Computing Paths in Massive Graphs

by

JUNHUA ZHANG

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

Australian Artificial Intelligence Institute (AAII)
Faculty of Engineering and Information Technology (FEIT)
University of Technology Sydney (UTS)

June, 2023

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, Junhua Zhang declare that this thesis, is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Junhua Zhang

Date: 20/06/2023

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude and appreciation to my advisor, Professor Lu Qin. His unwavering guidance and support have been essential in shaping my research career. As my mentor, his dedication, patience, and enlightened direction have guided me through the new terrain of research. His intelligence and professionalism have inspired me, sparking fresh ideas and perspectives each time we engaged in scientific discussion. Beyond a mentor-student relationship, I am privileged to consider him a valued friend whose care and willingness to support me in times of difficulty and failure has been a source of comfort. Without his invaluable contribution, the completion of this dissertation would have been impossible.

Secondly, I would like to thank my co-supervisor, Professor Ying Zhang, for his help and guidance. He has given me valuable advice and support for my research projects, which is indispensable for my Ph.D. study. In addition, he shared his valuable experience of research career with me and guided me for my future research career, which is crucial for my future research career.

Thirdly, I would like to thank Dr. Wentao Li and Prof. Yuan Long for their continuous guidance and support during my Ph.D. studies. Dr. Wentao Li is an experienced and hardworking researcher who served as a mentor during my Ph.D. studies. I have learned a lot from him, including research skills and presentation skills. In addition, he is a kind person and a good friend to me. He has given me a lot of advice and help not only in my research but also in my life. Prof. Long Yuan is also an experienced researcher who guided me in designing experiments and writing papers. All my research work was done in collaboration with them, and they played an important role in my research work.

Fourth, I would like to thank Prof. Xuemin Lin, Prof. Wenjie Zhang, Dr. Dong Wen and A/Prof. Lijun Chang for their valuable support and advice for my research work and research career. They are also my role models to become a good researcher.

I would also like to thank Dr. Xin Cao, Dr. Xiaoyang Wang, Prof. Fan Zhang, A/Prof. Yixiang Fang, A/Prof. Dian Ouyang, Dr. Cong Gai, Dr. You Peng, Dr. Yuxuan Qiu, Dr. Kai Wang, Dr. Hanchen Wang, Dr. Zhengyi Yang, Dr. Mingjie Li, Dr. Bohua Yang, Dr. Xiaoshuang Chen, Dr. Longbin Lai, Dr. Xubo Wang, Prof. Shiyu Yang, Prof. Peng Cheng, Dr. Yuren Mao, Dr. Yixing Yang, Dr. Yu Hao, Dr. Qingyuan Linghu, Dr. Michael Ruisi Yu, Dr. Chenji Huang, Mr. Yilun Huang, Mr. Yuanhang Yu, Mr. Rong Hu, Mr. Kongzhang Hao, Mr. Lantian Xu, Mr. Deming Chu, Mr. Jianwei Wang, Mr. Shiding Zhang, Mr. Yizhang He, Mr. Qingqiang Sun, Mr. Shunyang Li, Mr. Gengda Zhao, Mr. Qinghuai Feng, Ms. Yiqi Wang, Mr. Zhuo Ma, Ms. Yuting Zhang, Ms. Yanping Wu, Mr. Guangxin Su, Mr. Yufan Sheng, Dr. Ming Li, for their support and the good times we spent together.

Last but not least, I would like to thank my father, Mr Peigong Zhang, and my mother, Mrs Xuju Xiong, for their selfless love and endless encouragement. I would also like to thank my sister, Mrs Yu Zhang, and other relatives and friends for their love and support.

ABSTRACT

Graph is a widely used data structure for representing real-world entities and their relationships. Graph queries play an important role in the processing of graphs, and path queries are one of the most important types of query operations on graphs. A path query attempts to retrieve the path from one vertex to another, and has numerous real-world applications, including ontology reasoning, geographic navigation, and link analysis. Given the importance of path queries in real-world graphs, this thesis focuses on investigating efficient methods for computing path-related queries in massive graphs.

First, we study how to efficiently process reachability queries on distributed graphs, which check whether there is a path from one vertex to another. Real-world massive graphs are typically distributed across multiple data centers due to their size. When performing reachability queries on these distributed graphs, reachability labeling methods ensure fast query processing through lightweight indexes. One of the most well-known labeling methods is TOL; however, TOL is a *serial* algorithm that cannot handle distributed graphs. For this reason, we investigate the limitations of TOL and then propose a filtering-refinement framework for index construction. In addition, we design distributed labeling algorithms and batch-processing techniques to improve efficiency.

Second, we study shortest path queries on complex graphs, which return the shortest path between two vertices. To handle shortest path queries, one option is to use traversal-based methods (e.g., breadth-first search); another option is to use extension-based methods, i.e., extending existing methods that use indexes to handle shortest-distance queries to support shortest-path queries. These two approaches make different trade-offs regarding query time and space cost, but it lacks a comprehensive study of their performance on real-world graphs. Moreover, extension-based approaches use additional attributes to extend the index, resulting in high space costs after extension. To address these issues, we thoroughly compare the two approaches and propose a new extension-based approach, Monotonic Landmark Labeling (MLL), to reduce the required space cost while guaranteeing query time.

Third, we study label-constrained shortest path queries on road networks, which request the shortest path between two vertices in labeled graphs such that all edges on the path satisfy the label constraint. Computing the shortest path between two vertices is a fundamental problem for road networks. Most existing works assume that edges in the road networks do not have labels, but in many real applications, the edges have labels, and label constraints may be placed on the edges of a valid shortest path. Therefore, we study the label-constrained shortest path queries. To process these queries efficiently, we adopt an index-based approach and propose efficient algorithms for query processing and index construction with good performance guarantees.

We conduct extensive experiments for evaluation. The results validate the effectiveness and efficiency of our proposed methods.

PUBLICATIONS

- ***Junhua Zhang**, Wentao Li, Lu Qin, Ying Zhang, Dong Wen, Lizhen Cui, Xuemin Lin. "Reachability Labeling for Distributed Graphs." In 2022 IEEE 38th International Conference on Data Engineering (ICDE). IEEE, 2022. (**Chapter 3**)*

- ***Junhua Zhang**, Wentao Li, Long Yuan, Lu Qin, Ying Zhang, Lijun Chang. "Shortest-Path Queries on Complex Networks: Experiments, Analyses, and Improvement." In Proceedings of the 48th International Conference on Very Large Databases. VLDB, 15, 2022. (**Chapter 4**)*

- ***Junhua Zhang**, Long Yuan, Wentao Li, Lu Qin, Ying Zhang. "Efcient Label-Constrained Shortest Path Queries on Road Networks: A Tree Decomposition Approach." In Proceedings of the 48th International Conference on Very Large Databases. VLDB, 15(3): 686-698, 2022. (**Chapter 5**)*

- ***Junhua Zhang**, Long Yuan, Wentao Li, Lu Qin, Ying Zhang. "Label-Constrained Shortest Path Querying on Road Networks.". In submission.*

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Graphs are widely used data structures that represent real-world entities and their relationships [75]. Graph queries play an important role in processing graphs. Path queries, one of the most important types of query operations on graphs, involve finding the path from one vertex s to another vertex t in a graph G. These queries have numerous real-world applications, including ontology reasoning [97], geographic navigation, and link analysis. Furthermore, they serve as building blocks for various fields such as social sciences, computational biology, and software engineering [48].

In view of the importance of the paths in real-world graphs, this thesis identifies and studies three types of closely related path queries:

1. **Reachability Query.** Given two vertices, the reachability query asks if a path exists from one vertex to another; it tests the existence of a path. It has applications in areas like ontology reasoning and computational biology. It is a more generalized and simpler type of path query.

2. **Shortest Path Query.** In some optimization problems, checking the existence of paths could be insufficient; the shortest path is instead preferred.

For instance, in road navigation, one wants to find a shortest path to the destination; in social networks, the shortest path can be used to measure the closeness of two individuals. This query type is relatively more specialized and complex compared to the reachability query.

3. **Label Constrained Shortest Path Query.** In most cases, the shortest path should be sufficient and ideal. However, in some real-world applications, the edges in graphs have labels and label constraints may be placed on the edges. Consequently, a shortest path may not satisfy the label constraints. For example, the road network contains different types of roads, including toll roads, which some travelers may want to avoid such toll roads to save money. In such cases, the shortest path that doesn't include any toll roads is desired. This query type is more specialized and complex compared to the previous ones.

The reasons to study these three query types in this thesis is multi-fold. First, these three queries are all related to computing a path from one vertex to another, ranging from the more general and simpler reachability query to the more specialized and complex label constrained shortest path query. They are closely related and collectively serve the central topic of path computation in this thesis. Second, the techniques used to answer these path queries share similarities. Methods such as hub labeling, contraction hierarchies, and index-assisted traverse can all be applied to these path queries. The solution to one query may also provide understandings and insights for other queries. Third, although these path-related queries share many similarities, each of them presents unique challenges in different settings. Therefore, the goal of this thesis is to propose and devise algorithms and indexes to address these unique challenges faced by each of the path queries.

1.1 Distributed Reachability Labeling

As a common graph operation, reachability query $q(s,t)$ asks whether there is a path from vertex s to vertex t. Due to the importance of reachability query processing, many approaches have been proposed. These approaches are usually classified into three categories [97]: 1) index-free approaches that use the online search [11] (e.g., breadth-first search or depth-first search) on graphs to determine whether one vertex can reach another [80]; 2) index-assisted approaches that accelerate the online search using auxiliary information [46, 96, 89, 79]; 3) index-only approaches that *avoid* the online search using an offline index [29, 77, 27, 28, 101].

Most current approaches are centralized: they assume that graphs reside in the main memory [99]. However, as the graph size increases, real-world graphs are typically distributed in multiple data centers [34]. When performing reachability queries on distributed graphs, the pioneer work [34, 99] implements the online search in a distributed manner for query processing. However, the query latency can be high due to the need to access distributed graphs during the query. This makes methods that require the online search (i.e., index-free or index-assisted approaches) undesirable, especially when a large number of queries need to be processed.

To speed up query processing on distributed graphs, an alternative idea is to use index-only approaches — the index created offline eliminates the dependence on the original graph during querying [101]. Specifically, consider a graph $G(V,E)$ with vertex set V and edge set E, index-only approaches create an index that assigns an out-label set $\mathsf{L}_{\mathsf{out}}(v)$ and an in-label set $\mathsf{L}_{\mathsf{in}}(v)$ for each vertex $v \in V$. The out-label set $\mathsf{L}_{\mathsf{out}}(v)$ of v contains vertices that v can reach, while the in-label set $\mathsf{L}_{\mathsf{in}}(v)$ contains vertices that can reach v. To answer the query $q(s,t)$ between vertices s and t, we *only* need to check whether there exists a common vertex w between $\mathsf{L}_{\mathsf{out}}(s)$ and $\mathsf{L}_{\mathsf{in}}(t)$: the existence of w means that s can reach t via w.

The state-of-the-art index-only methods is Total Order Labeling (TOL) [101]. TOL assigns an order value to each vertex in a graph, and then selects vertices for labeling in a decreasing sequence of order values. When labeling a vertex v, TOL tries to add v to the in-label/out-label sets of other vertices, using a **pruning operation**. When all vertices have finished labeling, the in-label/out-label sets generated by TOL for each vertex are used as an index.

The pruning operation of TOL is a double-edged sword: TOL reduces the index size by eliminating redundancy through the pruning operation; however, each vertex v needs to wait for vertices whose order values are higher than v to finish labeling, thus making the pruning operation of v feasible. This means that the execution of TOL is inherently serial. In other words, TOL *does not work properly on distributed graphs.*

On the other hand, for a distributed graph, the index created by TOL can efficiently support queries. This is because the index of TOL is small enough that we can put it on a single machine to achieve fast in-memory queries. For example, the index size for graph SK (see Table 5.2 for graph's details) with billions of edges is bounded by 1 GB. The main purpose of our thesis is to design new labeling methods to handle a distributed graph while obtaining the same index as TOL.

For this purpose, we delve into the labeling process of TOL. We find that when labeling a vertex v, v joins the label sets of some specific vertices. These vertices are defined as the **backward label set** of v. The working process of TOL can be equated to finding the backward label set for each vertex v. To find v's backward label set, we use a filtering-and-refinement framework, thereby avoiding the pruning operation used by TOL. Specifically, we first generate a super-set of the backward label set of v as candidates, and then remove invalid elements from candidates to obtain the actual backward label set. This novel framework gets the same index as TOL while letting all vertices run in parallel. Based on this framework, we design efficient labeling algorithms and provide distributed implementations. In addition, we split vertices into batches for labeling to further improve efficiency.

The contributions are summarized as follows.

- Analysis of TOL's limitation. We investigate TOL's limitation, i.e., the pruning operation of TOL makes parallel work challenging. This motivates the design of new labeling methods.

- Novel labeling algorithms. We find that each vertex's labeling process can be replaced by finding the backward label set for that vertex. We use a filtering-and-refinement framework to find the backward label set of each vertex in parallel. Using this framework, we propose new labeling algorithms and provide implementations in a distributed system.

- Batch labeling optimization. To further improve the labeling algorithms' efficiency, we split vertices into batches and then construct the index in batches.

- Extensive empirical studies. We conduct numerous experiments to validate the efficiency of the proposed labeling algorithms. On medium-sized graphs, our algorithms can outperform TOL by nearly an order of magnitude. Furthermore, on billion-sized graphs, we can create indexes in half an hour while TOL cannot.

The details of this work are presented in Chapter 3.

1.2 Shortest-Path Queries on Complex Graphs

Many real-world graphs, such as social networks, web graphs, and biological networks, are called complex networks because of their complex topology [31, 15]. These complex networks can have millions or even billions of vertices and edges [60, 59], necessitating the development of efficient tools to support operations on these graphs [7, 6].

In a graph G, the shortest-path query $\mathsf{Q}_{\mathsf{P}}(s,t)$ returns the shortest path between two vertices s and t. One option to handle shortest-path queries is to use **traversal-based methods**, such as breadth-first search (BFS, for unweighted graphs) [11] or Dijkstra's algorithm (for weighted graphs) [50]. However, traversal on large graphs is slow because its runtime is proportional to the graph size [62]. To speed up graph traversal, several preprocessing techniques [78, 91], such as Highway Hierarchies [76], or Contraction Hierarchies [37], have been proposed to limit the traversal scope. However, traversal-based methods still *take a long time to process a query*, even with the preprocessing techniques [62]. Moreover, these preprocessing techniques often rely on the properties of road networks (e.g., planarity and hierarchical structures [3]), rendering them inapplicable to dealing with complex networks [62].

Another option is to use **extension-based methods**, i.e., extending methods designed for shortest-distance query processing to support shortest-path queries. The shortest-distance query is an operation closely related to the shortest-path query, which returns the length $dist(s,t)$ of the shortest path between two vertices s and t [59]. In recent years, many approaches have been proposed to build indexes for shortest-distance query processing in complex networks [62, 59, 60, 7, 6], and the well-known approaches are Pruned Landmark Labeling (PLL [6]) and Core-Tree Labeling (CTL [60]). To extend these methods, we can add an extra attribute to each entry in the index for path recovery.

Although the extension-based methods can handle shortest-path queries rapidly, *the introduction of extra attributes makes the required space cost too high.*

Motivations. Traversal and extension-based methods make different trade-offs in query time and space cost: traversal-based methods do not require high space cost but have no guarantee of query time; extension-based methods provide a quick query response, but their space costs are high. Yet, comprehensive studies of these two types of methods' performance on real-world graphs are lacking. This makes it hard for practitioners to select an appropriate method for applications that use shortest-path queries as a basic component. Also, extension-based methods typically add extra attributes to the original index designed for shortest-distance queries, thus allowing tracking of all vertices on the shortest path. Such an extension often leads to a too large index to support shortest-path queries.

Our Solution. To address the aforementioned issues, we thoroughly compare various methods for handling shortest-path queries. Also, we propose a new extension-based approach, Monotonic Landmark Labeling (MLL), to enable the index designed for shortest-distance queries to work for shortest-path queries. MLL non-trivially creates an additional lightweight index as a plug-in to the original index; instead of extending every index entry (which would result in an extended index nearly twice the size of the original index). As a result, MLL can still give rapid query responses but with a low (extra) space cost.

Contributions. The contributions are summarized as follows.

- Efficient extension of distance query processing methods. We extend the shortest-distance query processing methods PLL [6] and CTL [60] to allow them to efficiently handle shortest-path queries. Our idea is to add an extra attribute to each index entry so that the path can be found by tracking each vertex on the shortest path using the extra attribute. We discuss the correctness and time complexity of these extension-based methods in processing shortest-path queries.

- A new extension-based approach. We propose a new extension-based approach MLL tailored for shortest-path queries. MLL non-trivially builds an extra lightweight index on top of the index designed for shortest-distance queries. MLL works by decomposing the shortest path between two vertices into several subpaths, which are then indexed. At query time, the shortest path can be found efficiently by finding and splicing subpaths. We verify MLL consumes less space than extending each index entry with extra attributes while still having a fast query speed.

- Extending MLL to handle weighted and directed graphs. We describe how to adapt the proposed MLL approach to handle both weighted and directed graphs. We thus enable MLL to work for more general graphs.

- Comprehensive experimental studies. We select four traversal-based and three extension-based methods for experimental comparisons. We extensively evaluate the performance of various methods on ten real-world graphs. We also examine the impact of directed graphs on the index size of MLL. To the best of our knowledge, this is the first work to empirically compare shortest-path query processing methods on complex networks.

The details of this work are presented in Chapter 4.

1.3 Label Constrained Shortest Path on Road Networks

Computing the shortest path between two locations is one of the fundamental problems in road networks [10, 58, 63, 92, 38, 51, 76, 37, 88, 102, 69]. In real road networks, not all roads are the same, for example, highways allow faster travel, toll roads cost money, and the transport of hazardous goods is forbidden on roads in water protection areas. Therefore, many applications place constraints on the edges appearing on a valid shortest path when computing the shortest path, which leads to the study of label-constrained shortest path queries [71, 41]. Formally, given a road network G where each edge has a label, a source vertex s, a target vertex t, and an edge label set $\mathcal{L}$, a label-constrained shortest path query $q = (s, t, \mathcal{L})$ aims to compute the shortest path from s to t such that the labels of edges on the shortest path are contained in $\mathcal{L}$.

Label-constrained shortest path queries can be used in many real application scenarios such as personal routine planing [71] and emergency evacuation navigation [61]. For example, the shortest path from Irvine, CA to Riverside, CA travels along State Route 261, which is a local toll road through this area. For a user who does not wish to pay the toll fee, we can find the shortest path from Irvine to Riverside that actually avoids all toll roads by a label-constrained shortest path query $q = (''Irvine'', ''Riverside'', \mathcal{L})$ in which $\mathcal{L}$ does not contains the label representing toll road [71]. In China, new drivers who get the driver license in less than 12 months are not allowed to drive cars on expressway alone for safety [90]. Therefore, the expressway should be avoided when planning routines for these new drivers, which can be achieved by the label-constrained shortest path queries where $\mathcal{L}$ does not contain the label representing expressway. In emergency evacuation navigation, the recommended evacuation route should avoid the roads in dangerous areas [61], which can be achieved by the label-constrained shortest path queries where $\mathcal{L}$ does not contain the label representing roads in dangerous areas.

Motivation. A straightforward approach for label-constrained shortest path queries is to use Dijkstra's algorithm [33] by only visiting the edges whose edge label is in $\mathcal{L}$ during the traversal. Although this approach can compute the required shortest path, as the road network is large in real applications, it cannot satisfy the real-time requirements for the label-constrained shortest path queries as it may traverse the whole road network when s and t are far away from each other. As a result, researchers resort to index-based techniques to accelerate the label-constrained shortest path query processing [71, 41].

The state-of-the-art index-based approach is edge-disjoint partition (EDP) [41]. Intuitively, EDP partitions the road network based on each edge label and caches the computed shortest path information for processed queries in each partition as the index structure. When a new query comes, the cached information is used to accelerate the query processing. Obviously, the performance of EDP heavily depends on the hit ratio of the index. However, there are no theoretical guarantees on the hit ratio of EDP, as it just caches the computed shortest path in each partition for the processed queries, but the newly issued queries may distribute diversely and the label-constrained shortest path for a specific query maybe involve several partitions. Consequently, it is quite possible that the hit ratio for a specific query is low and EDP degenerates into an online search algorithm similar to direct using Dijikstra's algorithm. Even worse, the performance of EDP could be poorer than direct using Dijikstra's algorithm as more vertices may be visited due to the introduction of the index. Considering road networks in real world are typically large and label-constrained shortest path queries are issued frequently, EDP cannot satisfy the real-time requirements in practical applications either.

Motivated by this, in this thesis, we re-investigate the label-constrained shortest path queries on road networks and aim to design an efficient label-constrained shortest path query processing algorithm with non-trivial theoretical performance guarantees.

Our approach. We also resort to index-based techniques to accomplish efficient label-constrained shortest path query processing. As tree decomposition can decompose a road network into a tree-like structure with small treeheight and treewidth, it achieves great success in computing the shortest path on unlabelled road networks recently [68, 60]. Inspired by this, we revisit the tree decomposition based indexing techniques for shortest path problem.

We start from the shortest path queries on *unlabelled road networks*. Regarding this problem, the state-of-the-art tree decomposition based indexing approach processes a query with time complexity $O(h \cdot \omega^2)$, where h, ω is the treeheight, treewidth of the tree decomposition, respectively [88]. By carefully analyzing the properties of the tree decomposition, we present an algorithm based on the tree decomposition for the shortest path queries, and non-trivially prove that the time complexity of the algorithm to process a query can be bounded by $O(h \cdot \omega)$, which reduces the time complexity of [88] by a factor ω (refer to Theorem 13). Since h and ω are small for road networks, it means we can efficiently process the shortest path queries on unlabelled road networks based on tree decomposition with theoretical performance guarantees.

Based on the above findings, we explore the tree decomposition based indexing solution for label-constrained shortest path queries. A direct indexing solution is as follows: for each induced road network by one possible combination of the edge label set in Σ, where Σ is the finite alphabet used for the labels of edges in G, we treat the induced road network as an unlabelled road network and build the tree decomposition based index for it. Given a label-constrained shortest path $q = (s, t, \mathcal{L})$, we retrieve the corresponding index for the edge label set $\mathcal{L}$ and compute the shortest path accordingly. Clearly, this approach fully utilizes the efficiency of the tree decomposition based indexing technique for shortest path queries on unlabelled road networks. However, the total number of indices constructed in this approach is $2^{|\Sigma|}$. It is prohibitive to construct and maintain such a number of indices, which makes this approach unscalable to handle large road networks in real applications.

Observing the indices constructed in the direct solution, we find that lots of redundant information regarding label-constrained shortest path computation are stored among different indices. Following this observation, we conceive of reducing the redundant information among these $2^{|\Sigma|}$ indices and integrating them into a holistic compact index structure. To make our idea practically applicable, the following issues need to be addressed: (1) how to design such an index that the redundant information is reduced while the efficiency of query processing is not compromised? (2) how to efficiently construct the index for a road network, especially when the road network is large?

Contribution. In this thesis, we address the above issues and make the following contributions:

- A new tighter bound on the shortest path query processing on unlabelled road networks. We revisit the problem of tree decomposition based indexing for shortest path queries on unlabelled road networks and present an algorithm for this problem. We non-trivially prove the time complexity of the algorithm is $O(h \cdot \omega)$, while the state-of-the-art tree decomposition based indexing approach for this problem is $O(h \cdot \omega^2)$.

- Efficient algorithms for label-constrained shortest path queries with theoretical performance guarantees. We design a new tree decomposition based index for the label-constrained shortest path queries. Based on the index, we propose an algorithm to answer the queries. We also design an algorithm to construct the index. Moreover, considering the road networks in real applications could be very large, we exploit parallel computing techniques to further speed up the construction of the index. All these algorithms have bounded worst-case time complexities and can handle large road networks efficiently.

- Extensive performance studies on real road networks. We conduct extensive performance studies on eight real large road networks including the whole road network of the USA. The experimental results demonstrate that: 1) our algorithm can achieve up to 2 orders of magnitude speedup in query processing compared to the state-of-the-art approach while consuming much less index space. 2) our algorithms can efficiently construct the index for large road networks, especially the parallel index construct algorithm which completes the index construction for the whole road network of USA within 1,000 seconds.

The details of this work are presented in Chapter 5.

1.4 Roadmap

The rest of this thesis is organized as follows. Chapter 2 discuss related works. Chapter 3 introduce the distributed reachability labeling. Chapter 4 presents the shortest-path queries on complex graphs. Chapter 5 discuss the label constrained shortest path queries on road networks. Chapter 6 concludes the whole thesis.

Chapter 2

LITERATURE REVIEW

Due to the wide applications of path queries, efficient computation of paths has drawn a lot of research work. In this chapter, we will first survey literature on reachability queries, and then the research works on shortest path queries, and finally the label constrained path queries.

Path queries are gaining importance due to their wide range of applications, leading to significant attention from the research community on improve path computation. This chapter delves into related works on path computation and is structured as follows: We first examine the existing literature on reachability queries, and then explore research work devoted to shortest distance and path queries. Finally, we discuss works on label-constrained reachability and shortest path queries.

2.1 Reachability Queries

There are three streams of techniques used in literature to study the reachability queries: index-free, index-assisted and index-only approaches.

2.1.1 Index-free Approaches

The most straightforward method to answer reachability queries is to perform an online search on the graph [34]. This search can take the form of either a breadth-first search (BFS) [19] or a depth-first search (DFS) [9]. In practice, given a query examining if vertex u can reach vertex v, we can launch a BFS or DFS from u. If vertex v is encountered during the search process, it indicates that u can reach v. Conversely, if v is not found, it implies the opposite. However, this approach has limitations due to the need to use the graph at the query time, potentially leading to considerable query latency.

2.1.2 Index-assisted Approaches

Index-assisted methods speed up the online search by using auxiliary structures. The auxiliary structures can be subgraphs [46], multiple intervals [96], independent permutations [89], bloom filters [79], or partial label sets [99].

The state of the art approach in this category is BFL [79]. The basic idea of BFL is that if vertex s can reach vertex t, then s can reach all descendants $\mathsf{DES}(t)$ of t, i.e., $\mathsf{DES}(t) \subseteq \mathsf{DES}(s)$. Through a Bloom filter, BFL maps the descendants $\mathsf{DES}(v)$ of each vertex v to a subset as the out-label set of v [14]. At query time, the labels alone can be used to determine that s cannot reach t: if t's out-label set is not fully contained in s's out-label set, then $\mathsf{DES}(t) \not\subseteq \mathsf{DES}(s)$ and thus $s \not\rightarrow t$. However, if s can reach t, then BFL needs to perform a graph search to report the answer. Similarly, BFL uses the Bloom filter to map each vertex's ancestors to generate its in-label set. For BFL, as the index cannot answer all queries, the graph needs to be loaded into memory at query time (whereas the index-only approach eliminates the requirement of the original graph at query time).

2.1.3 Index-only Approaches

Many techniques pre-compute an index and answer reachability queries by only using the index. Most of them either compute and compress a transitive closure or build a 2-hop index.

Transitive Closure Compression. One category of techniques pre-computes and compresses the transitive closure (TC), which represents all vertices reachable by a given vertex. Given its large space consumption $O(n^2)$, various methods [67, 84, 4, 24, 49] have been proposed to compress the TC. To efficiently represent the transitive closure, some methods, such as interval list [67], interval-based [86] and optimal-tree[4], adopt a tree-based approach. The methods proposed in [85, 81, 22] are extensions of these methods in this category. Some other works [43, 24, 30] use chain-based methods to compresses the TC. In the chain-based methods, a directed acyclic graph is decomposed into chains, preserving the reachability backbone. Each chain and its vertices hold specific information to answer reachability queries. The efficiency of these methods is highlighted by the query time of $O(logk)$ and a space consumption of $O(nk)$, where n is the number of vertices and k is the number of chains.

2-hop Labeling. Originally proposed by Cohen [29] as a solution for answering reachability and distance queries over large graphs, 2-hop labeling has been further developed by many researchers using various methods [77, 27, 28, 26, 95, 101]. In 2-hop labeling, each vertex, denoted by u, in a graph is assigned two label sets: an in-label set, $\mathsf{L_{in}}(u)$, and an out-label set, $\mathsf{L_{out}}(u)$, both of which contain subsets of vertices. When a reachability query, i.e. $Q(u, v)$, is posed, the method checks for any common vertices between $\mathsf{L_{out}}(u)$ and $\mathsf{L_{in}}(v)$. If the intersection of $\mathsf{L_{out}}(u)$ and $\mathsf{L_{in}}(v)$ is not empty, it indicates that vertex u can reach v. If not, u cannot reach v.

Cohen et al.[29] were the first to present a 2-hop labeling approach for constructing a 2-hop index. For a given vertex u in graph G, they identify two sets: S, the set of vertices that can reach u, and D, the set of vertices that u can reach. By the transitive property of reachability, for every $s\ inS$ and $d\ inD$, s can reach d. Thus, S, u, and D form a cluster (S, u, D), and u is then removed from G. Repeating this process for all vertices in G produces a set of clusters that cover the reachability property of the entire graph, resulting in a 2-hop cover of G. However, Cohen et al .[29] show that computing a 2-hop cover with minimum size is NP-hard. As a result, they rely on heuristic methods to compute a small 2-hop cover. The core idea is to first identify the cluster (S, u, D) with the maximum value of $|S||D|(|S| + |D|)$, then insert u into the out-label set of each $s \in S$ and the in-label set of each $d \in D$. Then the vertex u and its incident edges are removed from the original graph G to form a new graph G_{n-1}. The above process is repeated with the new graph G_{n-1} until all vertices are removed from G, at which point the 2-hop labeling is complete. Despite the fact that the proposed method can generate a 2-hop labeling with a very small index size, the computation time of $O(n^3|TC|)$ is prohibitively high and unsuitable for large graphs.

Several heuristic methods [77, 27, 28, 26, 95, 6, 101] have been proposed to address the time-consuming nature of identifying the cluster with the maximum value of $|S||D|(|S| + |D|)$ in the 2-hop labeling method. These methods do not compute the transitive closure for all vertices in each iteration. Instead, they indirectly determine the order of the vertices to be labeled and compute only the cluster of the current vertex.

One such approach, proposed by Cheng et al. [27], is based on a 2-dimensional geometric map. This involves creating two spanning trees and constructing a 2-dimensional reachability map.

Another method proposed by Cheng et al. [26] uses topology folding to order vertices. This method compresses a graph G into a Directed Acyclic Graph (DAG), sorts the vertices based on topological order, and then divides them into several topological levels. The main advantage of this topological level-based method is its ability to reduce the computational cost of transitive closure.

In contrast, Jin et al. [48] order vertices based on their degree, using the value $|N_{in}(v)+1| \times |N_{out}(v)+1|$ to rank vertices. These heuristic ordering methods have been unified by Zhu et al. [101] into a Total Order Labeling(TOL) framework.

2.2 Shortest Distance and Path Queries

Due to the importance of shortest distance and path queries in graphs, there are plethora of works in the literature. In this section, we will first discuss the works on shortest distance and path queries and then the constrained shortest path queries.

2.2.1 Search Based Methods

The basic methods to compute shortest path queries are search based methods. BFS and Dijkstra [33] are two classic methods for computing single source shortest path tree on unweighted and weighted graphs in $O(m+n)$ and $O(m+n\log n)$ respectively. For point-to-point shortest path queries, a basic approach to improve BFS and Dijkstra is to use bidirectional search. The bidirectional search methods initiate two searches from the two query vertices. Once the two searches meet, then a shortest path is found. The bidirectional search improve the basic searches by reduce the number of vertices visited during the search. However, these methods are not efficient and many heuristic methods [38, 39, 42, 47, 66, 35] are proposed to accelerate the search procedure. ALT [38] pre-computes the shortest between a set of landmark vertices and other vertices and use triangle inequality to prune the search space of A*. Gutman [39] introduce a new concept REACH to check if a vertex is on the shortest path during search. Hilger et al. [42] use the concept of arg-flags to help remove unpromising edges during search. Similar to ALT [38], Wang et al. [35] pre-compute the shortest path from a set of landmark vertices to other vertices to accelerate the bidirectional BFS on unweighted graphs.

2.2.2 Hierarchical Methods

Hierarchical strategies are often used to handle shortest distance/path queries in road networks by exploiting the inherent hierarchy of these networks. The principle behind these strategies is that longer shortest paths ultimately connect to a condensed arterial network of major roads, such as highways [76, 37, 102, 51].

HiTi, proposed by Jung et al. [51], improves query processing speed by segmenting the graph and creating a hierarchical structure. Techniques such as Highway Hierarchies [76] and Contraction Hierarchies [37] have proven effective in reducing the search space. The Contraction Hierarchies technique constructs a topological order of a graph by forming shortcuts from lower ranked vertices to higher ranked ones, based on an overall order of vertices.

The Arterial Hierarchy (AH) approach, introduced by Zhu et al. [102], builds on the principles of Contraction Hierarchies. However, it creates shortcuts by superimposing 4×4 grids on the network, taking advantage of the 2-dimensional spatial properties within the network. The query processing in AH mirrors that of Contraction Hierarchies. It uses the bidirectional Dijkstra's algorithm and restricts the expansion direction to move only from lower to higher ranked vertices.

2.2.3 Labeling based Methods

Labeling methods, highlighted in numerous studies [29, 2, 1, 6, 5, 68, 25, 44], form a crucial category for shortest distance queries. They allow these queries to be solved using only an index, thus negating the need for graph traversal. These methods compute a label, $L(u)$, for each graph vertex u, which contains a set of vertices and their corresponding distances to u. A distinctive feature of these labeling methods is that they guarantee a shortest path cover property. This implies that for any two vertices s and t, the intersection of their labels, $L(s) \cap L(t)$, contains a vertex w that exists on the shortest path from s to t. Mathematically, this relation is expressed as $d(s,t) = d(s,w) + d(w,t)$. This coverage property is the basis for handling shortest path queries.

Cohen et al. [29] were the pioneers in proposing the 2-hop index for shortest path and reachability queries. They also proved that constructing a minimum 2-hop index is an NP-hard problem. As a result, they proposed heuristic methods for constructing a minimum 2-hop index, although these methods come with a significantly high computational cost.

Building on the concept of the 2-hop index, subsequent work by Abraham et al.[2, 1] used vertices visited by the upward traversal of vertex u on Contraction Hierarchies (CH) as hub vertices in u's label. The Pruned Landmark Labeling technique[6] computes a vertex order and then computes the labels of vertices by pruning landmarks according to this order. In contrast, the Hop-Doubling Labeling method [44] treats edges as the initial index and computes the 2-hop labels by doubling the path length in each iteration.

Inspired by pruned landmark labeling [6], Akiba et al. [5] introduced pruned highway labeling (PHL). Instead of using hubs as labels for each vertex, PHL uses paths, with the goal of encoding more information in each label. During the indexing phase, the algorithm decomposes the road network into disjoint shortest paths, and then computes a label for each vertex that contains the distance to vertices in a small subset of the computed paths.

Given the high cost associated with constructing 2-hop indexes, numerous techniques have also been proposed to parallelize the indexing process. Li et al. [59] developed a parallel distance labeling algorithm for unweighted graphs that builds indexes layer by layer. While Lakhotia et al.[55] provided a distributed distance labeling algorithm tailored for weighted graphs.

2.2.4 Tree Decomposition based Methods

Tree decomposition, first studied in [40] and further explored in [72], is a process that maps a graph into a tree, where each tree node comprises a subset of the vertices of the graph. This technique can be used to determine the treewidth of a graph and, by using dynamic programming, can speed up the solution of certain computational problems on the graph. It has been proved by [8] that determining whether the treewidth of a graph exceeds a given value is an NP-complete task.

Various heuristics for tree decomposition are outlined in [93], one of which is the widely used Minimum Degree Elimination (MDE) heuristic [13]. A notable feature of tree decomposition is its vertex-cut property, which makes it particularly suitable for shortest path queries.

TEDI [88] is the pioneering work that uses tree decomposition for shortest path queries. It first investigates the theoretical foundations of applying tree decomposition to such queries. Based on these foundations, it then designs an index, called TEDI, that relies on tree decomposition. For each tree node, TEDI precomputes the shortest path between each pair of vertices within the node and stores it in a hash table. This index takes up $O(n \cdot \omega)$ space, where n is the number of vertices and ω is the width of the tree. At query time, given two vertices s and t, TEDI first identifies the two tree nodes that contains s and t, respectively. It then finds the lowest common ancestor (LCA) of the two nodes in the tree. Using the vertex cut property of tree decomposition, it computes the shortest paths or distances from s (or t) to the vertices in their common ancestor. Since the LCA is also a vertex cut of the graph, the shortest path or distance can be determined using the previously computed shortest paths or distances. The query time of TEDI is $O(\omega^2 \cdot h)$, where h is the height of the tree.

