest path, where a user is driving on, may not be the shortest path any more due to the change of the traffic condition. As a result, the system should offer the user up-to-date shortest paths during his/her journey, which leads to another problem, *i.e.*, how to handle *adaptive PPSP queries on dynamic road networks*. Specifically, in the server-client model, the difficulty for the problem is, how to detect the affected queries, so that the client can avoid submitting continuous queries, which not only wastes the energy of the client, but also increases the burden of the server.

The single source shortest path and distance queries on dynamic disk-resident graphs. Different from the weight change in the dynamic road networks, the dynamic nature of the disk-resident graphs is on the graph structure (*i.e.*, insertion and deletion of nodes and edges). This is because most of the disk-resident graphs are unweighted graphs, such as, Web graphs, social networks and semantic graphs. To extend HoD for handling the updates on disk-resident graphs, there are two problems need considering. The first one is how to efficiently *distinguish the affected shortest paths* in the graph when an update incurs. This can be done by employing Dijkstra's traversals starting from the involved nodes of the update, and additional labels should be precomputed to avoid visiting unnecessary nodes during the traversal. The second one is how to *update the disk-resident index* with as few times of linear scans as possible. This involves ordering and

organizing the updates into batches according to the dependency relationships between the shortcuts, and additional triggers can be implanted into the index to speed up the linear scanning process by jumping through unaffected shortcuts. In summary, these two problems are crucial in extending HoD for dynamic disk-resident graphs, as they determine the major CPU and I/O costs of the updating process, and we will try to find out specific solutions in the future.

The point-to-point reachability queries on dynamic graphs. The dynamic nature of the graphs considered by reachability queries is the same as that of disk-resident graphs (i.e., insertion and deletion of nodes and edges), because the edge weight is also meaningless in reachability queries. Recall that the update algorithms presented in Section 5.4 focus on the node insertion and deletion in the processed graph. (Recall that each strongly connected component (SCC) in the original graph is concentrated into a node in the processed graph) Hence, to support updates in the original graph, we first derive the algorithms for edge updates in the processed graph directly from the node update algorithms in Section 5.4. Then, we utilize the dynamic SCC maintenance method in [96] to transfer each update on the original graph into a sequence of updates on the processed graph. As such, we correctly handle each update on the original graph by processing the corresponding sequence of updates on the processed graph. To further improve the efficiency, on one hand, we will consider the level adjustment in the edge insertion and deletion algorithms, which aims to minimize the index size after the update; on the other hand, we will develop our own SCC maintenance algorithm, which will cooperate better with our update algorithms. By addressing both issues, we look forward to work out a more efficient solution for handling updates on the original graph in the future.

6.2.2 Parallel Processing on Distributed Graphs

The point-to-point shortest path and distance queries on distributed graphs. Recall the index construction algorithm in Section 3.4.2. For each (4×4) -cell region processed, the algorithm only visits the nodes inside the region and edges with at least one endpoint inside the region to derive the next-level nodes and the necessary shortcuts. Moreover, the computation process on a (4×4) -cell region is independent to the computation of any other (4×4) -cell region. This enables an easy extension of AH to parallel processing on distributed systems like Hadoop. In a map-reduce iteration, each peer first runs the index construction algorithm on the regions that has been distributed to it from the previous iteration, since the computation units of different regions are independent to each other. And then, with a proper key coding algorithm, the generated shortcuts are distributed among the peers for the next iteration. As such, each iteration completes the index construction for one level, and pushes the process to the next level. Hence, at most

$\log \alpha$ iterations are needed to finish the preprocessing phase of AH, where $\alpha = d_{max}/d_{min}$, and d_{max} (resp. d_{min}) is the largest (resp. smallest) $\mathcal{L}_\infty$ distance between any two nodes in the road network. As $\log \alpha$ is always less than 30 on the earth, the extended preprocessing algorithm of AH is already quite efficient. However, the query algorithm of AH can hardly be extended to distributed systems directly. The key problem is how to partition the AH index among peers, so that each PPSP or PPD query can be answered with as little peer-wise information exchange as possible. To solve this problem, we will examine the nodes visited by AH for each query carefully, and work out an appropriate partition strategy in the future.

The single source shortest path and distance queries on distributed graphs.

To extend HoD for parallel processing on distributed systems, the challenge comes from both the preprocessing phase and the querying phase. Unlike the road networks that can be partitioned based on the spatial position of each node, general graphs do not contain a partition criterion in nature. Hence, in the preprocessing phase, the first problem is how to partition the graph, so that the load of each computing peer can be balanced, while the collisions among peers can be minimized. The basic idea for this problem is to use the high-degree nodes as the boundary nodes, which has two benefits: (i) in HoD, high-degree nodes are often removed later than low-degree nodes, and thus, using them as the boundary nodes keeps the partition steady during the preprocessing phase; (ii) the baseline edges generated from the high-degree nodes are more targeted, as the baseline edges across different partitions are avoided. This helps to clear redundant shortcuts more effectively. Secondly, how to organize the precomputed index in the distributed system is also challenging. The index organization strategy and the query algorithm should be appropriate to each other, so that an SSD or SSSP query can be answered with the minimum number of index scans over different peers. We will leave the specific algorithms as future work.

The point-to-point reachability queries on distributed graphs. Recall that TOL is defined on the processed graph (i.e., each strongly connected component (SCC) in the original graph is concentrated into a node in the processed graph), and the Butterfly algorithm also constructs a TOL from the processed graph. To handle distributed graphs, we either have to consider how to efficiently compute strongly connected components on distributed graphs, or we must define TOL in the original graph, and extend Butterfly to create the index straightly from the original graph. Furthermore, how to maintain the index on the created labels is also quite challenging, since these labels are fetched nearly in a random way during the labeling process. We will address these issues in the future work.

6.3 Summary

This chapter first concludes this thesis, and then presents several further directions for efficient query processing on large graphs. All the mentioned directions aim to produce better applications on large graphs, and I hope that more and more researchers will join the study of efficient query processing on large graphs and apply it in our daily lives.

Appendix A

Proofs

A.1 Unique Shortest Paths via Weight Perturbation

Let h be as defined in Section 3.3.1. The solutions in Chapter 3 rely on the following assumption:

Assumption A.1. *For any (4×4) -cell region B in the square grid R_i ($i \in [0, h]$), there do not exist two local shortest paths in B that share the same endpoints and have the same length.*

In the section, we show that Assumption A.1 can be enforced by adding a small perturbation to the weight of each edge in the road network G . Specifically, we associate each edge e in G with an integer $\rho(e)$ that is randomly selected in the range $[0, \tau - 1]$, where τ is a parameter to be specified shortly. We refer to $\rho(e)$ as the *nuance* of e , and we define the nuance of a path P as the sum of the nuance of the edges on the path, denoted as $\rho(P)$. For any two path P_1 and P_2 such that $l(P_1) = l(P_2)$, we consider P_1 shorter than P_2 if $\rho(P_1) < \rho(P_2)$. We will establish the following theorem.

Theorem A.1. *Let Δ be the largest degree of any node in G . If $\tau \geq 32hn^3\left(\frac{\Delta}{2}\right)$, and then Assumption A.1 holds with a probability at least $1 - \frac{1}{n}$.*

In other words, by setting τ to a sufficiently large value, we can ensure that Assumption A.1 holds with an overwhelming probability.

Our proof of Theorem A.1 is based on a few lemmas as follows.

Lemma A.1. *Let P and P' be two paths in G . Then, $\rho(P) = \rho(P')$ occurs with at most $1/\tau$ probability.*

Proof. Assume that $P = \langle e_1, \dots, e_i \rangle$ and $P' = \langle e'_1, \dots, e'_j \rangle$. Then,

$$\begin{aligned}
 & Pr \{ \rho(P) = \rho(P') \} \\
 &= Pr \left\{ \sum_{1 \leq k \leq i} \rho(e_k) = \sum_{1 \leq k \leq j} \rho(e'_k) \right\} \\
 &= Pr \left\{ \rho(e_1) = \sum_{1 \leq k \leq j} \rho(e'_k) - \sum_{2 \leq k \leq i} \rho(e_k) \right\} \\
 &= \sum_{0 \leq x \leq \tau-1} \left(Pr \{ \rho(e_1) = x \} \right. \\
 &\quad \left. \cdot Pr \left\{ \sum_{1 \leq k \leq j} \rho(e'_k) - \sum_{2 \leq k \leq i} \rho(e_k) = x \right\} \right) \\
 &= \sum_{0 \leq x \leq \tau-1} \left(\frac{1}{\tau} \cdot Pr \left\{ \sum_{1 \leq k \leq j} \rho(e'_k) - \sum_{2 \leq k \leq i} \rho(e_k) = x \right\} \right) \\
 &= \frac{1}{\tau} \cdot \sum_{0 \leq x \leq \tau-1} Pr \left\{ \sum_{1 \leq k \leq j} \rho(e'_k) - \sum_{2 \leq k \leq i} \rho(e_k) = x \right\} \\
 &= \frac{1}{\tau}
 \end{aligned}$$

□

Let Δ be the maximum degree of any node in G . Based on Lemma A.1, we have the following result:

Lemma A.2. *Let B be a (4×4) -cell region in R_i ($i \in [0, h]$). For a node s in B , let ζ (resp. ζ') be the event that there exists a another node v , such that the local shortest path from s to v (resp. from v to s) in B is not unique. Then, $Pr\{\zeta \vee \zeta'\} \leq \binom{\Delta}{2} \cdot 2n/\tau$.*

Proof. We will prove that $Pr\{\zeta\} \leq \binom{\Delta}{2} \cdot n/\tau$. By symmetry, it can also be shown that $Pr\{\zeta'\} \leq \binom{\Delta}{2} \cdot n/\tau$, leading to $Pr\{\zeta \vee \zeta'\} \leq \binom{\Delta}{2} \cdot 2n/\tau$.

Let $d_s(v)$ be length of the the local shortest path distance from s to v in B . Let $\langle v_0, v_1, \dots, v_k \rangle$ be a permutation of all the nodes that can be reached from s via local paths in B , such that $d_s(v_i) \leq d_s(v_j)$ for any $0 \leq i < j \leq k$. That is, $v_0, v_1, \dots, v_k$ are sorted in a non-decreasing order of their distances from s . Note that $v_0 = s$. Let ζ_i ($i \in [1, k]$) be the event that (i) the local shortest path from s to any v_j ($j \in [0, i-1]$) in B is unique, but (ii) the local shortest path from s to v_i is not unique. We have $Pr\{\zeta_1\} = 0$; otherwise, there must exist another node u such that $d_s(u) \leq d_s(v_1)$,

contradicts the definition of v_i ($i \in [0, k]$). In addition,

$$\begin{aligned} Pr\{\zeta\} &= Pr\left\{\bigcup_{1 \leq i \leq k} \zeta_i\right\} \\ &\leq Pr\{\zeta_1\} + Pr\left\{\bigcup_{2 \leq i \leq k} \zeta_i\right\} \\ &= Pr\left\{\bigcup_{2 \leq i \leq k} \zeta^i\right\}. \end{aligned}$$

Now let us consider $Pr\{\zeta^i\}$ for $i \in [2, n-1]$. Let $\mathcal{P}_{s,v_i} = \{P_1, \dots, P_q\}$ be the set of local shortest paths from s to v_i in B . For each P_j ($j \in [1, q]$), let $\langle u_j, v_i \rangle$ be the last edge on P_j . Then, u_j should be in $\{v_0, v_1, \dots, v_{i-1}\}$. By the definition of ζ_i , the local shortest path from s to u_j in B is unique. Furthermore, $q \leq \Delta$, since the degree of v_i is at most Δ . By Lemma A.1, we have

$$\begin{aligned} Pr(\zeta^i) &\leq \sum_{1 \leq j < k \leq q} Pr\{\rho(P_j) = \rho(P_k)\} \\ &\leq \sum_{1 \leq j < k \leq q} \frac{1}{m} = \frac{1}{m} \binom{q}{2} \leq \frac{1}{m} \binom{\Delta}{2}. \end{aligned}$$

Therefore,

$$Pr\left\{\bigcup_{2 \leq i \leq n-1} \zeta_i\right\} \leq \sum_{2 \leq i \leq n-1} Pr\{\zeta_i\} \leq \frac{n}{\tau} \binom{\Delta}{2},$$

which completes the proof. $\square$

Given Lemma A.2, we prove Theorem A.1 as follows:

Proof. Let s be an arbitrary node in G . For any R_i ($i \in [0, h]$), there exist at most 16 (4×4) -cell regions in R_i that contains s . By Lemma A.2, for each B of those 16 regions, there is at most $\binom{\Delta}{2} \cdot n/\tau$ probability that B contains non-unique local shortest paths between s and another node. Taking in account all possible choices of B in all R_i and all possible choices of s , Assumption A.1 fails with a probability at most

$$\binom{\Delta}{2} \cdot 32n/\tau \cdot h \cdot n = \binom{\Delta}{2} \cdot 32n^2h/\tau.$$

By setting $\tau \geq 32n^3h\binom{\Delta}{2}$, we can guarantee that the above probability is at most $1/n$. Therefore, Theorem A.1 is proved. $\square$

Remark. The above perturbation method requires generating random numbers in the integer range of $[0, \tau - 1]$, which causes practical concerns since $\tau - 1$ can be too large to be represented with a normal integer. We address this issue by using multiple random integers in a relatively narrow range to represent τ . In particular, to generate the nuance for an edge, we can use k random integers in the range of $[0, \tau' - 1]$, where $\tau' = \tau^{\frac{1}{k}}$. Accordingly, the nuance on each edge would be a k -dimensional vector. It can be verified that, under such edge perturbation, the results in this section still hold.

A.2 Proofs of Lemmas 3.1 and 3.2

Recall Lemma 3.1 *Any $(\alpha \times \alpha)$ -cell region in R_i contains $O(\alpha^2 \lambda)$ level- i nodes in H , where λ is the arterial dimension of G .*

Proof. Consider the $(\alpha \times \alpha)$ -cell region A in Figure A.1, as well as the $((\alpha+6) \times (\alpha+6))$ -cell region that is centered at A . The nodes at level i that fall in the A can be divided into two groups: (i) the endpoints of the arterial edges for a region B_1 completely covered by the $((\alpha+6) \times (\alpha+6))$ -cell region, and (ii) the endpoints of the arterial edges for a region (4×4) -cell region B_2 that is disjoint from A .

The first group contains at most $2 \cdot (\alpha + 3)^2 \cdot \lambda$ nodes. This is because (i) there are $(\alpha + 3)^2$ (4×4) -cell regions contained in the $((\alpha+6) \times (\alpha+6))$ -cell area, and (ii) each (4×4) -cell region has λ arterial edges. Meanwhile, all nodes in the second group also appear in the first group. To explain, observe that for any node u in A and any (4×4) -cell region B_2 that is disjoint from A , if u is the endpoint of an arterial edge for B_2 , and then the edge must (i) connect u to a node v in B_2 and (ii) lie on a spanning path P of B_2 . It can be verified that there should exist a (4×4) -cell region B in A , such that B contains u , and P is a spanning path of B , as exemplified in Figure A.2. In that case, the edge between u and v would also be an arterial edge for B . In other words, the node u is also contained in the first group mentioned before. As a consequence, the total number of level- i nodes in A is $2 \cdot (\alpha + 3)^2 \cdot \lambda$, which proves the lemma. $\square$

Recall Lemma 3.2 *Let P be a shortest path in G , such that no (3×3) -cell region in R_i ($i \in [1, h]$) can cover all nodes in P simultaneously. Then, P must contain an arterial edge of some (4×4) -cell region in R_i .*

To prove Lemma 3.2, we will show that there is always a (4×4) -cell region in R_i (denoted as B), such that a sub-path of P (denoted as P') is a spanning path of B . Such B and P' could be found by a *SlidingWindow* algorithm as presented in Figure A.3.

Figure A.1: B_1 is completely in the $(\alpha + 6) \times (\alpha + 6)$ area.

Figure A.2: B_2 is not completely in the $(\alpha + 6) \times (\alpha + 6)$ area.

To explain the *Sliding Window* Algorithm, consider a shortest path $P = \langle v_1, v_2, \dots, v_6, \rangle$ under R_i as shown in Figure A.4. We start with v_1 by in turn scanning each node on P to obtain a minimal bounding region B_θ (shown as the dotted lined rectangle) that covers all the nodes which have been scanned. After v_5 is scanned, B_θ is five cells in width. Then the iteration terminates, and the algorithm finds out a (4×4) -cell region

Algorithm *Sliding Window* ($P = \langle v_1, v_2, \dots, v_k \rangle, R_i$)

1. Let $\mathcal{B}$ be a set that contains any region B consisting of a consecutive block of $x \times y$ cells in R_i ($x, y > 0$).
 2. Initialize $\theta = 0$.
 3. For $i = 1$ to k
 4. $\theta = j$.
 5. Let B_j be the smallest region in $\mathcal{B}$ that covers $v_1, v_2, \dots, v_j$ simultaneously.
 6. if B_j is at least 4 cells in width or height, and then break.
 7. If B_θ is at least 4 cells in width
 8. Let v_α and v_β be the nodes in $\{v_1, v_2, \dots, v_\theta\}$ with the smallest and largest x-coordinates, respectively.
 9. Let B be any (4×4) -cell region in R_i such that (i) B covers $v_1, v_2, \dots, v_{\theta-1}$ simultaneously, and (ii) v_α is in the west strip of B .
 10. Else
 11. Let v_α and v_β be the nodes in $\{v_1, v_2, \dots, v_\theta\}$ with the smallest and largest y-coordinates, respectively.
 12. Let B be any (4×4) -cell region in R_i such that (i) B covers $v_1, v_2, \dots, v_{\theta-1}$ simultaneously, and (ii) v_α is in the south strip of B .
 13. Let $a = \min\{\alpha, \beta\}$, and $b = \max\{\alpha, \beta\}$.
 14. Return B and $P' = \langle v_a, v_{a+1}, \dots, v_b \rangle$.
-

Figure A.3: The *Sliding Window* Algorithm.

 Figure A.4: $P = \langle v_1, v_2, \dots, v_6 \rangle$ in R_i .

B (shown as the solid lined rectangle) that covers $v_1, v_2, \dots, v_4$. Since among the nodes $v_1, v_2, \dots, v_5$, v_2 (resp. v_5) has the smallest (resp. largest) x-coordinate, hence, we can obtain $P' = \langle v_2, v_3, \dots, v_5 \rangle$ as the spanning path of B .

Proof. By the *SlidingWindow* algorithm we can always find a (4×4) -cell region (denoted as B) in R_i , such that a sub-path of P (denoted as P') is a spanning path of B . Whenever such a region B exists, P must contain an arterial edge for B , and hence, the lemma proved. $\square$

A.3 Proofs of Lemmas 3.3 and 3.4

Our proofs of Lemmas 3.3 and 3.4 are based on a series of lemma on the property of AH's preprocessing algorithm. In what follows, we will first revisit AH's preprocessing algorithm, and then elaborate the proofs. Unless otherwise specified, we use the term *region* to refer to a (4×4) -cell region.

Given a road network G , AH selects *level i cores* (the nodes that are at least at level i) in an incremental manner. At the i -th iteration, AH performs two steps: (i) it computes the spanning paths so as to select the level- i cores, and (ii) it adds shortcuts concerning the boarder nodes under R_{i+1} and level- i cores to obtain a reduced graph G'_i for the next iteration.

More specifically, each original edge $\langle u, v \rangle \in G$ is considered an edge generated from a region B , if u is in B . Then, as for each iteration, AH selects level- $(i+1)$ cores in the following manner. First, AH imposes the grid R_{i+1} on G'_i (note that $G'_0 = G$). Then, for each region B in R_{i+1} , AH inspects each sub-graph of G'_i that overlaps with B , denoted as $G'_{i,B}$. For each boarder node s of B , AH invokes Dijkstra's algorithm to start a traversal on $G'_{i,B}$ from s ; for each node u visited, AH follows its outgoing edges $\langle u, v \rangle$ such that (i) v is a level- i core, and (ii) $\langle u, v \rangle$ satisfies the coverage condition. This results in a spanning tree T_s is obtained. Subsequently, for each node u on T_s , AH inspects its outgoing edges $\langle u, t \rangle$ in $G'_{i,B}$, such that (i) $\langle u, t \rangle$ satisfies the coverage condition, and (ii) t is a boarder

node in B , or t is not in B and t is a level- i core. As such, a path from s to t is obtained. Similarly, AH invokes Dijkstra's algorithm to start a traversal from s again, but with the difference that for each node visited, AH follows its incoming edges. We use $\mathcal{P}_{i+1,B}$ to denote the paths thus obtained. We will prove that each path $P \in \mathcal{P}_{i+1,B}$ is a spanning path of B in Lemma A.4, based on Lemma A.3 below.

For convenience, we define a few terms that will be frequently used in our proofs. Let B be a region in R_i and B' be a region in R_j ($i < j$). If B is completely contained in B' , we say that B is a *sub-region* of B' and B' is a *super-region* of B . Let P be a path from a node s to another node t . We say P is *contained* in a region B , if all the nodes on P are in B . Under the grid R_i , we say two nodes s and t are *far-apart* if they are not covered in the same 3×3 cell region.