H2H [68] is another method for handling shortest distance queries that uses the concept of tree decomposition. Unlike TEDI, H2H computes not only the shortest distance between vertices in each tree node, but also the shortest distance to all vertices in its ancestor nodes. The index size of H2H is $O(n \cdot h)$, where n is the number of vertices and h is the height of the tree. Despite having a larger index than TEDI, H2H processes shortest distance queries faster. Given two vertices s and t, after identifying the LCA of the tree nodes containing s and t, H2H directly uses the shortest distances from s (or t) to the vertices in the LCA to determine the distance between s and t. This significantly reduces the query time to $O(\omega)$, where ω is the tree width. Building on H2H, Chen et al. in P2H [25] further improve query efficiency by using vertex projection to identify a smaller vertex cut during query processing.

Many of the aforementioned studies either focus on road networks or conduct experiments on smaller graphs, and the effectiveness of tree decomposition on large, complex networks remains unproven. Research presented in [64, 68] suggests that complex graphs, such as web graphs and social networks, have a large dense core and tree-like fringe. These structures can have very large treewidth, making tree decomposition potentially unsuitable for them. Despite these limitations, efforts [60, 7, 64] have been made to use core-tree decomposition for shortest distance indexing.

Akiba et al. [7] were the first to apply core-tree decomposition to complex graphs. They used a tree-decomposition-based distance index for the tree-like fringe, while computing and storing pairwise shortest distances for vertices within the dense core. Both the tree index and the core index are used to answer shortest distance queries.

The recent work CTL [60] by Li et al. further improves the scalability of shortest distance queries by combining the advantages of the two approaches: Pruned Landmark Labeling (PLL) and Core-Tree Index. For the tree-like edge part of the core-tree decomposition, they use tree decomposition to derive multiple trees and index the distances from each vertex to its ancestors (excluding vertices only in the core part). For the core part, they use PLL to index the shortest distance. Like the approach in [7], both the core index and the tree index are used for shortest distance queries.

2.3 Label Constrained Path Queries

2.3.1 Label-Constrained Reachability Query

Label-constrained reachability queries have also been explored in the literature. Such a query, denoted as $q = (s, t, \mathcal{L})$, asks whether vertex s can reach vertex t following edges with labels contained in $\mathcal{L}$. The first approach for this problem is presented by Jin et al. [45], where they introduce a tree-based index framework that uses the directed maximal weighted spanning tree algorithm and sampling techniques to compress the generalized transitive closure for labeled graphs.

Zou et al. [103] propose a method that decomposes the graph into strongly connected components (SCCs), computes local transitive closures, and transforms each SCC into a bipartite graph via in-portals and out-portals. The result is an acyclic graph D. Using the topological order of D, they determine the label sets connecting the portals of different SCCs, and finally the internal vertices of each SCC. The generated index contains complete reachability information, allowing easy query resolution via index lookup.

Valstar et al. [83] proposed the LI+ algorithm, which uses landmark-based indexes and non-landmark index pruning to improve performance on large graphs. Building on this, the recent work by Peng et al [70] proposes PH+, an improvement over LI+ with more effective pruning rules and ordering strategies that can handle billion-scale graphs. A study based on LI+ by Chen et al. [23] further explores the problem of index maintenance on dynamic graphs.

2.3.2 Label-constrained shortest path queries.

In addition to label-constrained reachability, label-constrained shortest path query has recently received considerable attention. Rice et al. [71] proposed a method called CHLR, based on the Contraction Hierarchies (CH) approach [37], to handle such queries. The principal idea of CHLR is closely related to CH, but the main difference is that CHLR incorporates edge labels while contracting nodes in the underlying graph. Specifically, when contracting a node v, and there are shortcuts (e.g., (u, w)) added between neighbors of v (e.g., u and w), each shortcut is assigned the labels of the corresponding two edges, i.e., (v, u) and (v, w). Labeling the shortcuts allows CHLR to bypass these shortcuts if they contain a restricted label.

The state-of-the-art algorithm for this problem is the Edge-Disjoint Partitioning (EDP) method [41]. Given an edge-labeled graph, EDP partitions the graph according to the edge labels. The EDP index caches the discovered subpaths within each partition during query processing. As processing more queries causes the index size to exceed a certain threshold, EDP uses the Least Recently Used (LRU) replacement strategy to replace older paths with newly discovered shortest paths. During the query phase, EDP uses a greedy traversal paradigm similar to the Dijkstra algorithm. During this greedy traversal, partitions that do not satisfy the label constraints are skipped, and EDP's cached shortest paths are reused to speed up the query process.

Chapter 3

DISTRIBUTED REACHABILITY LABELING

3.1 Chapter Overview

In this chapter, we study the reachability labeling for distributed graphs. This chapter is structured as follows. Section 3.2 introduces the preliminaries of this chapter, including notations, TOL labeling approach and problem statement. Section 3.3 presents our proposed new labeling methods and the distributed implementation of them. Section 3.4 propose the batch labeling technique to accelerate the indexing process. Section 3.5 evaluates the proposed methods and Section 3.6 concludes this chapter.

3.2 Preliminary

We first introduce some notations in Section 3.2.1, then give the labeling algorithm TOL in Section 3.2.2, followed by the problem statement in Section 3.2.3. For ease of understanding, Table 3.1 lists frequently used notations.

3.2.1 Notations

Given a directed graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, we define the in-neighbor set $N^G_{in}(v)$ of a vertex $v \in V$ as $N^G_{in}(v) = \{u|(u, v) \in E\}$; the out-neighbor set $N^G_{out}(v)$ as $N^G_{out}(v) = \{u|(v, u) \in E\}$. The **in-degree** $d^G_{in}(v)$ (resp. the **out-degree** $d^G_{out}(v)$) of v is the size of its in-neighbor set (resp. out-neighbor set), i.e., $d^G_{in}(v) = |N^G_{in}(v)|$ (resp. $d^G_{out}(v) = |N^G_{out}(v)|$). The path between a vertex pair $s, t \in V$ is defined as $p^G(s, t) = (v_1 = s, v_2, \cdots, v_l = t)$, where $(v_i, v_{i+1}) \in E$, for $\forall i \in [1, l - 1]$. If there exists a path between s and t, then s can reach t (denoted as $s \to t$).

Definition 1. *The **ancestors*** $\mathsf{ANC}(v)$ *(resp. **descendants*** $\mathsf{DES}(v)$*) of a vertex* $v \in V$ *contain all the vertices that can reach* v *(resp. that* v *can reach). Vertex* v *is contained both in* $\mathsf{ANC}(v)$ *and* $\mathsf{DES}(v)$*.*

If the context is obvious, we drop G from notations. The **inverse graph** $\overline{G} = (V, \overline{E})$ of a graph $G(V, E)$ contains the same vertices but with all edges reversed in direction (i.e., $\overline{E} = \{(v, u)|(u, v) \in E(G)\}$).

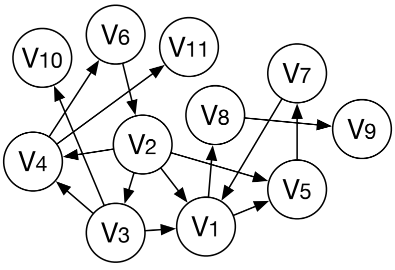


Figure 3.1: Graph G

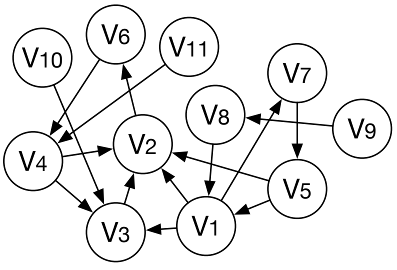


Figure 3.2: Inverse Graph $\overline{G}$

Example 1. *Consider the graph* G *in Fig. 3.1, which has* 11 *vertices and* 15 *edges. For vertex* v_2*,* $N_{in}(v_2) = \{v_6\}$ *and* $d_{in}(v_2) = 1$*;* $N_{out}(v_2) = \{v_1, v_3, v_4, v_5\}$ *and* $d_{out}(v_2) = 4$*. The vertex* v_2 *can reach vertex* v_7 *since there exists a path from* v_2 *to* v_7*. For* v_2*,* $\mathsf{ANC}(v_2) = \{v_2, v_3, v_4, v_6\}$ *and* $\mathsf{DES}(v_2) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$*. The inverse graph* $\overline{G}$ *of* G *is shown in Fig. 3.2.*

Table 3.1: Notations

Notation	Meaning
$G(V, E)$	graph
$\overline{G}(V, \overline{E})$	inverse graph
$d_{in}(v), d_{out}(v)$	in-degree, out-degree of v
$\mathsf{ANC}(v), \mathsf{DES}(v)$	ancestors, descendants of v
$\mathsf{L}_{\mathsf{in}}(v)$, $\mathsf{L}_{\mathsf{out}}(v)$	in-label, out-label sets of v
$\mathsf{L}^{-}_{\mathsf{in}}(v), \mathsf{L}^{-}_{\mathsf{out}}(v)$	backward in-label, out-label sets of v
$\mathsf{DES}_{\mathsf{hig}}(v)$	high-order descendants of v
$\mathsf{BFS}_{\mathsf{low}}(v), \mathsf{BFS}_{\mathsf{hig}}(v)$	low-order, high-order vertices in trimmed BFS
$\mathsf{IBFS}_{\mathsf{low}}(v)$	inverted list of v
$[V_1, V_2, \ldots, V_g]$	batch sequence of G
$\mathsf{L}^{V_i}_{\mathsf{in}}, \mathsf{L}^{V_i}_{\mathsf{out}}$	batch in-label, out-label sets regarding V_i

The in-label/out-label sets over all vertices form index L, which can be used to answer reachability queries in graph G.

Definition 2. *The **in-label set** $\mathsf{L}_{\mathsf{in}}(v)$ of v contains vertices that can reach v, i.e., $\mathsf{L}_{\mathsf{in}}(v) \subseteq \mathsf{ANC}(v)$; the **out-label set** $\mathsf{L}_{\mathsf{out}}(v)$ of v contains vertices that v can reach, i.e., $\mathsf{L}_{\mathsf{out}}(v) \subseteq \mathsf{DES}(v)$.*

Given an index L, the largest label size, denoted as Δ, is defined as $\Delta = \max_{v \in V}(\max(|\mathsf{L}_{\mathsf{in}}(v)|, |\mathsf{L}_{\mathsf{out}}(v)|))$. To answer the reachability query $q(s,t)$ between s, t, we check whether there are overlapping vertices between $\mathsf{L}_{\mathsf{out}}(s)$ and $\mathsf{L}_{\mathsf{in}}(t)$.

$$q(s,t) = \begin{cases} true, & \text{if } \mathsf{L}_{\mathsf{out}}(s) \cap \mathsf{L}_{\mathsf{in}}(t) \neq \varnothing; \\ false, & \text{otherwise.} \end{cases}$$

If the vertices in $\mathsf{L}_{\mathsf{out}}(s)$ and $\mathsf{L}_{\mathsf{in}}(t)$ are sorted by IDs, answering $q(s,t)$ takes $O(|\mathsf{L}_{\mathsf{out}}(s)| + |\mathsf{L}_{\mathsf{in}}(t)|)$ time [101]. To ensure that index L correctly answers all reachability queries on G, L needs to satisfy the cover constraint.

Definition 3 (Cover Constraint). *For $\forall s, t \in V$, $\mathsf{L}_{\mathsf{out}}(s) \cap \mathsf{L}_{\mathsf{in}}(t) \neq \varnothing$ if and only if ($\Leftrightarrow$) $s \to t$.*

Table 3.2: The Index L

Vertex	L_{in}	$\mathsf{L}_{\mathsf{out}}$
v_1	$\{v_1\}$	$\{v_1\}$
v_2	$\{v_2\}$	$\{v_1, v_2\}$
v_3	$\{v_2\}$	$\{v_1, v_2\}$
v_4	$\{v_2\}$	$\{v_1, v_2\}$
v_5	$\{v_1\}$	$\{v_1\}$
v_6	$\{v_2\}$	$\{v_1, v_2\}$
v_7	$\{v_1\}$	$\{v_1\}$
v_8	$\{v_1, v_8\}$	$\{v_8\}$
v_9	$\{v_1, v_8, v_9\}$	$\{v_9\}$
v_{10}	$\{v_2, v_{10}\}$	$\{v_{10}\}$
v_{11}	$\{v_2, v_{11}\}$	$\{v_{11}\}$

Example 2. *Consider the graph G in Fig. 3.1, where Table 3.2 lists an index L for G. For v_2, $\mathsf{L}_{\mathsf{out}}(v_2) = \{v_1, v_2\} \subseteq \mathsf{DES}(v_2)$; for v_3, $\mathsf{L}_{\mathsf{in}}(v_3) = \{v_2\} \subseteq \mathsf{ANC}(v_3)$. Since $\mathsf{L}_{\mathsf{out}}(v_2) \cap \mathsf{L}_{\mathsf{in}}(v_3) = \{v_2\} \neq \emptyset$, the query $q(v_2, v_3)$ returns "true".*

3.2.2 Total Order Labeling

Many labeling methods have been proposed to create reachability indexes for graphs [97], and one of the best-known is Total Order Labeling (TOL). TOL works for a total of n rounds, where one vertex is selected for labeling in each round. TOL gives each vertex v an order $\mathsf{ord}(v)$ and selects the vertex with the i-th largest order in round i. TOL uses degree to determine the order and vertex IDs to break the tie: we can define $\mathsf{ord}(v) = (d_{in}(v) + 1) \cdot (d_{out}(v) + 1) + \frac{ID(v)}{n+1}$ for a vertex v, where $ID(v)$ is the ID of v. There are other ways to define $\mathsf{ord}(v)$, but this way is cheap to calculate and works well in practice [95, 48].

Example 3. *Consider the graph G in Fig. 3.1, which has $n = 11$ vertices. For v_1, $\mathsf{ord}(v_1) = (d_{in}(v_1) + 1) \cdot (d_{out}(v_1) + 1) + \frac{1}{12} = 12.08$; for v_{10}, $\mathsf{ord}(v_{10}) = (d_{in}(v_{10}) + 1) \cdot (d_{out}(v_{10}) + 1) + \frac{10}{12} = 2.83$. Thus, $\mathsf{ord}(v_1) > \mathsf{ord}(v_{10})$, which means that the order of v_1 is higher than v_{10}.*

TOL **Algorithm.** Algorithm 1 shows how TOL creates an index L for graph G. We first initialize the in-label set $\mathsf{L}^1_{\mathsf{in}}(v)$ and out-label set $\mathsf{L}^1_{\mathsf{in}}(v)$ of all vertices v to an empty set (Line 1), and then copy G to G_1 (Line 2). Here, $\mathsf{L}^i_{\mathsf{in}}(v)$ (resp. $\mathsf{L}^i_{\mathsf{out}}$) refers to the label sets created by vertices $[v_1, v_2, \cdots, v_{i-1}]$ of order higher than v_i. The labeling process works in n rounds (Line 3). In round i, the vertex v_i with the i-th largest order starts labeling (Line 4).

Labeling v_i (Line 5-11). First, the descendants $\mathsf{DES}^{G_i}(v_i)$ and ancestors $\mathsf{ANC}^{G_i}(v_i)$ of v_i are obtained using the v_i-sourced BFS in G_i and $\overline{G_i}$, respectively (Line 5-6). Then, v_i is added to in-label/out-label sets of other vertices when it passes a **pruning operation**: for each vertex $w \in \mathsf{DES}^{G_i}(v_i)$, when $\mathsf{L}^i_{\mathsf{out}}(v_i)$ and $\mathsf{L}^i_{\mathsf{in}}(w)$ have no overlapping, v_i is appended to $\mathsf{L}^i_{\mathsf{in}}(w)$ to form $\mathsf{L}^{i+1}_{\mathsf{in}}(w)$ (Line 8); For each vertex $w \in \mathsf{ANC}^{G_i}(v_i)$, when $\mathsf{L}^i_{\mathsf{in}}(v_i)$ and $\mathsf{L}^i_{\mathsf{out}}(w)$ have no overlapping, v_i is appended to $\mathsf{L}^i_{\mathsf{out}}(w)$ to form $\mathsf{L}^{i+1}_{\mathsf{out}}(w)$ (Line 10). Then, v_i and the incident edges are removed from G_i (denoted as $G_i \setminus v_i$) to form G_{i+1} for the next round (Line 11). After n rounds, the label sets of all vertices are returned as index L (Line 12).

Example 4. *Consider the graph G in Fig. 3.1. We show how to create in-label sets for G, and out-label sets can be created similarly. In round 1, G is copied to G_1, and v_1 (with the highest order) is inserted into the in-label sets of its descendants $\mathsf{DES}^{G_1}(v_1) = \{v_1, v_5, v_7, v_8, v_9\}$ in G_1, as no pruning occurs. Then, v_1 and its adjacent edges are removed from G_1 to form G_2. In round 2, v_2 (with the second highest order) finds its descendants $\mathsf{DES}^{G_2}(v_2) = \{v_2, v_3, v_4, v_5, v_6, v_7, v_{10}, v_{11}\}$ in G_2. Then, v_2 performs a pruning operation (Line 8 of Algorithm 1) to test whether v_2 is added to the in-label sets of vertices in $\mathsf{DES}^{G_2}(v_2)$. For example, since $\mathsf{L}^2_{\mathsf{out}}(v_2) \cap \mathsf{L}^2_{\mathsf{in}}(v_5) = \{v_1\}$, v_2 is not inserted into $\mathsf{L}^3_{\mathsf{in}}(v_5)$ — pruning occurs. After pruning, v_2 is inserted into the in-label sets of $\{v_2, v_3, v_4, v_6, v_{10}, v_{11}\}$. After processing v_{11},* TOL *ends.*

```
Algorithm 1: TOL
  Input: Graph $G(V, E)$
  Output: Index $L$
  $\mathsf{L}^1_{\mathsf{in}}(v) \leftarrow \emptyset, \mathsf{L}^1_{\mathsf{out}}(v) \leftarrow \emptyset$, for each vertex $v \in V$;
  $G_1 \leftarrow G$;
  foreach $i \in [1, n]$ do
      $v_i \leftarrow$ the vertex whose order is the $i$-th largest;
      $\mathsf{DES}^{G_i}(v_i) \leftarrow$ a $v_i$-sourced BFS on $G_i$;
      $\mathsf{ANC}^{G_i}(v_i) \leftarrow$ a $v_i$-sourced BFS on $\overline{G_i}$;
      foreach $w \in \mathsf{DES}^{G_i}(v_i)$ do
          // pruning operation
          if $\mathsf{L}^i_{\mathsf{out}}(v_i) \cap \mathsf{L}^i_{\mathsf{in}}(w) = \emptyset$ then
              $\mathsf{L}^{i+1}_{\mathsf{in}}(w) \leftarrow \mathsf{L}^i_{\mathsf{in}}(w) \cup \{v_i\}$;
      foreach $w \in \mathsf{ANC}^{G_i}(v_i)$ do
          // pruning operation
          if $\mathsf{L}^i_{\mathsf{in}}(v_i) \cap \mathsf{L}^i_{\mathsf{out}}(w) = \emptyset$ then
              $\mathsf{L}^{i+1}_{\mathsf{out}}(w) \leftarrow \mathsf{L}^i_{\mathsf{out}}(w) \cup \{v_i\}$;
      $G_{i+1} \leftarrow G_i \setminus \{v_i\}$;
  return $L = \{\mathsf{L}^{n+1}_{\mathsf{in}}(v) \cup \mathsf{L}^{n+1}_{\mathsf{out}}(v) | v \in V\}$;
```

Limitation. TOL is a centralized algorithm [101], which means that the graph needs to be stored on one machine for processing. To make matters worse, TOL is non-trivial to be parallelized. To illustrate why, we focus on the process of labeling v_i. When labeling v_i, each vertex w has an in-label set $\mathsf{L}^i_{\mathsf{in}}(w)$ and an out-label set $\mathsf{L}^i_{\mathsf{out}}(w)$ created by vertices $[v_1, v_2, \cdots, v_{i-1}]$ of order higher than v_i. We collect the label sets of all vertices w and get the index $L^i = \bigcup_{w \in V}\{\mathsf{L}^i_{\mathsf{in}}(w) \cup \mathsf{L}^i_{\mathsf{out}}(w)\}$ generated by vertices of order higher than v_i. The index L^i is necessary when labeling v_i: L^i determines whether or not v_i is added to the label set of another vertex $w \in V$.

Lemma 1. *For $\forall w \in V$, whether or not v_i is in the label set of w depends on L^i, specifically,*

- $v_i \in \mathsf{L}_{\mathsf{in}}(w) \Leftrightarrow v_i \rightarrow w$, $\mathsf{L}^i_{\mathsf{out}}(v_i) \cap \mathsf{L}^i_{\mathsf{in}}(w) = \emptyset$;

- $v_i \in \mathsf{L}_{\mathsf{out}}(w) \Leftrightarrow w \to v_i$, $\mathsf{L}^i_{\mathsf{out}}(w) \cap \mathsf{L}^i_{\mathsf{in}}(v) = \emptyset$.

Proof. We verify only the case $v_i \in \mathsf{L}_{\mathsf{in}}(w)$.

- $\Leftarrow$: If $\mathsf{L}^i_{\mathsf{out}}(v_i) \cap \mathsf{L}^i_{\mathsf{in}}(w) = \emptyset$, then there is no vertex u with $\mathsf{ord}(u) > \mathsf{ord}(v_i)$ such that $v_i \to u \to w$. Then, v_i is the highest-order vertex on all paths from v_i to w. By Theorem 1, $v_i \in \mathsf{L}_{\mathsf{in}}(w)$.

- $\Rightarrow$: If $v_i \in \mathsf{L}_{\mathsf{in}}(w)$ but $\mathsf{L}^i_{\mathsf{out}}(v_i) \cap \mathsf{L}^i_{\mathsf{in}}(w) = S \neq \emptyset$, we choose $s \in S$ whose order is the highest in S. $s \neq v_i$ since $v_i \notin \mathsf{L}^i_{\mathsf{in}}(v_i)$ by definition. Moreover, the fact that $s \in S$ yields $v_i \to s \to u$ and $\mathsf{ord}(s) > \mathsf{ord}(v_i)$. By Theorem 1, $v_i \notin \mathsf{L}_{\mathsf{in}}(w)$, contradiction.

□

Lemma 1 shows that L^i is essential for labeling v_i. However, L^i is generated only after vertices $[v_1, v_2, \cdots, v_{i-1}]$ of order higher than v_i have completed labeling. This suggests that labeling v_i *cannot* begin until those vertices with higher orders have finished labeling. Such a strong *order dependency* prevents TOL from being parallelized. Motivated by this, we aim to design novel labeling methods that can work in parallel while obtaining the same indexes as TOL.

Remark. *[101] maintains* TOL*'s index for dynamic graphs, but we try to generate the same indexes as* TOL *for distributed graphs. We consider maintaining indexes on distributed dynamic graphs as future work.*

3.2.3 Problem Statement

We plan to use a **vertex-centric system** [82] to implement the proposed labeling methods. The vertex-centric system performs the tasks in a super-step fashion [36]. In each super-step, each active vertex v calls a user-defined function, compute(), to: 1) compute based on v's current state and the messages it received in the previous super-step; 2) update v's state; 3) send messages to other vertices (for the next super-step); and 4) (optionally) vote v to make it inactive. The whole computation terminates when there are no messages in the system, or all vertices become inactive. To avoid ambiguity, we use the term "vertex" to denote v as $v \in V$, and the term "node" to denote a computation unit in a cluster.

The problem addressed in this chapter is:

> Given a distributed graph G, design reachability labeling methods and implement them using a vertex-centric system to create the index as TOL.

Throughout this chapter, we do not assume that G is acyclic. This treatment is also used in [34, 98]. We treat it this way for two reasons: 1) our methods are general enough to handle both acyclic and non-acyclic graphs; 2) it is non-trivial to obtain and merge strongly connected components to make graphs acyclic in a distributed environment.

3.3 Distributed Reachability Labeling

We present the concept of backward label sets in Section 3.3.1, and then propose a filtering-and-refinement framework to find backward label sets in Section 3.3.2. New labeling algorithms based on this framework are designed in Section 3.3.3, followed by the algorithm implementation in a distributed system in Section 3.3.4.

3.3.1 TOL Revisited

As shown in Algorithm 1, TOL works in n rounds, where a vertex v is selected for labeling in each round. The process of labeling v is to use the pruning operation

Table 3.3: The Backward Label Sets

Vertex	$\mathsf{L}_{\mathsf{in}}^{-}$	$\mathsf{L}_{\mathsf{out}}^{-}$
v_1	$\{v_1, v_5, v_7, v_8, v_9\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$
v_2	$\{v_2, v_3, v_4, v_6, v_{10}, v_{11}\}$	$\{v_2, v_3, v_4, v_6\}$
v_3	$\emptyset$	$\emptyset$
v_4	$\emptyset$	$\emptyset$
v_5	$\emptyset$	$\emptyset$
v_6	$\emptyset$	$\emptyset$
v_7	$\emptyset$	$\emptyset$
v_8	$\{v_8, v_9\}$	$\{v_8\}$
v_9	$\{v_9\}$	$\{v_9\}$
v_{10}	$\{v_{10}\}$	$\{v_{10}\}$
v_{11}	$\{v_{11}\}$	$\{v_{11}\}$

to determine some vertices (in v's descendants/ancestors) such that v is added to their label sets. We define these determined vertices as the backward label set of v.

Definition 4. *Given $v \in V$ and index L of G, the **backward in-label set** of v is $\mathsf{L}_{\mathsf{in}}^{-}(v) = \{w | v \in \mathsf{L}_{\mathsf{in}}(w)\}$; the **backward out-label set** of v is $\mathsf{L}_{\mathsf{out}}^{-}(v) = \{w | v \in \mathsf{L}_{\mathsf{out}}(w)\}$.*

Example 5. *Consider the graph G in Fig. 3.1, where Table 3.2 shows the index L. In Table 3.3 we list the backward in-label/out-label sets for all vertices. For v_2, $\mathsf{L}_{\mathsf{in}}^{-}(v_2) = \{v_2, v_3, v_4, v_6, v_{10}, v_{11}\}$ since vertices $\{v_2, v_3, v_4, v_6, v_{10}, v_{11}\}$ contain v_2 in their in-label sets; For v_3, $\mathsf{L}_{\mathsf{in}}^{-}(v_3) = \emptyset$ since no vertex contains v_3 in its in-label set. Also, labeling v_2 is to add v_2 to the in-label sets of vertices in $\mathsf{L}_{\mathsf{in}}^{-}(v_2)$ and add v_2 to the out-label sets of vertices in $\mathsf{L}_{\mathsf{out}}^{-}(v_2)$.*

We re-describe the working process of TOL from the perspective of backward label sets: TOL chooses a vertex v to label in each round. The process of labeling v is to determine the vertices in backward in-label/out-label sets where v joins their labels. Recall that TOL applies the pruning operation to determine its backward label sets. According to the analysis in Section 3.2, the pruning operation causes TOL not to work in parallel. Therefore, the main contribution of this chapter is to replace the pruning operation of TOL but still get the backward label sets of each vertex $v \in V$. This allows all vertices to work in parallel and get the same index as TOL.

Remark. *Label sets and backward label sets are symmetric concepts — v is in $\mathsf{L}_{\mathsf{in}}(w)$ (resp. $\mathsf{L}_{\mathsf{out}}(w)$) implies that w is in $\mathsf{L}^{-}_{\mathsf{in}}(v)$ (resp. $\mathsf{L}^{-}_{\mathsf{out}}(v)$). For this reason, we aim to find the backward label sets $\mathsf{L}^{-}_{\mathsf{in}}(v)$ and $\mathsf{L}^{-}_{\mathsf{out}}(v)$ of each vertex v to create the index L. Also, since finding $\mathsf{L}^{-}_{\mathsf{out}}(v)$ on $\overline{G}$ is similar to finding $\mathsf{L}^{-}_{\mathsf{in}}(v)$ on G, we only discuss how to obtain $\mathsf{L}^{-}_{\mathsf{in}}(v)$ in the sequel. The discussions for $\mathsf{L}^{-}_{\mathsf{in}}(v)$ can be naturally extended to $\mathsf{L}^{-}_{\mathsf{out}}(v)$.*

3.3.2 Filtering-and-refinement Framework

To determine the backward label set $\mathsf{L}^{-}_{\mathsf{in}}(v)$ for each vertex v without relying on the pruning operation of TOL, we give the condition for a certain vertex w to lie in $\mathsf{L}^{-}_{\mathsf{in}}(v)$.

Theorem 1. *For $\forall v, w \in V$, $w \in \mathsf{L}^{-}_{\mathsf{in}}(v) \Leftrightarrow w \in \mathsf{DES}(v)$ and v is the highest-order vertex on all paths from v to w.*

Proof. We prove $w \in \mathsf{L}^{-}_{\mathsf{in}}(v)$, or equivalently $v \in \mathsf{L}_{\mathsf{in}}(w)$.

- $\Leftarrow$: If v is the highest-order vertex on all paths from v to w, then there is no vertex u such that $v \to u \to w$ and $\mathsf{ord}(u) > \mathsf{ord}(v)$. Because TOL processes vertices in a non-increasing sequence of vertex order, this means at the moment v starts labeling: 1) $w \in \mathsf{DES}^{G_i}(v)$ since no vertex on a path from v to w can be removed before labeling v; 2) $\mathsf{L}_{\mathsf{out}}(v) \cap \mathsf{L}_{\mathsf{in}}(w) = \emptyset$ since no vertex on a path from v to w finishes labeling. Therefore, by Line 8 of Algorithm 1, $v \in \mathsf{L}_{\mathsf{in}}(w)$.

With similar logic, we can prove that $v \in \mathsf{L}_{\mathsf{out}}(w)$ if v is the highest-order vertex on all paths from w to v.

- $\Rightarrow$: If $v \in \mathsf{L}_{\mathsf{in}}(w)$, let $u \neq v$ be the highest-order vertex on all paths from v to w. This means that u's order is the highest on all sub-paths from v to u and from u to w. We reuse the proof in $\Leftarrow$: 1) u's order is the highest on all u-w paths, then $u \in \mathsf{L}_{\mathsf{in}}(w)$; 2) u's order is the highest on all v-u paths, then $u \in \mathsf{L}_{\mathsf{out}}(v)$. Thus, $u \in \mathsf{L}_{\mathsf{out}}(v) \cap \mathsf{L}_{\mathsf{in}}(w) \neq \emptyset$ when labeling v. By Line 8 of Algorithm 1, $v \notin \mathsf{L}_{\mathsf{in}}(w)$, contradiction. □

Example 6. *Consider the graph G in Fig. 3.1. For vertex v_2, $v_3 \in \mathsf{L}^{-}_{\mathsf{in}}(v_2)$ since v_2 has the highest order on all paths from v_2 to v_3; $v_5 \notin \mathsf{L}^{-}_{\mathsf{in}}(v_2)$ because v_1, with order higher than v_2, lies on a path from v_2 to v_5.*

Theorem 1 paves the way for obtaining $\mathsf{L}^{-}_{\mathsf{in}}(v)$ of each vertex $v \in V$ in parallel. Specifically, by Theorem 1, $w \in \mathsf{DES}(v)$ is a necessary condition for $w \in \mathsf{L}^{-}_{\mathsf{in}}(v)$. In other words, $\mathsf{DES}(v)$ is a super-set of $\mathsf{L}^{-}_{\mathsf{in}}(v)$: $\mathsf{L}^{-}_{\mathsf{in}}(v) \subseteq \mathsf{DES}(v)$. To obtain $\mathsf{L}^{-}_{\mathsf{in}}(v)$, we need to remove invalid elements from $\mathsf{DES}(v)$. Thus, we define the higher-order descendants of v.

Definition 5. *The* ***higher-order descendants*** *of v, denoted as $\mathsf{DES}_{\mathsf{hig}}(v)$, are vertices $u \in \mathsf{DES}(v)$ whose order is higher than v, that is, $\mathsf{DES}_{\mathsf{hig}}(v) = \{u | u \in \mathsf{DES}(v), \mathsf{ord}(u) > \mathsf{ord}(v)\}$.*

Theorem 1 states that only when v is the highest-order vertex on all paths from v to w, then w is in $\mathsf{L}^{-}_{\mathsf{in}}(v)$. In other words, if there is a high-order vertex $u \in \mathsf{DES}_{\mathsf{hig}}(v)$ to reach w, w must not be in $\mathsf{L}^{-}_{\mathsf{in}}(v)$. Hence, we can use $\mathsf{DES}_{\mathsf{hig}}(v)$ to refine the super-set $\mathsf{DES}(v)$ by removing invalid elements.

Based on this idea, we propose the filtering-and-refinement framework to obtain $\mathsf{L}^{-}_{\mathsf{in}}(v)$: the filtering phase generates the super-set $\mathsf{DES}(v)$, and then invalid vertices that can be reached by vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$ are removed for refinement. The correctness of this framework is given in Theorem 2.

Theorem 2. $\mathsf{L}^{-}_{\mathsf{in}}(v) = \mathsf{DES}(v) - \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$.

Proof. We denote the right-hand side of the equation by RHS and verify that $\mathsf{L}^{-}_{\mathsf{in}}(v) = \mathsf{RHS}$.

- $\mathsf{L}^{-}_{\mathsf{in}}(v) \subseteq \mathsf{RHS}$: If $w \in \mathsf{L}^{-}_{\mathsf{in}}(v) \setminus \mathsf{RHS}$, then there are two possibilities: 1) $w \notin \mathsf{DES}(v)$, which contradicts $w \in \mathsf{L}^{-}_{\mathsf{in}}(v) \subseteq \mathsf{DES}(v)$; 2) $w \in \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$, but by Theorem 1, $w \notin \mathsf{L}^{-}_{\mathsf{in}}(v)$, contradiction.

- $\mathsf{RHS} \subseteq \mathsf{L}^{-}_{\mathsf{in}}(v)$: If $w \in \mathsf{RHS}$, then there is no vertex u on any path from v to w for which $\mathsf{ord}(u) > \mathsf{ord}(v)$. By Theorem 1, $w \in \mathsf{L}^{-}_{\mathsf{in}}(v)$. □

Example 7. *Consider the graph G in Fig. 3.1. We show how to obtain $\mathsf{L}^{-}_{\mathsf{in}}(v_3)$ of v_3. We first find $\mathsf{DES}(v_3) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$; next we get $\mathsf{DES}_{\mathsf{hig}}(v_3) = \{v_1, v_2\}$, and $\bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v_3)} \mathsf{DES}(u) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$. Thus, $\mathsf{L}^{-}_{\mathsf{in}}(v_3) = \mathsf{DES}(v_3) - \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v_3)} \mathsf{DES}(u) = \emptyset$.*

3.3.3 Two Labeling Methods

C-1. Basic Labeling Method

If we apply Theorem 2 to obtain $\mathsf{L}^{-}_{\mathsf{in}}(v)$ for each vertex $v \in V$, we need to perform a v-sourced breadth-first search (BFS) to obtain $\mathsf{DES}(v)$ and $\mathsf{DES}_{\mathsf{hig}}(v)$ in the filtering phase, and then perform $|\mathsf{DES}_{\mathsf{hig}}(v)|$ BFSs, one BFS for one vertex in $\mathsf{DES}_{\mathsf{hig}}(v)$ in the refinement phase. The refinement phase requires a large number of BFSs, rendering this solution inefficient. To improve efficiency, we find that not all vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$ are useful for refinement. For example, for vertices $a, b \in \mathsf{DES}_{\mathsf{hig}}(v)$, b is unnecessary when $\mathsf{DES}(b)$ is a subset of $\mathsf{DES}(a)$ — vertex a can reach all descendants of b.

So, how to identify unnecessary vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$? A simple rule is that it is safe to delete vertex b if vertex $a \in \mathsf{DES}_{\mathsf{hig}}(v)$ can reach b: a can reach all descendants of b. Based on this rule, we propose to use a v-sourced BFS but block the expansion branch upon meeting a vertex $a \in \mathsf{DES}_{\mathsf{hig}}(v)$, thus implicitly deleting vertices $b \in \mathsf{DES}_{\mathsf{hig}}(v)$ that can be reached by a. We denote this BFS as a trimmed BFS.

```
Algorithm 2: Trimmed BFS
   Input: Graph $G(V, E)$, $v$
   Output: $\mathsf{BFS}_{\mathsf{low}}(v)$, $\mathsf{BFS}_{\mathsf{hig}}(v)$
 1 queue $Q \leftarrow \emptyset$;
 2 $\mathsf{status}(u) \leftarrow$ ?⃝, for all vertices $u \in V$;
 3 push $v \rightarrow Q$ and $\mathsf{BFS}_{\mathsf{low}}(v)$;
 4 $\mathsf{status}(v) \leftarrow$ →⃝;
 5 while $Q$ is not empty do
 6     $u \leftarrow$ pop from $Q$;
 7     foreach $w \in N_{out}(u)$ do
 8         if $\mathsf{status}(w) \neq$ ?⃝ then continue;
 9         if $\mathsf{ord}(w) < \mathsf{ord}(v)$ then
10             $\mathsf{status}(w) \leftarrow$ →⃝, push $w \rightarrow Q$ and $\mathsf{BFS}_{\mathsf{low}}(v)$;
11         else
               // block the expansion via $w$
12             push $w \rightarrow \mathsf{BFS}_{\mathsf{hig}}(v)$;
13 return $\mathsf{BFS}_{\mathsf{low}}(v), \mathsf{BFS}_{\mathsf{hig}}(v)$;
```

Trimmed BFS. Algorithm 2 describes the v-sourced trimmed BFS. We initialize an empty queue Q and set the status of all vertices to unvisited (denoted as ?⃝) (Line 1-2). Then, v is inserted into Q and $\mathsf{BFS}_{\mathsf{low}}(v)$, and the status of v is set to visited (denoted as →⃝) (Line 3-4). Afterward, we pop a vertex u from Q and check each neighbor w of u (Line 6-7). If w is visited before, we do nothing (Line 8). Otherwise, depending on the order of w, we have two cases: 1) if w is

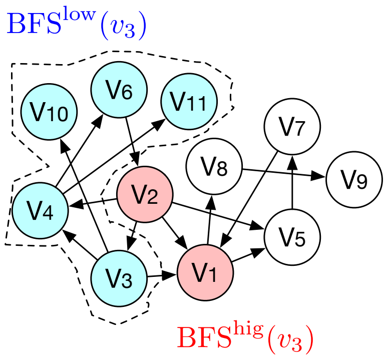

$\text{BFS}^{\text{low}}(v_3)$
V10
V6
V11
V7
V8
V9
V2
V4
V5
V3
V1
$\text{BFS}^{\text{hig}}(v_3)$

Figure 3.3: The v_3-sourced Trimmed BFS

lower in order than v, we continue expanding via w by setting the status of w as visited and inserting w both in Q and $\mathsf{BFS}_{\mathsf{low}}(v)$ (Line 9-10); 2) otherwise, we block the expansion via w and insert w into $\mathsf{BFS}_{\mathsf{hig}}(v)$ (Line 12). When the queue Q is empty, the BFS terminates, and $\mathsf{BFS}_{\mathsf{low}}(v)$ and $\mathsf{BFS}_{\mathsf{hig}}(v)$ are returned.