Lemma A.3. *Let B_{super} be a region in R_j . Let s, t be two nodes in B_{super} . If the shortest path P from s to t contained in B_{super} is not contained in any region in R_i (where $i < j$), and then there is a sub-region of B_{super} , say B , in R_i , that makes P' , a sub-path of P , a spanning path of B . Besides, either (i) P' is contained in B , or (ii) P' is not contained in B . Let v be the last node of P' and u the predecessor of v on P' , and then u and v are not in two adjacent cells under R_i .*

Proof. Similar to Lemma 3.2, we use the same *sliding window* algorithm to find out P' and B . We start with s and in turn scan each node on P . We also use *width* (resp. *height*) to denote the width (resp. height) of the window that tightly covers the scanned nodes on P . Without loss of generality suppose after a node u on P is scanned, $width, height < 4$, and after v , the successor of u on P is scanned, $width \geq 4$ and $width \geq height$. Without loss of generality suppose X_{max} is updated after v is scanned. We consider a square whose four corners are respectively (X_{min}, Y_{min}) , $(X_{min}, Y_{min} + 4)$, $(X_{min} + 4, Y_{min})$ and $(X_{min} + 4, Y_{min} + 4)$ at the time right before v is about to be scanned. Then that square is a region in R_i , and we denote it as B . On the path from s to v , let w be the node whose x_{min} equals to X_{min} . Then the sub-path P' from w to v of P is a spanning path of B .

Now we show that with P' and B defined above, P' satisfies one of the two conditions stated in the lemma. We consider the value of *width* after v is scanned in two cases: (i) $width = 4$ and (ii) $width > 4$. In case (i), as what we assume before, $width \geq height$ after v is scanned. Hence it follows that $height \leq 4$. Therefore, v is in B , which follows that P' is contained in B . In case (ii), v is not in B since $width > 4$ after v is scanned. Hence P' is not contained in B . On the other hand, since $width < 4$ right before v is scanned, and $width > 4$ after v is scanned. Hence u and v are not in two adjacent cells under R_i .

We can also prove the lemma in a similar way when X_{min} is changed after v is scanned, or the height of the rectangle grows to four or larger than four not later the width does.

Finally, we show that such B is a sub-region of B_{super} . If by contradiction B is not a sub-region of B_{super} , and then there should exist a node x on P , such that after x is scanned, B enlarges and part of B is not contained in B_{super} . If x is in B_{super} , and then B should be contained in B_{super} after x is scanned, because the side length of a cell in R_i is smaller than that of R_j , and B enlarges by the side length of a cell in R_i . It follows that in case part of B is not contained in B_{super} after x is scanned, x should be beyond B_{super} , which violates the hypothesis that P is contained in B_{super} . $\square$

Lemma A.4. *Let B be a region under R_i . Then the following statements are true:*

1. *Each $P \in \mathcal{P}_{i,B}$ is a spanning path of B .*
2. *For any $P \in \mathcal{P}_{i,B}$, a path from s to t , either (a) P contains only one edge and s, t are level- i cores, or (b) P contains more than one edge, a node w on P with $w \neq s, t$ is a level- i core, and w is in B .*
3. *Let B_{super} be a region under R_j with $j > i$. Let s, t be two nodes and P the shortest path from s to t contained in B_{super} . If s, t are far-apart in R_i , and then there exists a region B under R_i , where B is a sub-region of B_{super} , such that P covers a path in $\mathcal{P}_{i,B}$.*
4. *Let B_{super} be a region under R_j with $j > i$. Let s, t be two nodes and P the shortest path from s to t contained in B_{super} . If s, t are far-apart in R_i , and then P is covered by V_i . Further, if P contains multiple edges, and then there is a level- i core u on P where $u \neq s, t$.*

Proof. This lemma could be proved by mathematical induction.

For Statement 1, for simplicity, we only consider the paths from the west strip to the east strip of B . The lemma could be proved true for the paths in the other directions in a similar way. Within a region B , the algorithm finds out two types of paths. Suppose P is a path from s to t . Then, either (a) P ends at a boarder node t of B , and t lies in the east strip of B , or (b) t is to the east of B , while t , and the predecessor of t on P , are both level- $(i-1)$ cores. We prove Statement 1 respectively concerning these two types of paths. For ease exposition, we make a slight difference from the algorithm described above: after the level- i cores are selected, we add shortcuts concern all the nodes in G and the level- i cores to obtain G'_i . We will show in Lemma A.5 that if we only add shortcuts concerning the boarder nodes of R_{i+1} and the level- i cores, this proof also works. Furthermore, based on Statement 1, we prove Statements 2, 3, and 4 also hold.

To facilitate our proof, we make the following six claims. Claims 1.1 to 1.3 together prove Statement 1 true. Claims 1.4 to 1.6 are used for induction.

- 1.1. Let u, v be two level- $(i-1)$ cores in B . Then the local shortest path from u to v contained in B could be found by a Dijkstra search on G'_{i-1} within B from u : for each node w visited, the search only relaxes the edges $\langle w, x \rangle$ where x is a level- $(i-1)$ core and $\langle w, x \rangle$ satisfies the coverage condition.
- 1.2. Let u be a node in B and v a level- $(i-1)$ core in B . Then the shortest path from u to v contained in B could be found by a Dijkstra search as described in Claim 1.1.
- 1.3. Let u be a level- $(i-1)$ core in B and v a node in B . Then the shortest path from u to v contained in B could be found by a Dijkstra search which performs in two steps: (i) firstly perform a Dijkstra search as described in Claim 1.1, and (ii) after all the level- $(i-1)$ cores in B reachable from u are visited, a spanning tree from u is created; inspect the edges $\langle w, v \rangle$ where w is on the spanning tree and $\langle w, v \rangle$ satisfies the coverage condition to obtain the shortest path from u to v .
- 1.4. Let u, v be two level- i cores in B . Then the shortest path from u to v contained in B could be found by a Dijkstra search from u : for each node w visited, the search only relaxes the edges $\langle w, x \rangle$ where x is a level- i core and $\langle w, x \rangle$ is generated from B .
- 1.5. Let u be a node in B and v be a level- i core in B . Then the shortest path from u to v contained in B could be found by a Dijkstra search as described in Claim 1.4.
- 1.6. Let u be a level- i core in B and v a node in B . Then the shortest path from u to v contained in B could be found by a Dijkstra search which performs in two steps: (i) firstly perform a Dijkstra search as described in Claim 1.4, and (ii) similar to that of Claim 1.3, inspect the edges $\langle w, v \rangle$ where w is on the spanning tree obtained in (i), and $\langle w, v \rangle$ is generated from B to get the shortest path from u to v .

Our proof is organized as follow. At the beginning, we show all statements and claims are true when $i = 1$. Subsequently, assuming all statements and claims are true when $i = k$, we prove that they also hold when $i = k + 1$. In particular, we first prove that the six claims hold. Given Claim 1.4 and Statements 2 and 4 when $i = k$, we prove that Claim 1.1 holds when $i = k + 1$. Then, given Claim 1.5 (resp. Claim 1.6) when $i = k$ and Claim 1.1 when $i = k + 1$, we show that Claim 1.2 (resp. Claim 1.3) is true when $i = k + 1$. After that, given Claim 1.2 (resp. Claim 1.3) when $i = k$ and Claim 1.4 when $i = k + 1$, we prove Claim 1.5 (resp. Claim 1.6). Finally, given Claims 1.1 to 1.3 when $i = k + 1$, we prove that Statements 1 to 4 hold when $i = k + 1$.

In the following we show that the six claims are true when $i = 1$. Claim 1.1 to 1.3 are true when $i = 1$ given all the nodes in G are level-0 cores, and G'_0 is composed of the edges of G . Claim 1.4 is true when $i = 1$. Suppose that P is a local shortest path from

u to v , and $t_1, \dots, t_j$ are in turns the level-1 cores on P . Then the algorithm would add a shortcut $\langle u, t_1 \rangle$ (or $\langle u, t_1 \rangle$ is an original edge in G) whose length equals to the distance in G'_0 from u to t_1 , and $\langle u, t_1 \rangle$ is generated from B . So are the edges $\langle t_1, t_2 \rangle, \dots, \langle t_j, v \rangle$. Hence, Claim 1.4 holds when $i = 1$.

Claim 1.5 is similar to Claim 1.4, except that the source node u is not necessarily in V_1 . Since the algorithm also adds an edge from u to a level-1 core, similar to the case of Claim 1.4, Claim 1.5 also holds when $i = 1$.

As for Claim 1.6, Suppose P is a local shortest path from u to v , and $t_1, \dots, t_j$ are in turns the level-1 cores on P . Then $\langle t_j, v \rangle$, is a shortcut (or original edge) generated from B , and the weight of $\langle t_j, v \rangle$ equals that of the shortest path from t_j to v contained in B . Besides, by Claim 1.4, u could equally reach t_j by the edges generated from B . Hence Claim 1.6 holds when $i = 1$.

Apparently, Statement 1 is true when $i = 1$. This is because the spanning of B are found on G (*i.e.*, using the original edges) within a region B . In another word, when $i = 1$, both the type (a) and type (b) spanning paths of B could be found.

Statement 2 could be obtained by the algorithm when $i = 1$. If P contains one edge, both the endpoints of P would be promoted to a level-1 core. If P contains more than one edge, and then an internal node of P , say w , $w \neq s, t$ is promoted to a level-1 core. By Statement 1, P is a spanning path of B , and w is not an endpoint of P . Hence, w is in B .

As for Statement 3, when $i = 1$, firstly we show that there exists a region B under R_1 , such that a sub-path of P (including P itself) is a spanning path of B . If P is contained in a region B under R_1 , since s, t are far-apart in R_1 , P is a spanning path of B . Otherwise, if P is not contained in any region under R_1 , and then by Lemma A.3, there exists a region B under R_1 , which makes a sub-path of P a spanning path of B . Secondly, note that at the first iteration, given B all its spanning paths could be found because the paths are found on the original graph G . Hence, there should exist a region B under R_1 , such that P covers a path in $P_{1,B}$.

Statement 4 could be obtained by Statement 2 and 3. By Statement 3, there is a region B in R_i such that a path sub-path P' of P covers a path in $P_{i,B}$. If P' is exactly P , since P contains more than one edge, and then by Statement 2, there should be a node u on P other than s, t , such that u is a level- i core. If P' is not P , nevertheless where the positions of the two endpoints of P' on P are, by Statement 2, there should exist a node u ($u \neq s, t$) on P and u is a level- i core. Hence in either case, Statement 4 is true.

The above all show that the lemma is true when $i = 1$. We now turn to the induction phase. Suppose the statements and claims above are true when $i = k$, we show them true when $i = k + 1$.

As for Claim 1.1, P is a shortest path from u to v (u, v are level- k cores) contained in a region B under R_{k+1} . We prove the claim in two cases: (a) P is also contained in a sub-region B' of B , where B' is a region under R_k , and (b) P is not contained in any region under R_k . As for case (a), given Claim 1.4 true when $i = k$, P could be found by a Dijkstra search visiting the edges concerning level- k cores generated from B' , which satisfies the coverage condition. Hence Claim 1.1 is true. As for case (b), if P contains only one original edge $\langle u, v \rangle$, and then $\langle u, v \rangle$ is an edge that satisfies the coverage condition, hence Claim 1.1 is true. Subsequently we consider the case when P contains multiple edges. If u, v are far-apart under R_k , and then by Statement 4 when $i = k$, there should be a node w ($w \neq u, v$) on P and w is a level- k core. If u, v are not far-apart under R_k , given the hypothesis that P is not contained in any region under R_k , there should exist two nodes u' and v' such that u' and v' are far-apart under R_k . Again by Statement 4 the sub-path from u' to v' of P is covered by V_k , which follows that there also exist a node w on P ($w \neq u, v$) and w is a level- k core. Since u, w, v are all level- k cores, similarly we can consider the sub-path from u to w (and the sub-path from w to v as well) in the two cases above. Because P contains a finite number of nodes, P could not be infinitely decomposed. *i.e.*, we could always find a sub-path P' of P between two level- k cores, such that P' is contained in a region under R_k , and by Claim 1.4 P' could be discovered by the Dijkstra search relaxing the edges concerning level- k cores. Therefore, Claim 1.1 is true when $i = k + 1$.

Claim 1.2 is similar to Claim 1.1, except that the source u is a level-0 core instead of a level- k core. In a similar way as how Claim 1.4 is proved, we can consider P in two cases: whether it is contained in a region under R_k or not. We focus on the sub-path from u . In this way we can always find a node level- k core w , so that, either the sub-path from u to w is contained in a sub-region B' under R_k , or $\langle u, w \rangle$ is an original edge. In the former case, given Claim 1.5 true when $i = k$, Claim 1.2 is true when $i = k + 1$. In the latter case, $\langle u, w \rangle$ is an edge that satisfies the coverage condition, hence Claim 1.2 is also true.

Claim 1.3 could be similarly proved like Claim 1.2 given Claim 1.6 and Statements 2 and 4 true when $i = k$, and Claim 1.1 true when $i = k + 1$.

For further induction, we also show that Claim 1.4 to 1.6 are true when $i = k + 1$. In the following we prove Claim 1.5 true when $i = k + 1$. Consider a shortest path P from a level-0 core u to a level- $(k + 1)$ core v contained in a region B under R_{k+1} . By Claim 1.2 P could be equally found by a Dijkstra search which relaxes the edges concerning level- k cores. Along P let $t_1, \dots, t_j, v$ ($j \geq 0$) be the level- $(k + 1)$ cores. Then AH would add $\langle u, t_1 \rangle$ (if not existed in G) as a shortcut generated from B , and its weight equals the weight of the path from u to t_1 . So are the other shortcuts $\langle t_1, t_2 \rangle, \dots, \langle t_j, v \rangle$. Hence Claim 1.5 is true when $i = k + 1$.

Claim 1.4 and 1.6 could be proved in a similar way.

Based on Claim 1.1 to 1.3, we show that Statement 1 is true.

We show that type (a) path could be correctly found by the algorithm. Let s (resp. t) be a boarder node in the west (resp. east) strip of B . Then, for any pair of such s and t , the shortest path P from s to t contained in B is a spanning path of B . Besides, P is covered by a level- k core since s and t are far-apart in R_{k+1} , and followed by Statement 4. Subsequently, given Claim 1.1 to 1.3, P could be correctly found by the algorithm.

Then we show that type (b) path could also be correctly found. Let P be a local shortest path of B from s to t where t is beyond B . Let $\langle u, t \rangle$ be the last edge on P , and u a level- k core. Then, given Claim 1.1 and Claim 1.2, the shortest path from s to u contained in B could be correctly found. On the other hand, $\langle u, t \rangle$ satisfies the coverage condition. As a result, P could also be correctly found by the algorithm.

The above all shows that: (i) for a spanning path P of type (a) or type (b), P could be found by the algorithm supported by Claim 1.1 to 1.3. And (ii) every $P \in \mathcal{P}_{k+1,B}$ is a spanning path of B . Hence follows Statement 1.

We show that Statement 2 is true when $i = k + 1$. Similar to the case when $i = 1$, by the algorithm, each $P \in \mathcal{P}_{k+1,B}$, satisfies either condition stated in Statement 2.

As for Statement 3, we consider P in two cases: P is not contained in any region under R_{k+1} , or P is contained in a region under R_{k+1} . In the former case, by Lemma A.3, there should exist a region B under R_{k+1} , where B is a sub-region of B_{super} , such that P' , a sub-path of P is a spanning path of B . In the latter case, we put $P' = P$. Hence in the following, we consider P' in two cases: (a) P' is contained in B , and (b) P' is not contained in B . Suppose P' is from u to v . Without loss of generality suppose u is in the west strip of B and v the west strip.

In case (a), u is in the east strip of B , while v is in the west strip. We show that there should be two nodes u' and v' on P' , such that the sub-path of P' from u' to v' is a type (a) spanning path in $\mathcal{P}_{k+1,B}$. Starting with u , we in turn scan each node on P' . Stop until the first time such node v' is met: v' is in the east strip. In other words, the predecessor of v' on P' is to the west of the east strip. Such v' exists because P' ends at a node v in the east strip (v' might equal to v). Similarly, there should exist a node u' such that u' is in the west strip and the successor of u' on P' is to the east of the west strip (u' might equal to u too). On the other hand, the sub-path of P' from u' to v' is a type (a) spanning path that could be found by the algorithm, hence P covers a path in $\mathcal{P}_{k+1,B}$.

In case (b), we show that P' covers a type (b) spanning path in $\mathcal{P}_{k+1,B}$. Firstly, similar to case (a), there should be a node u' on P' , such that u' is in the west strip, and the successor of u' on P' is in to the east of the west strip. On the other hand, let v' be the predecessor of v on P' . By Lemma A.3, since P' is not contained in B , the v' and v are not in two adjacent cells under R_{k+1} , which follows that v' and v are far-apart in R_k .

Hence the sub-path of P' from u' to v is a type (b) spanning path in $\mathcal{P}_{k+1,B}$.

Finally, similar to the case when $i = 1$, Statement 4 is true when $i = k + 1$. Therefore, the lemma is proved. $\square$

Lemma A.5. *Lemma A.4 also holds if at each iteration AH only adds shortcuts concern the boarder nodes of R_{i+1} and the level- i cores.*

Proof. Let B be a region under to R_{i+1} ($i \geq 1$). Let P be a type (a) spanning path (described in Lemma A.4) from s to t in $\mathcal{P}_{i+1,B}$. Note that except for s and t , all the internal nodes on P are not level- i cores (*i.e.*, they are at a level less than i). It means that, the algorithm mainly uses the edges connecting the level- i edges. As for the other edges, only the edges from a boarder node (s) to a level- i core, or vice versa, the edges from a level- i core to a boarder node (t), are used. Similarly, so are the type (b) spanning paths. Hence, after the level- i cores are computed, it suffices to add the shortcuts concerning level- i cores and the boarder nodes of R_{i+1} . $\square$

Recall Lemma 3.3 *For any two nodes $u, v \in G$, if no (3×3) -cell region in R_i ($i \in [1, h]$) can cover u and v simultaneously, and then the shortest path from u to v must go through a node at level i or above.*

Proof. This lemma could be obtained by Statement 3 of Lemma A.4 when B_{super} is so large a region that covers the whole road network G . $\square$

Lemma A.6. *Let B be a region in R_i , $\langle u, v \rangle$ a shortcut created in B . Then the path contracted by $\langle u, v \rangle$ is contained in B .*

Proof. This lemma could be proved by mathematical induction. Let P be the path contracted by $\langle u, v \rangle$. Firstly we show that the lemma holds when $i = 1$. By contradiction suppose the path contracted by $\langle u, v \rangle$ is not contained in B . Then, there is a node x on P such that x is beyond B . $x \neq v$ because v is in B . On the other hand, let y be the successor of x on P . Such y exists because x is not the last node on P . In that case, $\langle x, y \rangle$ is not an edge generated from B because x is not in B . Hence P could not be found by AH, which follows that $\langle x, y \rangle$ could not be a shortcut created in B . Secondly, suppose the lemma holds when $i = k$, we show that it also holds when $i = k + 1$. By contradiction let x be the node on P that is not in B . Then $x \neq u, v$ because u, v are in B . If x is not a level- k core, x is not visited during the Dijkstra search because the search at level $k + 1$ would only visits the nodes level- k cores. It follows that x is contracted by a shortcut e where e is generated from a sub-region of B , and given the lemma true when $i = k$, x should be in B . If x is a level- k core, let $\langle x, y \rangle$ be the edge visited during the Dijkstra search. Then $\langle x, y \rangle$ violates the coverage condition since x is beyond B . Hence follows the lemma. $\square$

Lemma A.7. *Let B be a region under R_i , $P \in \mathcal{P}_{i,B}$ be a path from s to t , and u be the predecessor of t on P . If t is beyond B , and then u, t are level- $(i-1)$ cores.*

Proof. Apparently the lemma holds when $i = 1$ given all the nodes in G are level-0 cores. Suppose the lemma holds when $i = k$, in the following we show that true when $i = k + 1$. Firstly, t is a level- k cores, because, except for the source node s , the Dijkstra search only visits the level- k cores. Secondly, let $\langle x, t \rangle$ be the edge on P relaxed by the Dijkstra search. $\langle x, t \rangle$ could be a shortcut or an original edge. If $\langle x, t \rangle$ is a shortcut, by Lemma A.6, t should be in B . However, it violates the hypothesis that t is beyond B . Hence, $\langle x, t \rangle$ is an original edge in G . In another word, $\langle x, t \rangle$ is the last edge on P , and $\langle x, t \rangle$ is visited by the Dijkstra search. Hence x, t are level- k cores. Thus follows the lemma. $\square$

Definition A.1 ($\mathcal{Q}_{i,B}^s$, Set of spanning paths from s in B). *Let B be a region under R_i where $i \geq 2$, and s be a node in B . Define $\mathcal{Q}_{i,B}^s$ be the set of paths starting from s , where each path $P \in \mathcal{Q}_{i,B}^s$ from s to another node t satisfies the following conditions:*

- (i) P is a local shortest path in B .
- (ii) if s is at the west (resp. east) of the vertical bisector of B , and then t is to the east (resp. west) of s , and their horizontal grid-distance under R_{i-1} is at least two. Similarly, if s is at the north (resp. south) of the horizontal bisector of B , and then t is to the south (resp. north) of s , and their vertical grid-distance under R_{i-1} is at least two.
- (iii) Let u be the predecessor of t on P . If t is beyond B , and then u, t are level- $(i-2)$ cores.

Lemma A.8. *Each $P \in \mathcal{Q}_{i,B}^s$ covers a path in $\mathcal{P}_{i-1,B'}$, where B' is a sub-region of B .*

Proof. If P is contained in B , by Statement 3 of Lemma A.4, the lemma holds. Otherwise, t is beyond B . Let u be the predecessor of t on P . Then the sub-path $P_{s,u}$ from s to u is contained in B . We consider $P_{s,u}$ in two cases.

Case 1: there is no sub-region of B under R_{i-1} that contains $P_{s,u}$. In this case, there should exist two nodes x and y on $P_{s,u}$, such that x and y are far-apart under R_{i-1} . Such x, y exist, otherwise it violates the hypothesis in this case. Again by Statement 3 of Lemma A.4, this lemma holds.