Lemma 2. *The time cost of Algorithm 2 is $O(|E| + |V|)$.*

Example 8. *Fig. 3.3 shows the v_3-sourced trimmed BFS. First, v_3 is inserted into both Q and $\mathsf{BFS}_{\mathsf{low}}(v_3)$. Then, v_3 is popped from Q, and for v_3's out-neighbors $\{v_1, v_4, v_{10}\}$: v_4 and v_{10} are inserted into both $\mathsf{BFS}_{\mathsf{low}}(v_3)$ and Q because they are of lower order than v_3; the expansion via v_1 is pruned since $\mathsf{ord}(v_1) > \mathsf{ord}(v_3)$, and v_1 is inserted into $\mathsf{BFS}_{\mathsf{hig}}(v_3)$. Then, v_4 is popped from Q, and v_4's out-neighbors $\{v_6, v_{11}\}$ are examined. The BFS terminates when Q is empty, and we get $\mathsf{BFS}_{\mathsf{low}}(v_3) = \{v_3, v_4, v_{10}, v_6, v_{11}\}$, $\mathsf{BFS}_{\mathsf{hig}}(v_3) = \{v_1, v_2\}$.*

Modified Framework. During the trimmed BFS sourced from v, we obtain $\mathsf{BFS}_{\mathsf{low}}(v)$ (vertices visited by BFS and of order lower than v) and $\mathsf{BFS}_{\mathsf{hig}}(v)$ (vertices with higher order that block the expansion). Using $\mathsf{BFS}_{\mathsf{low}}(v)$ and $\mathsf{BFS}_{\mathsf{hig}}(v)$, we optimize the original filtering-and-refinement framework.

Refinement. We first show that in the refinement phase, $\mathsf{BFS}_{\mathsf{hig}}(v)$ can replace $\mathsf{DES}_{\mathsf{hig}}(v)$ since vertices in $\mathsf{BFS}_{\mathsf{hig}}(v)$ reach all the descendants of vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$.

Lemma 3. $\bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u) = \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$.

Proof. Let LHS be $\bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$ and RHS be $\bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$.

- $\mathsf{LHS} \subseteq \mathsf{RHS}$ follows from the fact $\mathsf{BFS}_{\mathsf{hig}}(v) \subseteq \mathsf{DES}_{\mathsf{hig}}(v)$.

- $\mathsf{RHS} \subseteq \mathsf{LHS}$: If $\exists s \in \mathsf{RHS} \setminus \mathsf{LHS}$, $s \in \mathsf{RHS}$ implies there is a path from v to s containing some vertex $w \in \mathsf{DES}_{\mathsf{hig}}(v)$. On all paths from v to s, we collect the inner vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$ and insert them in set S (S is not empty as $w \in \mathsf{DES}_{\mathsf{hig}}(v)$ is such a vertex). We choose the vertex $u \in S$ with the smallest distance to v: there is no other higher-order vertex on the v-u path. By Algorithm 2, $u \in \mathsf{BFS}_{\mathsf{hig}}(v)$. Thus, $s \in \bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u) = \mathsf{LHS}$, contradiction. □

Example 9. *Consider the graph G in Fig. 3.1. For v_3, $\mathsf{BFS}_{\mathsf{hig}}(v_3)$ can replace $\mathsf{DES}_{\mathsf{hig}}(v_3)$ for refinement since $\mathsf{BFS}_{\mathsf{hig}}(v_3) = \{v_1, v_2\}$ reaches all descendants of vertices in $\mathsf{DES}_{\mathsf{hig}}(v_3)$: $\bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v_3)} \mathsf{DES}(u) = \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v_3)} \mathsf{DES}(u) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$.*

Filtering. Next, we show $\mathsf{BFS}_{\mathsf{low}}(v)$ is a super-set of $\mathsf{L}^-_{\mathsf{in}}(v)$, meaning that $\mathsf{BFS}_{\mathsf{low}}(v)$ can replace $\mathsf{DES}(v)$ for filtering.

Lemma 4. $\mathsf{L}^-_{\mathsf{in}}(v) \subseteq \mathsf{BFS}_{\mathsf{low}}(v)$.

Proof. Suppose there is $s \in \mathsf{L}^-_{\mathsf{in}}(v) \setminus \mathsf{BFS}_{\mathsf{low}}(v)$, then there must be a path from v to s through a high-order vertex $u \in \mathsf{BFS}_{\mathsf{hig}}(v)$. By Theorem 1, $s \notin \mathsf{L}^-_{\mathsf{in}}(v)$, contradiction. □

Example 10. *Consider the graph G in Fig. 3.1. For v_3, $\mathsf{BFS}_{\mathsf{low}}(v_3)$ can replace $\mathsf{DES}(v_3)$ for filtering, because $\mathsf{BFS}_{\mathsf{low}}(v_3) = \{v_3, v_4, v_6, v_{10}, v_{11}\}$ includes all vertices in $\mathsf{L}^-_{\mathsf{in}}(v_3) = \emptyset$.*

Basic Labeling Method. Lemma 4 shows that $\mathsf{BFS}_{\mathsf{low}}(v)$ is a super-set of $\mathsf{L}^{-}_{\mathsf{in}}(v)$, while Lemma 3 shows that $\mathsf{BFS}_{\mathsf{hig}}(v)$ is sufficient to eliminate invalid elements not in $\mathsf{L}^{-}_{\mathsf{in}}(v)$. Thus, we give the basic labeling method for labeling $v \in V$.

Step 1. In the filtering phase, we use a v-sourced BFS to find $\mathsf{BFS}_{\mathsf{low}}(v)$ and $\mathsf{BFS}_{\mathsf{hig}}(v)$;

Step 2. In the refinement phase, we perform a BFS for each vertex in $\mathsf{BFS}_{\mathsf{hig}}(v)$.

Step 3. Return $\mathsf{BFS}_{\mathsf{low}}(v) - \bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$ as $\mathsf{L}^{-}_{\mathsf{in}}(v)$.

Combing Lemma 3, Lemma 4, and Theorem 2, the correctness of this method is given in Theorem 3.

Theorem 3. $\mathsf{L}^{-}_{\mathsf{in}}(v) = \mathsf{BFS}_{\mathsf{low}}(v) - \bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$.

C-2. Improved Labeling Method

Compared with the framework given in Theorem 2, the basic method based on Theorem 3 reduces the number of BFSs needed in the refinement phase from $|\mathsf{DES}_{\mathsf{hig}}(v)|$ to $|\mathsf{BFS}_{\mathsf{hig}}(v)|$. But the number $|\mathsf{BFS}_{\mathsf{hig}}(v)|$ may still be very large. Therefore, can we avoid using a large number of BFSs in the refinement phase?

To answer this question, we revisit the refinement phase of the basic method (i.e., Lemma 3): when labeling v, w is eliminated when a vertex in $\mathsf{BFS}_{\mathsf{hig}}(v)$ can reach w. We focus on a specific vertex $u \in \mathsf{BFS}_{\mathsf{hig}}(v)$, which has the highest order on paths from v to w: when performing a u-sourced trimmed BFS in G, u can reach w, so w is in $\mathsf{BFS}_{\mathsf{low}}(u)$[1]; when performing a u-sourced trimmed BFS in the inverse graph $\overline{G}$, u can reach v, so v is in $\mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$. Thus, by examining whether there exist vertices u of order higher than v such that $w \in \mathsf{BFS}_{\mathsf{low}}(u)$

[1] Without ambiguity, $\mathsf{BFS}_{\mathsf{low}}(u)$ and $\mathsf{BFS}^{\mathsf{G}}_{\mathsf{low}}(u)$ refers to the same thing.

and $v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$, we can eliminate w to complete the refinement without using any BFSs: the existence of higher-order vertices u on the v-w paths implies that w can be eliminated.

Example 11. *Consider the graph G in Fig. 3.1. For v_3 in G, v_4 can be eliminated because $\exists v_2 \in \mathsf{BFS}_{\mathsf{hig}}(v_3)$ s.t., 1) in G (Fig. 3.1), v_2-sourced BFS visits v_4 and hence $v_4 \in \mathsf{BFS}_{\mathsf{low}}(v_2)$; 2) in $\overline{G}$ (Fig. 3.2), v_2-sourced BFS visits v_3, and hence $v_3 \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(v_2)$.*

Improved Refinement. Based on the above idea, in the refinement phase of labeling v, to check whether some vertex $w \in \mathsf{BFS}_{\mathsf{low}}(v)$ should be removed, we need to check if there exists $u \in \mathsf{BFS}_{\mathsf{hig}}(v)$ such that $w \in \mathsf{BFS}_{\mathsf{low}}(u)$ and $v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$. Determining $w \in \mathsf{BFS}_{\mathsf{low}}(u)$ can be done intuitively in G, since $\mathsf{BFS}_{\mathsf{low}}(u)$ is known; but determining $v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$ is not so simple, since the information on $\overline{G}$ needs to be used.

To make it feasible to determine $v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$, we create an inverted list $\mathsf{IBFS}_{\mathsf{low}}(v)$ for vertex $v \in V$.

Definition 6. *If v is visited by the u-sourced trimmed BFS in $\overline{G}$, vertex u is in the* ***inverse list*** *$\mathsf{IBFS}_{\mathsf{low}}(v)$ of v, i.e., $\mathsf{IBFS}_{\mathsf{low}}(v) = \{u | v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)\}$.*

With $\mathsf{IBFS}_{\mathsf{low}}(v)$, we can eliminate w by Lemma 5.

Lemma 5. *For a vertex $w \in \mathsf{BFS}_{\mathsf{low}}(v)$, $w \notin \mathsf{L}^{-}_{\mathsf{in}}(v)$ if $\exists u \in \mathsf{IBFS}_{\mathsf{low}}(v)$, and $w \in \mathsf{BFS}_{\mathsf{low}}(u)$.*

Proof. A vertex $u \in \mathsf{IBFS}_{\mathsf{low}}(v)$ with $w \in \mathsf{BFS}_{\mathsf{low}}(u)$ means there is a higher-order vertex u on the path from v to w. Then, $w \notin \mathsf{L}^{-}_{\mathsf{in}}(v)$ by Theorem 1. □

Improved Labeling Method. With the refinement given in Lemma 5, we give an improved labeling method for labeling v.

Step 1. In the filtering phase, we use a v-sourced BFS in G to find $\mathsf{BFS}_{\mathsf{low}}(v)$ and $\mathsf{BFS}_{\mathsf{hig}}(v)$, for $\forall v \in V$;

Step 2. We use a v-sourced BFS in $\overline{G}$ to find $\mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(v)$, and then get $\mathsf{IBFS}_{\mathsf{low}}(v)$ by Definition 6, for $\forall v \in V$;

Step 3. In the refinement phase, if $\exists u \in \mathsf{IBFS}_{\mathsf{low}}(v)$, and $w \in \mathsf{BFS}_{\mathsf{low}}(u)$, the vertex w can be eliminated;

Step 4. Return the non-eliminated vertices as $\mathsf{L}^{-}_{\mathsf{in}}(v)$.

Combing Lemma 5 and Theorem 3, the correctness of this method is given below.

Theorem 4. $\mathsf{L}^{-}_{\mathsf{in}}(v) = \mathsf{BFS}_{\mathsf{low}}(v) - S$, *where* $S = \{w | w \in \mathsf{BFS}_{\mathsf{low}}(v), \exists u \in \mathsf{IBFS}_{\mathsf{low}}(v), w \in \mathsf{BFS}_{\mathsf{low}}(u)\}$.

Note that in Step 2 of the improved labeling method, we need a v-sourced trimmed BFS on $\overline{G}$ to obtain $\mathsf{L}^{-}_{\mathsf{in}}(v)$. This step does not introduce additional costs because $\mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(v)$ is needed to obtain $\mathsf{L}^{-}_{\mathsf{out}}(v)$. In other words, only trimmed BFSs are required to obtain both $\mathsf{L}^{-}_{\mathsf{in}}(v)$ and $\mathsf{L}^{-}_{\mathsf{out}}(v)$.

So far, in addition to finding the backward label sets using the framework given in Theorem 2, we have proposed a basic method based on Theorem 3 and an improved method based on Theorem 4. We give in Table 3.4 the number of BFSs required in the filtering and refinement phases for each method.

Table 3.4: The Comparison Between Labeling Methods

Method	Filtering	Refinement
Theorem 2	1	$\lvert \mathsf{DES}_{\mathsf{hig}}(v) \rvert$
Theorem 3 (Basic)	1	$\lvert \mathsf{BFS}_{\mathsf{hig}}(v) \rvert \leq \lvert \mathsf{DES}_{\mathsf{hig}}(v) \rvert$
Theorem 4 (Improved)	1	1

3.3.4 Distributed Implementation

To handle distributed graphs, we implement the improved labeling method using a vertex-centric system, which is denoted as DRL. We omit the distributed implementation of the basic labeling method, as this can be implemented in a similar way.

Algorithm 3: Compute() for DRL

```
Data: in-msgs ← messages from in-neighbors;
      out-msgs ← messages to out-neighbors;
if super-step = 1 then
    // $w$ is vertex to perform computations
    $w = vertex\_id()$;
    $w$.status($z$) ← $\textcircled{?}$, for each vertex $z \in V$;
    $w$.status($w$) ← $\textcircled{\rightarrow}$;
    // message format:$\{ID, order\}$
    message ← $\{w, \mathsf{ord}(w)\}$;
    send message to out-neighbors;
foreach message ∈ in-msgs do
    // $v$ is the source to do trimmed BFS
    $v$ ← message.ID;
    ord($v$) ← messge.order;
    if $w$.status($v$) = $\textcircled{\rightarrow}$ then continue;
    if ord($v$) > ord($w$) then
        if Check($v, w$)=true then continue;
        $w$.status($v$) ← $\textcircled{\rightarrow}$;
        message ← $\{v, \mathsf{ord}(v)\}$;
        send message to out-neighbors;
        // works on $\overline{G}$
        insert $v$ into $\mathsf{IBFS}_{\mathsf{low}}(w)$;
// only run after the final super-step
foreach $v$, s.t., $w$.status($v$) = $\textcircled{\rightarrow}$ do
    if Check($v, w$)=true then $w$.status($v$) ← $\textcircled{?}$;

Procedure Check($v$, $w$)
    foreach $u \in \mathsf{IBFS}_{\mathsf{low}}(v)$ do
        if $w$.status($u$) = $\textcircled{\rightarrow}$ then return true
    return false
```

Algorithm. Algorithm 3 describes DRL, where the compute() function is executed on each vertex $w \in V$ in super-steps. We record the visited status of w using a status array[2] $w.\mathsf{status}$. Specifically, if the value of $w.\mathsf{status}(v)$ is ?⃝, then w is not visited by the vertex v; if the value is →⃝, then w is visited by v. By reading the values of status arrays, the backward in-label sets of all vertices can be obtained.

In the first super-step (Line 3), vertex w initializes its status array by assigning the unvisited status ?⃝ to all vertices (Line 5), except for w itself, which is assigned as →⃝ (Line 6). Then, w sends the message containing its vertex ID (w) and vertex order ($\mathsf{ord}(w)$) to out-neighbors (Line 7-8). In subsequent super-steps, once vertex w receives the message from in-neighbors (Line 9), w extracts vertex ID v (Line 10) and order $\mathsf{ord}(v)$ (Line 11) of the message. If $w.\mathsf{status}(v)$ is →⃝, we do nothing as v visited w before (Line 12).

If $w.\mathsf{status}(v)$ is ?⃝ and the order v is higher than w, we continue the v-sourced trimmed BFS via w (Line 13). We mark the status of $w.\mathsf{status}(v)$ as →⃝ (Line 15), and we send the message $\{v, \mathsf{ord}(v)\}$ to w's out-neighbors to continue the v-sourced BFS. Also, on the inverse graph $\overline{G}$, if v can reach w, then v is inserted in $\mathsf{IBFS}_{\mathsf{low}}(w)$ (Line 18). Note that we will call the procedure $\mathsf{Check}(v, w)$ (Line 21-24) for an expansion pruning (Line 14): if $\mathsf{IBFS}_{\mathsf{low}}(v)$ contains a vertex u that can reach w, it follows from Lemma 5 that w is not in $\mathsf{L}^{-}_{\mathsf{in}}(v)$, and we prune the expansion of v-sourced BFS via w.

In the final super-step, we check for w the vertices v for which $w.\mathsf{status}(v)$ is →⃝: if the procedure $\mathsf{Check}(v, w)$ returns true, we reset $w.\mathsf{status}(v)$ to ?⃝ (Line 19-20). After this check, the vertices w for which $w.\mathsf{status}(v)$ is →⃝ form the backward in-label set $\mathsf{L}^{-}_{\mathsf{in}}(v)$ of v. Finally, we can collect the backward label sets of each vertex on one machine to obtain an index the same as TOL to support

[2] In the implementation, the hash table can be used to replace the array because of the sparsity of the array.

reachability queries.

Analysis. We give the correctness analysis of DRL, i.e., the vertices w whose $w.\mathsf{status}(v)$ value is →⃝ form $\mathsf{L}^-_{\mathsf{in}}(v)$ of v.

Theorem 5. *Given a graph G and a vertex v, $\mathsf{L}^-_{\mathsf{in}}(v) = \{w|w.\mathsf{status}(v) =$ →⃝$\}$ for Algorithm 3.*

Proof. We denote RHS by $\{w|w.\mathsf{status}(v) =$ →⃝$\}$ and we prove $\mathsf{L}^-_{\mathsf{in}}(v) = \mathsf{RHS}$.

- $\mathsf{RHS} \subseteq \mathsf{L}^-_{\mathsf{in}}(v)$: $w.\mathsf{status}(v) =$ →⃝ means that v can reach w and $\mathsf{ord}(w) < \mathsf{ord}(v)$. Then we verify that there are no higher-order vertices on any path from v to w, thus deriving $w \in \mathsf{L}^-_{\mathsf{in}}(v)$ by Theorem 1. Suppose there are high-order vertices on paths from v to w, we choose the highest-order vertex u. Since u's order is the highest on all v-w paths, u can reach v in $\overline{G}$, thus $u \in \mathsf{IBFS}_{\mathsf{low}}(v)$; u can reach w in G, thus $w.\mathsf{status}(u) =$ →⃝. So the procedure $\mathsf{Check(v,w)}$ will set $w.\mathsf{status}(v) =$?⃝, contradiction.

- $\mathsf{L}^-_{\mathsf{in}}(v) \subseteq \mathsf{RHS}$: Suppose there exists a vertex $w \in \mathsf{L}^-_{\mathsf{in}}(v)$ and $w.\mathsf{status}(v) =$?⃝, then there are two cases: 1) if v cannot reach w, then $w \notin \mathsf{L}^-_{\mathsf{in}}(v)$ by Theorem 1, contradiction; 2) there exists a higher-order vertex u in $\mathsf{IBFS}_{\mathsf{low}}(v)$ to block the expansion branch from v to w to set $w.\mathsf{status}(v)$ to ?⃝ (Line 14) or to reset $w.\mathsf{status}(v)$ to ?⃝ (Line 19-20), which contradicts with Theorem 1. □

We then analyze the computation and communication costs.

Lemma 6. *The computation cost of labeling a vertex $v \in V$ using Algorithm 3 is $O(|E| + |\mathsf{IBFS}_{\mathsf{low}}(v)| \cdot |V|)$, where $|\mathsf{IBFS}_{\mathsf{low}}(v)|$ is the inverted list size of v.*

Proof. The time required to perform a v-sourced trimmed BFS is $O(|E|)$. Also, Algorithm 3 triggers at most $|V|$ times of the $\mathsf{Check}()$ procedure for v, each requiring $|\mathsf{IBFS}_{\mathsf{low}}(v)|$ time. □

Lemma 7. *The communication cost of labeling a vertex $v \in V$ using Algorithm 3 is $O(|E| + |\mathsf{IBFS}_{\mathsf{low}}(v)|)$.*

Proof. Each vertex needs to send/read a message to its neighbors at most once for labeling a certain vertex v. In addition, we need to share $\mathsf{IBFS}_{\mathsf{low}}(v)$ to implement the $\mathsf{Check}()$ procedure. Hence, the communication cost is $\sum_{v \in V} d_{out}(v) + \sum_{v \in V} d_{in}(v) + |\mathsf{IBFS}_{\mathsf{low}}(v)| = |E| + |\mathsf{IBFS}_{\mathsf{low}}(v)|$. □

Remark. Although each vertex v needs to share its inverted list $\mathsf{IBFS}_{\mathsf{low}}(v)$, the size of $\mathsf{IBFS}_{\mathsf{low}}(v)$ is pretty small (empirical studies show that the average size of $\mathsf{IBFS}_{\mathsf{low}}(v)$ of each vertex v is less than one), so the communication overhead associated with sharing the inverted list is not significant. The efficiency of DRL is validated in Section 3.5.

3.4 Batch Labeling Optimization

DRL creates backward label sets for all vertices in parallel. However, DRL misses the opportunity provided by the serial execution of TOL — the already processed high-order vertices strongly prune the search space when labeling the current vertex. As a remedy, we further improve the labeling efficiency by splitting vertices into batches to trade-off between pruning power and parallelization.

Batch Sequence. We split the vertices into a batch sequence for *batch labeling*: we label all the vertices within a batch simultaneously, while vertices in different batches perform the labeling process sequentially.

Definition 7. *$[V_1, V_2, \dots, V_g]$ is a **batch sequence** when*

- *$\bigcup_{i \in [1,g]} V_i = V$ and $V_i \cap V_j = \emptyset$, for $\forall i \neq j$;*
- *for vertex $u \in V_i$ and vertex $v \in V_j$ with $i < j$, it must be ensured that $\mathsf{ord}(u) > \mathsf{ord}(v)$.*

The batch sequence $[V_1, V_2, \ldots, V_g]$ is a graph partition since it disjointly covers all vertices. Also, the vertices with high order are placed before the vertices with low order in the sequence. When the batch size $|V_i|$ $(1 \leq i \leq g)$ is fixed to one, we get $|V|$ batches of vertices for labeling. This fully serial execution is howTOL works. When the batch size is fixed to $|V|$, we get 1 batch of vertices for labeling. This fully parallel execution is how DRL works. By setting the batch size flexibly, we make a trade-off between TOL and DRL.

To obtain a valid batch sequence, we need two parameters: an initial batch size variable b (ranging from 1 to $|V|$) and an increment factor k. The specific procedure is given below.

Step 1. Sort vertices V in a non-increasing order of ord values, and then copy sorted vertices into the set S;

Step 2. In iteration i, remove b vertices with the highest order from S to form V_i (i.e., $S \leftarrow S \setminus V_i$), and then multiply b by k for the next iteration (i.e., $\mathsf{b} \leftarrow \mathsf{b} \cdot \mathsf{k}$);

Step 3. Stop at round $g + 1$ when $S = \emptyset$ and return $[V_1, V_2, \cdots, V_g]$; otherwise, increase i by 1 and go to Step 2.

The number of vertices in the last batch V_g may not exceed b. We set the values of both b and k to 2. The effect of b and k on the labeling efficiency is discussed in Section 5.

Example 12. *Consider the graph G in Fig. 3.1. Suppose* $\mathsf{b} = 2$ *and* $\mathsf{k} = 2$. *In the first round (*$\mathsf{b} = 2$*), we get* $V_1 = \{v_1, v_2\}$*; in the second round (*$\mathsf{b} = 4$*), we get* $V_2 = \{v_3, v_4, v_5, v_6\}$*; in the third round (*$\mathsf{b} = 8$*), we get* $V_3 = \{v_7, v_8, v_9, v_{10}, v_{11}\}$. $[V_1, V_2, V_3]$ *is a batch sequence of G.*

Batch Label Sets. Since we process vertices in batches, the vertices in previous batches $[V_1, V_2, \cdots, V_{i-1}]$ completed labeling before the current batch V_i begins. We define label sets generated by vertices in batches $[V_1, V_2, \cdots, V_{i-1}]$ as batch label sets regarding V_i.

Definition 8. *Given the batch V_i, the **batch in-label set** $\mathsf{L}_{\mathsf{in}}^{V_i}(w)$ of a vertex $w \in V$ is defined as $\mathsf{L}_{\mathsf{in}}^{V_i}(w) = \{u | u \in \mathsf{L}_{\mathsf{in}}(w), \mathsf{ord}(u) > \mathsf{ord}(V_i)\}$; the **batch out-label set** $\mathsf{L}_{\mathsf{out}}^{V_i}(w)$ of a vertex $w \in V$ is defined as $\mathsf{L}_{\mathsf{out}}^{V_i}(w) = \{u | u \in \mathsf{L}_{\mathsf{out}}(w), \mathsf{ord}(u) > \mathsf{ord}(V_i)\}$, where $\mathsf{ord}(V_i) = \max\{\mathsf{ord}(v) | v \in V_i\}$.*

Similar to TOL, we can use the batch label sets to perform the pruning operation during the current batch, thereby optimizing the efficiency of DRL.

Example 13. *Consider the graph G in Fig. 3.1. Suppose v_1 and v_2 finished labeling in the previous batch V_1 and only v_3 is in current batch V_2. Before V_2 starts labeling, the batch in-label set $\mathsf{L}_{\mathsf{in}}^{V_2}(v_9)$ of v_9 is $\{v_1\}$: v_1 in $\mathsf{L}_{\mathsf{in}}(v_9) = \{v_1, v_8, v_9\}$ has a higher order than v_3; the batch out-label set $\mathsf{L}_{\mathsf{out}}^{V_2}(v_9)$ of v_9 is $\emptyset$: $\mathsf{L}_{\mathsf{out}}(v_9) = \{v_9\}$ has no vertex of higher order than v_3.*

Algorithm. We incorporate the idea of batch labeling into DRL and implement it on a vertex-centric system to obtain algorithm $\mathsf{DRL_b}$ (Algorithm 4). $\mathsf{DRL_b}$ resembles DRL (Algorithm 3), and we only list the differences. First, only vertices in the current batch V_i are selected for labeling (Line 6); Also, if there is a higher-order vertex on the path from w to w ($\mathsf{L}_{\mathsf{out}}^{V_i}(w) \cap \mathsf{L}_{\mathsf{in}}^{V_i}(w) \neq \emptyset$), w is pruned (note that the graph is unnecessary to be acyclic) (Line 6). The batch label sets are then sent to all computation nodes (Line 8). The batch label sets are also used for pruning in Line 12. Then, at the end of batch V_i, vertices v with $w.\mathsf{status}(v) =$ →⃝ form batch in-label set of w for the next round (Line 14).

Example 14. *Fig. 3.4 shows how batch labeling works. When labeling v_3 in*

Algorithm 4: Compute() for $\mathsf{DRL_b}$

```
Data: $in\text{-}msgs \leftarrow$ messages from in-neighbors;
      $out\text{-}msgs \leftarrow$ messages to out-neighbors;
Input: $V_i$;
if $super\text{-}step = 1$ then
    // $w$ is vertex to perform computations
    $w = vertex\_id()$;
    if $w \notin V_i$ or $\mathsf{L}_{\mathsf{out}}^{V_i}(w) \cap \mathsf{L}_{\mathsf{in}}^{V_i}(w) \neq \emptyset$ then Return;
    the same as Line 5-8 of Algorithm 3;
    broadcast $\mathsf{L}_{\mathsf{out}}^{V_i}(w)$ and $\mathsf{L}_{\mathsf{in}}^{V_i}(w)$ to all computation nodes;
for $each\ message \in in\text{-}msgs$ do
    // $v$ is source to do trimmed BFS
    $v \leftarrow message.ID$;
    $\mathsf{ord}(v) \leftarrow messge.order$;
    if $\mathsf{L}_{\mathsf{out}}^{V_i}(w) \cap \mathsf{L}_{\mathsf{in}}^{V_i}(w) \neq \emptyset$ then Continue;
    the same as Line 12-20 in Algorithm 3;
// only run after the final super-step
$\mathsf{L}_{\mathsf{in}}^{V_{i+1}}(w) \leftarrow \{v | w.\mathsf{status}(v) =$ ➲$\}$;
```

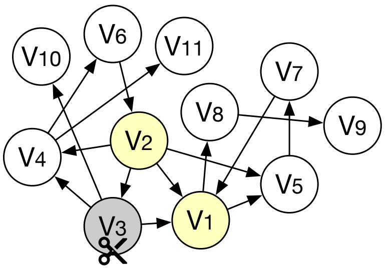


Figure 3.4: The Illustration of Batch Labeling

the current batch V_i, as $\mathsf{L}_{\mathsf{in}}^{V_i}(v_3) = \{v_2\}$ intersects with $\mathsf{L}_{\mathsf{out}}^{V_i}(v_3) = \{v_1, v_2\}$, v_3 is pruned immediately — the search space for labeling v_3 is dramatically reduced.

Analysis. We analyze the correctness of $\mathsf{DRL_b}$.

Theorem 6. *Given a graph G and a vertex v, $\mathsf{L}_{\mathsf{in}}^{-}(v) = \{w | w.\mathsf{status}(v) =$ ➲$\}$ for Algorithm 4.*

Proof. We denote $\{w | w.\mathsf{status}(v) =$ ➲$\}$ by RHS and prove that $\mathsf{L}_{\mathsf{in}}^{-}(v) = \mathsf{RHS}$.

- $\mathsf{RHS} \subseteq \mathsf{L}_{\mathsf{in}}^{-}(v)$: Suppose there is a vertex w with $w.\mathsf{status}(v) =$ ➲ but $w \notin \mathsf{L}_{\mathsf{in}}^{-}(v)$, $w \notin \mathsf{L}_{\mathsf{in}}^{-}(v)$ implies 1) $v \not\rightarrow w$, which shows that $w.\mathsf{status}(v) =$?⃝, or 2) there is a vertex with order higher than v on a path from v to w, so we select the highest-order vertex s. If s is in the previous batches and the current batch is denoted as V_i, since there are no vertices to prune s, $s \in \mathsf{L}_{\mathsf{out}}^{V_i}(v)$ and $s \in \mathsf{L}_{\mathsf{in}}^{V_i}(w)$. Hence, $w.\mathsf{status}(v) = \textcircled{?}$ by Line 12 of Algorithm 4; or s is in the current batch, then $w.\mathsf{status}(v) = \textcircled{?}$ by the correctness of DRL, contradiction.

- $\mathsf{L}_{\mathsf{in}}^{-}(v) \subseteq \mathsf{RHS}$: We prove this by induction on the batch number. When $v \in V_1$, since no pruning occurs, then $\mathsf{DRL_b}$ is correct by the correction of DRL. Suppose V_{i-1} finishes labeling and $\mathsf{DRL_b}$ is correct, we prove $\mathsf{DRL_b}$ is correct for $v \in V_i$. Since $w \in \mathsf{L}_{\mathsf{in}}^{-}(v)$, then v can reach w and no pruning occurs at Line 12 of Algorithm 4. Therefore, by the correction of DRL, $\mathsf{DRL_b}$ is correct for $v \in V_i$. □

We then provide its computation and communication costs.

Lemma 8. *The computation cost of labeling a vertex $v \in V$ using Algorithm 4 is $O(|E'| + (|\mathsf{IBFS_{low}}(v)| + \Delta) \cdot |V|)$, where $E' \subseteq E$, and Δ is the largest label size.*

Proof. For each vertex v, Algorithm 4 needs to explore the reduced search space (denoted as E', $E' \subseteq E$) due to the pruning operation. Moreover, Algorithm 4 requires at most $|V|$ times of $\mathsf{Check}()$ procedure (each costing $O(|\mathsf{IBFS_{low}}(v)|)$) and $|V|$ label queries (each costing $O(\Delta)$). □

Lemma 9. *The communication cost of labeling a vertex $v \in V$ using Algorithm 4 is $O(|E'| + |\mathsf{IBFS_{low}}(v)| + \Delta)$.*

Proof. The cost comes from: 1) sharing label sets with other computation nodes, which incurs $O(\Delta)$ cost; 2) sending $\mathsf{IBFS_{low}}(v)$ for refinement; 3) reduced search space E'. □

Remark. *Compared to* DRL *(Algorithm 3),* $\mathsf{DRL_b}$ *(Algorithm 4) requires additional costs to share and query batch label sets. However, empirical studies in Section 3.5 show that the benefit of reducing the search space from E to $E' \subseteq E$ outweighs the additional overhead.*

3.5 Experiments

3.5.1 Settings

Algorithms. We aim to propose distributed labeling algorithms that produce the same index as TOL. Our methods include:

- DRL (Algorithm 3), a distributed labeling algorithm based on Theorem 4.
- DRL^-, a basic labeling algorithm based on Theorem 3. Since its distributed implementation is similar to DRL, we omit its implementation details.
- $\mathsf{DRL_b}$ (Algorithm 4), a distributed algorithm obtained by applying batch labeling to DRL.

Datasets. The experiments were conducted on 18 real-world directed graphs that are widely used in recent work related to reachability queries [26, 48, 46]. The properties of the graphs are shown in Table 3.5. The largest graph has more than 3.7 billion edges. All the datasets are from Stanford Large Network Dataset Collection[3] [57], Koblenz Network Collection[4] [54], Laboratory for Web Algorithms[5] [17, 16], Network Data Repository[6] [74], and the links in [100]. Note that, to verify the generality of our algorithms for processing distributed graphs,

[3] http://snap.stanford.edu/data/

[4] http://konect.uni-koblenz.de/

[5] http://law.di.unimi.it

[6] http://networkrepository.com/

Table 3.5: Datasets

Name	Dataset	$\|V\|$	$\|E\|$	Type
WEBW	Web-wikipedia	1,864,433	4,507,315	Web
DBPE	Dbpedia	3,365,623	7,989,191	Knowledge
CITE	Citeseerx	6,540,401	15,011,260	Citation
CITP	Cit-patent	3,774,768	16,518,947	Citation
TW	Twitter	18,121,168	18,359,487	Social
GO	Go-uniprot	6967956	34,770,235	Biology
SINA	Soc-sinaweibo	58,655,849	261,321,071	Social
LINK	Wikipedia-link	13,593,032	437,217,424	Web
WEBB	Webbase-2001	118,142,155	1,019,903,190	Web
GRPH	Graph500	17,043,780	1,046,934,896	Synthetic
TWIT	Twitter-2010	41,652,230	1,468,365,182	Social
HOST	Host-linkage	57,383,985	1,643,624,227	Web
GSH	Gsh-2015-host	68,660,142	1,802,747,600	Web
SK	Sk-2005	50,636,154	1,949,412,601	Web
TWIM	Twitter-mpi	52,579,682	1,963,263,821	Social
FRIE	Friendster	68,349,466	2,586,147,869	Social
UK	Uk-2006-05	77,741,046	2,965,197,340	Web
WEBS	Webspam-uk	105,896,555	3,738,733,648	Web

we do not transform the graphs into acyclic graphs, but build the indexes directly on the original graphs.

Environment. We implement all algorithms in C++ and compile them using GNU GCC 4.8.5. We use MPICH to implement the distributed algorithms. Our algorithms are executed on a cluster of 32 computation nodes — each node contains an Intel Xeon 2.7 GHz CPU, 32 GB main memory, and runs Linux (Red Hat Linux 4.8.5, 64 bits). In contrast, centralized algorithms such as TOL [101] and BFL [79] are executed on only one node with the same settings. If not explicitly stated, we only run one thread on each computation node. We set the cut-off time to 2 hours. If the algorithm runs out of memory or cannot complete the computation within the cut-off time, the execution time is marked as "INF".

Graph Partition Strategy. For the distributed algorithms, we adopt a vertex-centric paradigm and partition the vertices in the graph based on their hash

Table 3.6: The Comparison with Competitor Methods

	Index Time (sec)					Index Size (MB)					Query Time (sec)				
Name	$\mathsf{BFL}^{\mathsf{C}}$	$\mathsf{BFL}^{\mathsf{D}}$	TOL	$\mathsf{DRL}_{\mathsf{b}}$	$\mathsf{DRL}_{\mathsf{b}}^{\mathsf{M}}$	$\mathsf{BFL}^{\mathsf{C}}$	$\mathsf{BFL}^{\mathsf{D}}$	TOL	$\mathsf{DRL}_{\mathsf{b}}$	$\mathsf{DRL}_{\mathsf{b}}^{\mathsf{M}}$	$\mathsf{BFL}^{\mathsf{C}}$	$\mathsf{BFL}^{\mathsf{D}}$	TOL	$\mathsf{DRL}_{\mathsf{b}}$	$\mathsf{DRL}_{\mathsf{b}}^{\mathsf{M}}$
WEBW	1.51	59.21	61.84	9.08	7.31	85.35	85.35	432.06	432.06	432.06	8.58E-07	5.39E-05	2.09E-07	2.09E-07	2.09E-07
DBPE	2.10	110.17	2.21	0.92	0.64	154.07	154.07	63.91	63.91	63.91	2.25E-07	4.63E-05	1.51E-07	1.51E-07	1.51E-07
CITE	3.39	195.62	4.95	2.34	1.42	299.40	299.40	138.80	138.80	138.80	1.24E-07	4.52E-05	1.78E-07	1.78E-07	1.78E-07
CITP	5.10	138.30	125.21	13.36	11.17	172.80	172.80	622.04	622.04	622.04	5.68E-07	5.07E-05	3.12E-07	3.12E-07	3.12E-07
TW	3.74	469.34	7.27	1.13	1.40	829.52	829.52	271.60	271.60	271.60	1.95E-07	6.72E-05	1.84E-07	1.84E-07	1.84E-07
GO	3.56	365.38	7.40	1.76	2.03	318.97	318.97	274.43	274.43	274.43	1.11E-07	4.41E-05	2.02E-07	2.02E-07	2.02E-07
SINA	41.35	2,822.48	–	136.32	–	2,685.05	2,685.05	–	13,691.20	–	2.82E-06	8.64E-05	–	6.76E-07	–
LINK	16.29	213.43	55.16	15.64	9.38	622.24	622.24	239.76	239.76	239.76	2.29E-07	7.31E-05	1.35E-07	1.35E-07	1.35E-07
WEBB	–	1,181.08	–	103.98	–	–	5,408.12	–	2,578.85	–	–	2.37E-04	–	1.84E-07	–
GRPH	46.30	6.36	76.31	24.00	16.44	780.20	780.20	325.01	325.01	325.01	9.61E-08	7.03E-05	8.71E-08	8.71E-08	8.71E-08
TWIT	57.44	304.55	134.87	62.82	35.79	1,906.69	1,906.69	766.17	766.17	766.17	1.55E-07	1.43E-04	1.06E-07	1.06E-07	1.06E-07
HOST	–	1,655.19	–	66.87	–	–	2,626.83	–	926.77	–	–	5.30E-04	–	2.26E-07	–
GSH	–	512.04	–	77.93	–	–	3,143.01	–	1,266.78	–	–	2.85E-04	–	1.37E-07	–
SK	–	219.47	–	82.85	–	–	2,317.94	–	975.83	–	–	1.80E-04	–	1.01E-07	–
TWIM	–	359.05	–	68.36	–	–	2,406.91	–	958.17	–	–	2.60E-04	–	2.76E-07	–
FRIE	–	688.47	–	112.77	–	–	3,128.79	–	1,240.01	–	–	3.58E-04	–	4.38E-07	–
UK	–	296.52	–	217.94	–	–	3,558.70	–	1,567.59	–	–	2.25E-04	–	2.81E-07	–
WEBS	–	747.92	–	188.58	–	–	4,847.56	–	2,063.97	–	–	4.81E-04	–	4.15E-07	–

values. This ensures that each partition has roughly the same number of vertices. However, for the scale-free networks, there may be vertices with large degrees, which can lead to imbalanced workloads and high communication costs. To address these issues, we adopt the vertex mirroring technique introduced in [94]. This technique creates mirrors for the high-degree vertices in each partition. The edges adjacent to these vertices are partitioned based on the location of the other endpoints and are stored with the mirrors. The messages from the original vertex are first sent to its mirrors and then relayed to its neighbors. For vertices with small degrees, we do not create mirrors and store their adjacent edges with them.

3.5.2 Comparison with Competitor Methods

Exp 1: Comparison with TOL. TOL is an index-only algorithm [101]. To illustrate the necessity of the proposed methods, we compare our best method $\mathsf{DRL_b}$ with TOL. The results of the comparison are given in Table 3.6. When a method cannot complete index creation because it exceeds the memory limit, we mark its results with the notation "-" in the table.

On index time. The time of $\mathsf{DRL_b}$ includes both computation time and communication time. On a medium-sized graph that can be accommodated by a single computation node, $\mathsf{DRL_b}$'s index time can be at most 9.37 times faster than TOL. Note that TOL is a centralized algorithm and cannot handle distributed graphs. When a single computation node cannot accommodate a graph (e.g., WEBS), TOL fails to work. In contrast, $\mathsf{DRL_b}$ can index all graphs within half an hour. This shows that our method can efficiently handle large-scale graphs that are beyond the ability of TOL.

On index size. For $\mathsf{DRL_b}$, we aggregate the index distributed on different computation nodes on one node. Hence, our algorithms have the same index size as TOL. The index size of $\mathsf{DRL_b}$ (and TOL) on all graphs is very small. For example, the largest graph WEBS has an index size of 2.06 GB. This indicates that although distributed graphs may be large, the generated index can be accommodated on an ordinary machine to support in-memory queries. So, it is feasible to create the index of TOL for a distributed graph because the index size is not so large.

On query time. $\mathsf{DRL_b}$ creates the same index as TOL, so the query time is the same for TOL and $\mathsf{DRL_b}$. The query time of $\mathsf{DRL_b}$ (and TOL) on all graphs is less than one microsecond. This result reinforces our research motivation that efficient reachability queries on a distributed graph can be supported by proposing new labeling methods to obtain the same index as TOL.

Exp 2: Comparison with BFL. We compare our best method $\mathsf{DRL_b}$ with BFL, which is an index-assisted method [79]. BFL uses DFS to create the index and may also use the online search (e.g. DFS) at query time. We use the code provided in [79] to implement the *centralized* BFL algorithm, which is denoted as $\mathsf{BFL^C}$, where all parameters are set by default. Also, we implemented a distributed version of DFS (which is a core operation of BFL) to obtain the *distributed* BFL algorithm, which is denoted as $\mathsf{BFL^D}$. $\mathsf{BFL^C}$ runs on one computation node, while $\mathsf{BFL^D}$ runs on 32 nodes. We compare $\mathsf{DRL_b}$, $\mathsf{BFL^C}$, and $\mathsf{BFL^D}$ and record the results in Table 3.6.

On index time. (1) First, we compare $\mathsf{DRL_b}$ with the centralized algorithm $\mathsf{BFL^C}$. On medium-sized graphs, the index time of $\mathsf{BFL^C}$ is normally better than that of $\mathsf{DRL_b}$. But the index time of $\mathsf{DRL_b}$ is comparable to that of $\mathsf{BFL^C}$: the index time of $\mathsf{DRL_b}$ is within seven times that of $\mathsf{BFL^C}$. Moreover, $\mathsf{DRL_b}$ can handle large-scale graphs for which $\mathsf{BFL^C}$ cannot create indexes. (2) Then, we compare $\mathsf{DRL_b}$ with the distributed algorithm $\mathsf{BFL^D}$. $\mathsf{BFL^D}$ can process large-scale graphs by partitioning them to different computation nodes. But because $\mathsf{BFL^D}$ requires the distributed DFS to create indexes, its index time is very high: $\mathsf{BFL}^{\mathsf{D}}$ runs on average 52.54 times slower than $\mathsf{DRL}_{\mathsf{b}}$.

On index size. Since the index sizes of $\mathsf{BFL}^{\mathsf{C}}$ and $\mathsf{BFL}^{\mathsf{D}}$ are the same on all graphs, we only compare the index sizes of $\mathsf{BFL}^{\mathsf{D}}$ and $\mathsf{DRL}_{\mathsf{b}}$. It can be seen that the index size of $\mathsf{BFL}^{\mathsf{D}}$ is small on all graphs (no larger than 6 GB), and the index of $\mathsf{BFL}^{\mathsf{D}}$ is smaller than that of $\mathsf{DRL}_{\mathsf{b}}$ on WEBW, CITP, and SINA. However, the index size of $\mathsf{DRL}_{\mathsf{b}}$ is smaller than that of $\mathsf{BFL}^{\mathsf{D}}$ on the other graphs: the index size of $\mathsf{DRL}_{\mathsf{b}}$ is on average 2.38 times smaller than that of $\mathsf{BFL}^{\mathsf{D}}$ on these graphs.

On query time. (1) We first compare the query time of $\mathsf{BFL}^{\mathsf{C}}$ and $\mathsf{DRL}_{\mathsf{b}}$. Both $\mathsf{BFL}^{\mathsf{C}}$ and $\mathsf{DRL}_{\mathsf{b}}$ can answer queries in microseconds, but the performance of $\mathsf{DRL}_{\mathsf{b}}$ is better than $\mathsf{BFL}^{\mathsf{C}}$: on average, the query time of $\mathsf{DRL}_{\mathsf{b}}$ is 1.8 times faster than that of $\mathsf{BFL}^{\mathsf{C}}$ on graphs that $\mathsf{BFL}^{\mathsf{C}}$ can process. (2) Then we compare the query time of $\mathsf{BFL}^{\mathsf{D}}$ and $\mathsf{DRL}_{\mathsf{b}}$. Because $\mathsf{BFL}^{\mathsf{D}}$ cannot avoid traversing the distributed graph to answer queries, the performance of $\mathsf{BFL}^{\mathsf{D}}$ can be very bad: the query time of $\mathsf{BFL}^{\mathsf{D}}$ is on average 867.6 times longer than that of $\mathsf{DRL}_{\mathsf{b}}$.

Overall, BFL performs well only when index creation can be done on a single node (see performance of $\mathsf{BFL}^{\mathsf{C}}$); when dealing with large graphs, BFL has high index time and query time due to the high cost of distributed DFS and graph search (see performance of $\mathsf{BFL}^{\mathsf{D}}$). This clarifies why we parallelize the index-only method TOL instead of BFL. Note that the graphs we use are not converted to be acyclic because of the high cost of performing such conversions in a distributed environment using DFS. This partly explains why there are some minor inconsistencies between our conclusions and [79].

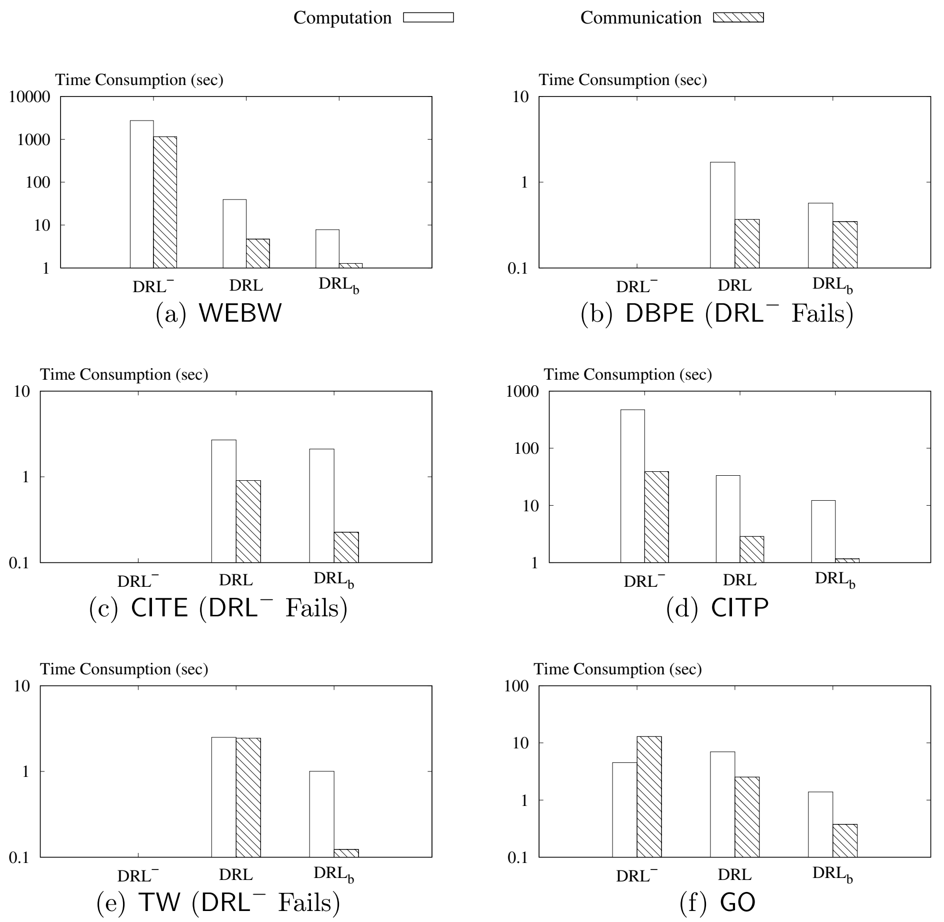
Exp 3: Comparison with Multi-core Version. Our algorithm $\mathsf{DRL}_{\mathsf{b}}$ (see Algorithm 4) achieves parallelism among multiple computation nodes in a distributed environment. Besides, we can also achieve parallelism among multiple threads (instead of multiple nodes), resulting in a multi-core version of $\mathsf{DRL}_{\mathsf{b}}$, which is denoted as $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$. For a fair comparison, we test $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ in a machine configured similarly to the single computation node used by DRL_b, and this machine contains 32 cores and has a memory size of 32 GB. OpenMP [32] is used to implement $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$. Since DRL_b and $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ have the same index size and query time, we compare only the index time between them. We record the results in Table 3.6 and have the following findings.

On medium-sized graphs. Because $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ can use shared memory for data exchange [20], it avoids the communication cost of DRL_b. This leads to a better index time for $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ than for DRL_b in most cases: $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ is 1.34 times faster than DRL_b on graphs where $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ can create indexes. However, this speedup is limited. One possible reason is that the communication cost of DRL_b is relatively small compared to the computation cost (see the communication time and the computation time of DRL_b in Exp 4 for details), leading to a less prominent advantage of shared memory.

On large-scale graphs. Since $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ is a centralized algorithm, the usability of $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ is limited by memory and therefore cannot build indexes for massive graphs. For example, $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ ran out of memory when building the index for WEBS. On the other hand, DRL_b can allocate graphs to multiple computation nodes and thus is more suitable for large graph processing.

3.5.3 Comparison Between Proposed Algorithms

Exp 4: Communication and Computation Time. We compare our proposed labeling algorithms, DRL^-, DRL, and DRL_b. We divide the index time of the proposed algorithms into computation time and communication time. If an algorithm is unable to finish labeling within the cut-off time, we do not report its time and mark the failure at the title of that graph. Due to space constraints, we present in Fig. 3.5 the results on the first 6 graphs (WEBW, DBPE, CITE,



(a) WEBW

(b) DBPE (DRL⁻ Fails)

(c) CITE (DRL⁻ Fails)

(d) CITP

(e) TW (DRL⁻ Fails)

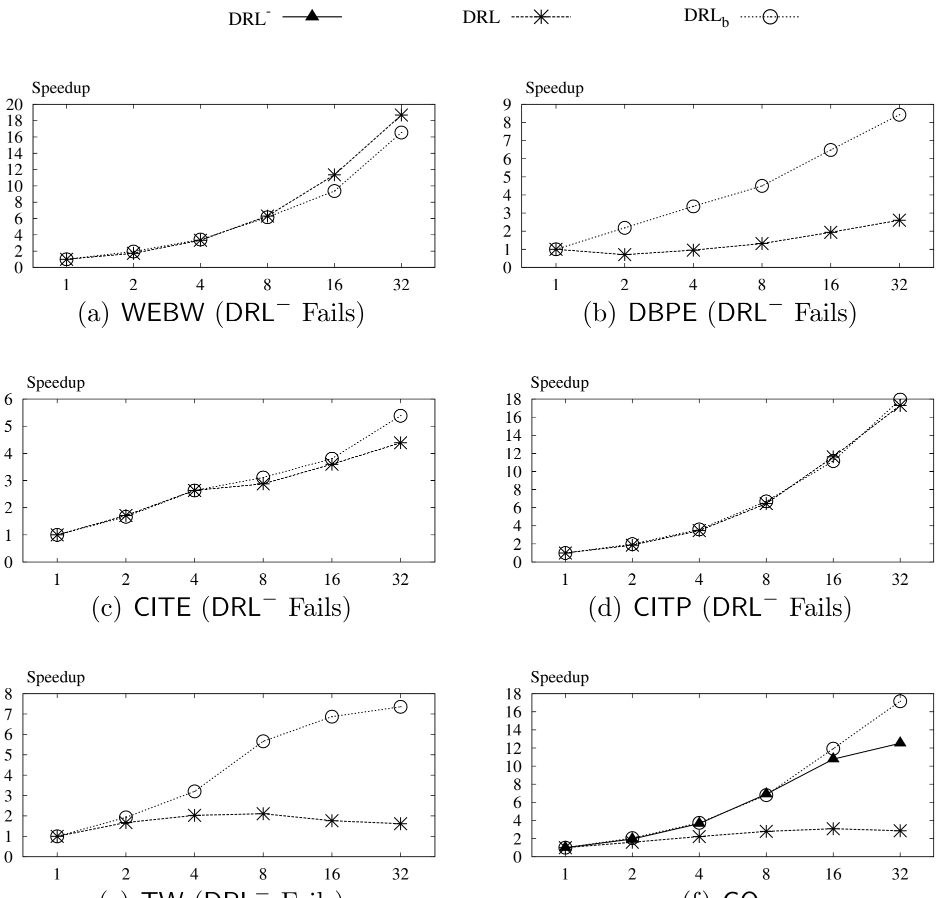
(f) GO

Figure 3.5: The Comparison of Communication and Computation Time

CITP, TW, and GO) with the following findings.

Comparison of DRL⁻ *and* DRL. Compared to DRL⁻, DRL uses the inverted list to implement the refinement phase. We find that DRL can index on DBPE, CITE and TW, while DRL⁻ cannot. In addition, on the other three graphs, DRL has an average of 88.2 times shorter index time than DRL⁻, thanks to the new refinement technique used by DRL.

Comparison of DRL *and* DRL$_b$. DRL$_b$ uses batch labeling to further optimize DRL. We find that DRL$_b$ has an average of 3.5 times shorter index time over DRL. Moreover, DRL$_b$ reduces the computation time while substantially reducing the communication cost of DRL, which validates the effectiveness of the optimization strategy used by $\mathsf{DRL_b}$.



(a) WEBW (DRL⁻ Fails)

(b) DBPE (DRL⁻ Fails)

(c) CITE (DRL⁻ Fails)

(d) CITP (DRL⁻ Fails)

(e) TW (DRL⁻ Fails)

(f) GO

Figure 3.6: The Effect of # of Nodes on the Index Time

Exp 5: Effect of Node Number. We used 32 computation nodes by default. To test the impact of the number of computation nodes on the proposed algorithms, we varied the number of nodes from 1, 2, 4, 8, 16 to 32 and recorded the corresponding index time. We define the **speedup** as the ratio of the index time on one node to the index time on x nodes, i.e., $speedup = \frac{\text{the index time on 1 node}}{\text{the index time on x node}}$. If an algorithm fails to finish labeling within the cut-off time on 1 node, we do not report its speedup ratio and mark the failure at the title of that graph. We show the speedup ratios on the first six graphs in Fig. 3.6 with the following findings. $\mathsf{DRL_b}$ *has a satisfying speedup ratio.* The maximum speedup ratio for $\mathsf{DRL_b}$ with 32 nodes is 17.93 (on CITP) compared to using a single node. Moreover, the speedup ratio of $\mathsf{DRL_b}$ shows an increasing trend as the number of nodes increases.

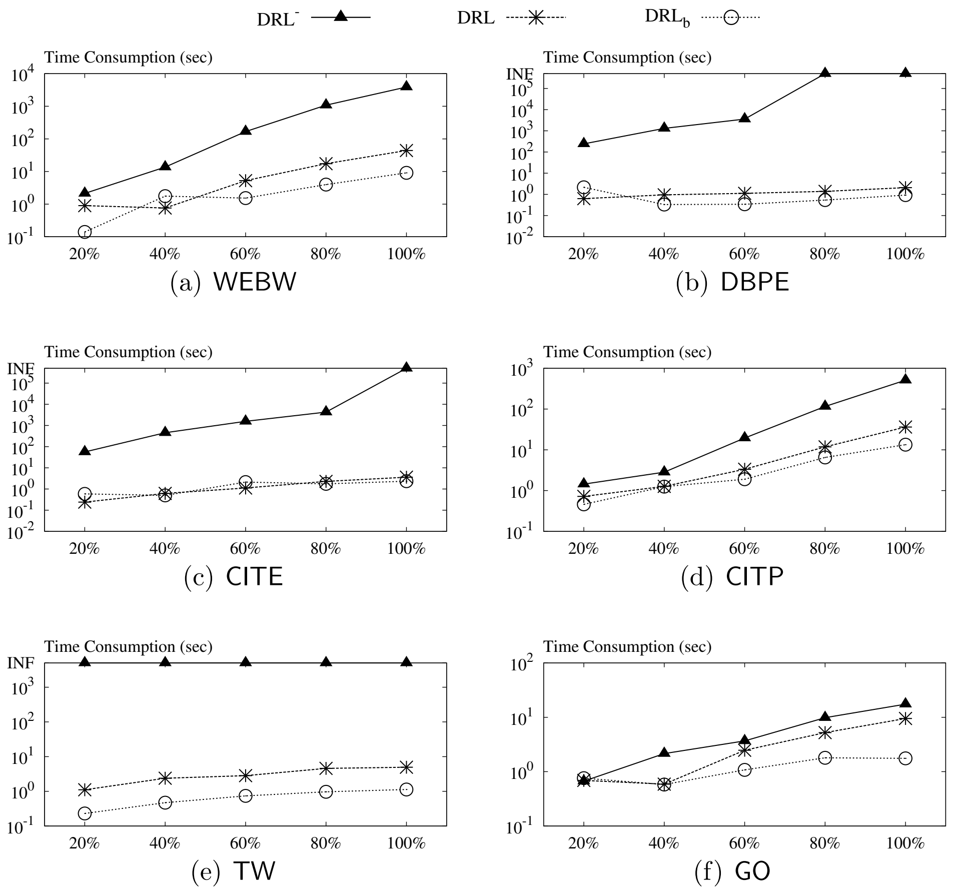
The speedup of DRL^- *and* DRL *has limitations.* Although the maximum speedup ratio of DRL^- is 12.54 (on GO), DRL^- cannot finish labeling on other five graphs using a single node within the cut-off time. On the other hand, although the maximum speedup ratio of DRL is 18.69 (on WEBW), on TW, the ratio of DRL is only 2.86 while that of $\mathsf{DRL_b}$ is 17.2, which shows that introducing the batch labeling optimization maintains a better speedup ratio.

Exp 6: Test of Scalability. In testing the scalability of the proposed algorithms, we divide the edges of the graph into five disjoint groups, each group consisting of $\frac{1}{5}$ edges of the original graph. We generate five test graphs, where the i-th test graph contains edges in the first i groups. The experiments are conducted on five test graphs for each dataset. Since the index size and query time are the same for all algorithms, we omit the discussion of them and only provide the effect on the index time in Fig. 3.7. We have the following finding.

The proposed algorithms exhibit good scalability. The index time of all algorithms improves as the graph size becomes larger. However, the increase of all methods is smooth. For example, the index time of $\mathsf{DRL_b}$ for the test graph with 100% edges is 4.8 times longer than that of the test graph with 20% edges on TW. This indicates the good scalability of the proposed algorithms.

3.5.4 Effect of Parameters on Index Time

In Exp-4, we verified that using the batch labeling optimization (forming $\mathsf{DRL_b}$) can speed up the index time of DRL. For $\mathsf{DRL_b}$, two parameters need to be set to



(a) WEBW (b) DBPE

(c) CITE (d) CITP

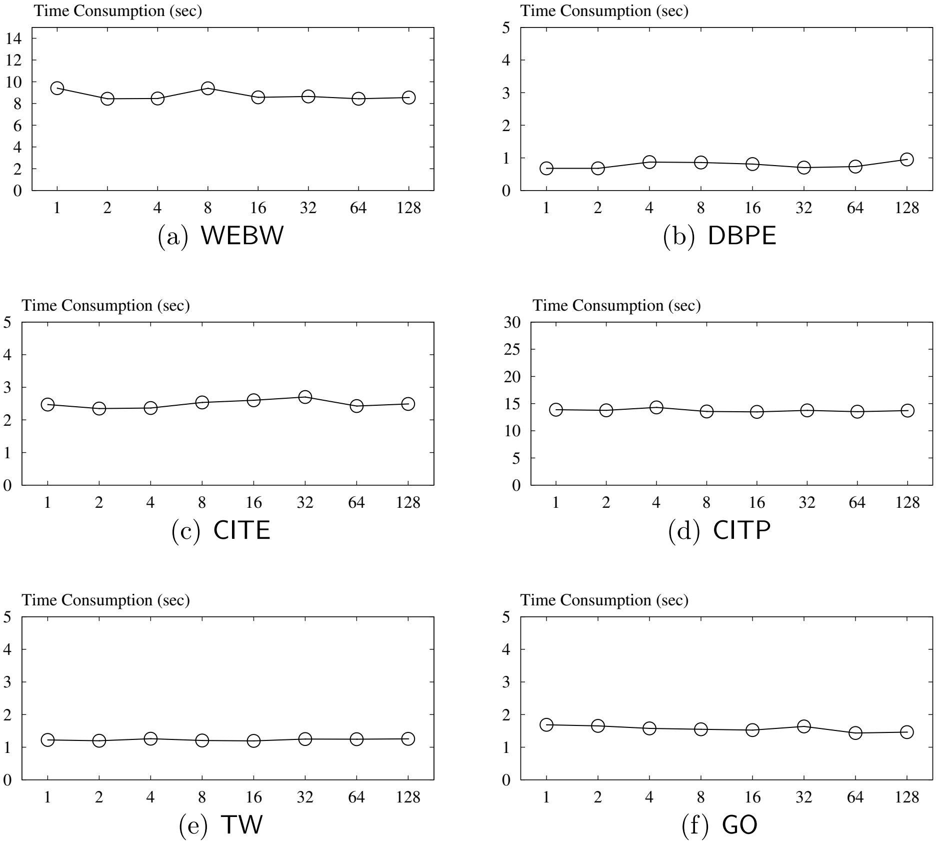
(e) TW (f) GO

Figure 3.7: The Test of the Scalability on the Index Time

generate the batch sequence: the initial batch size b and the incremental factor k. We set both parameters to 2 by default, but we need to further test the effect of these two parameters on the index performance (time) of $\mathsf{DRL_b}$.

Exp 7: Effect of Initial Batch Size b. We first analyze the effect of b. We vary the value of b from 1, 2, 4, 8, 16, 32, 64, to 128. We record the index time of $\mathsf{DRL_b}$ for different values of b and present the results of $\mathsf{DRL_b}$ on the first 6 graphs in Fig 3.8. We have the following findings.

b *has little effect on the index time.* As the value of b varies, the difference between the maximum index time and the minimum index time on all used graphs



(a) WEBW

(b) DBPE

(c) CITE

(d) CITP

(e) TW

(f) GO

Figure 3.8: The Effect of Initial Batch Size b on the Index Time

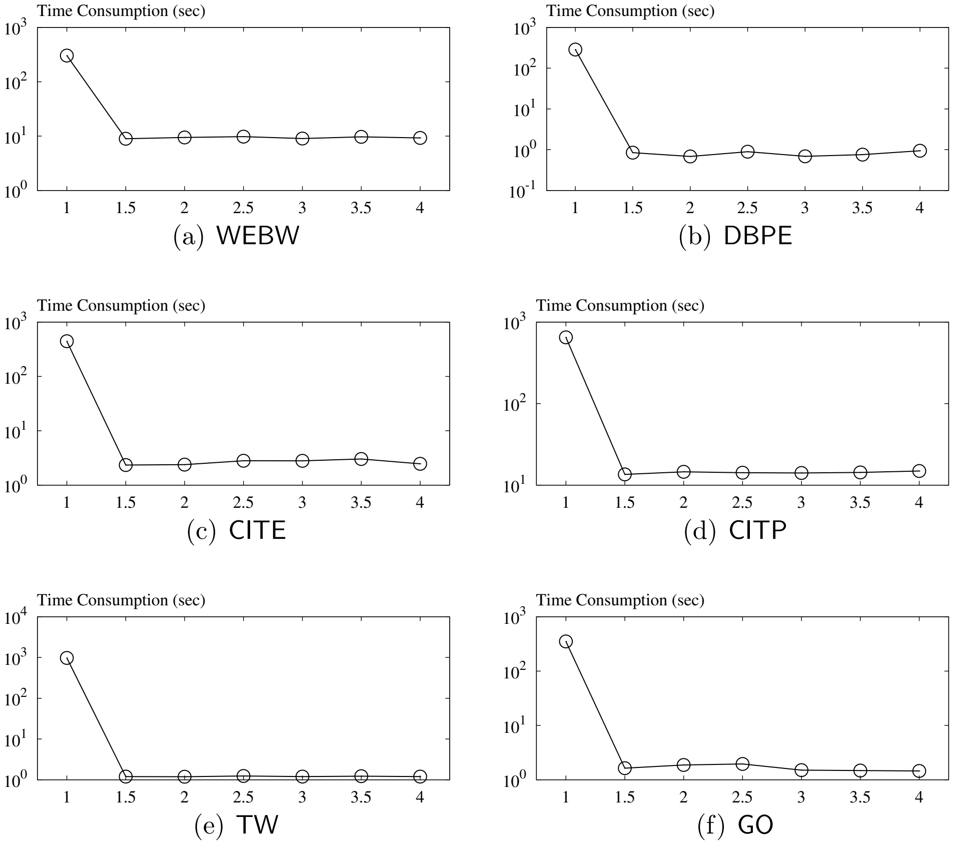
is no more than 1.5 times. This indicates that $\mathsf{DRL_b}$ is not sensitive to the parameter b.

The default value of 2 *is a good choice.* On some graphs (e.g., WEBW and DBPE), setting b to 2 leads to a local minimum index time. This explains why 2 is used as the default value.

Exp 8: Effect of Factor k. We analyze the effect of another parameter k on the index time. We vary the value of k from 1, 1.5, 2, 2.5, 3, 3.5, to 4. We report the index time of $\mathsf{DRL_b}$ for different k in Fig. 3.9 and obtain the following findings.

When k *is not* 1. When k is taken other than 1, the index time does not vary much: the difference between the maximum and minimum index time on all graphs does not exceed 1.4. Also, on some graphs (e.g., DBPE and CITE), the index time reaches a local minimum when k is 2, so 2 is used as the default value.

When k *is* 1. The index time becomes very slow when k is taken as 1: the index time is up to 812 times slower when k is 1 than when k is taken as other. This further explains why k needs to be set to 2 as the default value.



(a) WEBW

(b) DBPE

(c) CITE

(d) CITP

(e) TW

(f) GO

Figure 3.9: The Effect of Factor k on the Index Time

3.6 Chapter Summary

We develop novel labeling methods to produce the same indexes as TOL on distributed graphs. To overcome the limitation that TOL cannot be executed in parallel, we resort to finding the backward label set of each vertex. We propose to use a filtering-and-refinement framework to find backward label sets. Using this framework, we design new labeling algorithms and further improve the efficiency by batch labeling optimization. Experimental results show that our algorithms can efficiently handle distributed graphs.

For future work, there could be three directions: 1) maintaining the indexes for distributed dynamic graphs, 2) further exploring the features of distributed and multi-core systems to accelerate the construction of indexes and 3) designing distributed algorithms for label constrained reachability. Label constrained reachability presents additional challenges. But we can adopt a similar approach to parallelizing the indexing procedure, to reduce the number of vertices being explored during the Bread-First Searches, we can exploit the dominance relationship between paths and prune the paths dominated by others.

Label-constrained reachability presents additional challenges. However, we can adopt a similar approach used for the parallelization of the indexing procedure, i.e.,, issue parallel Bread-First Searches (BFSs) to index multiple vertices in a batch. By exploiting the dominance relationship between paths, we can first explore the paths that are less likely to be dominated by others and use them to prune other paths. This could effectively reduce the number of vertices explored, making the process more efficient.

Chapter 4

SHORTEST-PATH QUERIES ON COMPLEX GRAPHS

4.1 Chapter Overview

In this chapter, we study the shortest-path query on complex graphs. This chapter is structured as follows. Section 4.2 introduces the preliminaries of this chapter. Section 4.3 introduces the index-based approaches PLL and CTL for shortest-distance queries and extends presents them to answer shortest path queries. Section 4.4 presents our proposed new shortest path extension MLL based on CTL. Section 4.5 evaluates the proposed methods and Section 4.6 concludes this chapter.

4.2 Preliminary

Let $G(V, E)$ be an undirected unweighted graph with $n = |V(G)|$ vertices and $m = |E(G)|$ edges. The neighbors $N(v, G)$ of each vertex $v \in V(G)$ are defined as $N(v, G) = \{u|(u, v) \in E(G)\}$. The size of $N(v, G)$ is the degree $deg(v, G)$ of

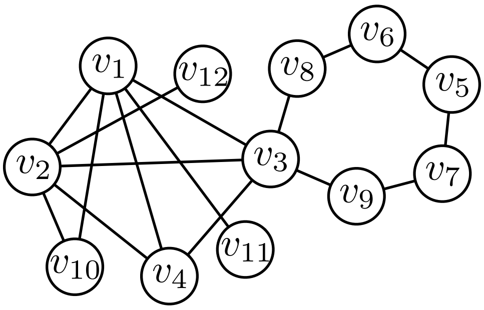


Figure 4.1: The Example Graph G

v, i.e., $deg(v, G) = |N(v, G)|$. The length of each edge $(u, v) \in E(G)$ is a positive value $\delta(u, v, G)$. In an unweighted graph, the length of all edges is 1.

The path $p(s, t, G)$ between two vertices $s, t \in V(G)$ is a sequence of vertices, i.e., $p(s, t, G) = \{s = v_0, v_1, \cdots, t = v_l\}$. The inner vertices of $p(s, t, G)$ are vertices v_i, where $v_i \neq s, v_i \neq t$, and $i \in [1, l-1]$. The precursor $prev(v_i)$ of a vertex v_i on the path $p(s, t, G)$ is v_{i-1}, where $i \in [1, l]$; the successor $succ(v_i)$ of a vertex v_i on the path $p(s, t, G)$ is v_{i+1}, where $i \in [0, l-1]$. Given two paths $p_1 = \{v_0, v_1, \cdots, v_a\}$ and $p_2 = \{w_0, w_1, \cdots, w_b\}$, when $(v_a, w_0) \in E(G)$, then the splicing operation on p_1 and p_2 is defined as $p_1 + p_2 = \{v_0, v_1, \cdots, v_a, w_0, w_1, \cdots, w_b\}$; or $v_a = w_0$, then $p_1 + p_2 = \{v_0, v_1, \cdots, v_a = w_0, w_1, \cdots, w_b\}$.

The length of $p(s, t, G)$ is defined as $|p(s, t, G)| = \Sigma_{(v_i, v_{i+1})}\delta(v_i, v_{i+1}, G)$, where $i \in [0, l-1]$. The shortest path between s and t is the one with the minimum length among all s-t paths, and the shortest distance $dist(s, t, G)$ is the length of the s-t shortest path. Without loss of generality, we assume that graph G is connected since otherwise, we can work on each connected component separately (as the shortest distance between the vertices of different connected components is positive infinity). If the context is obvious, we remove G from notations for simplicity.

Example 15. *Fig. 4.1 gives the example graph G, which contains $n = 12$ vertices and $m = 16$ edges. For the vertex v_5, $N(v_5) = \{v_6, v_7\}$, so $deg(v_5) = 2$.*

$p(v_5, v_3) = \{v_5, v_6, v_8, v_3\}$ is a v_5-v_3 path. The inner vertices of $p(v_5, v_3)$ are v_6 and v_8. The precursor (resp. successor) of v_6 is v_5 (resp. v_8) on $p(v_5, v_3)$. $p(v_5, v_3)$ is the shortest path between v_5 and v_3, thus $dist(v_5, v_3) = 3$. $p(v_4, v_2) = \{v_4, v_2\}$ is a v_4-v_2 path. As $(v_3, v_4) \in E$, we use the splicing operation to get $p(v_5, v_3) + p(v_4, v_2) = \{v_5, v_6, v_8, v_3, v_4, v_2\}$, which is a v_5-v_2 path.

Complex Graphs. Complex graphs are a specific category of graphs distinguished by their non-trivial topological properties, which are neither completely regular nor completely random. They are characterized by three core properties: a scale-free degree distribution, a high clustering coefficient, and small-world phenomena [52]. The scale-free property indicates a power-law distribution in the number of connections per node. A high clustering coefficient denotes the presence of well-connected subgroups within the graph. The small-world property illustrates that most nodes can be reached from others through a small number of steps. Despite their unweighted and undirected nature, these complex topological properties pose significant computational challenges in the study of graph problems such as the shortest path problem. The focus of this chapter is on these complex graphs.