Case 2: there is a sub-region of B under R_{i-1} that contains $P_{s,u}$. Since s and t are far-apart under R_{i-1} , without loss of generality, suppose s locates at the west of the vertical bisector of B . Let s_l be the west most node on $P_{s,u}$, i.e., among all the nodes on $P_{s,u}$, s_l has the smallest X-coordinate value. Then there should exist a sub-region of B under R_{i-1} , say B' , which makes s_l locating in its west strip, and the path from s_l to t a spanning path of B' . On the path from s_l to t , among the nodes locating in the west strip of B' , let s' be the east most one, i.e., s' has the largest X-coordinate value. Given the definition of $\mathcal{Q}_{i,B}$ that t is beyond B , u, t are level- $(i-2)$ cores. And further by Lemma A.4 the path from s' to t is in $\mathcal{P}_{i-1,B'}$. Hence above all, the lemma holds. $\square$

Lemma A.9. *Let B be a region under R_i ($i \geq 2$), $P \in \mathcal{P}_{i,B}$ a path from s to t . Let $\langle a, b \rangle$ be an arterial edge on P . Then the sub-path from a to t is in $Q_{i,B}^a$.*

Proof. Suppose $P_{a,t}$ is the sub-path of P from a to t . By Statement 1 of Lemma A.4, P is a spanning path of B , hence its sub-path $P_{a,t}$ is a local shortest path of B , which satisfies property (i) of $Q_{i,B}^a$. Without loss of generality suppose s is in the west strip of B , and t is two-grid distance away (under R_i) to the east of s . By definition a is an endpoint of an arterial edge on P , hence in this case a is at the west of the vertical bisector of B . It follows that the horizontal grid-distance from a to t is at least under R_{i-1} , which satisfies property (ii) of $Q_{i,B}^a$. If t is not in B , let u be the predecessor of t on P . By Lemma A.7 u, t are level- $(i-1)$ cores, which leads to u, t are level- $(i-2)$ cores. Then $P_{a,t}$ satisfies property (iii) of $Q_{i,B}^a$. Hence follows the lemma. $\square$

Lemma A.10. *Let B be a region under R_i . Then totally at most $50\lambda^2$ nodes are promoted to level- i cores by the spanning paths of B .*

Proof. When $i = 1$, the spanning paths are found by a Dijkstra search which relaxes only the original edges in G . By assumption there are at most λ arterial edges in B . In the worst case both endpoints of those arterial edges are promoted, and the number of nodes promoted is still less than $50\lambda^2$. In the following we show that the lemma also holds when $i \geq 2$.

Let AE be the set of pseudo-arterial edges of the spanning paths of B . Then each $e \in AE$ is either an original edge in G or a shortcut. We consider two disjoint subsets AE_1 and AE_2 where AE_1 contains all the original edges and AE_2 the shortcuts. Suppose $|AE_1| = \lambda_1$ and $|AE_2| = \lambda_2$.

Set AE_1 : by the *Node Selection* algorithm, each edge in AE_1 contributes at most two level- i cores.

Set AE_2 : we divide AE_2 into several disjoint subsets according to the arterial edges $e \in AE_2$ contracts. *i.e.*, the shortcuts that contract the same arterial edge are in the same subset. We show that each subset contributes at most 50λ level- i cores. For each subset AE_{sub} , suppose the shortcuts in AE_{sub} are $\langle X_1, Y_1 \rangle, \dots, \langle X_k, Y_k \rangle$, and they are respectively on the paths $P_1, \dots, P_k$ to the nodes $t_1, \dots, t_k$. We use $\langle a, b \rangle$ to denote the arterial edge $\langle X_1, Y_1 \rangle, \dots, \langle X_k, Y_k \rangle$ contract. By Lemma A.9, the sub-path from a to t_j is in $Q_{i,B}^a$ for $1 \leq j \leq k$, because the arterial edge $\langle a, b \rangle$ is on P_j where $P_j \in \mathcal{P}_{i,B}$. We consider $Q_{i,B}^a$. Let G_a be a graph whose edges are composed of the edges on a path P' where $P' \in Q_{i,B}^a$. Then G_a is a tree, otherwise, it violates the hypothesis that the shortest paths in G_r are unique. Besides, $Y_1, \dots, Y_k$ is in G_a , the tree. We show that there are at most 25λ distinct nodes among $Y_1, \dots, Y_k$. This is because, for each $P \in Q_{i,B}^a$, there is a sub-region of B under R_{i-1} , say B' , such that a subpath P' of P is a spanning path of B' . And by Statement 2 of Lemma A.4, P' is covered by a level- $(i-1)$ core.

Since B' contains at most λ arterial edges, and there are at most twenty-five sub-region of B , hence there are at most 25λ level- $(i-1)$ cores that cover the paths in $Q_{i,B}^a$. On the other hand, on the path from a to t_j , among the nodes in V_{i-1} , Y_j is the closet one to a , otherwise, if a level- $(i-1)$ core u is on the path from a to Y_j , and then $\langle X_j, Y_j \rangle$ contracts a level- $(i-1)$ core, which violates the algorithm. Hence there are at most 25λ distinct nodes of $Y_1, \dots, Y_k$. Similarly, the number of distinct nodes of $X_1, \dots, X_k$ is at most 25λ as well ¹. In addition, the nodes promoted to level- i cores are selected from $X_1, \dots, X_k, Y_1, \dots, Y_k$. Hence in the worst case at most 50λ nodes are promoted. What's more, given $|AE_2| = \lambda_2$, there are at most λ_2 disjoint subsets, hence at most $50\lambda \cdot \lambda_2$ are promoted.

On the other hand, if $e \in AE$ is an original edge, e cannot be contracted by other shortcuts in AE at the same time because both of the two endpoints of e are level- $(i-1)$ cores. Hence $\lambda_1 + \lambda_2 = \lambda$. So totally at most $50\lambda^2$ level- i cores are promoted by the spanning paths of B . $\square$

Recall Lemma 3.4 *Any $(\alpha \times \alpha)$ -cell region in R_i contains $O(\alpha^2 \lambda^2)$ nodes whose level in H^* are no lower than i , where λ is the arterial dimension of G .*

Proof. The proof is similar to that of Lemma 3.1. It could be obtained by Statement 2 of Lemma A.4 and Lemma A.10. $\square$

A.4 Proof of Theorem 3.1

We prove Theorem 3.1 by presenting a series of lemmas and theorems that establish the space and time complexities of AH, as well as the correctness of AH's query processing algorithms.

Lemma A.11. *Each node in H^* has $O(\lambda^2)$ non-elevating edges.*

Proof. Let u be a node at level i , and $\langle u, v \rangle$ a non-elevating edge. Under the grid R_{i+1} , consider the cell C that contains u . Then v is in the (5×5) -cell region centred at C . This is because the algorithm firstly builds a *SPT* (see definition 3.3) from u in the (5×5) -cell region centred at u , and then adds non-elevating edges from u to the nodes on the tree. Besides, by Lemma 3.4, there are $O(\lambda^2)$ nodes in V_i in the (5×5) -cell region centred at C , which is a 10×10 cell region under R_i . Hence, the number of non-elevating edges is $O(\lambda^2)$. $\square$

¹This conclusion is obtained by considering the paths from the source of each path P_j to b . If we modify the definition of $Q_{i,B}^s$, which consider the paths in a reverse direction, *i.e.*, the path from a far-apart node t , to a fixed node s , we can have the same conclusion

Lemma A.12. *Each node in H^* has $O(h\lambda^2)$ elevating edges.*

Proof. Let $\langle u, v \rangle$ be an elevating edge. Then v has a higher rank than that of u . Suppose v is at level i . It suffices to show that for a fixed i , the number of such elevating edges is $O(\lambda^2)$. This is because $\langle u, v \rangle$ is obtained from a *SPT* rooted at u generated in a (5×5) -cell region centred at u under R_i , and by Lemma 3.4 the number of such node v is $O(\lambda^2)$. In the worst case, u is at level 0 and adds elevating edges to the level- i nodes where $i \in [1, h]$. Hence, u has $O(h\lambda^2)$ elevating edges. $\square$

Theorem A.2. *The space overhead of AH is $O(hn)$.*

Proof. By Lemmas A.11 and A.12, each node has $O(h\lambda^2)$ edges, and there are n nodes in G . Hence the space overhead is $O(hn)$ when λ is a constant. $\square$

Theorem A.3. *AH answers a distance query in $O(h \log h)$ time. Besides, it answers a shortest path query in $O(h \log h + k)$ time, where k is the number of edges in the shortest path.*

Proof. AH answers any distance query with two traversals of H^* , starting from the source s and destination t of the query, respectively. Due to the proximity constraint, in the i -th level of H^* ($i \in [0, h]$), each traversal of AH only visits the nodes in a (5×5) -cell region in R_{i+1} . By Lemma 3.4, such a region only contains $O(\lambda^2)$ nodes at level i because a (5×5) -cell region in R_{i+1} is a (10×10) -cell region in R_i . Hence, the total number of nodes traversed by AH is $O(h\lambda^2)$. Furthermore, for each node v visited during a traversal, AH either follows the elevating edges of v to a certain level of H^* , or moves along the non-elevating edges of v that satisfy the rank and proximity constraints. As previously discussed in Lemma A.12, v has $O(\lambda^2)$ elevating edges to each level of H^* , and has $O(\lambda^2)$ non-elevating edges. Therefore, the total number of edges visited by AH is $O(h\lambda^4)$. Since each traversal is performed using Dijkstra's algorithm, its overall time complexity is $O(h\lambda^2 \log(h\lambda^2) + h\lambda^4)$. Consequently, when λ is a constant, the time complexity of AH for a distance query is $O(h \log h)$.

To answer a shortest path query from s to t , AH first processes its corresponding distance query to retrieve the shortest path P' from s to t in H^* , and then it transforms P' into the actual shortest path P from s to t in G . For each shortcut e , it requires $O(1)$ time to decompose e into two edges, and an original edge cannot be further decomposed. Hence the transformation from P' to P takes $O(k)$ time, where k is the number of edges in P . Therefore, AH requires $O(h \log h + k)$ time to answer a shortest path query. $\square$

Lemma A.13. *AH requires $O(hn^2)$ time to construct H^* .*

Proof. The preprocessing algorithm of AH consists of three steps: (i) assigning nodes to each level of H^* , (ii) deriving the strict total order on nodes at the same level, and (iii) creating shortcuts in H^* .