Problem Definition. Given an undirected unweighted graph $G(V, E)$, a shortest-path query is defined as $\mathsf{Q_P}(s, t)$, where $s, t \in V$. The answer to query $\mathsf{Q_P}(s, t)$ is an s-t shortest path $p(s, t)$. If multiple s-t shortest paths exist, an arbitrary one can be returned. The studied problem is how to process query $\mathsf{Q_P}(s, t)$ efficiently on G.

4.3 Distance Queries and Extensions

The shortest-distance query is an operation closely related to the shortest-path query, which returns the shortest path length of two vertices. Many methods have been proposed to create indexes to process shortest-distance queries on complex networks; the well-known methods are PLL [6] and CTL [59]. This section describes how to extend PLL and CTL to support shortest-path queries.

4.3.1 PLL and Its Extension

Pruned landmark labeling (PLL) is a classical method for processing shortest-distance queries. PLL supports queries by creating an index $\mathsf{L}^{\mathsf{PLL}}$ offline. The index assigns a label $\mathsf{L}^{\mathsf{PLL}}(u) = \{(v, dist(u, v))\}$ to each vertex $u \in V$ in the graph G. $\mathsf{L}^{\mathsf{PLL}}(u)$ contains some selected **landmarks** v and the corresponding shortest distance $dist(u, v)$ for u. The number of landmarks contained in the label of u is defined as the label size $|\mathsf{L}^{\mathsf{PLL}}(u)|$. The maximum label size Δ^{PLL} of PLL is defined as the value of the largest label size among all vertices. The index size $|\mathsf{L}^{\mathsf{PLL}}|$ is the sum of the label size over each vertex $u \in V$, i.e., $|\mathsf{L}^{\mathsf{PLL}}| = \Sigma_{u\in V}|\mathsf{L}^{\mathsf{PLL}}(u)|$.

Distance Query Processing. To report the shortest distance $dist(s, t)$ between vertices s and t in G, we only need to use the labels of s and t, as given in Equation 4.1.

$$dist(s, t) = \min_{w \in \mathsf{L}^{\mathsf{PLL}}(s) \cap \mathsf{L}^{\mathsf{PLL}}(t)} dist(s, w) + dist(w, t). \tag{4.1}$$

We find common landmarks w in the labels of s and t, and select the smallest distance $dist(s, w) + dist(w, t)$ through w as a result. The time cost to compute $dist(s, t)$ is $O(|\mathsf{L}^{\mathsf{PLL}}(s)| + |\mathsf{L}^{\mathsf{PLL}}(t)|)$.

Example 16. *Consider the graph G in Fig. 4.1. The* PLL *column of Table 4.1 shows* $\mathsf{L}^{\mathsf{PLL}}$ *(ignore the third attribute marked blue in each entry) of G. For v_2, its label $\mathsf{L}^{\mathsf{PLL}}(v_2) = \{(v_1, 1), (v_2, 0)\}$ has two landmarks, v_1 and v_2, so $|\mathsf{L}^{\mathsf{PLL}}(v_2)| = 2$. The index size is $|\mathsf{L}^{\mathsf{PLL}}| = 44$. To compute $dist(v_2, v_3)$, we use the labels of v_2 and v_3 and get the common landmarks $\{v_1, v_2\}$. As $dist(v_2, v_1) + dist(v_3, v_1) =$*

Table 4.1: Comparison of Different Approaches

	PLL	CTL	MLL
v_1	$(v_1, 0, -)$	$(v_1, 0, -)$	
v_2	$(v_1, 1, -), (v_2, 0, -)$	$(v_1, 1, -), (v_2, 0, -)$	$(v_1, -)$
v_3	$(v_1, 1, -), (v_2, 1, -), (v_3, 0, -)$	$(v_1, 1, -), (v_2, 1, -), (v_3, 0, -)$	$(v_1, -), (v_2, -)$
v_4	$(v_1, 1, -), (v_2, 1, -), (v_3, 1, -), (v_4, 0, -)$	$(v_1, 1, -), (v_2, 1, -), (v_3, 1, -), (v_4, 0, -)$	$(v_1, -), (v_2, -), (v_3, -)$
v_5	$(v_1, 4, v_6), (v_2, 4, v_6), (v_3, 3, v_6), (v_5, 0, -)$	$(v_3, 3, v_6)$	(v_3, v_6)
v_6	$(v_1, 3, v_8), (v_2, 3, v_8), (v_3, 2, v_8), (v_5, 1, -), (v_6, 0, -)$	$(v_3, 2, v_8), (v_5, 1, -)$	$(v_3, v_8), (v_5, -)$
v_7	$(v_1, 3, v_9), (v_2, 3, v_9), (v_3, 2, v_9), (v_5, 1, -), (v_7, 0, -)$	$(v_3, 2, v_9), (v_5, 1, -)$	$(v_3, v_9), (v_5, -)$
v_8	$(v_1, 2, v_3), (v_2, 2, v_3), (v_3, 1, -), (v_5, 2, v_6), (v_6, 1, -), (v_8, 0, -)$	$(v_3, 1, -), (v_5, 2, v_6), (v_6, 1, -)$	$(v_3, -), (v_6, -)$
v_9	$(v_1, 2, v_3), (v_2, 2, v_3), (v_3, 1, -), (v_5, 2, v_7), (v_7, 1, -), (v_9, 0, -)$	$(v_3, 1, -), (v_5, 2, v_7), (v_7, 1, -)$	$(v_3, -), (v_7, -)$
v_{10}	$(v_1, 1, -), (v_2, 1, -), (v_{10}, 0, -)$	$(v_1, 1, -), (v_2, 1, -)$	$(v_1, -), (v_2, -)$
v_{11}	$(v_1, 1, -), (v_{11}, 0, -)$	$(v_1, 1, -)$	$(v_1, -)$
v_{12}	$(v_1, 2, v_2), (v_2, 1, -), (v_{12}, 0, -)$	$(v_2, 1, -)$	$(v_2, -)$

$1+1=2$ *and* $dist(v_2, v_2) + dist(v_2, v_3) = 0 + 1 = 1$*, by Equation 4.1, we know that* $dist(v_2, v_3) = 1$ *and* v_2 *is the landmark on* v_2*-*v_3 *shortest path.*

PLL Index Construction. PLL constructs the index $\mathsf{L}^{\mathsf{PLL}}$ for the graph $G(V, E)$ using the following steps.

1. Give each vertex $v \in V$ an order $r(v)$. PLL sets the order[1] by degree and breaks tie by vertex IDs, i.e., $r(v) = deg(v) + \frac{n-\text{ID}(v)}{n}$. Without loss of generality, let $r(v_1) > r(v_2) > \cdots > r(v_n)$.

2. Select each $v_i \in V$ sequentially according to the vertex order (v_1 is the first, v_n is the last) to do v_i-sourced pruned BFS.

When doing v_i-sourced pruned BFS, there are two cases if a vertex u is encountered while v_i is performing BFS. (1) If the shortest distance $dist(v_i, u)$ between v_i and u can be answered correctly by Equation 4.1 using the existing labels, then the expansion branch from u is pruned. (2) Otherwise, v_i is added to the label $\mathsf{L}^{\mathsf{PLL}}(u)$ of u, and the BFS continues. When v_n finishes pruned BFS, we obtain the index $\mathsf{L}^{\mathsf{PLL}}$ for return.

Theorem 7 presents the condition for determining whether vertex v will join the label of u as a landmark.

Theorem 7 ([59])**.** *The entry* $(v, dist(u, v))$ *is in* $\mathsf{L}^{\mathsf{PLL}}(u)$ *iff* $r(v) \geq r(w)$ *for* $\forall w$ *on all* v*-*u *shortest paths.*

Example 17. *Consider the graph* G *in Fig. 4.1.* $(v_2, 2) \in \mathsf{L}^{\mathsf{PLL}}(v_8)$ *since* v_2 *has the highest order among vertices on all* v_2*-*v_8 *shortest paths.* $(v_2, 2) \notin \mathsf{L}^{\mathsf{PLL}}(v_{11})$ *as* v_1*, whose order is higher than* v_2*, is on the* v_2*-*v_{11} *shortest path.*

[1] PLL can use other methods to order the vertices. For example, PLL can set the order of vertices in a similar way to the minimum degree elimination used by CTL.

Algorithm 5: Processing $\mathsf{Q_P}(s,t)$ Based on $\mathsf{L_E^{PLL}}$

```
   Input: $\mathsf{Q_P}(s,t)$, index $\mathsf{L_E^{PLL}}$
   Output: The shortest path $p(s,t)$
1  $w, dist(s,t) \leftarrow$ Equation 4.1;
2  if $dist(s,t) = 0$ then $p(s,t) \leftarrow \{s\}$; return $p(s,t)$;
3  if $dist(s,t) = 1$ then $p(s,t) \leftarrow \{s,t\}$; return $p(s,t)$;
4  $p_1 \leftarrow \{s\}, p_2 \leftarrow \{t\}$;
5  while $dist(s,w) > 1$ do
6      $s \leftarrow succ(s)$, where $(w, dist(s,w), succ(s)) \in \mathsf{L_E^{PLL}}(s)$;
7      $p_1 \leftarrow p_1 + \{s\}$;
8  while $dist(w,t) > 1$ do
9      $t \leftarrow succ(t)$, where $(w, dist(t,w), succ(t)) \in \mathsf{L_E^{PLL}}(t)$;
10     $p_2 \leftarrow \{t\} + p_2$;
11 return $p(s,t) = p_1 + \{w\} + p_2$
```

Extension to Path Queries. We next extend the PLL index to support shortest-path queries. The extended index $\mathsf{L_E^{PLL}}$ assigns a label $\mathsf{L_E^{PLL}}(u) = \{(v, dist(u,v), succ(u))\}$ to each vertex u. We add an extra attribute $succ(u)$, the successor of u on the shortest path $\{v_0 = u, v_1 = succ(u), \cdots, v_{l-1}, v_l = v\}$ from u to v, in the label $\mathsf{L_E^{PLL}}(u)$ of u. If $dist(u,v) < 2$, then the successor $succ(u)$ is meaningless, and we store "$-$" instead. The successor can be used to track all vertices on the shortest path and thus recover this path.

Example 18. *Consider the graph G in Fig. 4.1, where Table 4.1 lists the extended index $\mathsf{L_E^{PLL}}$. For v_6, $(v_3, 2, v_8) \in \mathsf{L_E^{PLL}}(v_6)$, where v_3 is the landmark and v_8 is the successor of v_6 on v_6-v_3 shortest path.*

Path Query Processing. We describe how to answer $\mathsf{Q_P}(s,t)$ using index $\mathsf{L_E^{PLL}}$ in Algorithm 5. First, we use Equation 4.1 to get $dist(s,t)$ and the landmark w in the labels of s and t, s.t., $dist(s,t) = dist(s,w) + dist(w,t)$ (Line 1). If $dist(s,t)$ is 0, then $s = t$ and we return $\{s\}$ as a path (Line 2); if $dist(s,t)$ is 1, then $\{s,t\}$ is an edge and we return this edge as a path (Line 3). Otherwise, w is used to decompose the s-t shortest path into two subpaths p_1 and p_2. p_1 is first initialized to contain only vertex s (Line 4), then we use $\mathsf{L_E^{PLL}}(s)$ to get the successor $succ(s)$ of s on the s-w subpath and iteratively add the vertices on the s-w subpath to p_1 by assigning $succ(s)$ to s (Line 5-7); similarly, we use $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(t)$ to iteratively add the vertices on the t-w subpath to p_2 (Line 8-10). Finally, p_1 and p_2 are spliced with w to produce the answer (Line 11).

Example 19. *Consider the graph G in Fig. 4.1. Given $\mathsf{Q}_{\mathsf{P}}(v_6, v_3)$, we use Equation 4.1 to get $dist(v_6, v_3) = 2$ and the landmark v_3 on the v_6-v_3 shortest path $p(v_6, v_3)$. The shortest path is divided into two subpaths p_1, p_2 by the landmark v_3. We initialize $p_1 = \{v_6\}$ and inquire $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(v_6)$ to obtain the successor $succ(v_6) = v_8$ of the starting vertex v_6 on the v_6-v_3 subpath. We add v_8 to p_1 and use v_8 as the new starting vertex for the next round of iterations. The iteration stops here because $dist(v_8, v_3) = 1$. We then initialize $p_2 = \{v_3\}$, but the iteration ends because $dist(v_3, v_3) = 0$. Finally, we return the answer $p_1 + \{v_3\} + p_2 = \{v_6, v_8, v_3\}$.*

Lemma 10. *Algorithm 5 correctly answers the query $\mathsf{Q}_{\mathsf{P}}(s, t)$.*

Proof. If $dist(s, t) \leq 1$, then the s-t shortest path $p(s, t)$ can be returned correctly by Line 2-3 of Algorithm 5. Otherwise, locate the landmark w on the path $p(s, t)$ and divide $p(s, t)$ into two subpaths $p_1 = p(s, prev(w))$ and $p_2 = p(succ(w), t)$, where $p(s, t) = p_1 + \{w\} + p_2$. We show how to find $p(s, x = prev(w))$, and a similar proof can be used for $p(succ(w), t)$. We use induction on the shortest distance: if $dist(s, x) < 1$, then $p(s, x) = \{s\}$ is found correctly according to Line 4 of Algorithm 5. Assume that the shortest path with distance $< dist(s, x)$ can be found correctly.

Theorem 7 shows that $(w, dist(u, w), succ(u)) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(u)$ for $\forall u \in p(s, x)$, since w is the highest-order vertex on all s-x shortest paths. By the induction hypothesis, $p(s, prev(x) = y)$ can be found, and the edge $\{y, x\}$ can be found correctly since $(w, dist(w, y), succ(y) = x) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(y)$, thus $p(s, x) = p(x, y) + \{y, x\}$ can be found correctly. □

Lemma 11. *Algorithm 5 requires*[2] $O(dist(s,t) \times \log \Delta^{\mathsf{PLL}})$ *to find the s-t shortest path, where* Δ^{PLL} *is* PLL*'s maximum label size.*

Proof. In the worse case, Algorithm 5 requires one label lookup for each vertex on the path (to determine the successor), and the complexity of each lookup via binary search is $\log \Delta^{\mathsf{PLL}}$. □

4.3.2 CTL and Its Extension

CTL is proposed to avoid the oversized indexes of PLL [59, 60] for shortest-distance queries. CTL relies on the concept of core-tree decomposition, which is a special kind of tree decomposition.

Definition 9 (Tree Decomposition)**.** *The tree decomposition of a graph $G(V,E)$, denoted as T_G, is a rooted tree, where every node $X \in V(T_G)$ in the tree is a subset of vertices of the graph G, i.e., $X \subseteq V(G)$. T_G meets the following three conditions.*

(1) $\bigcup_{X \in V(T_G)} X = V(G)$;

(2) For every edge $(u,v) \in E(G)$ in the graph G, there exists a node X in $V(T_G)$ such that $u \in X$ and $v \in X$;

(3) For every vertex $v \in V(G)$ in the graph G, the set $T(v) = \{X \in V(T_G) | v \in X\}$ is a connected subtree.

The root of subtree $T(v)$ is $X(v)$, for $\forall v \in V(G)$. The ***treewidth*** *of T_G is defined as $tw(T_G) = \max_{X \in V(T_G)} |X| - 1$.*

[2] If additional data structures such as hash tables or pointers are used to store the labels for the quick label lookup, the complexity can be reduced to $O(dist(s,t))$. We disregard this optimization due to the oversized PLL index.

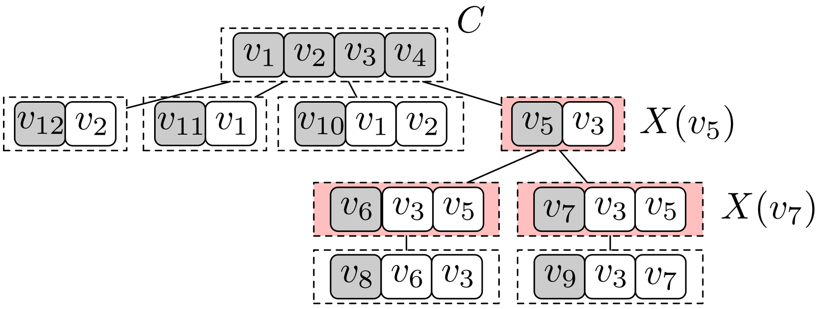


Figure 4.2: (Core-)Tree Decomposition of G

We refer to each $v \in V(G)$ as a vertex and each $X \in V(T_G)$ as a node. The **ancestor nodes** $\mathsf{ANC}(X)$ of node X are all the nodes on the shortest path from X to the root in T_G; the **parent node** $\mathsf{PAR}(X)$ of X is the ancestor node connecting to X.

Example 20. *Fig. 4.2 is the tree decomposition T_G of G in Fig. 4.1. We verify three conditions for T_G. (1) Each vertex of G (say v_1) appears in some node (say $C = \{v_1, v_2, v_3, v_4\}$) of T_G. (2) For each edge of G, say (v_1, v_2), we find a node C containing both endpoints v_1, v_2. (3) For vertex v_5 of G, all (marked in red) nodes in T_G containing v_5 form a connected subtree $T(v_5)$. The root of subtree $T(v_5)$ is $X(v_5)$. Similarly, $X(v_7)$ is the root of the subtree consisting of the nodes containing v_7. On T_G, the ancestor nodes of $X(v_7)$ are $\mathsf{ANC}(X(v_7)) = \{X(v_7), X(v_5), C\}$, and the parent node of $X(v_7)$ is $\mathsf{PAR}(X(v_7)) = X(v_5)$. The treewidth of T_G is $tw(T_G) = |C| - 1 = 3$.*

Definition 10 (Core-Tree Decomposition)**.** *Given a parameter d, the core-tree decomposition T_G of graph G is a tree decomposition with the fourth condition: there is a special node (defined as the* **core part***) $C \in V(T_G)$, s.t., $|C| > d + 1$; for the other nodes (defined as the* **tree part***) $X \in V(T_G) \setminus C$, $|X| \leq d + 1$.*

Example 21. *Consider the graph G in Fig. 4.1 and given $d = 2$, the tree decomposition T_G in Fig. 4.2 is also the core-tree decomposition of G. We verify* *the fourth condition: for $V(T_G)$, only the root C has size $|C| > 3$, while all other nodes in $V(T_G) \setminus C$ have size ≤ 3.*

Decomposing a Graph. To obtain a core-tree decomposition T_G (with parameter d) of graph G, we can use minimum degree elimination (MDE) [13]. MDE creates nodes and then edges of T_G.

Node elimination. MDE selects the smallest degree vertex for elimination each time. When the degree of the vertex selected at some time is $\geq d+1$, elimination stops. We initialize $G_1 = G$, and then every time i, we eliminate vertex v with the smallest degree in G_i.

1. v: the vertex with the smallest degree in the graph G_i (or any of them if there is a tie), and the order of v is set to $r(v) = i$.

2. For v's neighbors $N(v, G_i)$ in G_i, we add extra edges to make $N(v, G_i)$ form a clique (i.e., complete graph).

   (a) If $u, w \in N(v, G_i)$ is not an edge in G_i, we add edge (u, w) whose length equals the length of the path $\{u, v, w\}$, i.e., $\delta(u, v, G_i) + \delta(w, v, G_i)$. To record that (u, w) is made by removing v, we set v as the **elimination vertex** of (u, w).

   (b) Otherwise (u, w) is an edge of G_i, then we set its length to the smallest of the current length and the length of the path $\{u, v, w\}$, i.e. $\min\{\delta(u, v, G_i), \delta(u, v, G_i) + \delta(w, v, G_i)\}$. If the edge length is updated with a smaller value, we set v as the elimination vertex of (u, w).

3. Form a node $X(v) = v \cup N(v, G_i)$.

4. Delete v from G_i to form G_{i+1} for the next round ($i \leftarrow i + 1$).

Example 22. *Given $d = 2$, we use MDE on G of Fig. 4.1. First, we delete v_{12} from $G_1 = G$ to get G_2, and obtain $X(v_{12}) = \{v_{12}, v_2\}$, $r(v_{12}) = 1$. Then, we delete v_{11} from G_2 to get G_3. Next, we delete v_{10} from G_3 to get G_4. Here, for v_{10}'s neighbors $\{v_1, v_2\}$ in G_3, since (v_1, v_2) is an edge, its length is set to $\min\{\delta(v_1, v_2), \delta(v_1, v_{10}) + \delta(v_2, v_{10})\} = 1$. We also get $X(v_{10}) = \{v_{10}, v_1, v_2\}$, $r(v_{10}) = 3$. We continue deleting v_9, v_8, v_7, v_6, until v_5 to get G_9. At this point, the degree of vertices in $G_\lambda = G_9 = \{v_1, v_2, v_3, v_4\}$ is ≥ 3, and we stop.*

Edge generation. We next describe how to generate edges by imposing parent relations for the nodes in T_G. When the vertex elimination stops, suppose the value of i is λ, then we get a graph G_λ. We get all the nodes $X(v)$ in the tree part of T_G, where $r(v) \in [1, \lambda - 1]$.

1. Take all vertices $V(G_\lambda)$ in G_λ as the core part C of T_G.

2. For each node $X(v)$ in the tree part,

   (a) if the vertices of $X(v) \setminus \{v\}$ all belong to C, then make C the parent of $X(v)$;

   (b) otherwise, find the vertex $u \notin C$ with the lowest order $r(u)$ in $X(v) \setminus \{v\}$, and make $X(u)$ the parent of $X(v)$.

After the whole process, we get the core-tree decomposition T_G.

Example 23. *We describe how to create $E(T_G)$. We first take vertices $\{v_1, v_2, v_3, v_4\}$ in $G_\lambda = G_9$ as C. Then for each node $X(v) \in V(T_G) \setminus C$, we find its parent in T_G. For example, for $X(v_{12}) = \{v_{12}, v_2\}$, since $X(v_{12}) \setminus \{v_{12}\} \subseteq C$, $\mathsf{PAR}(X(v_{12})) = C$. For $X(v_7) = \{v_7, v_3, v_5\}$, the vertex (not in C) with the lowest order in $X(v_7) \setminus \{v_7\}$ is v_5, then $\mathsf{PAR}(X(v_7)) = X(v_5)$.*

CTL **Index.** Given a core-tree decomposition T_G of G, we create separate indexes for the core part C and the tree part $V(T_G) \setminus C$. For vertices v in the tree part, the order $r(v)$ is set to the moment i when v is eliminated; For vertices v in the core part, the order $r(v)$ is set according to the degree (as in PLL). The order of vertices in the core part is forced to be set higher than the vertices in the tree part. The indexes of the two parts together form the index $\mathsf{L}^{\mathsf{CTL}}$.

Core index. For the core part C, we create the index in the graph G_λ using PLL, thus assigning a core label for each vertex in C.

Tree index. For each node $X(v) \in V(T_G) \setminus C$ in the tree part, we assign a tree label for the corresponding vertex v of $X(v)$. The label of v contains the distances between v and its landmarks u where $u \in \bigcup_{X \in \mathsf{ANC}(X(v)) \setminus C} X$, and $u \neq v$.

When using the index $\mathsf{L}^{\mathsf{CTL}}$ to obtain the distance $dist(s,t)$ between two vertices s and t, if both s and t are in the core part, then we can just use the core index to complete the distance query according to Equation 4.1; otherwise, if one of the vertices is not in the core part, then we need to use both the core index and the tree index to complete the distance query[3]. The time cost of shortest-distance queries using $\mathsf{L}^{\mathsf{CTL}}$ is $O(d \cdot \log |C| \cdot tw(T_G))$, where $tw(T_G)$ is the treewidth of T_G.

Example 24. *Consider the graph G in Fig. 4.1, and the* CTL *column of Table 4.1 shows* $\mathsf{L}^{\mathsf{CTL}}$ *(ignore the third attribute marked blue in each entry, which is used for extension). (1) Core index. For vertices in C, we build the core index using* PLL *in $G_\lambda = G_9$. (2) Tree index. For vertex $v \in V(V_G) \setminus C$, we build the tree index. For example, for v_9,* $\mathsf{ANC}(X(v_9)) \setminus C = \{X(v_9), X(v_7), X(v_5)\}$*, so its tree label contains landmarks in $X(v_9) \cup X(v_7) \cup X(v_5) \setminus \{v_9\} = \{v_3, v_5, v_7\}$.*

[3] For a more detailed distance query process, please refer to the literature [60].

Extension to Path Queries.

To handle shortest-distance queries, CTL assigns a (core or tree) label $\mathsf{L}^{\mathsf{CTL}}(u) = \{(v, dist(u,v))\}$ to each vertex $u \in V$ in the graph, where v is the landmark of u. To make CTL support shortest-path queries, like PLL, we need to add an extra attribute $auxi(u)$ to the label entry $(v, dist(u,v))$ of each vertex u, thus obtaining the extended label $\mathsf{L}^{\mathsf{CTL}}_{\mathsf{E}}(u) = \{(v, dist(u,v), auxi(u))\}$. The extra attribute $auxi(u)$ is an inner vertex on the u-v shortest path; it is used to help find the shortest path between two vertices. If $dist(u,v) \leq 1$, then $auxi(u)$ is unnecessary, and we store "-" instead. Because CTL separates the vertices into two parts, we discuss two different extended labels.

Extended Core Label. For a vertex $u \in C$ in the core part, we extend the label of u in a similar way as we extend PLL, i.e., for any landmark v of u, we set $auxi(u)$ to $succ(u)$, the successor of u on the u-v shortest path in G_λ. The only difference is that for an edge (u,v) in G_λ, its weight $\delta(u,v,G_\lambda)$ may be greater than 1. In this case, we need to further find the corresponding u-v shortest path in G for this edge (u,v) in G_λ. To do so, instead of assigning $auxi(u)$ the value "-", we assign the elimination vertex w of (u,v) — eliminating w forms the edge (u,v) — to $auxi(u)$.

Extended Tree Label. For each node $X(u) \in V(T_G) \setminus C$ in the tree part, we extend the tree label of the vertex u corresponding to that node. Specifically, for each landmark v of u, if $dist(u,v) < 2$, $auxi(u)$ is set to "-"; otherwise, we choose some vertex on the u-v shortest path to be assigned to $auxi(u)$. (1) If $v \notin X(u)$, an inner vertex on the u-v shortest path can be picked from $X(u)$ as $auxi(u)$ — According to Lemma 3 of [21], $X(u) \setminus u$ is the cut that separates u and v, so $X(u)$ must contain some inner vertex on the u-v shortest path; (2) Otherwise, an inner vertex on the u-v shortest path can be found from either $X(u)$ or the elimination vertex w of (u,v) as $auxi(u)$ — Lemma 3 of [21] does not necessarily apply to this case, and the u-v shortest path may contain the elimination vertex w, since w must be on the (local) u-v shortest path whose inner vertices all do not belong to $X(u)$ [60].

Example 25. *Consider the core-tree decomposition in Fig. 4.2 and the extended* CTL *index* $\mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}$ *in Table 4.1. Given the vertex* v_8 *in the tree part, for landmark* v_5 *of* v_8*, as* $v_5 \notin X(v_8)$*, we pick an inner vertex* $v_6 \in X(v_8)$ *on the* v_8*-*v_5 *shortest path from* $X(v_8) = \{v_8, v_6, v_3\}$ *as* $auxi(v_8)$ *to extend the label entry. Given the vertex* v_6 *in the tree part, for landmark* v_3 *of* v_6*, as* $v_3 \in X(v_6)$*, we find an inner vertex* v_8 *on the* v_6*-*v_3 *shortest path as* $auxi(v_6)$ *to extend the label entry, where* v_8 *is the elimination vertex of the edge* (v_6, v_3)*.*

Path Query Processing

We use the extended CTL index $\mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}$ to process the path query $\mathsf{Q}_{\mathsf{P}}(s,t)$, thus getting the s-t shortest path $p(s,t)$. There is some complexity in using the extended CTL index for path queries. For the sake of convenience, we first discuss two special cases of queries, and then introduce how to handle the general case of path queries based on these two special cases.

Special Cases. We first give two special cases ($\mathsf{sp}1$-$\mathsf{sp}2$) of queries.

$\mathsf{sp}1$*: $p(s,t)$ contains only the vertices of the tree part.* We find the s-t shortest path $p(s,t)$ using the following steps.

1. We first use tree index[4] and a similar function to Equation 4.1 to find $dist(s,t)$ as well as the landmark vertex w on $p(s,t)$; we divide the path $p(s,t)$ into two parts by landmark w: s-w subpath p_1 and w-t subpath p_2. The following steps only explain how to find p_1, and the process of finding p_2 is similar.

[4] Note that each vertex u of the tree part uses itself as landmark during query processing, i.e., generate a new label entry $(u, 0, \text{“}-\text{''})$.

2. If $dist(s, w) = 0$, we return $p_1 = \{s\}$; if $dist(s, w) = 1$, return $p_1 = \{s, w\}$. Otherwise, we find $(w, dist(w, s), auxi(s)) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(s)$. We use attribute $auxi(w)$ to decompose p_1 into s-$auxi(s)$ and $auxi(s)$-w subpaths. We recursively call Step (2) to find them. Finally, they are spliced together as p_1 to return.

3. Return $p_1 + \{w\} + p_2$ as the s-t shortest path.

sp2*: $p(s, t)$ contains vertices of both parts, $s \notin C$, $t \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(s) \cap C$.* If $dist(s, t) = 1$, then $p(s, t) = \{s, t\}$. Otherwise, since $t \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(s)$, we can obtain the label entry $(t, dist(s, t), auxi(s))$ to get the extra attribute $w = auxi(s)$. We then use w to find the path $p(s, t)$ recursively (similar to Step (2) of sp1).

Example 26. *Consider the extended* CTL *index in Table 4.1. For the query* $\mathsf{Q_P}(v_9, v_5)$*, since the shortest path* $p(v_9, v_5)$ *does not pass through the vertices of the core part, it belongs to* sp1*. To process* $\mathsf{Q_P}(v_9, v_5)$*, we use the tree index to find a landmark* v_5 *on the* v_9*-*v_5 *path to divide the path into subpath* $p1(v_9, v_5)$ *and the trivial subpath* $p2(v_5, v_5)$*. (Step (1)). Then, to find* $p1(v_9, v_5)$*, we query* $\mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(v_9)$ *to get* $(v_5, 2, auxi(v_5) = v_7)$*, and then use the extra attribute* v_7 *to query and splice paths recursively until* $p1(v_9, v_5) = \{v_9, v_7, v_5\}$ *is found (Step (2)). For the query* $\mathsf{Q_P}(v_5, v_3)$*, since* $v_5 \notin C$ *and* $v_3 \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(v_5) \cap C$*, it belongs to* sp2*. To process* $\mathsf{Q_P}(v_5, v_3)$*, we first query* $\mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(v_5)$ *to get* $(v_3, 3, auxi(v_5) = v_6)$*, and then use the extra attribute* v_6 *to query and spice paths recursively until* $p(v_5, v_3) = \{v_5, v_6, v_8, v_3\}$ *is returned.*

General Cases. Based on sp1 and sp2, we are now ready to give query processing in general.

Case 1: $s, t \in C$. First, we use a similar method to Algorithm 5 to get the shortest path $p(s, t, G_\lambda)$ between s and t in G_λ. For each edge (u, v) on the path, if

$dist(u, v) = 1$, then the edge (u, v) is returned directly; otherwise, the edge (u, v) needs to be unfolded to the u-v shortest path on the original graph — suppose $r(u) > r(v)$, then $(u, dist(u, v), auxi(v) = w) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(v)$, we divide $p(u, v)$ into u-w subpath p_1 and w-v subpath p_2. Since eliminating $w \notin C$ produces the edge (u, v) in G_λ, then $u, v \in X(w)$ by the core-tree decomposition process and thus u and v are the landmarks of w. Therefore, the subpaths p_1 and p_2 can be both found using $\mathsf{sp2}$, which can be spliced to produce $p(u, v)$. The s-t shortest path in G is obtained by applying the above process to all edges in $p(s, t, G_\lambda)$.

Case 2: $s \notin C, t \in C$ (or vice versa). Using the distance query of CTL, we can get the vertex c on $p(s, t)$, where $c \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(s) \cap C$ (c is the interface [60]). We divide the s-t shortest path $p(s, t)$ into two segments, $p(s, c)$ and $p(c, t)$, where the subpath $p(s, c)$ can be found by $\mathsf{sp2}$ and the subpath $p(c, t)$ can be processed by Case 1. Finally, $p(s, c)$ and $p(c, t)$ can be spliced to obtain $p(s, t)$.

Case 3: $s, t \notin C$. Using the distance query of CTL, we can get the vertex c (resp. d) on $p(s, t)$, where $c \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(s) \cap C$ (resp. $d \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(s) \cap C$). If both c and d do not exist, this indicates that s and t do not pass through the core part, then $p(s, t)$ is obtained directly using $\mathsf{sp1}$; otherwise, since $c \in C$, the subpath $p(s, c)$ can be handled by Case 2; since both $c, d \in C$, the subpath $p(c, d)$ can be handled by Case 1; since $d \in C$, the subpath $p(d, t)$ can be processed by Case 2. $p(s, t)$ can be obtained by splicing $p(s, c)$, $p(c, d)$, and $p(d, t)$.

Example 27. *Consider the extended index $\mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}$ in Table 1. We show how to process the query $\mathsf{Q_P}(v_5, v_{10})$ (Case 3). (1) We first find $c = v_3 \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(v_5)$ on the v_5-v_{10} path using the distance query of CTL. We can find the subpath $p(v_5, v_3) = \{v_5, v_6, v_8, v_3\}$ (Case 2). (2) We then find $d = v_1 \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(v_{10})$ on the v_5-v_{10} path by querying the CTL index. We can find the subpath $p(v_1, v_{10}) = \{v_1, v_{10}\}$ (Case 2). (3) We find the subpath $p(v_3, v_1) = \{v_3, v_1\}$ via the extended core index (Case 1). Hence, $p(v_5, v_{10}) = p(v_5, v_3) + p(v_3, v_1) + p(v_1, v_{10}) = \{v_5, v_6, v_8, v_3, v_1, v_{10}\}$.*

Lemma 12. *Using $\mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}$ can correctly answer the query $\mathsf{Q}_{\mathsf{P}}(s,t)$.*

Proof. We first prove that the two special cases are correct.

(1) sp1: For $p(s,t)$, we can correctly find a landmark w and thus split $p(s,t)$ into s-w subpath $p(s,w)$ and w-t subpath $p(w,t)$ by Step (1) of sp1. In the following, we prove by induction that Step (2) of sp1 can correctly return $p(s,w)$ (and similarly $p(w,t)$), so that $p(s,t) = p(s,w) + \{w\} + p(w,t)$ can be correctly found.

When $d(s,w) \leq 1$, Step (2) can correctly return $\{s = w\}$ (when $dist(s,w) = 0$) or $\{s,w\}$ (when $dist(s,w) = 1$) as $p(s,w)$. Assuming that Step (2) can return paths of length $< dist(s,w)$, we prove that paths of length $dist(s,w)$ can also be returned. Without loss of generality, suppose $(w, dist(w,s), auxi(s)) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(s)$, we first prove the correctness of the claim that s (resp. w) is a landmark of $auxi(s)$ or $auxi(s)$ is a landmark of s (resp. w). This is because w is a landmark of s, then by the definition of tree labels, $X(w)$ must be an ancestor of $X(s)$. 1) If $auxi(s)$ is an eliminating vertex, then $X(s)$ is an ancestor of $X(auxi(s))$ and both w and s are landmarks of $auxi(s)$. 2) Or $auxi(s) \in X(s)$, then $X(auxi(s))$ is an ancestor of $X(s)$, so $auxi(s)$ is a landmark of s. In this case, a) either $X(w)$ is an ancestor of $X(auxi(s))$, and w is a landmark of $auxi(s)$; b) or $X(auxi(s))$ is an ancestor of $X(w)$, and $auxi(s)$ is a landmark of w.

If s (resp. w) is a landmark of $x = auxi(s)$, then $(s, dist(s,x), auxi(x)) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(x)$ (resp. $(w, dist(w,x), auxi(x)) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(x)$), and thus the s-$auxi(s)$ subpath $p(s, auxi(s))$ and the $auxi(s)$-w subpath $p(auxi(s), w)$ can be found recursively using Step (2). Under the induction hypothesis, subpaths $p(s, auxi(s))$ and $p(auxi(s), w)$ can be correctly returned by Step (2), and it follows that $p(s,w) = p(s, auxi(s)) + p(auxi(s), w)$; If $auxi(s)$ is a landmark of s (resp. w), the correctness can be proved similarly.

(2) sp2: similar to the proof of sp1.

We then prove three cases are correct. (1) Case 1: It is correct by the correctness of Algorithm 5, and $\mathsf{sp2}$. (2) Case 2: It is correct by the correctness of Case 1, and $\mathsf{sp2}$. (3) Case 3: It is correct by the correctness of Case 1, Case 2, and $\mathsf{sp1}$. □

Lemma 13. *Using* $\mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}$ *requires* $O(dist(s,t) \times \log \Delta^{\mathsf{CTL}})$ *to answer the query* $\mathsf{Q}_{\mathsf{P}}(s,t)$*, where* Δ^{CTL} *is* CTL*'s maximum label size.*

Proof. In the worse case, we require one label lookup for each vertex on the path (to determine the extra attribute), and the complexity of each lookup via binary search is $\log \Delta^{\mathsf{CTL}}$. □

4.4 Monotonic Landmark Labeling

Section 4.3 describes how to extend PLL and CTL to support shortest-path queries. Implementing the extensions requires adding an extra attribute to each index entry to enable fast pathfinding. However, adding the extra attributes makes the extended PLL and CTL indexes too large: both the extended PLL and CTL indexes occupy about twice the size of the original index. On the other hand, although using the traversal-based approach does not require high space cost, there is no way to guarantee query time.