When assigning nodes to the i -th level of H^* ($i \in [0, h - 1]$), AH inspects each non-empty (4×4) -cell region in R_{i+1} , and construct a subgraph that consists of the level- i cores and border nodes in the region. For each boarder node u , AH needs to apply Dijkstra's algorithm to traverse the subgraph. By Lemma 3.4, $O(\lambda^2)$ level- i cores are visited during the traversal, and each node visited has $O(\lambda^2)$ edges. Hence building a Dijkstra tree from u require $O(\lambda^4)$ time. After the Dijkstra tree is constructed, AH needs to inspect each node in the tree and the boarder nodes to find out a spanning path which requires $O(n\lambda^2)$, because there are $O(\lambda^2)$ nodes in the tree and n boarder nodes for a loose estimation. On the other hand, u is contained in a constant number of (4×4) -cell region in R_i , hence it requires $O(n\lambda^2)$ time to find out the spanning paths from u . As such, it requires $O(n^2\lambda^2)$ time to find out all the spanning paths.

Meanwhile, AH takes only $O(n)$ time to derive the strict total order at level i of H^* , since the derivation is based on a linear time algorithm for vertex cover.

To construct shortcuts at the i -th level of H^* , AH needs to inspect a graph G_i^* reduced from G . For each node u in G_i^* , AH invokes a Dijkstra traversal in the (5×5) -cell region in R_{i+1} that is centered at u , and it creates shortcuts from u by traversing the nodes in the Dijkstra tree. By Lemma 3.4 there are $O(\lambda^2)$ nodes in the tree, hence it costs $O(\lambda^2)$ time to generate level- i shortcuts from u . As such, the time required to create shortcuts at level i of H^* is $O(n\lambda^2)$. There are h levels, and therefore AH costs $O(hn\lambda^2)$ to create shortcuts.

Since λ is a constant, AH totally requires $O(hn^2)$ time to construct H^* . $\square$

Lemma A.14. *For any two nodes s and t , there exists a path P from s to t such that (i) the weight of P equals the weight of the shortest path from s to t in G , and (ii) the rank sequence of P is unimodal.*

Proof. Among the nodes on the shortest path from s to t in G , let u be the one with the highest rank. Let $P_{s,u}$ be the shortest path from s to u in G . It suffices show that there is a path P_f from s to u such that (i) the weight of P_f equals the weight of $P_{s,u}$, and (ii) the rank sequence of P_f is increasing. We use $rank(u)$ to denote the rank of a node u .

Among the nodes on $P_{s,u}$ let v be such a node that: (i) $rank(s) < rank(v)$, and (ii) if v' is another node on $P_{s,u}$ that has a higher rank value than s , v is closer to s than v' . Let $P_{s,v}$ be the sub-path of $P_{s,u}$ from s to v . Then $P_{s,v}$ is also a shortest path, and among the nodes on $P_{s,v}$ v has the highest rank and s comes the second. Otherwise it violates the second property of v . If $P_{s,v}$ contains multiple edges, by the algorithm $< s, v >$ is a level- $rank(s)$ edge, and the weight of $< s, v >$ equals the weight of $P_{s,v}$. It means that $< s, v >$ is an edge on P_f . Then in a similar way we continue to consider the path from v to u . Therefore, the lemma is proved. $\square$

Theorem A.4. *AH correctly answers any distance query.*

Proof. Suppose $P_{s,t}$ is a shortest path from s to t in G . Lemma A.14 shows there exists a unimodal rank sequence path P from s to t where the weight of P equals the weight of $P_{s,t}$. *i.e.*, the *Rank Constraint* would not affect the query correctness. Now we show that the *Proximity constraint* would not affect the correctness either.

Let u be the node with the highest rank among all the nodes on $P_{s,t}$. Let P_f be the shortest path from s to u and the rank sequence of P_f is increasing. It suffices to show that the boundary check in the forward search would not affect the discovery of P_f . By contradiction suppose v is a level- i node on P_f , but v is beyond the (5×5) -cell region under R_{i+1} centred at the cell that contains s . Let $P_{s,v}$ be the shortest path from s to v . Then in this case $P_{s,v}$ contains multiple edges, otherwise, since s and v are far-apart under R_{i+1} , and followed by Statement 2 of Lemma A.4, s and v are both level- $(i+1)$ cores, which violates the assumption v as at level i . Besides, by Statement 4 of Lemma A.4, there is a node level- $(i+1)$ core x on $P_{s,v}$ and $x \neq v$. Then it violates the rank-increasing property of P_f because x comes before v on P_f .

Finally we show that the elevating edges would not affect the correctness. Let R_j ($j \in [1, h]$) be the coarsest grid where no (4×4) -cell region contains both s and t . By Lemma 3.3, the shortest path from s to t should pass through at least one node whose level is at least j . This indicates that AH's traversal from s would meet its traversal from t at level i or above of H . As such, omitting visiting the nodes that are lower than level j would not affect the correctness. Therefore, the lemma is proved. $\square$

Theorem A.5. *AH correctly answers any shortest path query.*

Proof. By theorem A.4, given two nodes s and t , the query algorithm can discover a unimodal node rank sequence path P . It remains to show that every shortcut e on P can reconstruct the path e contracts. There are two types of shortcuts: elevating edges and non-elevating edges (*i.e.*, the level- i edges).

As for the non-elevating edges, it suffices to show that the non-elevating edges on the rank-increasing path discovered by the forward search from s can be reconstructed. Suppose $\langle u, v \rangle$ is a non-elevating edge on the path discovered by the forward search. Then the rank of v is higher than that of u . In addition, $\langle u, v \rangle$ contracts the shortest path $P_{u,v}$ from u to v , otherwise it violates the hypothesis that $\langle u, v \rangle$ is on a shortest path. Suppose $\langle u, v \rangle$ is marked with a node w . Then the sub-path $P_{w,v}$ (resp. $P_{u,w}$) from w to v (resp. from u to w) of $P_{u,v}$ is also a shortest path. It suffices to show that $\langle w, v \rangle$ (resp. $\langle u, w \rangle$) is an original edge or a non-elevating edge. We consider $P_{w,v}$. If $P_{w,v}$ contains multiple edges, since by the meaning of w , among the nodes on $P_{w,v}$, v has the highest rank and w comes the second, hence the algorithm would add level- i edge $\langle w, v \rangle$ where

w is at level- i . On the other hand, we can also come to a similar conclusion of $\langle u, w \rangle$ if we build the non-elevating edge to w from a *Backward SPT* rooted at w .

As for the elevating edges, we use mathematical induction to prove that every elevating edge e on the shortest path P could be reconstructed. Firstly, let $\langle u, v \rangle$ be an elevating edge where u is a node at level-0, and v at level-1, and is marked with a node w . Then w is the first node that has higher rank than u does on the shortest path from u to v . In that case, $\langle u, w \rangle$ is a level-0 edge, and as discussed before, $\langle u, w \rangle$ could be reconstructed. If $w \neq v$, we continue to consider $\langle w, v \rangle$. Since w is also at level-0, and v is the first node at level-1 on the shortest path from w to v . Then $\langle w, v \rangle$ is also an elevating edge. Similar to the discussion aforementioned, $\langle w, v \rangle$ could also be reconstructed. As such, all the elevating edges to a node at level 1 on the shortest path could be reconstructed. Suppose the elevating edges e to a level- i node on a shortest path P could be reconstructed. We show that the elevating edges e to a level- $(i + 1)$ node could also be reconstructed. Let $\langle u, v \rangle$ be an elevating edge where v is at level- i . There are two types of elevating edges: (a) u is a boarder node under R_i , and (b) u is a node at level $i - 1$. We firstly consider (a). Let T be an SPT from u . Let w be the node immediately follows u on the branch from u to v in T . Then w is a level- $(i-1)$ core. In that case, $\langle u, w \rangle$ is also an elevating edge because: (i) u is a boarder node in R_i implies that u is also a boarder node in R_{i-1} , and (ii) w is a node at level- $(i-1)$ that is closet to u on the path from u to v . By induction, $\langle u, w \rangle$ could be reconstructed. If $w \neq v$, we continue to consider $\langle w, v \rangle$. Since the shortest path from w to v does not go through another node at level i , $\langle w, v \rangle$ is an elevating shortcut of type (b). It remains to show that the type (b) elevating edges could be reconstructed. Let $\langle u, v \rangle$ be a type (b) elevating edge, *i.e.*, u is at level $i - 1$ and v at level i . By the algorithm $\langle u, v \rangle$ is marked with a node w , the first node that has a higher rank than u does on the shortest path from u to v . Then, $\langle u, w \rangle$ is a level- $(i-1)$ edge and could be reconstructed. If $w \neq v$, we continue to consider w , and similarly $\langle w, v \rangle$ is also an elevating edge of type (b) marked with w' . And finally, there is an edge $\langle w', v \rangle$ such that w' is at level $i - 1$, and the nodes on the shortest path from w' to v (excluding w' and v) have lower ranks than w' does. In that case, $\langle w', v \rangle$ is also a level- $(i-1)$ edge, and could be reconstructed. Similarly, all the elevating edges from a level- i node to the boarder nodes of R_i and the level- $(i-1)$ nodes could also be reconstructed. $\square$

A.5 Analysis of HoD on Deterministic Scale-free Networks

In contrast to the efficiency of HoD in practice, the worst-case analysis in Section 4.4.6 shows the complexity of HoD's preprocessing algorithm is prohibitive. This is because,

Figure A.5: Construction of the deterministic scale-free network.

the performance of the preprocessing algorithm fully relies on the structure properties of the input graph G , but in the worst-case study, these properties are ignored from the analysis. Thus, for a more accurate and clear evaluation on scale-free networks, we utilize the well-known deterministic scale-free networks [14] which are generated in a deterministic fashion for the analysis in this section. Because it is much more easier to capture the structure properties of the deterministic graphs.

The construction of the deterministic scale-free network is shown in Figure A.5. The undirected graph is derived in an iterative manner, with each iteration replicating the graph generated in the previous step and connecting them together, as follows:

Step 0: We start from a single node, and we denote it as the root of the graph.

Step 1: We add two more nodes, and connect each of them to the root.

Step 2: We add two replicas of three nodes graph, with each replica identical to the graph created in the previous iteration (step 1), and we connect each of the bottom nodes of these two replicas to the root. As a result, the root will gain four more new links.

Step 3: We add two replicas of the nine nodes graph generated in the previous iteration, and connect all eight bottom nodes of the two new replicas to the root.

In general, step k would involve the following operation:

Step k : Add two replicas of 3^{k-1} nodes graph generated in the previous iteration (step $k-1$), and connect each of the 2^k bottom nodes of these two replicas to the root of the graph.

Let the deterministic scale-free network created above with $k=3$ as the input graph G of HoD (as shown in Figure A.6(a)). In the following, we demonstrate how the pre-processing algorithm of HoD constructs the index on G step by step, and then analyze the costs incurred by the preprocessing algorithm.