In this section, we propose a new extension-based approach, Monotonic Landmark Labeling (MLL), to further balance the space cost and query time. Considering that CTL can handle large graphs that PLL cannot [60], we choose to extend CTL. Instead of adding extra attributes to each entry of the CTL index, our approach MLL is to non-trivially create an additional lightweight index (i.e., the MLL index $\mathsf{L}^{\mathsf{MLL}}$) on top of the CTL index as a plug-in to facilitate shortest-path queries. This lightweight index not only avoids the excessive space cost caused by the extra attributes but also guarantees query time. We will introduce the new MLL index in Section 4.4.1, followed by a description of how to use both the CTL index and the MLL index to support queries in Section 4.4.2, and finally, we will introduce how to create the MLL index in Section 4.4.3.

4.4.1 Index Structure

The concept of the monotonic shortest path underpins our method MLL. We set the vertex order of MLL using the same order as CTL.

Definition 11 (Monotonic Shortest Path)**.** *Given two vertices s, t of G, the s-t shortest path $p(s,t) = \{v_0 = s, v_1, \cdots, v_l = t\}$ is monotonic iff $r(v_i) < \min\{r(s), r(t)\}$ for $\forall v_i \in p(s,t)$, $i \in [1, l-1]$.*

The shortest path $p(s,t)$ is monotonic if the order of inner vertices (i.e., vertices excluding s and t) is lower than both s and t. Any edge in the graph is a trivial monotonic shortest path. We show any shortest path can be split into several monotonic shortest paths.

Lemma 14. *The shortest path $p(s,t)$ between any two vertices s and t can be split into a set of monotonic shortest paths $\{\tilde{p_1}, \tilde{p_2}, \cdots, \tilde{p_l}\}$, s.t., $p(s,t) = \tilde{p_1} + \tilde{p_2} \cdots \tilde{p_l}$.*

Proof. The proof holds if $s = t$ or $p(s,t)$ is monotonic. Assume $r(s) < r(t)$; we prove by construction. We start with $s = v_0$ and find the first vertex v_i of order higher than s, forming a monotonic shortest path $\tilde{p_1}$. We start from v_i and repeat the process until t is met. So we decompose $p(s,t)$ into $\{\tilde{p_i}\}$. □

Example 28. *Consider the graph G in Fig. 4.1. We assume that $r(v_1) > r(v_2) > \cdots > r(v_{12})$. The v_3-v_5 shortest path $p(v_3, v_5) = \{v_3, v_9, v_7, v_5\}$ is monotonic as the order of inner vertices $\{v_9, v_7\}$ is lower than v_3 and v_5. The v_5-v_4 shortest path $p(v_5, v_4) = \{v_5, v_7, v_9, v_3, v_4\}$ is not monotonic as the order of the inner vertex v_3 is higher than v_5. We describe how to decompose*

$p(v_5, v_4)$ into several monotonic shortest paths. First, starting from v_5, we find a vertex v_3 of order higher than v_4 and stop, forming the first monotonic path $\tilde{p_1}(v_5, v_3) = \{v_5, v_7, v_9, v_3\}$. Then starting from v_4 until meeting v_3, we get the second monotonic path $\tilde{p_2}(v_3, v_4) = \{v_3, v_4\}$. It follows that $p(v_5, v_4) = \tilde{p_1}(v_5, v_3) + \tilde{p_2}(v_3, v_4)$.

MLL Index. Lemma 14 shows that any shortest path can be split into several monotonic shortest paths. Our basic idea is to index monotonic shortest paths, and these indexed shortest paths can be stitched together to answer any shortest-path query. Based on this, we create a new kind of index $\mathsf{L}^{\mathsf{MLL}}$. The index $\mathsf{L}^{\mathsf{MLL}}$ assigns a label $\mathsf{L}^{\mathsf{MLL}}(u)$ to each vertex $u \in V$ in the graph, which includes some landmark v selected for u and the auxiliary vertex $h(u)$.

Definition 12. *The entry $(v, h(u))$ is in $\mathsf{L}^{\mathsf{MLL}}(u)$ iff*

1. *$r(v) \geq r(w)$ for $\forall w$ on all u-v shortest paths, and $v \neq u$;*

2. *All u-v shortest paths are monotonic.*

$h(u)$ is the highest-order inner vertex on all u-v shortest paths.

For MLL, if v is a landmark of u, then v needs to satisfy two conditions. The first condition is that $r(v)$ is the highest over all vertices in u-v shortest paths. Note that PLL only requires this condition to add v as the landmark of u (see Theorem 7). Also, we require $v \neq u$ to forbid v to become its own landmark. The second condition is that all u-v shortest paths must be monotonic. Without this condition, MLL will index the shortest paths (between any two vertices) as in PLL. Because of this condition, MLL only indexes the monotonic shortest paths, causing MLL to have a significantly smaller index size than PLL.

Also, we record the inner vertex $h(u)$ that has the highest order among all u-v shortest paths. If $dist(u, v) < 2$, then there is no inner vertex on the u-v

shortest path, so we store "-" instead. The role of $h(u)$ is similar to that of the precursor used to extend PLL for shortest-path queries (see Algorithm 5), i.e., using $h(u)$, we can track all vertices on a shortest path and thus recover the path.

Example 29. *Consider the graph G in Fig. 4.1, where the* MLL *column of Table 4.1 gives the* MLL *index. For vertex v_6, $(v_3, v_8) \in \mathsf{L}^{\mathsf{MLL}}(v_6)$ because all v_3-v_6 shortest paths are monotonic and v_3 has the highest order on all paths; $h(v_6) = v_8$ as v_8 is the inner vertex with the highest order on all v_3-v_6 paths. Similarly, for v_2, $(v_1, -) \in \mathsf{L}^{\mathsf{MLL}}(v_2)$ because all v_1-v_2 shortest paths are monotonic and v_1 has the highest order; $h(v_2) = -$ because $dist(v_1, v_2) = 1$.*

A careful reading of Definition 12 reveals redundancy. The second condition (all u-v shortest paths are monotonic) implies that all inner vertices w on u-v shortest paths are of a lower order than v, i.e., $r(v) > r(w)$. We give a more intuitive condition to decide if v is a landmark of u.

Theorem 8. *The entry $(v, h(u))$ is in $\mathsf{L}^{\mathsf{MLL}}(u)$ iff all u-v shortest paths are monotonic, and $r(v) > r(u)$.*

Index Size. MLL supports path queries by building an additional MLL index on top of the CTL index, so the total index size of MLL includes the CTL index size and the MLL index size (as the extra space cost). On the other hand, extending CTL (and PLL) by extra attributes also introduces an extra space cost. According to empirical results, the extra space required for the extended CTL (and PLL) method occupies almost the same size as the original index. To intuitively compare the *extra space cost* required by MLL and the extended CTL (and PLL) method, we compare the MLL index size with the original CTL (and PLL) index size.

We define the label size $|\mathsf{L}^{\mathsf{MLL}}(u)|$ of u as the number of landmarks contained in $\mathsf{L}^{\mathsf{MLL}}(u)$. Then the MLL index size is defined as $|\mathsf{L}^{\mathsf{MLL}}| = \Sigma_{u \in V} |\mathsf{L}^{\mathsf{MLL}}(u)|$. We show the PLL index size exceeds the MLL index size (suppose PLL and MLL use the same vertex order.).

Theorem 9. $|\mathsf{L}^{\mathsf{MLL}}| < |\mathsf{L}^{\mathsf{PLL}}|$, *where* $\mathsf{L}^{\mathsf{MLL}}$ *is the index of* MLL *created on top of* CTL.

Proof. To complete the proof, we show that for any vertex u, if landmark v belongs to $\mathsf{L}^{\mathsf{MLL}}(u)$, then v belongs to $\mathsf{L}^{\mathsf{PLL}}(u)$. According to Definition 12, v is the vertex with the highest order in all (monotonic) shortest paths from v to u. By Theorem 7, v is also in $\mathsf{L}^{\mathsf{PLL}}(u)$. The inequality strictly holds because u will act as its own landmark in PLL, but not in MLL (as $v \neq u$ is required). □

We show that the CTL index size exceeds the MLL index size. We define the label size of each vertex u as the number of u's landmarks in u's label for CTL. So the CTL index size $|\mathsf{L}^{\mathsf{CTL}}|$ is the total label size of all vertices.

Theorem 10. $|\mathsf{L}^{\mathsf{MLL}}| < |\mathsf{L}^{\mathsf{CTL}}|$, *where* $\mathsf{L}^{\mathsf{MLL}}$ *is the index of* MLL *created on top of* CTL.

Proof. We first analyze the core index of CTL. The core index is created on G_λ with vertices in the core part C. The edges of G_λ consist of edges $\{(u, v) \in E(G) | u, v \in C\}$ in $E(G)$ and extra edges formed by eliminating (lower order) vertices not in C. Since we use PLL to create the core-index on G_λ, Theorem 7 continues to hold. Then by Theorem 9, the size of the core index for CTL exceeds the total size of labels assigned to vertices in C for MLL.

Then we analyze the tree index of $\mathsf{L}^{\mathsf{CTL}}$. For any vertex $u \in V(G) \setminus C$, we prove that if v is a landmark of $\mathsf{L}^{\mathsf{MLL}}(u)$, then it must be a landmark of v by CTL. By Definition 12, all v-u shortest paths must be monotonic, i.e., the order of inner vertices w on all v-u shortest paths is lower than both v and u. All inner vertices w are eliminated before v and u for CTL, so there is an edge between u and v when eliminating u. As a result, $v \in X(u)$ and v is a landmark for u. The inequality strictly holds as long as C is not empty. □

Example 30. *Table 4.1 compares the three different indexes.* $|\mathsf{L}^{\mathsf{MLL}}| < |\mathsf{L}^{\mathsf{PLL}}|$*: the index of* MLL *contains 19 landmarks while* PLL *contains 44.* $|\mathsf{L}^{\mathsf{MLL}}| < |\mathsf{L}^{\mathsf{CTL}}|$*: the index of* MLL *contains 19 landmarks while* CTL *contains 25.*

Remark. If the graph G with n vertices is a star graph, and the center vertex has the highest order, then the MLL index size reaches the lower bound value $O(n)$; if the graph G is a clique, then the MLL index size reaches the upper bound value $O(n^2)$. In practice, the MLL index size is small. According to the experimental results in Section 4.6, the MLL index size on all graphs does not exceed 23 GB. Compared to the index sizes of PLL and CTL, the size of the MLL index is on average 22 times and 5.19 times smaller than that of the PLL and CTL indexes (before the extension). Considering that extending CTL (and PLL) using extra attributes requires extra space cost close to the index size itself, building a lightweight MLL index on top of the CTL index consumes less extra space cost.

4.4.2 Query Processing

We first describe how to process queries using $\mathsf{L}^{\mathsf{MLL}}$ (and $\mathsf{L}^{\mathsf{CTL}}$) if all shortest paths between two vertices are monotonic, and then show how to process queries in general. The entire query process is given in Algorithm 6.

Case 1: All Paths Are Monotonic. For MLL, by Definition 12, if v is a landmark of u, i.e., $(v, x = h(u)) \in \mathsf{L}^{\mathsf{MLL}}(u)$, then all v-u shortest paths are monotonic. To find the v-u shortest path $p(u, v)$ in this case, we use an idea similar to the one used in the PLL extension, i.e., to employ an auxiliary vertex $h(u)$ (similar to $succ(u)$ in Algorithm 5) to track all vertices on a shortest path. Specifically, we define Procedure $\mathsf{Unfold}(u, v, x)$ in Algorithm 6 (Line 12-18).

We use $x = h(u)$ to decompose $p(u, v)$ into subpaths $p(u, x)$ and $p(x, v)$ to find them separately. For $p(u, x)$, if $dist(u, x) = 1$, then (u, x) is an edge and is returned as $p(u, x)$ (Line 14). Otherwise, we take out $(u, h(x))$ from $\mathsf{L}^{\mathsf{MLL}}(x)$, and call $\mathsf{Unfold}(u, x, h(x))$ recursively to find $p(u, x)$ (Line 15). Similarly, we find $p(x, v)$ (Line 16-17) and return $p(u, x) + p(x, v)$ as a result (Line 18).

Example 31. *Consider the graph G in Fig. 4.1. v_3 is a landmark of v_5: $(v_3, v_6) \in \mathsf{L}^{\mathsf{MLL}}(v_5)$. To find the v_3-v_5 shortest path, we call Procedure $\mathsf{Unfold}(v_3, v_5, v_6)$. (1) Since $dist(v_3, v_6) > 1$ and $(v_3, v_8) \in \mathsf{L}^{\mathsf{MLL}}(v_6)$, we recursively call $\mathsf{Unfold}(v_3, v_6, v_8)$ to get path $p(v_3, v_6) = \{v_3, v_8, v_6\}$. (2) Since $dist(v_6, v_5) = 1$, we directly return $\{v_6, v_5\}$ as the v_6-v_5 shortest path $p(v_6, v_5)$. $p(v_3, v_6) + p(v_6, v_5) = \{v_3, v_8, v_6, v_5\}$ is returned as $p(v_3, v_5)$.*

Lemma 15. *Procedure $\mathsf{Unfold}(u, v, x = h(u))$ correctly returns the u-v shortest path $p(u, v)$ when $(v, h(u)) \in \mathsf{L}^{\mathsf{MLL}}(u)$.*

Proof. On the $p(u, v)$ path, x is the inner vertex with the highest order. We split $p(u, v)$ into subpaths $p(u, x)$ and $p(x, v)$, and $p(u, v) = p(u, x) + p(x, v)$. We show that the subpath $p(u, x)$ can be answered correctly, and a similar proof can be used for $p(x, v)$. We use the induction on the shortest distance: if $dist(u, x) = 1$, then (u, x) is an edge that $p(u, x)$ is found correctly. Assume that shortest paths with length $< dist(u, x)$ can be found correctly.

We next prove that $(u, h(x)) \in \mathsf{L}^{\mathsf{MLL}}(x)$ since otherwise there is vertex $w \neq x$ on the u-x shortest path of order higher than x. Since the u-x shortest path is a subpath of the u-v shortest path, w is also an inner vertex of u-v shortest path. This contradicts the fact that x is the inner vertex with the highest

Algorithm 6: Processing $\mathsf{Q_P}(s,t)$ for MLL

```
Input: $\mathsf{Q_P}(s,t)$, index $\mathsf{L^{CTL}}$, index $\mathsf{L^{MLL}}$
Output: The shortest path $p(s,t)$
if $r(s) > r(t)$ then swap $s$ and $t$;
$dist(s,t) \leftarrow$ query by $\mathsf{L^{CTL}}$;
if $dist(s,t) = 0$ then $p(s,t) \leftarrow \{s\}$, return $p(s,t)$;
if $dist(s,t) = 1$ then $p(s,t) \leftarrow \{s,t\}$, return $p(s,t)$;
for $(w, h(s)) \in \mathsf{L^{MLL}}(s)$ do
    $dist(s,w), dist(t,w) \leftarrow$ query by $\mathsf{L^{CTL}}$;
    if $dist(s,w) + dist(t,w) = dist(s,t)$ then break;
if $dist(s,w) = 1$ then $p(s,w) \leftarrow \{s,w\}$;
  else $p(s,w) \leftarrow$ Unfold($s$, $w$, $h(s)$);
$p(w,t) \leftarrow$ Algorithm 6$(w,t)$;
return $p(s,w) + p(w,t)$

Procedure Unfold($u$, $v$, $x$)
    $dist(u,x), dist(x,v) \leftarrow$ query by $\mathsf{L^{CTL}}$;
    if $dist(u,x) = 1$ then $p(u,x) \leftarrow \{u,x\}$;
    else $p(u,x) \leftarrow$ Unfold($u$, $x$, $h(x)$), where $(u, h(x)) \in \mathsf{L^{MLL}}(x)$;
    if $dist(x,v) = 1$ then $p(x,v) \leftarrow \{x,v\}$;
    else $p(x,v) \leftarrow$ Unfold($x$, $v$, $h(x)$), where $(v, h(x)) \in \mathsf{L^{MLL}}(x)$;
return $p(u,x) + p(x,v)$
```

order. Then, by $h(x)$, $p(u,x)$ is decomposed into $p(u,h(x))$ and $p(h(x),x)$. By the induction hypothesis, $p(u,h(x))$ and $p(h(x),x)$ can be found, and $p(u,x) = p(u,h(x)) + p(h(x),x)$ can also be found correctly. □

Lemma 16. *Procedure* Unfold$(u,v,x = h(u))$ *requires* $O(dist(u,v))$ *shortest-distance queries to return the u-v shortest path* $p(u,v)$ *when* $(v,h(u)) \in \mathsf{L^{MLL}}(u)$.

Proof. To return $p(u,v)$, Unfold(u,v,x) is called $O(dist(u,v))$ times; each call incurs a constant number of distance queries. □

Case 2: General Case. If not all shortest paths between two vertices s and t are monotonic, then we cannot use Procedure Unfold to find the s-t shortest path $p(s,t)$: t is not a landmark of s (assume $r(s) \leq r(t)$). To handle this case, we resort to Lemma 14, which states that any shortest path can be decomposed into several monotonic shortest (sub)paths. When $p(s,t)$ is broken down into monotonic shortest subpaths, Procedure Unfold can find each of them. By splicing these subpaths, we can find out $p(s,t)$.

Algorithm 6 describes how to answer $\mathsf{Q_P}(s,t)$ in a general case. We swap s and t to make $r(s) \leq r(t)$ (Line 1). Then we run a distance query using $\mathsf{L^{CTL}}$ to get $dist(s,t)$ (Line 2). If $dist(s,t)$ is 0 or 1, we can return the path $p(s,t)$ directly (Line 3-4). Otherwise, we enumerate all landmarks in $\mathsf{L^{MLL}}(s)$ and find the one w that is on $p(s,t)$ (Line 5-7). If $dist(s,w) = 1$, we set the edge $\{s,w\}$ as $p(s,w)$ directly (Line 8); otherwise, since w is a landmark of s, the s-w shortest path $p(s,w)$ can be discovered by Procedure Unfold (because all s-w shortest paths are monotonic) (Line 9). Here, $p(s,w)$ is the first monotonic shortest subpath of $p(s,t)$, we then set w as s to continue with Algorithm 6, until all monotonic shortest subpaths of $p(s,t)$ are found (Line 10-11).

Example 32. *Consider the graph G in Fig. 4.1. When answering $\mathsf{Q_P}(v_6,v_4)$ that finds the v_6-v_4 shortest path $p(v_6,v_4)$, we first find the landmark v_3 from the label of $s = v_6$, which is on $p(v_6,v_4)$. Since $dist(v_6,v_3) > 1$, we call $\mathsf{Unfold}(v_6,v_3,h(v_6) = v_8)$ to find the v_6-v_3 shortest path $p(v_6,v_3) = \{v_6,v_8,v_3\}$. Then we set $s = v_3$ and since $dist(v_3,v_4) = 1$, we find $p(v_3,v_4) = \{v_3,v_4\}$ directly. Splicing $p(v_6,v_3)$ with $p(v_3,v_4)$ yields $p(v_6,v_4) = \{v_6,v_8,v_3,v_4\}$.*

Theorem 11. *Algorithm 6 correctly answers the query $\mathsf{Q_P}(s,t)$.*

Proof. We use induction on the number of monotonic shortest subpaths contained in $p(s,t)$. If all s-t shortest paths are monotonic, $t \in \mathsf{L^{MLL}}(s)$ by Definition 6, $p(s,t)$ is found correctly by Lemma 15. Assume Algorithm 6 correctly answers the shortest-path query for vertices s,v containing $k-1$ monotonic shortest subpaths. If $p(s,t)$ contains k monotonic shortest subpaths, we denote the last monotonic subpath of $p(s,t)$ as $\tilde{p}(v,t)$. As all v-t shortest paths are monotonic, $t \in \mathsf{L^{MLL}}(v)$ and $\tilde{p}(v,t)$ is found correctly by Lemma 15. Splicing $p(s,v)$ and $\tilde{p}(v,t)$ yields $p(s,t)$. □

Lemma 17. *Algorithm 6 requires* $O(\Sigma_{v \in p(s,t)} |\mathsf{L}^{\mathsf{MLL}}(v)|)$ *shortest-distance queries to answer the query* $\mathsf{Q_P}(s,t)$.

Proof. The worst time occurs when each vertex in $p(s,t)$ requires checking its label to find landmark w on $p(s,t)$, incurring $O(\Sigma_{v \in p(s,t)} |\mathsf{L}^{\mathsf{MLL}}(v)|)$ shortest-distance queries. □

Lemma 18. $\mathsf{L}^{\mathsf{MLL}}$ *is minimal for correct query processing.*

Proof. Suppose we remove a landmark v from an arbitrary vertex u, then $\mathsf{Q_P}(u,v)$ cannot be answered. Otherwise, if Algorithm 6 can answer $\mathsf{Q_P}(u,v)$, there is a landmark $w \neq v$ of u on u-v shortest paths, and $r(w) > r(u)$. But v is the landmark of u implies that all u-v shortest paths are monotonic, i.e., all inner vertices (including w) on u-v shortest paths are lower in order than u, contradiction. □

Remark. The best-case scenario for MLL's query time is similar to extending PLL and CTL using extra attributes, so MLL's query speed will be slightly inferior. However, MLL's sacrifices in query time allow us to use less space cost to support queries. Moreover, the query time of MLL (Algorithm 6) is related to the length of the shortest path (bounded by the graph diameter D) and the label size of MLL. For complex networks (used in this chapter), since both the diameter D (mostly under 50) and the average label size (all less than 150) are small, the query time for MLL is still fast in practice — shortest-path queries using MLL on all graphs can be completed in less than 2 milliseconds.

4.4.3 Index Construction

We create the MLL index $\mathsf{L}^{\mathsf{MLL}}$ based on the label condition given by Theorem 8, which states that vertex v is added to the label of vertex u as the landmark if

Algorithm 7: MLL Index Construction

```
Input: Graph $G(V, E)$, Index $\mathsf{L}^{\mathsf{CTL}}$
Output: The index $\mathsf{L}^{\mathsf{MLL}}$
for each vertex $v \in V$ in parallel do
    $Q \leftarrow$ a queue with only vertex $v$;
    $dist(v, v) \leftarrow 0$ and $dist(v, u) \leftarrow \infty, \forall u \in V \setminus \{v\}$;
    $h(u) \leftarrow -, \forall u \in V$;
    while $Q \neq \varnothing$ do
        $u \leftarrow Q.pop()$;
        if $r(u) > r(v)$ then continue;
        // Check if all $u$-$v$ shortest paths are monotonic
        if Check($u$, $v$, $dist(v,u)$) then
            Insert $(v, h(u))$ into $\mathsf{L}^{\mathsf{MLL}}(u)$;
        for $w \in N(u)$ do
            if $dist(v, w) = \infty$ then
                $dist(v, w) \leftarrow dist(v, u) + 1$; $Q.push(w)$;
            if $dist(v, w) = dist(v, u) + 1$ and $dist(v, w) > 1$ then
                $h(w) \leftarrow \text{argmax}_{x \in \{u, h(u), h(w)\}} r(x)$;
return $\mathsf{L}^{\mathsf{MLL}}$

Procedure Check($u$, $v$, $d$)
    if $u \in C$ then $\mathsf{L}(u) \leftarrow$ the core label of $x$ from the CTL index;
    if $u \notin C$ then $\mathsf{L}(u) \leftarrow X(u)$;
    if $v \notin \mathsf{L}(u)$ then return False;
    for $w \in \mathsf{L}(u) \setminus \{u, v\}$ do
        $dist(u, w), dist(w, v) \leftarrow$ query by $\mathsf{L}^{\mathsf{CTL}}$;
        if $d = dist(u, w) + dist(w, v)$ return False;
    return True;
```

all v-u shortest paths are monotonic and $r(v) > r(u)$. If we can check whether all v-u shortest paths are monotonic, then indexing vertex v becomes a process of adding v to lower-order vertices u. This process can be completed using a v-sourced BFS. Since there is no dependency between the BFSs of different vertices, the indexing process of all vertices can be executed in parallel. Once all vertices complete BFS for the indexing process, $\mathsf{L}^{\mathsf{MLL}}$ is created.

Indexing Algorithm. Algorithm 7 describes how to create $\mathsf{L}^{\mathsf{MLL}}$ in parallel for each vertex v. First, each vertex v is inserted into the queue Q, and a v-sourced

BFS begins (Line 2). We initialize the distances from v to all vertices, and set the highest-order inner vertex $h(u)$ on all v-u shortest paths as nil (denoted by "$-$") (Line 3-4). Then, we pop an element u from Q (Line 6). If the order of u is higher than v, then the expansion from u is pruned (Line 7). Otherwise, we need to check if all v-u shortest paths are monotonic. We use Procedure $\mathsf{Check}(u, v, dist(v, u))$ for this purpose, which will be described later. If True, v is added as a landmark to the label of u (Line 8-9). For each unvisited neighbor vertex $w \in N(u)$ of u, we update $dist(v, w)$ and add w to Q (Line 11-12). If $dist(v, w) + 1 = dist(v, u)$, we set $h(w)$ to the one x with the highest order in u, $h(u)$, and $h(w)$, i.e., $\text{argmax}_{x\in\{u,h(u),h(w)\}} r(x)$ (Line 15-16).

Example 33. *Consider the graph G in Fig. 4.1. We run in parallel to index each vertex in G. Taking v_3 as an example. v_3 first adds itself to Q. Then, v_3 is popped from Q and we push v_3's neighbors $N(v_3) = \{v_1, v_2, v_4, v_8, v_9\}$ into Q. Next, v_1 and v_2 is popped and pruned. Then, v_4 is popped, and v_3 is added as a landmark of v_4 since $r(v_4) < r(v_3)$ and all v_4-v_3 shortest paths are monotonic. Next, v_8 is popped, and v_3 is added as a landmark of v_8; meanwhile, we insert $v_6 \in N(v_8)$ to Q and update $h(v_6) = argmax_{x\in\{v_8,h(v_8)=-,h(v_6)=-\}} r(x) = v_8$. After Q becomes empty, we stop.*

Procedure $\mathsf{Check}(u, v, d)$. Next, we introduce Procedure $\mathsf{Check}(u, v, d)$ (Line 16-23), where $r(v) > r(u)$, $d = dist(v, u)$. The purpose is to examine if all v-u shortest paths are monotonic. A simple way is to enumerate all u-v shortest paths, and then compare the order of each inner vertex with u. However, this approach is inefficient, and we instead use the CTL index to speed up the checking. Given the core-tree decomposition T_G (under parameter d) of graph G, CTL uses PLL to assign a core label to each vertex of the core part C to form the core index.

The CTL index is sufficient to check if all v-u shortest paths are monotonic.

We first determine whether vertex u belongs to the core part C, and if so, we set $\mathsf{L}(u)$ as the core label of u from the CTL index (Line 17). If u does not belong to C, we find the corresponding tree node $X(u)$ in T_G and assign $X(u)$ to $\mathsf{L}(u)$ (Line 18). If v is not in $\mathsf{L}(u)$, we return False, i.e., not all v-u shortest paths are monotonic (Line 19). Otherwise, we enumerate the vertices w in $\mathsf{L}(u)$, where $w \neq u, w \neq v$, and obtain distances $dist(u, w)$ and $dist(w, v)$ by querying the CTL index (Line 20-21). If there is a vertex $w \in \mathsf{L}(u) \setminus \{u, v\}$ on the u-v shortest path (i.e., $dist(u, w) + dist(w, v) = dist(u, v)$), then False is returned (Line 22); otherwise True is returned (Line 23).

Example 34. *Consider the graph G in Fig. 4.1, where the core-tree decomposition ($d = 2$) is in Fig. 4.2. For $\mathsf{Check}(v_8, v_4, 2)$, since $v_8 \notin C$, $X(v_8) = \{v_6, v_3\}$ is assigned to $\mathsf{L}(v_8)$. As $v_4 \notin \mathsf{L}(v_8)$, we return False. For $\mathsf{Check}(v_2, v_1, 1)$, since $v_2 \in C$, the label $\{(v_1, 0), (v_2, 0)\}$ of v_2 from the core index is assigned to $\mathsf{L}(v_2)$. As $v_1 \in \mathsf{L}(v_2)$ and $\not\exists w \in \mathsf{L}(v_2) \setminus \{v_1, v_2\}$ on the v_1-v_2 shortest path, True is returned.*

Lemma 19. *$\mathsf{Check}(u, v, d)$ correctly checks if all u-v shortest paths are monotonic ($r(v) > r(u)$).*

Proof. If $u \in C$, then $\mathsf{L}(u)$ is the core label assigned to u (by PLL). If $v \notin \mathsf{L}(u)$, by Theorem 7, there exists a landmark $w \in \mathsf{L}(u) \cap \mathsf{L}(v)$ on the u-v shortest path as an inner vertex and $r(w) > r(v) > r(u)$. So u-v shortest paths are not monotonic. If $v \in \mathsf{L}(u)$ and there exists an inner vertex on the u-v shortest path as a landmark w in $\mathsf{L}(u)$, then by Theorem 7, $r(u) < r(w)$, so the u-v shortest path is not monotonic. If $v \in \mathsf{L}(u)$ but there is no landmark $w \in \mathsf{L}(u) \setminus \{u, v\}$ on the u-v shortest path, we show that all v-u shortest paths are monotonic, since otherwise we can always find the closest inner vertex w to u on the v-u shortest path with $r(w) > r(u)$. By Theorem 7, $w \in \mathsf{L}(u)$, contradiction.

If $u \notin C$, then $\mathsf{L}(u) = X(u)$. If $v \notin X(u)$, $X(u) \setminus u$ is a cut of u and v [21], i.e., it contains an inner vertex w on the u-v shortest path. The fact $w \in X(u)$ implies $r(w) > r(u)$ [13]. So not all v-u shortest paths are monotonic. If $v \in X(u)$ and there exists an inner vertex $w \in X(u)$ on the u-v shortest path, then the v-u shortest path is not monotonic (as $r(w) > r(u)$). If $v \in X(u)$ but there is no inner vertices $w \in X(u)$ on the u-v shortest path, we show that all v-u shortest paths are monotonic, since otherwise we can find the closest inner vertex w to u on the v-u shortest path with $r(w) > r(u)$. $w \in X(u)$ when we use MDE [13], contradiction. □

Lemma 20. *Algorithm 7 correctly creates the* MLL *index.*

Proof. If the index created by Algorithm 7 is L_1 and the index defined in Definition 12 is L_2, we prove $\mathsf{L}_1(u) = \mathsf{L}_2(u)$, for $\forall u \in V$.

We first prove that $\mathsf{L}_1(u) \subseteq \mathsf{L}_2(u)$. Otherwise, suppose there is a landmark v, $v \in \mathsf{L}_1(u)$ but $v \notin \mathsf{L}_2(u)$. $v \notin \mathsf{L}_2(u)$ implies that (1) there is an inner vertex $w = h(u)$ on the v-u shortest path, which is of order higher than v, then by Line 7 of Algorithm 7, $v \notin \mathsf{L}_1(u)$, contradiction; (2) not all v-u shortest paths are monotonic (by Lemma 19, we can correctly identify this case), then by Line 8 of Algorithm 7, $v \notin \mathsf{L}_1(u)$, contradiction.

We next prove that $\mathsf{L}_2(u) \subseteq \mathsf{L}_1(u)$. Otherwise, suppose there is a landmark v, $v \in \mathsf{L}_2(u)$ but $v \notin \mathsf{L}_1(u)$. $v \notin \mathsf{L}_1(u)$ implies that (1) there is an inner vertex $h(u)$ of order higher than v, then v is not the highest-order vertex on all u-v paths, and by Definition 12, $v \notin \mathsf{L}_2(u)$, contradiction. (2) not all v-u shortest paths are monotonic, and by Definition 12, $v \notin \mathsf{L}_2(u)$, contradiction. □

Lemma 21. *Algorithm 7 requires* $O(\Delta^{\mathsf{CTL}} \times |\mathsf{L}^{\mathsf{CTL}}|)$ *shortest-distance queries, where* Δ^{CTL} *is* CTL*'s maximum label size.*

Proof. Each time v visits a vertex u, we need to call `Check` once. If $v \in \mathsf{L}(u)$

($\mathsf{L}(u)$ is u's core label or $X(u)$), we need to check if there is a vertex in $\mathsf{L}(u)$ on the shortest path from u to v, this needs at most Δ^{CTL} shortest-distance queries. Due to the if condition of Line 19, Algorithm 7 will reach Line 20 $|\mathsf{L}^{\mathsf{CTL}}|$ times. Hence, Algorithm 7 requires $O(\Delta^{\mathsf{CTL}} \times |\mathsf{L}^{\mathsf{CTL}}|)$ shortest-distance queries. □

4.5 Extension of MLL

In this section, we generalize our method MLL to weighted and directed graphs.

Extension for Weighted Graphs. MLL relies on the CTL index, and both the tree index (refer to the tree decomposition algorithm on the weighted graph [68]) and the core index (refer to the PLL algorithm on the weighted graph [7]) of CTL can be extended to the weighted graph. Then, we only need to discuss how to modify the MLL index construction and its query process for weighted graphs.

Index construction. In constructing MLL, Algorithm 7 uses the BFS algorithm, thus trying to add each vertex $v \in V$ to the vertices in the graph as labels. To generalize to weighted graphs, we use a similar algorithm to Algorithm 7; the main difference is that we use Dijkstra's algorithm instead of the BFS algorithm to complete the indexing process of each vertex v.

Query processing. The query algorithm for MLL (Algorithm 6) can be naturally generalized to weighted graphs. Specifically, we firstly remove Line 4 because for weighted graphs, a length equal to 1 does not mean the shortest path contains only one edge. Then for the conditions checking (i.e., "if $dist(\cdot) = 1$") in Line 8, 14, and 16, we use "if $h(\cdot) = -$" instead, which means that the monotonic path cannot be further unfolded (i.e., we find an edge in the graph).

Performance Analysis. Our method is based on CTL. For unweighted graphs, CTL uses the parallel BFSs to build index entries for each distance level by level, i.e., it builds the index entries with distance $d + 1$ after it finishes building the ones with distance d, and the index entries with distance $d+1$ rely on the ones with distance less than $d+1$. However, for weighted graphs, we cannot build the index in a similar way because the number of possible distances is large, and we do not know which distance of indexes to build. Therefore, there is no efficient way to parallelize CTL for weighted graphs. As a result, the performance of CTL for weighted graphs will be slightly slower than the single-threaded version of CTL for unweighted graphs because it requires the Dijkstra's algorithm instead of the BFS algorithm.

As for the MLL index entries, it does not have similar limitations as CTL. Therefore, it can be parallelized in a similar way as for unweighted graphs.

Extension for Directed Graphs. In Section 4.4, we assumed that the complex network is undirected for MLL. MLL can also be extended to support shortest-path queries on directed graphs. Given a directed graph $G(V, E)$, if $(u, v) \in E$, then u is an in-neighbor of v and v is an out-neighbor of u.

Index construction. MLL relies on the CTL index, but building the CTL index on directed graphs is challenging due to performing core-tree decomposition on directed graphs. There are two differences when *decomposing a directed graph* compared to an undirected graph. (1) We perform the decomposition using minimum degree elimination (MDE) [13], which iteratively finds the vertex v with the smallest degree (the total number of in-/out-neighbors) in a graph. In eliminating the vertex v, we connect *any* two neighbors u, w of v by directed edges, set the edge weight, and remove v using similar methods to that of undirected graphs; the other parts are the same. (2) For each vertex u in the tree node $X(v)$, we need to store two shortest distances: the forward distance $dist_+(u, v)$ from u to v and the backward distance $dist_-(u, v)$ from v to u.

When the graph is decomposed, we get a directed graph G_λ, and we can use PLL to create a forward label $\mathsf{L}_+^{\mathsf{CTL}}(u)$ for each vertex $u \in C$. Then, the direction of the edges of G_λ is reversed to get a **reverse graph**, and we create a backward label $\mathsf{L}_{-}^{\mathsf{CTL}}(u)$ for each $u \in C$ by using PLL on the reverse graph of G_λ. As for the tree index, for each landmark v of u, we calculate the distance from v to u as a forward label and the distance from u to v as a backward label. Similarly, for our MLL index, we use Algorithm 7 to create a forward label $\mathsf{L}_{+}^{\mathsf{MLL}}(u)$ for each vertex $u \in V$ on the original graph G and a backward label $\mathsf{L}_{-}^{\mathsf{MLL}}(u)$ on the reverse graph of G.

Query processing. To perform shortest-path queries on directed graphs, we can use a similar way as on undirected graphs (by Algorithm 6). One thing to note is that we need to consider whether to use forward or backward labels for each vertex. For example, when we need to obtain the shortest path from s to t, we should use the forward label $\mathsf{L}_{+}^{\mathsf{MLL}}(s)$ for s and the backward label $\mathsf{L}_{-}^{\mathsf{MLL}}(t)$ for t. Algorithm 6 also needs the support of shortest-distance queries using CTL, which can be adapted similarly on directed graphs.

4.6 Experiments

Algorithms. We intend to conduct thorough experimental comparisons of traversal-based and extension-based approaches for dealing with shortest-path queries on complex networks.

- Traversal-based: we select and implement four representative methods, all of which need graph traversal to process queries.
  - BFS. We start a BFS from s until t is met to process $\mathsf{Q_P}(s, t)$.
  - BiBFS. Bidirectional BFS, i.e., search the path from both the s and t sides to process $\mathsf{Q_P}(s, t)$. We use an implementation similar to [87] to incorporate some heuristics for speedup.

  - $\mathsf{PLL_B}$. This method uses both the index and graph traversal. To reduce the PLL index size, we construct a partial PLL by only constructing labels within a specific distance (≤ 5 in this chapter) and ignoring others with larger distance values. This partial PLL index is extended to support shortest-path queries. When dealing with $\mathsf{Q_P}(s, t)$, if the partial PLL index finds $dist(s, t) \leq 5$, then the s-t shortest path can be returned using Algorithm 5; otherwise, the query is handled by BiBFS.
  - $\mathsf{CTL_B}$. We use CTL as preprocessing to speed up BFS, that is, when processing $\mathsf{Q_P}(s, t)$, the CTL index can provide distance information to determine whether a vertex w is on the s-t shortest path, i.e., whether $dist(s, t) = dist(s, w) + dist(w, t)$. Vertices not on the s-t shortest path can be pruned directly.

- Extension-based: we select the extended PLL and CTL introduced in Section 4.3 and MLL introduced in Section 4.4, all of which use only indexes to process queries.

  - $\mathsf{PLL_E}$. We use the extended PLL index, i.e., add an extra attribute to each index entry, for query processing. The query method is in Algorithm 5. We use the parallel version of PLL to speed up index creation, whose source code is provided by the authors of [59].
  - $\mathsf{CTL_E}$. We use the extended CTL index, i.e., add an extra attribute to each index entry, for query processing. The query processing method is introduced in Section 4.3.2. The index construction of CTL can be accelerated using multiple cores. The source code of CTL is provided by the authors of [60].
  - MLL. MLL uses both the CTL index and the MLL index for query processing (see Algorithm 6). The index of MLL can be constructed in parallel,

Table 4.2: Dataset Description

	Datasets	n	m	Type	Diameter	$dist_{avg}$	Deg_{avg}
DELI	Delicious[5]	536,109	1,365,961	Social	14	5.16	5.10
DIGT	DIGT[5]	4,000,151	8,649,016	Social	15	7.81	4.32
FRIE	Friendster[5]	8,658,745	55,170,227	Social	25	5.37	12.74
STAC	Stack[5]	6,024,271	63,497,050	Interaction	11	3.86	21.08
LIVE	LiveJournal[6]	5,363,260	79,023,142	Social	20	5.45	29.47
FACE	Facebook[5]	58,790,783	92,208,195	Social	24	7.25	3.14
TWIT	Soc-Twitter[5]	21,297,772	265,025,809	Social	26	4.87	24.89
SK05	SK-2005[6]	50,636,154	1,949,412,601	Web	40	5.20	77.00
UK06	UK-2006[6]	77,741,046	2,965,197,340	Web	42	6.16	76.28
UK07	UK-2007[6]	133,633,040	5,507,679,822	Web	257	6.22	82.43

and the construction algorithm is presented in Algorithm 7.

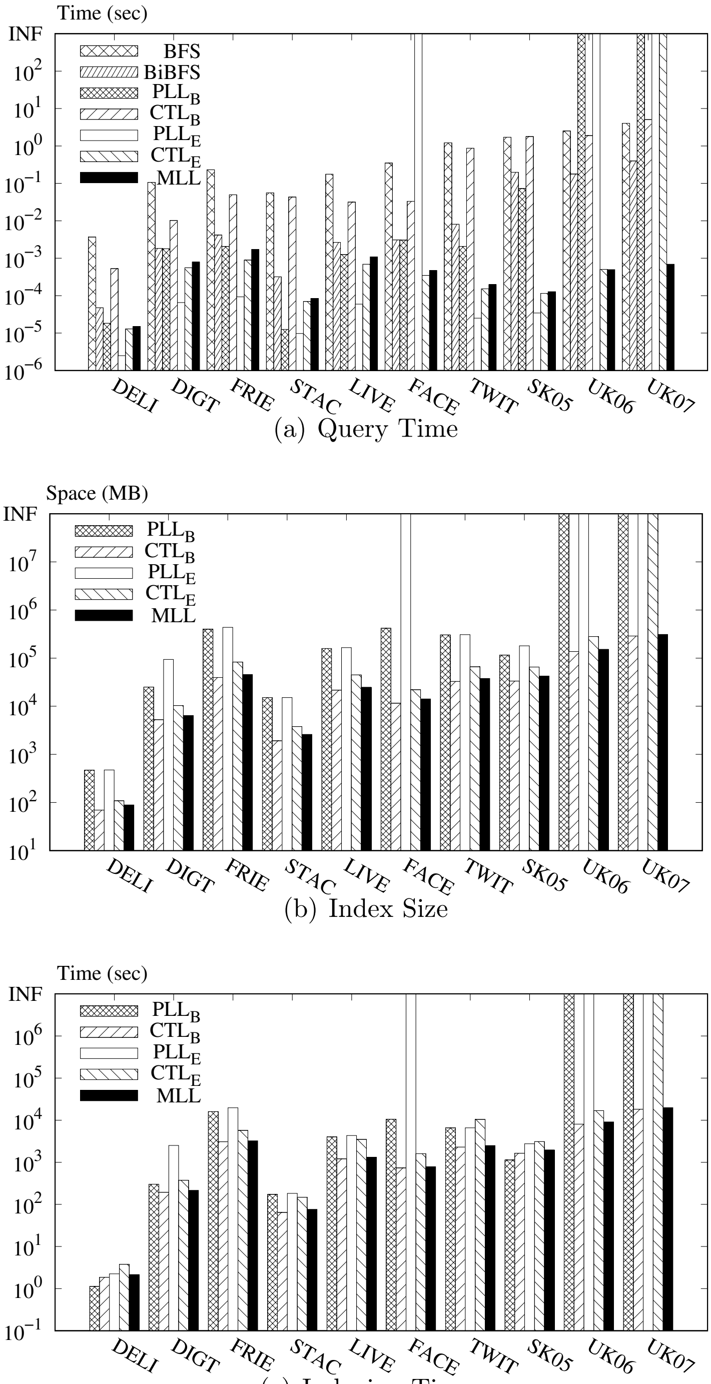
We implemented all the algorithms using C++ and compiled them using GNU GCC 4.8.5 and -O3 level optimizations. We use OpenMP to support the implementation of the parallel algorithms. All experiments were conducted on a machine with 64 CPU cores and 500 GB main memory running Linux (Red Hat Linux 4.8.5, 64bit). Each CPU core is Intel Xeon 2.4GHz.

Datasets. We ran experiments on 10 real-world graphs, whose details are given in Table 4.2. The largest graph has over 5.5 billion edges. These graphs are small-world graphs, most of which have diameters (longest shortest distances) less than 50, and the average distance between two vertices of all graphs, i.e., $dist_{avg}$, is less than 10. The average degree of these graphs, Deg_{avg}, varies between 3.14 and 82.43. The dataset comes from various complex networks, including social networks, web graphs, and interaction networks. All graphs were downloaded from Network Repository[5][74] and Laboratory for Web Algorithms[6][18].

Summary of Findings. For traversal-based methods (i.e., BFS, BiBFS, $\mathsf{PLL_B}$, and $\mathsf{CTL_B}$), BFS and BiBFS can process queries without building indexes, but their query speed is slow. $\mathsf{CTL_B}$ tries to use the CTL index to accelerate BFS, but it cannot guarantee query time. $\mathsf{PLL_B}$ can speed up the query process by

[5] https://networkrepository.com/networks.php

[6] https://vigna.di.unimi.it



(a) Query Time

(b) Index Size

(c) Indexing Time

Figure 4.3: The Comparison of Different Methods

creating a partial PLL index, but it still cannot guarantee query time for shortest-path queries with long distances. Thus, the traversal-based approaches are only applicable when the query speed is not so demanding while the space budget is low.

On the other hand, extension-based methods (i.e., PLL$_E$, CTL$_E$, and MLL) use the pre-computed index for query processing, and their query speed is much faster than traversal-based methods since they avoid graph traversal at query time. Moreover, the three extension-based methods make a different trade-off between query time and space cost: among them, MLL has the smallest index size and PLL^E has the largest index size, while the index size of MLL is in between; MLL has the slowest query speed and PLL^E has the fastest query speed, while the query speed of CTL is in between.

Ex-1: Query Time Comparison. We compare the query time of all methods. For the approach using indexes for query processing, we set the query time to "INF" if the index cannot be built. We generated 1000 random queries and obtained the average query processing time. We show the results in Fig. 4.3(a).

Comparison among extension-based methods. Since the extension-based methods (i.e., PLL$_E$, CTL$_E$, and MLL) do not rely on graph traversal, they can process queries very quickly: all of them handle shortest-path queries within two milliseconds. Among them, PLL$_E$ has the fastest query speed, and the query time of PLL$_E$ is on average 10.53 times shorter than that of MLL. The query speed of MLL is comparable to that of CTL$_E$, and the query time of CTL$_E$ is on average 1.94 times shorter than that of MLL. Considering the performance of extension-based methods in query processing, they are suitable for applications requiring high query speed.

Comparison with traversal-based methods. Due to the inevitable need for graph traversal, the query speed of traversal-based methods (i.e., BFS, BiBFS, PLL$_B$, and $\mathsf{CTL_B}$) is much slower than that of extension-based methods. We take MLL as a representative to compare extension-based methods with traversal-based methods.

(1) Comparison with BFS and BiBFS. The query time of BFS is, on average, 3265.86 times longer than MLL and up to four orders of magnitude longer than MLL. BiBFS uses bi-directional search to reduce the overhead of graph traversal compared to BFS, but BiBFS still takes a long time to process queries: BiBFS is on average 254 times and up to three orders of magnitude slower than MLL.

(2) Comparison with $\mathsf{PLL_B}$. When processing a query, if the distance between two vertices is short, $\mathsf{PLL_B}$ can use the index to avoid traversing the graph. However, $\mathsf{PLL_B}$ cannot totally avoid graph traversal, which leads to the query time of $\mathsf{PLL_B}$ being 102.46 times longer than that of MLL on average.

(3) Comparison with $\mathsf{CTL_B}$. $\mathsf{CTL_B}$ narrows the search space of BFS by distance queries, so $\mathsf{CTL_B}$ is faster than BFS on some graphs; for example, on DELI, the query time of $\mathsf{CTL_B}$ is 0.14 times that of BFS. But distance queries used by $\mathsf{CTL_B}$ are not free; for example, on UK07, $\mathsf{CTL_B}$ takes 1.25 times longer than BFS. $\mathsf{CTL_B}$ is also much slower than MLL: MLL is on average 3027.45 times and at most four orders of magnitude faster than $\mathsf{CTL_B}$.

Ex-2: Index Size Comparison. There are five methods that require the use of indexes (including $\mathsf{PLL_B}$, $\mathsf{CTL_B}$, $\mathsf{PLL_E}$, $\mathsf{CTL_E}$, and MLL) for query processing. We compare the index size of these five methods and present the results in Fig. 4.3(b).

Index Size of $\mathsf{PLL_E}$*,* $\mathsf{CTL_E}$*, and* MLL. Among extension-based methods, $\mathsf{PLL_E}$ has the largest index, while MLL has the smallest index.

(1) The total size of the indexes (including the CTL and the MLL indexes) used by MLL is 6.9 times smaller than the size of indexes used by $\mathsf{PLL_E}$. Because of the oversized indexes, $\mathsf{PLL_E}$ cannot handle large graphs such as FACE, TWIT,

and UK07.

(2) Both MLL and CTL$_E$ are extended based on CTL to support path queries. For CTL$_E$, the extra space brought by the extension is 0.96 times that of the original CTL index, while the size of the extra space (i.e., the MLL index) required by MLL is 0.2 times that of the original CTL index. Also, due to the difference in the extra space used, CTL$_E$ cannot handle graph UK07 while MLL can.

Index Size of PLL$_B$. By limiting the distance in the labels, PLL$_B$ constructs a partial PLL index. Thus, the index size of PLL$_B$ is 0.82 times that of PLL$_E$. However, the index size of PLL$_B$ is on average 8.05 times[7] that of MLL, and PLL$_B$ cannot handle large graphs such as FACE and UK07. This shows that even building a partial PLL index still requires a much larger index size than MLL.

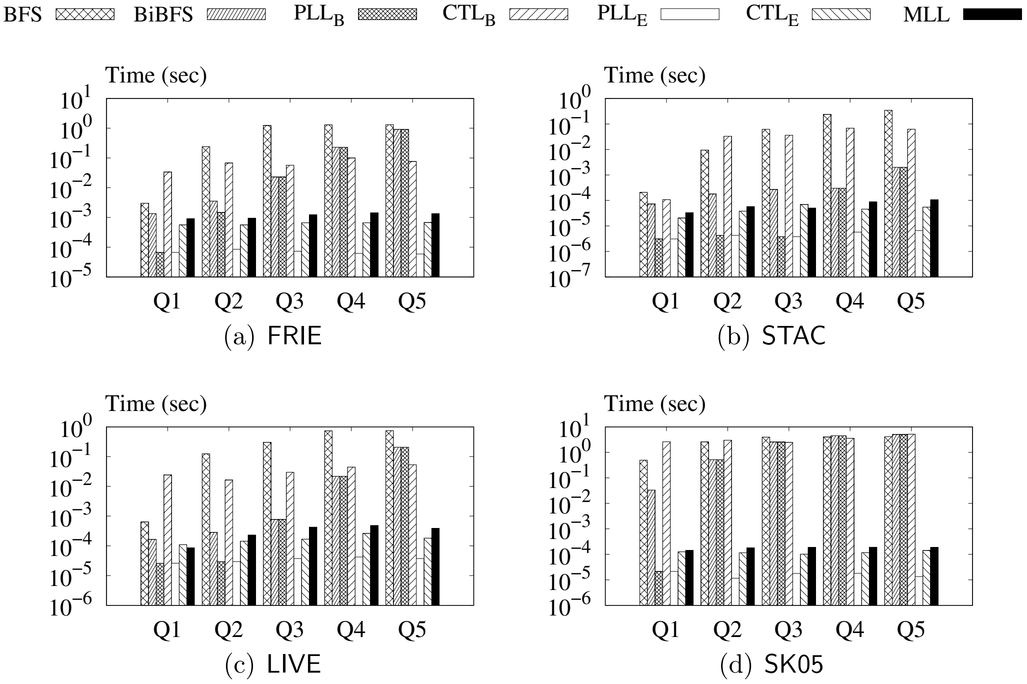
Index Size of CTL$_B$. Instead of extending the CTL index, CTL$_B$ uses the original CTL index for distance queries to reduce the search range of BFS. CTL$_B$ has a smaller index size than the extension-based approaches. However, the total index size of MLL is only 1.2 times that of CTL$_B$, indicating that MLL does not add significantly extra space to the original CTL index.

Ex-3: Indexing Time Comparison. We compare the indexing time of five methods (including PLL$_B$, CTL$_B$, PLL$_E$, CTL$_E$, and MLL) that require the use of indexes for query processing. We show the results in Fig. 4.3(c).

Indexing Time of PLL$_E$, CTL$_E$, and MLL. Among three extension-based methods, MLL has the shortest indexing time: the total indexing time of MLL (including the time to build the CTL index and the MLL index) is on average 4.06 times shorter than that of PLL^E, and also on average 2.15 times shorter than that of CTL^E.

Indexing Time of PLL$_B$. On the graphs that PLL$_E$ can index, PLL$_B$ is 2.44 times faster than PLL$_E$: PLL$_B$ only needs to build a partial PLL index while PLL$_E$ needs

[7] There is no conflict with former results as PLL$_B$ can index FACE while PLL$_E$ cannot.



(a) FRIE (b) STAC

(c) LIVE (d) SK05

Figure 4.4: The Test of the Query Time at Different Distance Ranges

a complete one. However, the indexing time of $\mathsf{PLL_B}$ is on average 1.72 times and 3.6 times longer than that of $\mathsf{CTL_E}$ and MLL, respectively.

Indexing Time of $\mathsf{CTL_B}$. $\mathsf{CTL_B}$ only needs to build the CTL index, while $\mathsf{CTL_E}$ needs to add an extra attribute to each entry of the CTL index, which causes the indexing time of $\mathsf{CTL_E}$ to be 2.41 times longer than that of $\mathsf{CTL_B}$. MLL also needs to create the CTL index first. Still, the additional building of lightweight indexes results in the indexing time of MLL being only 1.12 times that of $\mathsf{CTL_B}$, indicating that MLL does not incur much additional indexing time cost to support shortest-path queries.

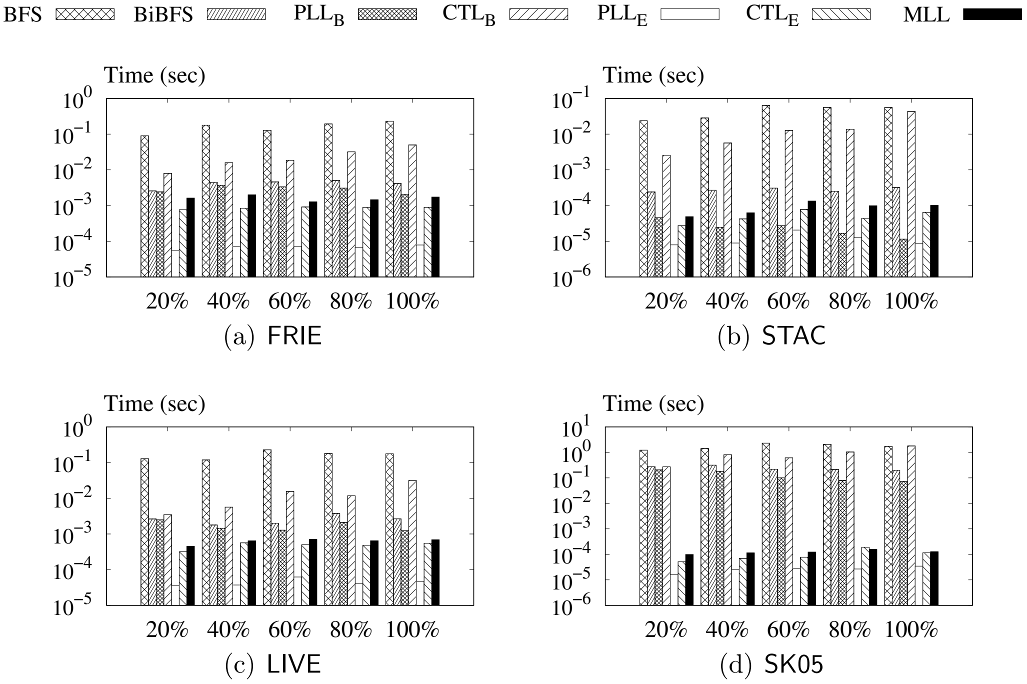
Ex-4: Test of Query Time at Different Distance Ranges. We test the performance of all methods in handling queries in different distance ranges. We randomly generate five sets of queries $Q = \{Q_1, Q_2, Q_3, Q_4, Q_5\}$, where each set $Q_i \in Q$, $i \in [1, 5]$, consists of 1000 random queries. For each query $\mathsf{Q_P}(s, t) \in Q_i$, we control the shortest distance between s and t located in the range between $\frac{D}{5} \times (i-1)$ and $\frac{D}{5} \times i$, where D is the diameter of the graph. We report the average time for answering queries in each set Q_i. Since various graphs have similar conclusions, we only show the results for FRIE, STAC, LIVE, and SK05 in Fig. 4.4.

Effect of distance on query time. For all methods, the time to answer queries in Q_1 tends to be shorter, while the time to answer queries in Q_5 tends to be longer. For example, on graph FRIE, MLL takes 1.48 times longer to process queries in Q_5 than in Q_1; BFS takes 433.77 times longer to process queries in Q_5 than in Q_1. One reason for this trend is that a longer path always means examining more labels or visiting more graph vertices to find the path. It is worth noting that the query time for $\mathsf{PLL_B}$ increases dramatically as the query distance increases. This is because $\mathsf{PLL_B}$ can use indexes to answer queries when the distance is less than a certain value (we set it to 5), whereas graph traversal is required for larger distances.

Comparison of traversal-based and extension-based Methods. The extension-based methods are faster than the traversal-based methods in processing queries in any Q_i on all graphs. Taking MLL as an example, MLL is on average 1511.34, 44.95, 44.95 and 91.75 times faster than BFS, BiBFS, $\mathsf{PLL_B}$ and $\mathsf{CTL_B}$ in processing queries in Q_4 of LIVE.

Ex-5: Scalability Test on Query Time. We test the scalability of all methods. To do this, we randomly divide the edges E of graph $G(V, E)$ into five groups with equal size and then generate five test graphs such that the i-th test graph contains the first i groups of edges. Thus, the five test graphs include $20\%, 40\%, 60\%, 80\%, 100\%$ of the edges in G, respectively. We conduct experiments on each of the five test graphs.

We first evaluate the graph size against the query time of all methods. Since



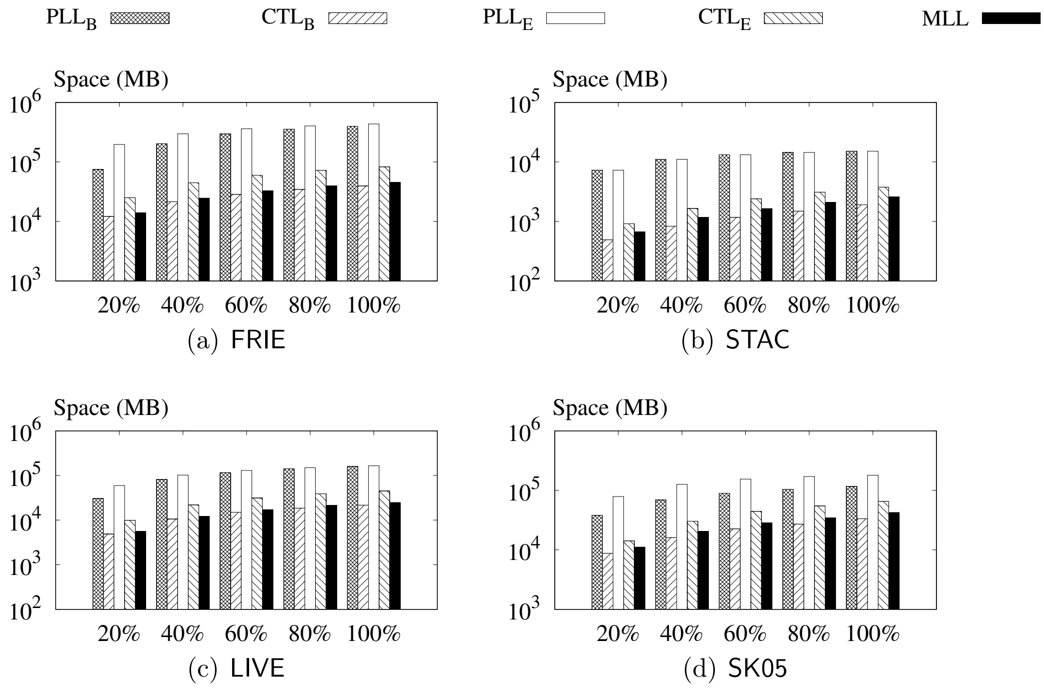
(a) FRIE (b) STAC

(c) LIVE (d) SK05

Figure 4.5: The Test of Scalability on the Query Time

various graphs have similar conclusions, we only present the results for FRIE, STAC, LIVE, and SK05 in Fig. 4.5. We find that, as the graph size increases, the query time of some methods shows an increasing trend. For example, on STAC, for CTL_B, the query time on the test graphs containing 40%, 60%, 80% and 100% edges is 2.2, 4.94, 5.33 and 16.77 times longer than the query time on the test graph with 20% edges. Yet, there is not just an upward trend observed; for example, on STAC, for MLL, the query time on the test graph containing 60% edges is 1.76 times longer than that on the test graph with 80% edges.

The reasons for the query time fluctuations are manifold. First, most of the query complexity is related to the graph scale, and a large graph generally implies an increase in complexity; however, query processing is also related to other factors, such as graph density and graph diameter. According to [56], the graph diameter decreases as the number of edges increases. Thus, the query time



(a) FRIE (b) STAC

(c) LIVE (d) SK05

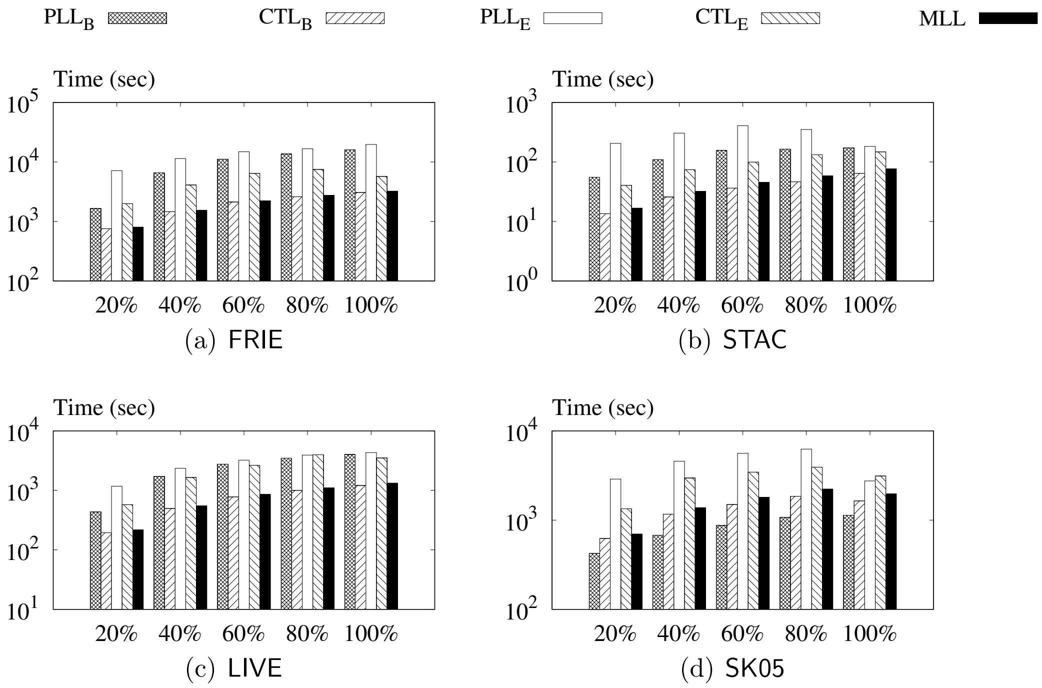
Figure 4.6: The Test of Scalability on the Index Size

shows a fluctuating trend under the interaction of various factors.

Ex-6: Scalability Test on Index Size. We investigate the effect of the graph size on the index size. We use the same experimental setup as Ex-5 and compare the index size of the five methods (including PLL$_B$, CTL$_B$, PLL$_E$, CTL$_E$, and MLL) that use indexes for query processing. The results are given in Fig. 4.5.

We find that the index size of all the five methods increases as the graph size grows. For example, on FRIE, the index built by MLL on the test graph containing 40% edges is 1.75 times larger than the index built on the test graph containing 20% edges, while the index built by MLL on the test graph with 100% edges is 3.23 times larger than the index built on the test graph with 20% edges.

Ex-7: Scalability Test on Indexing Time. We next study the effect of the graph size on the indexing time of the five methods (including PLL$_B$, CTL$_B$, PLL$_E$, CTL$_E$, and MLL) that rely on indexes for query processing. The experiments use



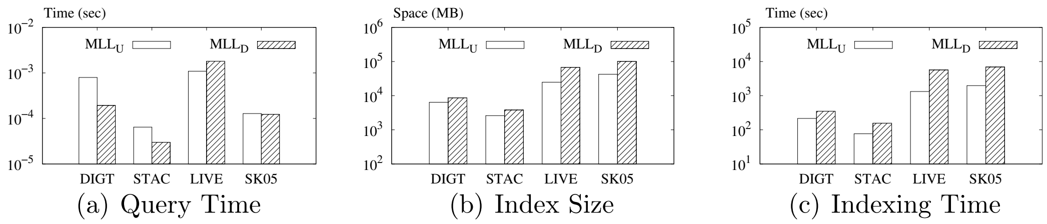
(a) FRIE (b) STAC

(c) LIVE (d) SK05

Figure 4.7: The Test of Scalability on the Indexing Time

the same settings as Ex-5, and we report the results in Fig. 4.7.

It can be found from Fig. 4.7 that the indexing time of all the five methods increases as the graph size increases. Taking MLL as an example, on graph FRIE, the indexing time of MLL is 1.92, 2.77, 3.41, and 4.02 times longer on test graphs containing 40%, 60%, 80%, and 100% edges, respectively, than on the test graph containing 20% edges. Similar phenomena can be observed in other methods.



(a) Query Time (b) Index Size (c) Indexing Time

Figure 4.8: The Performance of MLL on Directed Graphs

Ex-8: Test of MLL on Directed Graphs. Section 4.5 introduces how to extend MLL to directed graphs. To test the effectiveness of the extension, we run experiments on four real datasets (DIGT, STAC, LIVE, and SK05). Note that these graphs used are directed, while in the previous experiments, we ignored edge directions to create undirected graphs. The original MLL method is called $\mathsf{MLL_U}$ since it works with undirected graphs; the extended MLL method is called $\mathsf{MLL_D}$ since it works with directed graphs. The results are in Fig. 4.8.

Query time. The query time of $\mathsf{MLL_D}$ is normally faster than $\mathsf{MLL_U}$, for example, on DIGT, the average query time of $\mathsf{MLL_D}$ is 4.13 times shorter than that of $\mathsf{MLL_U}$. One reason could be that when querying on directed graphs, we only use the labels in one direction (and ignore the labels in the opposite direction). However, due to the randomness of queries and the larger index size, $\mathsf{MLL_D}$ may be slower than $\mathsf{MLL_U}$ in some cases; for example, on LIVE, the average query time of $\mathsf{MLL_D}$ is 1.66 times longer than $\mathsf{MLL_U}$.

Index size. The index size of $\mathsf{MLL_D}$ is generally larger than that of $\mathsf{MLL_U}$: the average index size of $\mathsf{MLL_D}$ is 1.98 times larger than that of $\mathsf{MLL_U}$. One possible explanation for this result is that forming a path in a directed graph is more difficult (a path in a directed graph must be a path in the corresponding version of an undirected graph, and the reverse does not hold). When indexing a directed graph, pruning the index using existing path information is more difficult, resulting in a large index size.

Indexing time. The indexing time for $\mathsf{MLL_D}$ is generally longer than that for $\mathsf{MLL_U}$: on the four graphs used, the indexing time of $\mathsf{MLL_D}$ is on average 2.87 times longer than that of $\mathsf{MLL_U}$.

4.7 Chapter Summary

This chapter studies shortest-path queries on complex networks. The distance query processing methods PLL and CTL are extended to support shortest-path queries. To reduce the space cost required for extensions, MLL is proposed. MLL is also adapted for weighted and directed graphs. Extensive experiments are conducted to investigate the performance of various methods in answering shortest-path queries. The experimental results can help practitioners choose the appropriate method for a specific application.

Chapter 5

LABEL CONSTRAINED SHORTEST PATH ON ROAD NETWORKS

5.1 Chapter Overview

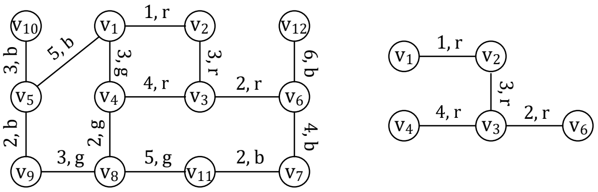
In this chapter, we study the label-constrained shortest-path query on road networks. This chapter is structured as follows. Section 5.2 provides the problem definition. Section 5.3 introduces the state-of-the-art algorithm. Section 5.4 shows a naive index-based approach. Section 5.5 further improves index structure. Section 5.6 optimizes the index construction procedure through parallelization. Section 5.7 evaluates the proposed algorithms and Section 5.9 concludes this chapter.

Table 5.1: List of Notations

Notation	Description
$G = (V, E)$	graph G with vertex set V and edge set E
$\phi(\cdot)$, $\ell(\cdot)$	weight and label of an edge/path
Σ	alphabet of edge labels
$G[\Sigma_s]$	Σ_s-induced subgraph of G
$\mathsf{dist}^{\mathcal{L}}_{G}(s, t)$	label-constrained shortest distance
T_G	tree decomposition of G
X, $X(v)$	tree node
$\omega(T_G)/\omega, h(T_G)/h$	treewidth and treeheight of T_G
$\mathcal{S}$, $\mathcal{S}(u, v)$	label-constrained shortest distance set(LSDS)
ρ	the maximum size of LSDS
$\mathcal{G}_i$	label-constrained distance preserved graph

5.2 Preliminaries

Let $G = (V, E, \phi, \ell, \Sigma)$ be a labelled road network, where $V(G)$ is a set of vertices, $E(G)$ is a set of edges, $\phi : E(G) \to \mathbb{R}^+$ is a function that assigns each edge $e \in E(G)$ a positive number $\phi(e, G)$ as its weight, Σ is a finite alphabet of edge labels, and $\ell : E(G) \to \Sigma$ is a function assigns each edge $e \in E(G)$ a label $\ell(e, G) \in \Sigma$. We use $n = |V(G)|$ (resp. $m = |E(G)|$) to denote the number of vertices (resp. edges) in G. For each vertex $v \in V(G)$, the neighbors of v, denoted by $\mathsf{nbr}(v, G)$, is defined as $\mathsf{nbr}(v, G) = \{u|(u, v) \in E(G)\}$. The degree of a vertex v is the number of neighbors of v. Given a subset of labels $\Sigma_s \subseteq \Sigma$, the Σ_s-induced subgraph of G, denoted by $G[\Sigma_s]$, is the subgraph that contains all edges in G with labels in Σ_s. A path p in G is a sequence of vertices $p = (v_0, v_1, v_2 ... v_k)$, where $(v_i, v_{i+1}) \in E(G)$ for each $0 \leq i \leq k - 1$. We use $P(s, t, G)$ to denote all paths from s to t. The weight of the path, denoted by $\phi(p, G)$, is defined as $\phi(p, G) = \sum_{0 \leq i \leq k-1} \phi(v_i, v_{i+1})$. Given two vertices s and t, the shortest path from s to t is the path with minimum weight in $P(s, t, G)$. The shortest distance, denoted by $\mathsf{dist}_G(s, t)$, is the weight of the shortest path



(a) Road network G (b) $G[r]$

Figure 5.1: A Road Network and Label-induced Subgraph

between s and t. For a given path p in G, the label of p, denoted by $\ell(p, G)$, is the union of edge labels in p, i.e., $\ell(p, G) = \bigcup_{e \in p} \ell(e, G)$. For simplicity, we omit G in the notations if the context is self-evident. For ease of reference, we summarize the frequently used notations in Table 5.1.

Definition 13. ***(Label-Constrained Path)*** *Given two vertices s, t in a road network $G = (V, E, \phi, \ell, \Sigma)$ and a set of edge labels $\mathcal{L} \subseteq \Sigma$, a path from s to t is a label-constrained path regarding $\mathcal{L}$ if $\ell(p) \subseteq \mathcal{L}$.*

Definition 14. ***(Label-Constrained Shortest Path)*** *Given two vertices s, t in a road network $G = (V, E, \phi, \ell, \Sigma)$ and a set of edge labels $\mathcal{L} \subseteq \Sigma$, the label-constrained shortest path from s to t is the path with the minimum weight among the label-constrained paths from s to t regarding $\mathcal{L}$.*

Problem statement. Given a road network $G = (V, E, \phi, \ell, \Sigma)$, a label-constrained shortest path query is defined as $q = (s, t, \mathcal{L})$, where $s, t \in V(G)$, $\mathcal{L} \subseteq \Sigma$, and the answer is the label-constrained shortest path from s to t regarding $\mathcal{L}$. In this chapter, we aim to develop effective indexing techniques to answer q efficiently.

For ease of explanation, we first consider that G is undirected, and discuss how to extend the techniques to handle directed road networks in a separate section (Section 5.5.5).