Recall the node selection criteria in Section 4.4.2, HoD prefers to remove nodes with smaller degrees in G . Thus, in the first iteration, nodes with degrees of 1 or 2 are removed from G . For instance, in the left branch, v_2, v_4, v_6, v_8 are removed. Specifically, the removal of v_4 and v_6 does not incur shortcut generation, because they are degree-one nodes, while the removal of v_2 and v_8 might bring in shortcuts (denoted by blue dashed lines), if the number of baseline edges generated by v_5 is not enough. Figure A.6(b) shows the residual graph G_1 after the first iteration. Then in the second iteration, the bottom nodes in each of the three branches will be removed. Recall the node removal criteria in Section 4.4.1, v_7 and v_9 will not be removed together, as there is an edge (i.e., the shortcut created in the first iteration) between them. Meanwhile, the removal of v_3 brings in a shortcut between v_5 and v_0 , but it might be counteracted if a baseline edge between v_5 and v_0 is created via v_1 . Without loss of generality, suppose such counteraction happens in half cases, then two bottom nodes are remained in each branch, which derives G_2 (as shown in Figure A.6(c)) after the second iteration. Notice that, G_2 is equivalent to the deterministic scale-free network with $k=2$ in Figure A.5, which means the number of nodes in G_2 is $\frac{1}{3}$ of that in G_0 . Moreover, G_2 is the mother structure of G_0 , i.e., G_2 and two replicas compose G_0 . This indicates that the reduction process of HoD is the inverse process of the deterministic scale-free network generation process, and thus, HoD takes $2 \cdot \log_3(n) - 1$ steps to derive the core graph with only the root node left.

In the first iteration, HoD generates $O(n)$ shortcuts as well as $O(n)$ baseline edges, because the number of baseline edges is linear to the number of shortcuts as shown in Section 4.4.3. Similarly, $O(n)$ shortcuts and baseline edges are created in the second iteration. Given the major I/O and CPU costs are incurred by sorting the edge triplets in each iteration, we know that the I/O and CPU costs of the first two iterations are $O(\frac{n}{M})$ and $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ respectively. Then for the entire complexity, we have $T(n) = O(\frac{n}{M}) + T(\frac{n}{3})$ for the I/O cost, since the size of the residual graph reduces to $\frac{1}{3}$ of that before, after every two iterations. By the master theorem, we have $T(n) = O(\frac{n}{M})$. Similarly, we also derive the entire CPU cost as $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$.

The deterministic scale-free network is an extreme example for HoD, but it reveals the efficiency of HoD in practice. Because real-world graphs tend to have a small portion of nodes that are crucial for shortest path and distance queries while the majority are marginal nodes, and coincidentally, HoD is able to eliminate the unimportant nodes step by step, while keep the nodes that are crucial for shortest path and distance queries as the core graph.

A.6 Proof of Theorem 4.1

Our proof of Theorem 4.1 utilizes the concepts of *rank sequences* and *arch paths*, defined as follows.

Definition A.2 (Rank Sequence and Arch Path). *Let P^* be a path in the augmented graph G^* of HoD, such that P^* contains k nodes. The **rank sequence** of P^* is a sequence of integers $\langle r_1, r_2, \dots, r_k \rangle$, such that r_i ($i \in [1, k]$) equals the rank of the i -th node in P^* . P^* is an **arch path**, if its rank sequence can be divided into three subsequences $\langle r_1, \dots, r_x \rangle$, $\langle r_x, \dots, r_y \rangle$, and $\langle r_y, \dots, r_k \rangle$, such that*

- (i) $r_1 < r_2 < \dots < r_x$, and
- (ii) $r_x = r_{x+1} = \dots = r_y$, and
- (iii) $r_y > r_{y+1} > \dots > r_k$. □

For instance, consider the path $P^* = \langle v_1, v_9, v_{10}, v_8, v_4 \rangle$ in Figure 4.1e. The path's rank sequence is $\langle 1, 4, 4, 3, 2 \rangle$. This rank sequence can be divided into three subsequences $\langle 1, 4 \rangle$, $\langle 4, 4 \rangle$, and $\langle 4, 3, 2 \rangle$. By Definition A.2, P^* is an arch path.

Let s and t be any two nodes in the original graph G , and P be the shortest path from s to t in G . If there exist multiple shortest paths from s to t , we choose P to be a path where the highest-rank node ranks no lower than any other node on any other shortest path from s to t . In the following, we will prove three propositions:

- **Proposition 1:** For any path P^* from s to t in the augmented graph G^* , its length is no shorter than P 's.
- **Proposition 2:** There exists an arch path P° in G^* , such that $l(P^\circ) = l(P)$, i.e., P° and P have the same length.
- **Proposition 3:** When HoD's answers an SSD query from s , it will traverse a path no longer than P° .

The combination of the above three propositions will establish Theorem 4.1.

Proof of Proposition 1. If all edges in P^* appear in the original graph G , the proposition trivially holds since any path from s to t in G should be no shorter than P . In the following, we consider only the case when P^* contains at least one shortcut, and we will prove that there exists a path P' in G , such that $l(P^*) = l(P') \geq l(P)$.

Assume without loss of generality that P^* consists of a sequence of k nodes $\langle v_1, v_2, \dots, v_k \rangle$, where $v_1 = s$ and $v_k = t$. Further assume that $\langle v_i, v_{i+1} \rangle$ ($i \in [1, k]$) is a shortcut. By the preprocessing algorithm of HoD, this shortcut must be constructed when HoD removes a certain node v from the reduced graph, such that v_i and v_{i+1} are incoming and outgoing

neighbors of v , respectively. This indicates that the augmented graph G^* must contain two edges $\langle v_i, v \rangle$ and $\langle v, v_{i+1} \rangle$, such that their total length equals $l(\langle v_i, v_{i+1} \rangle)$. Given those two edges, we change P^* by replacing $\langle v_i, v_{i+1} \rangle$ with a two-hop path $\langle v_i, v, v_{i+1} \rangle$. This results in a modified path from s to t in G^* that has the same length with the original P^* .

If the modified P^* does not contain any shortcut, and then it is a path in G , and hence, $l(P^*) \geq l(P)$. On the other hand, if the modified P^* contains any shortcut, and then we can replace the shortcut with a two-hop path in G^* , as in the previous case for the shortcut $\langle v_i, v_{i+1} \rangle$. By recursively applying this replacement procedure on the shortcuts in P^* , we can transform P^* into a path P' that (i) goes from s to t , (ii) contains only edges in G , and (iii) has the same length with P^* . Given that P is the shortest path from s to t in G , we have $l(P) \leq l(P') = l(P^*)$, which completes the proof.

Proof of Proposition 2. The proposition trivially holds if P itself is an arch path in G^* . In the following, we assume that P is not an arch path, and we show that P can be transformed into a arch path P° in G^* with the same length.

Let r_{max} be the highest rank of the nodes in P . Let v_x and v_y be the first and last nodes in P whose ranks equal r_{max} . We divide P into three subsequences P_1 , P_2 , and P_3 , such that

- (i) P_1 is the sequence of nodes in P before v_x (including v_x).
- (ii) P_2 is the sequence of nodes in P between v_x and v_y (including v_x and v_y).
- (iii) P_3 is the sequence of nodes in P after v_y (including v_y).

Let us first consider P_1 . We say that a node v in P_1 is *pit*, if v ranks no higher than the node that immediately precedes v in P_1 . Among the pits in P_1 , let v' be one with the lowest rank. Let u (resp. w) be the node that immediately precedes (resp. follows) v' in P_1 . By the preprocessing algorithm of HoD, when HoD removes v' from the reduce graph, it would generate a candidate edge $e_1 = \langle u, w \rangle$, such that the edge has the same length with the two-hop path $\langle u, v', w \rangle$. If e_1 is in G^* (*i.e.*, it is retained by HoD during preprocessing), and then we transform P_1 into another path in G^* , by using e_1 to replace the two-hop path $\langle u, v', w \rangle$ in P_1 . The resulting path has the same length with P_1 , and it has one less pits than P_1 .

On the other hand, if e_1 is not in G^* , there are two possibilities:

- (i) There already exists an edge $e_2 = \langle u, w \rangle$ in the reduced graph, such that $l(e_2) = l(e_1)$. (Note that $l(e_2) < l(e_1)$ is impossible given Proposition 1 and the fact that P_1 is the shortest path from s to v_x in G .) In that case, e_2 must appear in G^* . Therefore, if we modify P_1 by using e_2 to replace the two-hop path $\langle u, v', w \rangle$ in P_1 , we can still obtain a modified path that retains the length of P_1 but contains one less pits.

- (ii) There exists a two-hop path $\langle u, v^\diamond, w \rangle$ in the reduced graph, such that (i) $l(e_3) = l(e_1)$, and (ii) $v^\diamond$ has a higher rank than v' . (Note that $l(e_3) < l(e_1)$ cannot occur due to Proposition 1 and the fact that P_1 is the shortest path from s to v_x in G .) In that case, we transform P_1 by replacing v' with $v^\diamond$. This may not decrease the number of pits in P_1 , but it retains the length of P_1 and replaces a node in P_1 with a higher rank node.

In summary, the above transformation procedure preserves the length of P_1 , and it either (i) reduces the number of pits in P_1 or (ii) substitute a node in P_1 with a higher-rank node. Given that the ranks of nodes are bounded, if we recursively apply the procedure on the lowest-rank pit in P_1 , eventually we should obtain a path P_1^* in G^* without any pit, such that (i) $l(P_1^*) = l(P_1)$, and (ii) P_1^* starts at s and ends at v_x . In that case, the rank sequence of P_1^* must be an ascending sequence.

Now consider P_2 and P_3 . We say that a node in P_2 is a pit if its rank is smaller than r_{max} , and we define a node v in P_3 as a pit if v ranks no higher than the node that immediately follows v in P_3 . By applying the same transformation procedure as in the case of P_1 , we can convert P_2 and P_3 into two paths P_2^* and P_3^* in G^* , such that (i) P_2^* and P_3^* have no pit, (ii) $l(P_2^*) = l(P_2)$ and $l(P_3^*) = l(P_3)$, (iii) P_2^* starts at v_x and ends at v_y , and (iii) P_3^* starts at v_y and ends at t . It can be verified that all nodes in P_2^* should have a rank r_{max} , and P_3^* 's rank sequence should be a descending sequence.

Let P° be a path in G^* obtained by concatenating P_1^* , P_2^* , and P_3^* . By Definition A.2, P° is an arch path, which completes the proof.

Proof of Proposition 3. Let v_x , v_y , P° , P_1^* , P_2^* , and P_3^* be as defined in the proof of Proposition 2. Without loss of generality, assume that each of P_1^* , P_2^* , and P_3^* contains at least two nodes. We will prove the proposition by showing that, given an SSD query from s , the query processing algorithm of HoD will traverse (i) a path from s to v_x that is no longer than P_1^* , (ii) a path from v_x to v_y that is no longer than P_2^* , and (iii) a path from v_y to t that is no longer than P_3^* .

Recall that HoD's query algorithm consists of three phases: a forward search in the forward graph G_f , followed by a core search in the core graph G_c , and finally a backward search in the backward graph G_b . The forward search is a variant of Dijkstra's algorithm that follows only the outgoing edges whose endpoints rank higher than the starting points. Since the rank sequence of P_1^* is in ascending order, P_1^* should be in the search space of the forward search. Furthermore, by Proposition 1 and the construction of P_1^* , G_f does not contain any path from s to v_x that is shorter than P_1^* . Therefore, when the forward search terminates, HoD should either identify P_1^* as the shortest path from s to v_x in G_f , or identify another path from s to v_x that is no longer than P_1^* . In either case, HoD will correctly derive $dist(s, v_x)$, *i.e.*, the distance from s to v_x in G .

Now consider P_2^* , where each node has the same rank. By the preprocessing algorithm of HoD, all nodes in P_2^* must be in the core graph G_c , since any node not in the core graph can only have outgoing edges to higher-rank nodes (see Section 4.4). Meanwhile, recall that the core search of HoD is a continuation of the forward search in G_c , using Dijkstra's algorithm. By the correctness of Dijkstra's algorithm and the fact that P_2^* is the shortest path from v_x to v_y in G_c , the core search of HoD should traverse a path from v_x to v_y that is no longer than P_2^* .

It remains to prove that HoD's backward search will traverse P_3^* . Assume without loss of generality that P_3^* contains a sequence of k nodes $\langle v_1, v_2, \dots, v_k \rangle$, where $v_1 = v_x$ and $v_k = t$. Recall that the backward search of HoD examines nodes in descending order of their ranks. Since P_3^* has a descending rank sequence, the backward search of HoD should examine v_i before v_{i+1} , for any $i \in [1, k-1]$.

We will prove by induction that, for any v_i ($i \in [1, k]$), the backward search can correctly derive $\text{dist}(s, v_i)$. First, given that P_1^* and P_2^* have been traversed by HoD before the backward search, HoD should be able to compute the precise value of $\text{dist}(s, v_1)$ when after it visits v_1 . Now assume that, after examining v_j ($j \in [1, k-1]$), HoD correctly calculates $\text{dist}(s, v_j)$. Then, when HoD inspects v_{j+1} , it would identify $\text{dist}(s, v_j) + l(\langle v_j, v_{j+1} \rangle)$ as the length of a path from s to v_{j+1} . Given that P_1^* is a shortest path from s to t in G^* and that v_{j+1} immediately follows v_j on P_1^* , we have $\text{dist}(s, v_{j+1}) = \text{dist}(s, v_j) + l(\langle v_j, v_{j+1} \rangle)$. This indicates that HoD will correctly derive $\text{dist}(s, v_{j+1})$, which completes the proof.

(a) G_0 .

(b) G_1 .

(c) G_2 .

Figure A.6: HoD on the deterministic scale-free network.

Publication List:

- **Andy Diwen Zhu**, Wenqing Lin, Sibbo Wang, and Xiaokui Xiao. Reachability Queries on Large Dynamic Graphs: A Total Order Approach. Proceedings of the ACM SIGMOD International Conference on Management of Data (**SIGMOD**), pages 1323-1334, 2014.
- **Andy Diwen Zhu**, Xiaokui Xiao, Sibbo Wang and Wenqing Lin. Efficient Single-Source Shortest Path and Distance Queries on Large Graphs. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (**SIGKDD**), pages 998-1006, 2013.
- **Andy Diwen Zhu**, Hui Ma, Xiaokui Xiao, Siqiang Luo, Youze Tang and Shuigeng Zhou. Shortest Path and Distance Queries on Road Networks: Towards Bridging Theory and Practice. Proceedings of the ACM SIGMOD International Conference on Management of Data (**SIGMOD**), pages 857-868, 2013.
- Youze Tang, **Andy Diwen Zhu**, and Xiaokui Xiao. An Efficient Algorithm for Mapping Vehicle Trajectories onto Road Networks. The 20th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (**GIS**), 2012.
- Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, **Andy Diwen Zhu**, and Shuigeng Zhou. Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. Proceedings of the 38th International Conference on Very Large Data Bases (**VLDB**), pages 406-417, 2012.

References

- [1] <https://sourceforge.net/projects/ntu-sp-exp/>.
- [2] <http://algo2.itl.kit.edu/routeplanning.php>.
- [3] <http://www.dis.uniroma1.it/~challenge9/>.
- [4] <http://www.dis.uniroma1.it/challenge9/download.shtml>.
- [5] <http://vmlion25.deri.ie/>.
- [6] <http://barcelona.research.yahoo.net/webspam/datasets/uk2007/links/>.
- [7] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, pages 230–241, 2011.
- [8] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35, 2012.
- [9] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA*, pages 782–793, 2010.
- [10] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.
- [11] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.*, 28(4):1167–1181, 1999.
- [12] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory bfs algorithms. In *SODA*, pages 601–610, 2006.
- [13] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, pages 124–137, 2007.

REFERENCES

- [14] A.-L. Barabási, E. Ravasz, and T. Vicsek. Deterministic scale-free networks. *Physica A: Statistical Mechanics and its Applications*, 299(3):559–564, 2001.
- [15] H. Bast, S. Funke, and D. Matijevic. Transit: ultrafast shortest-path queries with linear-time preprocessing. In *Proc. of the 9th DIMACS Implementation Challenge*, pages 175–192, 2006.
- [16] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *ALENEX*, 2007.
- [17] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.
- [18] R. Bramandia, B. Choi, and W. K. Ng. Incremental maintenance of 2-hop labeling of large graphs. *TKDE*, 22(5):682–698, 2010.
- [19] J. Cai and C. K. Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *CIKM*, pages 119–128, 2010.
- [20] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, pages 493–504, 2005.
- [21] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, pages 893–902, 2008.
- [22] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*, pages 193–204, 2013.
- [23] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD*, pages 457–468, 2012.
- [24] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-reach: Who is in your small world. *PVLDB*, 5(11):1292–1303, 2012.
- [25] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, pages 193–204, 2008.
- [26] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

REFERENCES

- [27] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- [28] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, pages 117–139, 2009.
- [29] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *J. Comput. Syst. Sci.*, 72(5):813–837, 2006.
- [30] R. B. Dial. Algorithm 360: shortest-path forest with topological ordering [h]. *Commun. ACM*, 12(11):632–633, 1969.
- [31] R. B. Dial. Toward faster algorithms for dynamic traffic assignment. i. parametric quickest-path trees. *Networks*, 41(1):36–43, 2003.
- [32] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.
- [33] D. Eppstein and J. Wang. Fast approximation of centrality. *J. Graph Algorithms Appl.*, 8:39–45, 2004.
- [34] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006.
- [35] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
- [36] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.
- [37] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *PVLDB*, 6(6):457–468, 2013.
- [38] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.
- [39] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. Walking in facebook: A case study of unbiased sampling of osns. In *INFOCOM*, pages 2498–2506, 2010.

REFERENCES

- [40] F. Glover, R. Glover, and D. Klingman. Computational study of an improved shortest path algorithm. *Networks*, 14(1):25–36, 1984.
- [41] A. V. Goldberg. Shortest path algorithms: Engineering aspects. In *ISAAC*, pages 502–513, 2001.
- [42] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA*, pages 156–165, 2005.
- [43] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *ALENEX*, pages 129–143, 2006.
- [44] S. Gupta, S. Kopparty, and C. V. Ravishankar. Roads, codes and spatiotemporal queries. In *PODS*, pages 115–124, 2004.
- [45] T. Hagerup. Improved shortest paths on the word ram. In *ICALP*, pages 61–72, 2000.
- [46] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *FOCS*, pages 664–672, 1995.
- [47] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In *Proc. of the 9th DIMACS Implementation Challenge*, pages 73–92, 2006.
- [48] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.
- [49] R. Jin, N. Ruan, S. Dey, and J. X. Yu. Scarab: scaling reachability computation on large graphs. In *SIGMOD*, pages 169–180, 2012.
- [50] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, 36(1):7, 2011.
- [51] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *PVLDB*, 6(14):1978–1989, 2013.
- [52] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.

REFERENCES

- [53] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, pages 595–608, 2008.
- [54] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *TKDE*, 10(3):409–432, 1998.
- [55] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *TKDE*, 14(5):1029–1046, 2002.
- [56] P. N. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$ -time algorithm. *ACM Transactions on Algorithms*, 6(2), 2010.
- [57] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, 1997.
- [58] I. Krommidas and C. D. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *ACM Journal of Experimental Algorithmics*, 12, 2008.
- [59] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *SPDP*, pages 169–176, 1996.
- [60] J. Leskovec, L. Backstrom, J. M. Kleinberg, and J. M. Kleinberg. Meme-tracking and the dynamics of the news cycle. In *KDD*, pages 497–506, 2009.
- [61] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, 2007.
- [62] U. Meyer. Via detours to i/o-efficient shortest paths. In *Efficient Algorithms*, pages 219–232, 2009.
- [63] U. Meyer and V. Osipov. Design and implementation of a practical i/o-efficient shortest paths algorithm. In *ALENEX*, pages 85–96, 2009.
- [64] U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.

REFERENCES

- [65] U. Meyer and N. Zeh. I/o-efficient undirected shortest paths. In *ESA*, pages 434–445, 2003.
- [66] U. Meyer and N. Zeh. I/o-efficient shortest path algorithms for undirected graphs with random or bounded edge lengths. *ACM Transactions on Algorithms*, 8(3):22, 2012.
- [67] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *SODA*, pages 209–222, 2012.
- [68] K. Munagala and A. G. Ranade. I/o-complexity of graph algorithms. In *SODA*, pages 687–694, 1999.
- [69] I. Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.
- [70] R. Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81–87, 1997.
- [71] M. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *PVLDB*, 4:69–80, 2010.
- [72] L. Roditty. Decremental maintenance of strongly connected components. In *SODA*, pages 1143–1150, 2013.
- [73] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC*, pages 184–191, 2004.
- [74] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.
- [75] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, pages 568–579, 2005.
- [76] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS*, pages 200–209, 2005.
- [77] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *TKDE*, 22(8):1158–1175, 2010.
- [78] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2:1210–1221, 2009.

REFERENCES

- [79] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *EDBT*, pages 237–255, 2004.
- [80] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371, 2005.
- [81] D. Schultes. *Route Planning in Road Networks*. PhD thesis, University Karlsruhe (TH), 2008.
- [82] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*, pages 1009–1020, 2013.
- [83] H. Shi and T. H. Spencer. Time-work tradeoffs of the single-source shortest paths problem. *J. Algorithms*, 30(1):19–32, 1999.
- [84] C. Sommer. *Approximate Shortest Path and Distance Queries in Networks*. PhD thesis, The University of Tokyo, 2010.
- [85] F. W. Takes and W. A. Kusters. Determining the diameter of small world networks. In *CIKM*, pages 1191–1196, 2011.
- [86] Y. Tao, C. Sheng, and J. Pei. On k-skip shortest paths. In *SIGMOD*, pages 421–432, 2011.
- [87] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
- [88] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.
- [89] J. L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21(9):1505–1532, 1995.
- [90] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.
- [91] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, pages 913–924, 2011.

REFERENCES

- [92] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.
- [93] 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.
- [94] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pages 1601–1606, 2013.
- [95] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.
- [96] H. Yildirim, V. Chaoji, and M. J. Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. *CoRR*, abs/1301.0977, 2013.