Example 35. *Consider the road network G in Figure 5.1 (a), the weight and the label of each edge is shown beside the corresponding edges. For example,*

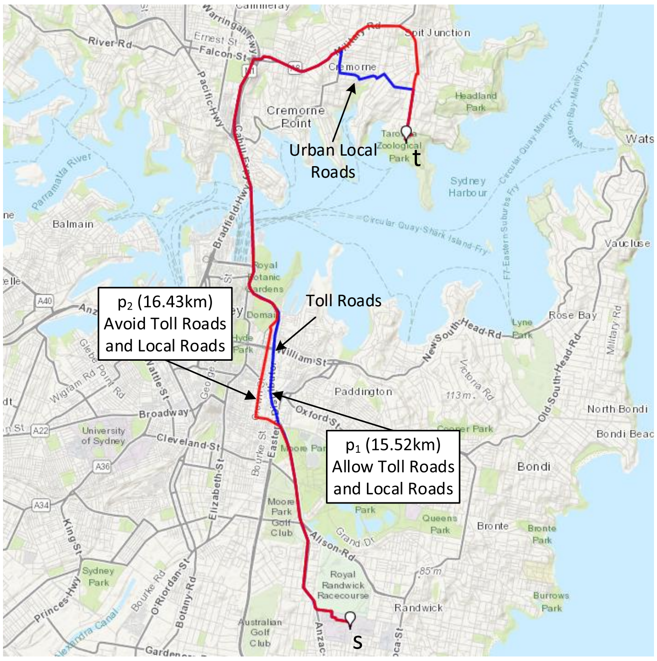


Figure 5.2: Case Study

$\phi((v_1, v_2)) = 1$ and $\ell((v_1, v_2)) = r$. For an edge label set $\{r\}$, the $\{r\}$-induced subgraph $G[r]$ is shown in Figure 5.1 (b), which consists of edges with label r. Given vertices v_5, v_6 and a label set $\{b, g\}$, there are two label-constrained paths between v_5 and v_6: $\{(v_5, v_1, v_4, v_8, v_{11}, v_7, v_6), (v_5, v_9, v_8, v_{11}, v_7, v_6)\}$ and the second one is the label-constrained shortest path with weight 16.

Case Study. Figure 5.2 demonstrates a real-world example of label-constrained shortest path queries. In Sydney, we can briefly divide the roads into three categories: toll road (T), main road (M), and local road (L). Assume that the students from UNSW plan to go to Tarango Zoo by car at weekends. If they only want to get to the zoo as fast as possible, then, they can obtain their route by the query q = (*"UNSW"*, *"Tarango Zoo"*,”TML”), which returns p_1 with 15.52km. On the other hand, if they also want to get to the zoo as fast as possible, but are not willing to pass the toll road or local road, they can obtain their route by the query q = (*"UNSW"*, *"Tarango Zoo"*,”M”), which return p_2

with 16.43km. From this example, we can see that different label-constrained shortest path queries can satisfy different users' requirements in route planning.

5.3 Existing Solution

Edge-disjoint partitioning (EDP) [41] is the state-of-the-art solution for the label-constrained shortest path queries. EDP is an index-based approach consisting of two components:

EDP **indexing.** Given a road network G, EDP first partitions G by the labels of edges. For each label $l \in \Sigma$, the partition Part_l contains the edges with label l, i.e., $\mathsf{Part}_l = G[l]$. It is clear that each edge label uniquely corresponds to a partition, we use them interchangeably when the context is self-evident. Based on the partitions, a vertex v in a partition Part_{l_i} is a *bridge vertex* if there exists an edge $(v, u) \in \mathsf{Part}_{l_j}$, and $l_i \neq l_j$. For a bridge vertex $v \in \mathsf{Part}_{l_i}$, its *OtherHosts* is other partitions containing v. When processing queries, it computes the shortest paths in each partition. These computed paths are all cached in the EDP index. As more queries are processed, which leads to the index size exceeds a specified threshold, EDP uses the least recently used (LRU) replacement strategy to replace the old paths with new computed shortest paths.

Query processing. For a query $q = (s, t, \mathcal{L})$, EDP adopts a greedy traversal paradigm similar to Dijkstra's algorithm to compute the label-constrained shortest path. During the traversal, it maintains a min-priority queue Q, each element of Q has three attributes: (1) Part: the identifier of a partition, (2) v: a vertex id, and (3) d: currently observed distance from s to v. Q is keyed by (Part, v) and ordered by d. EDP initially inserts $(\mathsf{Part}_{l_s}, s, 0)$ into Q, where Part_{l_s} is the partition in which s resides. Then, EDP iteratively extracts elements e' from Q, expands the traversal, and inserts frontier discovered vertex into Q until

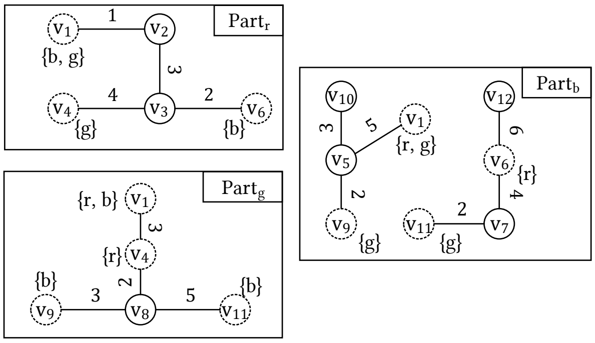


Figure 5.3: EDP Indexing

t is reached or Q becomes empty. During the expansion, EDP first computes the shortest distances d from $e'.v$ to bridge vertices v' in e'.Part, then, for each Part $\in \{\mathcal{L}' \cap \mathcal{L}\}$, where $\mathcal{L}'$ represents the labels of $v'.OtherHosts$ in e'.Part, it inserts $(\mathsf{Part}, v', d + e'.d)$ to Q (if t is in e'.Part, the same procedure is applied to t as well). When computing the distances from $e'.v$ to a bridge vertices v' in a partition, EDP directly obtains it if it is already cached in the index; Otherwise, it performs Dijkstra's algorithm and caches the result in the index.

Example 36. *Consider G shown in Figure 5.1 (a), Figure 5.3 demonstrates the edge-disjoint partitioning of G. The vertices with dashed circle are* bridge vertices*, and the label sets near the bridge vertices are the* OtherHosts List*. For example, v_1 in* Part_r *is a bridge vertex because it has two adjacent edges with labels b and g, and its otherHosts list is* $\{\mathsf{Part}_b, \mathsf{Part}_g\}$. *Given a query $q = (v_5, v_6, \{g, b\})$. As v_5 is located at* Part_b *as shown in Figure 5.3,* EDP *first pushes* $(\mathsf{Part}_b, v_5, 0)$ *into the Q. Then, it pops out* $(\mathsf{Part}_b, v_5, 0)$ *from Q and computes the distances from v_5 to bridge vertices v_1 and v_9. Because the otherHosts of v_1 and v_9 has a common label g with the constraint labels $\{g, b\}$,* EDP *pushes* $(\mathsf{Part}_g, v_1, 5)$ *and* $(\mathsf{Part}_g, v_9, 2)$ *into Q. Then,* $(\mathsf{Part}_g, v_9, 2)$ *is popped out and* EDP *finds the distance from v_9 to bridge vertices in* Part_g*, and* $(\mathsf{Part}_b, v_{11}, 10)$ *is pushed into Q. At this time, there are only two elements in Q:* $(\mathsf{Part}_g, v_1, 5)$ *and* $(\mathsf{Part}_b, v_{11}, 10)$. $(\mathsf{Part}_g, v_1, 5)$ *is popped out and processed, and no new element is pushed into Q.*

Finally, $(\mathsf{Part}_b, v_{11}, 10)$ *is popped out.* EDP *computes the shortest distance from* v_{11} *to* v_6*. As* v_6 *is the target vertex,* EDP *obtains the label-constrained shortest path between* v_5 *and* v_6*, namely* $(v_5, v_9, v_8, v_{11}, v_7, v_6)$ *with weight 16.*

During processing the query, the sub-paths between v_5 *and* v_1/v_9*, between* v_9 *and* v_{11}*, etc., are cached in the index. When a new query is processed and needed to compute, for instance, the shortest distance from* v_9 *to bridge vertices in* Part_g*,* EDP *directly uses the cached result instead of computing it from scratch.*

Drawbacks of EDP. EDP processes the label-constrained shortest path queries correctly, but it has the following two drawbacks in efficiency: (1) theoretically, there is no non-trivial tight bound on its query processing time. The worst-case time complexity of EDP is not better than the online search following Dijkstra's algorithm, which limits the ability of EDP to handle adversarial queries. (2) practically, EDP just caches the computed shortest paths in each partition for the processed queries, but the newly issued queries may distribute diversely and the label-constrained shortest path for the query may involve several partitions, it is quite possible that most of the needed information for a specific query is not cached. In this case, EDP degenerates into an online traversal based algorithm similar to the Dijikstra's algorithm, which implies the processing time could be very large when s and t are far apart in G.

5.4 A Naive Indexing Approach

As analyzed in Section 5.3, EDP is unable to provide efficient query processing regarding label-constrained shortest path queries. In this section, we first introduce the tree decomposition and present a naive tree decomposition based indexing approach, which paves the way to our new index presented in the next section.

5.4.1 Tree Decomposition

Tree decomposition [72] decomposes a graph into a tree-like structure to speed up solving graph problems, it is defined as:

Definition 15. ***(Tree Decomposition)*** *Given a graph G, a tree decomposition T_G of G is a rooted tree with nodes $\{X_1, \cdots, X_n\}$, where each node is a subset of $V(G)$ (i.e., $X_i \subseteq V(G)$), such that:*

1. *$\bigcup_{X \in V(T_G)} X = V(G)$;*

2. *for each edge $(u, v) \in E(G)$, there is a node $X \in T_G$ such that $u \in X$ and $v \in X$;*

3. *for each $v \in V(G)$, the nodes containing v (i.e., $\{X | v \in X\}$) form a connected subtree of T_G.*

Definition 16. ***(Treewidth and Treeheight)*** *Given a tree decomposition T_G of G, the treewidth of T_G, denoted by $\omega(T_G)$ is one less than the maximum size of all nodes in T_G, i.e., $\omega(T_G) = \max_{X \in V(T_G)} |X| - 1$. The treeheight of T_G, denoted by $h(T_G)$, is the maximum depth (the depth of a node in T_G is the distance from the node to the root node of T_G) of all nodes in T_G.*

For ease of presentation, we refer to $v \in V(G)$ in G as a vertex and refer to $X \in V(T_G)$ in T_G as a node. We use ω and h to denote the treewidth and treeheight of the tree decomposition T_G if the context is self-evident. The treewidth of a graph G is the minimum treewidth over all possible tree decompositions of G.

It has been proved that to determine whether a given graph G has treewidth at most a given variable is NP-Complete [8]. Existing techniques to compute the optimal tree decomposition with the minimum treewidth can only handle small graphs [53]. Therefore, in this thesis, we adopt a suboptimal but practically effective algorithm, MDE, to conduct the tree decomposition [93].

Minimum degree elimination based tree decomposition. MDE conducts the tree decomposition in two steps: (1) it iteratively eliminates a vertex v with the minimum degree in G, and then adds edges between all neighbors of v, v's neighbors form a clique in G. Clearly, after the elimination of v, v's neighbors become its neighbor's neighbor. It proceeds the elimination until G becomes empty. For each elimination, the eliminated vertex v and its neighbors $\mathsf{nbr}(v)$ form a node $X(v)$ in T_G. (2) After all the vertices are eliminated, for each node $X(v)$, $X(u)$ is set as the parent of $X(v)$ in T_G, where $X(u)$ is the node created by the first eliminated vertex u in $X(v)\backslash\{v\}$.

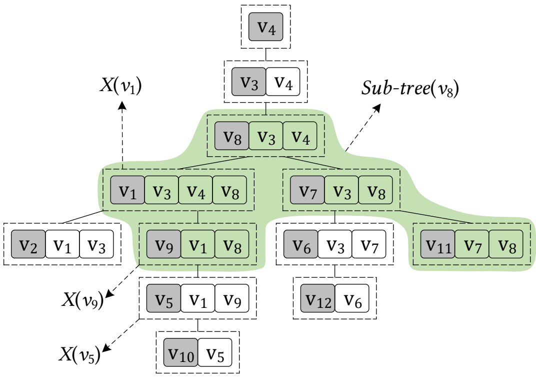


Figure 5.4: A Tree Decomposition T_G of G

Example 37. *Figure 5.4 shows the tree decomposition T_G of G in Figure 5.1 (a) generated by* MDE. *T_G has 12 nodes. The vertex elimination order is v_{10}, v_{12}, v_5, v_2, v_6, v_{11}, v_7, v_9, v_1, v_8, v_3, v_4. The elimination of a vertex v leads to a unique node $X(v)$ in T_G. For example, the elimination of v_5 creates node $X(v_5) = \{v_5, v_1, v_9\}$. The nodes that contains v_8 form a connected subtree of T_G (the green area). Since the nodes in T_G contain at most 4 vertices, the treewidth $\omega = 3$, and treeheight $h = 6$.*

5.4.2 A Naive Indexing Approach

Given $G = (V, E, \phi, \ell, \Sigma)$, there are $2^{|\Sigma|}$ possible edge label combinations. Therefore, we can build $2^{|\Sigma|}$ indices and each index is built upon the induced subgraph by one possible combination of the edge label in Σ. As all the possible edge label combinations are considered, for each index, we only need to treat the corresponding induced subgraph as unlabelled and build the index following the shortest path indexing technique for the unlabelled road networks. To answer a query $q = (s, t, \mathcal{L})$, the index for $\mathcal{L}$ is utilized to retrieve the shortest path. Following this idea, we present a naive indexing approach based on tree decomposition.

Before presenting the naive indexing approach, we first introduce the vertex cut property of the tree decomposition, this property is the key to apply tree decomposition to shortest path queries.

Definition 17. ***(Vertex Cut)*** *Given a graph G, a subset of vertices $C \subset V(G)$ is a vertex cut of G if the deletion of C from G splits G into multiple connected components. Given two vertices s and t in G, the vertex cut C is a s-t cut if the deletion of C from G disconnects s and t, and we say C separates s and t.*

Lemma 22. *[73] Given a tree decomposition T_G of G, for any non-root node X_c and its parent X_p, if there exist $s \in X_c \setminus X_p$ and $t \in X_p \setminus X_c$, then $X_c \cap X_p$ is a vertex cut of G and it separates s and t.*

Lemma 23. *[21] Given a tree decomposition T_G of G, for any two vertices s and t in $V(G)$, suppose $X(s)$ is not an ancestor/decedent of $X(t)$ in T_G, let X_{lca} be the lowest common ancestor (LCA) of $X(s)$ and $X(t)$ in T_G, then X_{lca} is a vertex cut of G and it separates s and t.*

Given a s-t cut C, it is obvious that every path from s to t passes at least one vertex in C. Accordingly, we have:

Algorithm 8: NaiveQuery $(s, t, \mathcal{L}, T)$

```
1  $X_{\mathsf{lca}} \leftarrow$ find LCA of $X(s)$ and $X(t)$ in $T_{G[\mathcal{L}]}$;
   // compute shortest distance from $s$ to vertices in $X_{\mathsf{lca}}$
2  $d_s(\cdot) \leftarrow \infty$, $d_s(s) \leftarrow 0$ ;
3  foreach $w \in X(s) \backslash \{s\}$ do
4      $d_s(w) \leftarrow \mathsf{dist}_{G[\mathcal{L}]}(s, w)$;
5  $X'_s \leftarrow X(s)$;
6  while $X_{\mathsf{lca}} \neq X'_s$ do
7      $X_p \leftarrow$ parent of $X'_s$ in $T_{G[\mathcal{L}]}$;
8      for $u \in X_p \setminus X'_s$ do
9          for $v \in X_p \cap X'_s$ do
10             $d_s(u) = \min\{d_s(u), d_s(v) + \mathsf{dist}_{G[\mathcal{L}]}(v, u)\}$;
11     $X'_s \leftarrow X_p$;
12 Repeat line 2-11 by replacing $s$ with $t$;
13 return $\min_{w \in X_{\mathsf{lca}}}\{d_s(w) + d_t(w)\}$;
```

Lemma 24. *[68] Given two vertices s and t in G, let C be a s-t cut, then* $\mathsf{dist}(s, t) = \min_{v \in C}\{\mathsf{dist}(s, v) + \mathsf{dist}(v, t)\}$.

Example 38. *Consider the tree decomposition T_G of G shown in Figure 5.4. For $X(v_9)$ and its parent node $X(v_1)$, $X(v_9) \cap X(v_1) = \{v_1, v_8\}$ is a vertex cut of G, which separates v_9 and v_3. As shown in Figure 5.4, for $X(v_{10})$ and $X(v_{12})$, their* LCA *is $X(v_8)$, we know* $\mathsf{dist}(v_{10}, v_3) = 12$*,* $\mathsf{dist}(v_{10}, v_4) = 10$*,* $\mathsf{dist}(v_{10}, v_8) = 8$*;* $\mathsf{dist}(v_{12}, v_3) = 8$*,* $\mathsf{dist}(v_{12}, v_4) = 12$ *and* $\mathsf{dist}(v_{12}, v_8) = 14$*, the shortest distance from v_{10} to v_{12} is* $\mathsf{dist}(v_{10}, v_{12}) = \min(12 + 8, 10 + 12, 8 + 14) = 20$.

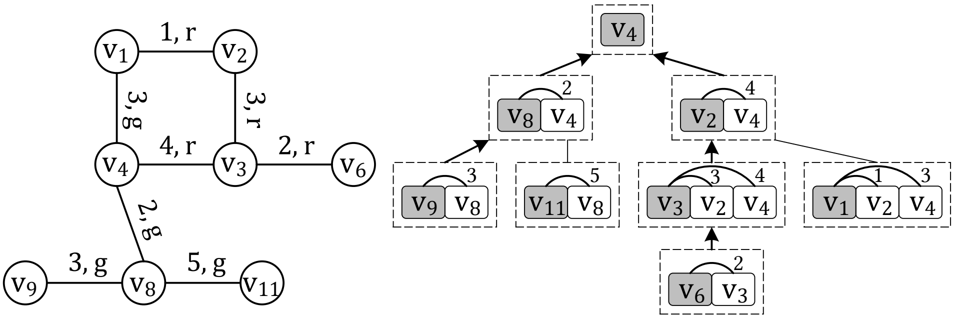
Since the label-constrained shortest path between two vertices can be easily obtained if their label-constrained shortest distance is determined with our algorithms, we focus on the *computation of the label-constrained shortest distance* between two vertices hereafter for clearness and discuss how to obtain the corresponding shortest path in Section 5.5.4. For brevity, given two vertices u, v, and an edge label set $\mathcal{L} \subseteq \Sigma$, we use $\mathsf{dist}^{\mathcal{L}}_G(u, v)$ to denote the label-constrained shortest distance from u to v regarding $\mathcal{L}$ in G.

The naive indexing approach. Based on the above lemmas, we can devise a straightforward indexing approach to compute label-constrained shortest distance between two vertices as follows:

• *Indexing.* For each possible edge label set $\Sigma_s \subseteq \Sigma$, we first retrieve the Σ_s-induced subgraph $G[\Sigma_s]$. Based on $G[\Sigma_s]$, we compute the tree decomposition $T_{G[\Sigma_s]}$ with MDE. After that, for each $X(v) \in T_{G[\Sigma_s]}$, we compute the $\mathsf{dist}_{G[\Sigma_s]}(v, u)$ for any $u \in X(v) \setminus \{v\}$ and store them in node $X(v)$ using hash table. Note that we also maintain the mapping from vertex v to node $X(v)$ in the index for ease of query processing.

• *Query Processing.* Given a query $q = (s, t, \mathcal{L})$, we can compute $\mathsf{dist}_G^{\mathcal{L}}(s, t)$ based on the index $T_{G[\mathcal{L}]}$ built on $G[\mathcal{L}]$. The detailed procedure is shown in Algorithm 8. It first computes the lowest common ancestor X_{lca} of $X(s)$ and $X(t)$ in $T_{G[\mathcal{L}]}$ (line 1). After that, it computes the distance from s to vertices in X_{lca}. Based on Lemma 22, for a node X_s' and its parent X_p, where X_s' is an ancestor node of $X(s)$, and assume that the shortest distances from s to all vertices in X_s' are already computed, then the shortest distances from s to vertices in $u \in X_p \setminus X_s'$ can be calculated as $\mathsf{dist}_{G[\mathcal{L}]}(s, u) = \min_{w \in X_s' \cap X_p}\{\mathsf{dist}_{G[\mathcal{L}]}(s, w) + \mathsf{dist}_{G[\mathcal{L}]}(w, u)\}$, where $\mathsf{dist}_{G[\mathcal{L}]}(w, u)\}$ can be accessed by looking up the hash table in $X(w)$ or $X(u)$. Hence, we can iteratively compute the shortest distances from s to vertices in X_{lca} along the tree path from X_s to X_{lca} (line 3-11). The distances from t to vertices in X_{lca} can be computed similarly (line 12). Finally, $\mathsf{dist}_G^{\mathcal{L}}(s, t)$ is obtained via the vertices in X_{lca} based on Lemma 23 and Lemma 24 (line 13).

Example 39. *Reconsider the road network shown in Figure 5.1 (a). Figure 5.5 (a) shows the $\{g, r\}$-induced subgraph $G[\{g, r\}]$. Figure 5.5 (b) shows the corresponding index $T_{G[\{g,r\}]}$ built on $G[\{g, r\}]$. For the node in $T_{G[\{g,r\}]}$, such as $X(v_1) = \{v_1, v_2, v_4\}$, we store $\mathsf{dist}_{G[\{g,r\}]}(v_1, v_2) = 1$ and $\mathsf{dist}_{G[\{g,r\}]}(v_1, v_4) = 3$ in it. For a query $q = (v_9, v_6, \{g, r\})$, the arrows in Figure 5.5 (b) demonstrate the query processing procedure. The LCA of $X(v_9)$ and $X(v_6)$ is $X(v_4)$. It it-*



(a) $G[\{g, r\}]$ (b) The Index of $G[\{g, r\}]$

Figure 5.5: The Naive Indexing Approach

eratively computes the shortest distance from v_9 *and* v_6 *to the vertices in* $X(v_4)$ *following the arrows. For example, when computing the shortest distances from* v_9 *to* $v_4 \in X(v_8)$*, as* $X(v_9) \cap X(v_8) = v_8$*,* $\mathsf{dist}_{G[\{g,r\}]}(v_9, v_4) = \mathsf{dist}_{G[\{g,r\}]}(v_9, v_8) + \mathsf{dist}_{G[\{g,r\}]}(v_8, v_4) = 5$*. Similarly, it computes* $\mathsf{dist}_{G[\{g,r\}]}(v_6, v_4) = 6$*. Hence,* $\mathsf{dist}_{G[\{g,r\}]}(v_9, v_6) = \mathsf{dist}_{G[\{g,r\}]}(v_9, v_4) + \mathsf{dist}_{G[\{g,r\}]}(v_6, v_4) = 11$*.*

Theorem 12. *Given a query* $q = (s, t, \mathcal{L})$*, Algorithm 8 computes* $\mathsf{dist}_G^{\mathcal{L}}(s, t)$ *correctly.*

Proof. This theorem can be directly proved based on Lemma 22, Lemma 23 and Lemma 24. □

Lemma 25. *Given a tree decomposition* T_G *of* G *generated by* MDE*, for a node* X *of* T_G*,* $|X \bigcup_{X_a \in \mathcal{A}(X)} X_a| \leq h$*, where* $\mathcal{A}(X)$ *represents the ancestors of* X *in* T_G*.*

Proof. According to step (2) of MDE, given a non-root node $X(v) \in T_G$, for $u \in X(v) \setminus \{v\}$, $X(u)$ must be an ancestor node of $X(v)$, i.e., $X(u) \in \mathcal{A}(X(v))$. Suppose the tree path from X to root node X_r in T_G is $(X(v_1) = X, X(v_2), \cdots, X(v_k) = X_r)$, then $\bigcup_{i=1}^{k} X(v_i) = \{v_1, v_2, \cdots, v_k\}$. Therefore, $|X \bigcup_{X_a \in \mathcal{A}(X)} X_a| = |\bigcup_{i=1}^{k} X(v_i)| = k \leq h$. Thus, the lemma holds. □

Lemma 26. *Given a tree decomposition T_G of G, for a node $X(v)$ and a non-root ancestor node $X(u)$ of $X(v)$, let $X_p(v)$ (resp. $X_p(u)$) be the parent node of $X(v)$ (resp. $X(u)$), then $\{X_p(v) \setminus X(v)\} \cap \{X_p(u) \setminus X(u)\} = \emptyset$.*

Proof. We prove this by contradiction. Suppose there exists a vertex $w \in \{X_p(v) \setminus X(v)\} \cap \{X_p(u) \setminus X(u)\}$, then $w \in X_p(v)$, $w \in X_p(u)$ and $w \notin X(u)$. Meanwhile, the possible relationships between $X(u)$ and $X(v)$ are: (1) $X(u)$ is the parent node of $X(v)$. It means $X(u) = X_p(v)$. This contradicts with $w \notin X(u)$ and $w \in X_p(v)$. (2) $X(u)$ is not the parent node of $X(v)$. Then $X(u)$ must be the ancestor node of $X_p(v)$. Thus, we can derive that $X_p(v)$ and $X_p(u)$ that contain w are split by $X(u)$ which doesn't contain w. This contradicts with Definition 15 in which the nodes contains w form a connected-subtree in T_G. Thus, the lemma holds. □

Theorem 13. *Given a query $q = (s, t, \mathcal{L})$, Algorithm 8 computes $\mathsf{dist}_G^{\mathcal{L}}(s, t)$ in $O(h \cdot \omega)$.*

Proof. In Algorithm 8, the computation of LCA of $X(s)$ and $X(t)$ in line 1 can be finished in $O(1)$ time [12]. Suppose the tree path from $X(s)$ to X_{lca} is $(X(s) = X_1, X_2, \cdots, X_{\mathsf{lca}} = X_k)$, where X_i is the parent of X_{i-1}. According to Lemma 25 and Lemma 26, $\Sigma_{i=2}^{k}|X_i \setminus X_{i-1}| < h$. Thus, the total number of vertices u visited in line 8 of Algorithm 8 is less than h. For line 9-10, the distance computation of $d_s(u)$ can be finished in $O(\omega)$. Therefore, the overall time complexity of Algorithm 8 is $O(h \cdot \omega)$. □

Remark. TEDI [88], the state-of-the-art tree decomposition based indexing approach for the shortest path queries on unlabelled road networks, presents a query processing algorithm using a similar idea as Algorithm 8 with time complexity $O(h \cdot \omega^2)$. As shown in Theorem 13, our presented algorithm has a time complexity of $O(h \cdot \omega)$, which reduces that of TEDI by a factor ω.

5.5 Our new indexing approach

As shown in our experiments (Table 5.2), MDE generally generates a tree decomposition with small treeheight h and treewidth ω for road networks. For example, h and ω for the whole USA road network is $2,886$ and 579, respectively. Hence, Algorithm 8 permits an efficient query processing regarding a label-constrained query. However, the naive approach needs to construct $2^{|\Sigma|}$ separate indices. Obviously, it is prohibitive to construct and maintain such $2^{|\Sigma|}$ separate indices. In this section, we exploit the dominance relationships between edge-labelled paths and present a new index based on the tree decomposition. The new index can overcome the problem of the naive index with little additional cost for the query processing.

5.5.1 A New Index Structure

Reconsider the road network G in Figure 5.1, due to the existence of path $\{v_1, v_2, v_3, v_6\}$, the shortest distance between v_1 and v_6 in $\{b, g, r\}$-induced subgraph is 5. Meanwhile, in $\{b, r\}$, $\{g, r\}$ and $\{r\}$ induced subgraphs, the shortest distance between v_1 to v_6 is 5 as well. In this case, if we have already stored the shortest distance 5 between v_1 and v_6 in the index constructed on $G[\{r\}]$, it is redundant to store the same information in the indices constructed on $G[\{b, r\}]$, $G[\{g, r\}]$, and $G[\{b, g, r\}]$. Based on this observation, instead of considering all the possible edge label sets regarding Σ separately, we can treat these possible edge label sets as a whole and design a holistic compact index that covers all the shortest distance information without storing any redundant information. Following this idea, we have the following lemma:

Lemma 27. *Given two vertices u, v in G, let p be a path between u and v in G, then u can reach v in distance d regarding a label set $\mathcal{L}$ if $\ell(p) \subseteq \mathcal{L}$, $\phi(p) \leq d$.*

Proof. This lemma can be proved directly based on Definition 13. □

Following Lemma 27, we define the label-constrained shortest distance set between two vertices as follows:

Definition 18. ***(Label-constrained Shortest Distance Set)*** *Given a road network G and two vertices u and v in G, the label-constrained shortest distance set (*LSDS*) of u, v, denoted by $\mathcal{S}(u, v)$, is a set of label-distance pairs $\{(L_1, d_1), (L_2, d_2), \dots\}$ such that:*

1. *For each $(L_i, d_i) \in \mathcal{S}(u, v)$, there exists a path p from u to v with $L_i = \ell(p)$ and $d_i = \phi(p)$.*

2. *For any path p from u to v, there exists a $(L_i, d_i) \in \mathcal{S}(u, v)$, $L_i \subseteq \ell(p)$ and $d_i \leq \phi(p)$;*

3. *For any path p from u to v and $(L_i, d_i) \in \mathcal{S}(u, v)$, if $\ell(p) \subset L_i$, then $d_i < \phi(p)$; if $\ell(p) = L_i$, then $d_i \leq \phi(p)$.*

Condition (1) ensures that each label-distance pair corresponds to a path in G. Condition (2) guarantees that $\mathcal{S}(u, v)$ covers all possible label-constrained shortest distances between u and v. Condition (3) ensures that the set is minimum and there is no redundancy label-distance pair regarding label-constrained shortest distance according to Lemma 27. Based on Definition 18, for a given G and its tree decomposition T_G, we construct the label-constrained shortest distance index as follows:

Definition 19. ***(Label-constrained Shortest Distance Index)*** *Given a road network G, let T_G be its tree decomposition, the label-constrained shortest distance index of G, denoted by* LSD-Index*, is built on T_G. For each node $X(v) \in V(T_G)$, the label-constrained shortest distance set between v to other vertices $u \in X(v) \setminus \{v\}$ are precomputed and stored.*

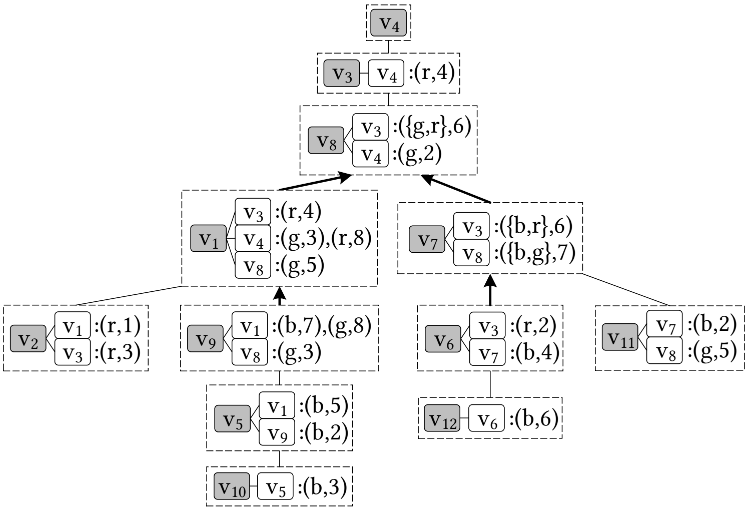


Figure 5.6: The LSD-Index

Example 40. *Figure 5.6 shows the* LSD-Index *of G in Figure 5.1 (a). Each node stores the corresponding* LSDS*. Take $\mathcal{S}(v_1, v_4)$ as an example. $\mathcal{S}(v_1, v_4) = \{(g, 3), (r, 8)\}$ because (1) $(g, 3)$ and $(r, 8)$ correspond to path (v_1, v_4) and path (v_1, v_2, v_3, v_4) between v_1 and v_4, respectively, (2) any other paths between v_1 and v_4, such as $(v_1, v_5, v_9, v_8, v_4)$, can be covered by these two paths, and (3) there is no redundancy in $\{(g, 3), (r, 8)\}$.*

5.5.2 Query Processing by LSD-Index

With LSD-Index, we can easily obtain a query processing algorithm similar to Algorithm 8. The details are shown in Algorithm 9. Given a query $q = (s, t, \mathcal{L})$, Algorithm 9 first computes the lowest common ancestor X_{lca} of $X(s)$ and $X(t)$ (line 1). Then, it computes the label-constrained distance from s (resp. t) to the vertices in X_{lca} along the tree path similarly to Algorithm 8 (line 3-11). Finally, it computes the label-constrained shortest distance by iterating over vertices in X_{lca} (line 13). Procedure dist computes the label-constrained shortest distance between u and v regarding an edge label set L, it iterates the label-distance pair (L', d') in $\mathcal{S}(u, v)$ (line 15) and returns the shortest distance d' such that $L' \subseteq L$

(line 16-17).

Example 41. *Reconsider the query $q = (v_9, v_6, \{g, r\})$, the arrows in Figure 5.6 demonstrate the query process. The* LCA *of $X(v_9)$ and $X(v_6)$ is $X(v_8)$. For v_9, $\mathsf{dist}_G^{\{g,r\}}(v_9, v_1) = 8$, as $\mathcal{S}(v_9, v_1)$ contains $(g, 8)$. Similarly, $\mathsf{dist}_G^{\{g,r\}}(v_9, v_8) = 3$. Following the arrow, $X(v_9) \cap X(v_1) = \{v_1, v_8\}$ and $X(v_1) \setminus X(v_9) = \{v_3, v_4\}$. Hence $\mathsf{dist}_G^{\{g,r\}}(v_9, v_3) = \min\{\mathsf{dist}_G^{\{g,r\}}(v_9, v_1) + \mathsf{dist}_G^{\{g,r\}}(v_1, v_3), \mathsf{dist}_G^{\{g,r\}}(v_9, v_8) + \mathsf{dist}_G^{\{g,r\}}(v_8, v_3)\} = 9$. Similarly, $\mathsf{dist}_G^{\{g,r\}}(v_9, v_4) = 5$. For v_6, $\mathsf{dist}_G^{\{g,r\}}(v_6, v_3) = 2$, $\mathsf{dist}_G^{\{g,r\}}(v_6, v_4) = 6$, and $\mathsf{dist}_G^{\{g,r\}}(v_6, v_8) = 8$ can be computed similarly. Then, $\mathsf{dist}_G^{\{g,r\}}(v_9, v_6) = \min_{w \in \{v_3, v_4, v_8\}}\{\mathsf{dist}_G^{\{g,r\}}(v_9, w) + \mathsf{dist}_G^{\{g,r\}}(w, v_6)\} = 11$.*

Theorem 14. *Given a query $q = (s, t, \mathcal{L})$, Algorithm 9 correctly computes $\mathsf{dist}_G^{\mathcal{L}}(s, t)$.*

Proof. This theorem can be proved similarly to Theorem 12 based on Definition 18 and Definition 19. □

Theorem 15. *Given a road network G, the size of the* LSD-Index *is $O(n \cdot \omega \cdot \rho)$, where ρ represents the maximum size of* LSDS *stored in* LSD-Index.

Proof. There are n tree nodes in T_G, and each tree node $X(v)$ has at most ω vertices in $X(v) \setminus \{x\}$. For each $u \in X(v) \setminus \{x\}$, we store the $\mathcal{S}(v, u)$ whose size is at most ρ in LSD-Index. Therefore, the size of the LSD-Index is $O(n \cdot \omega \cdot \rho)$. □

Theorem 16. *Given a query $q = (s, t, \mathcal{L})$, Algorithm 9 computes $\mathsf{dist}_G^{L}(s, t)$ in $O(h \cdot \omega \cdot \rho)$.*

Proof. This theorem can be proved similarly to Theorem 13. The only difference is that `dist`$(v, u, \mathcal{L})$ is invoked in line 10. The time complexity of `dist`$(v, u, \mathcal{L})$ is bounded by $O(\rho)$. Therefore, the time complexity of Algorithm 9 for answering a query is $O(h \cdot \omega \cdot \rho)$. □

Algorithm 9: LSD-Index-Query $(s, t, \mathcal{L}, \mathsf{LSD\text{-}Index}\ T)$

```
1  $X_{\mathsf{lca}} \leftarrow$ find LCA of $X(s)$ and $X(t)$ in $T$;
   // compute LSD from $s$ to vertices in $X_{\mathsf{lca}}$
2  $d_s^{\mathcal{L}}(\cdot) \leftarrow \infty$, $d_s^{\mathcal{L}}(s) \leftarrow 0$;
3  foreach $w \in X(s) \backslash \{s\}$ do
4      $d_s^{\mathcal{L}}(w) \leftarrow \mathtt{dist}(s, w, \mathcal{L})$;
5  $X_s' \leftarrow X(s)$;
6  while $X_{\mathsf{lca}} \neq X_s'$ do
7      $X_p \leftarrow$ parent of $X_s'$ in $T$;
8      for $u \in X_p \setminus X_s'$ do
9          for $v \in X_p \cap X_s'$ do
10             $d_s^{\mathcal{L}}(u) \leftarrow \min(d_s^{\mathcal{L}}(u), d_s(v) + \mathtt{dist}(v, u, \mathcal{L}))$;
11     $X_s' \leftarrow X_p$;
12 Repeat line 2-11 by replacing $s$ with $t$;
13 return $\min_{w \in X_{\mathsf{lca}}}(d_s^{\mathcal{L}}(w) + d_t^{\mathcal{L}}(w))$;
14 Procedure dist($u$, $v$, $L$)
       // assume $\mathcal{S}(u,v)$ is ordered by distance
15     for $(L', d') \in \mathcal{S}(u, v)$ do
16         if $L' \subseteq L$ then
17             return $d'$;
```

Remark. Compared with the naive approach, an additional factor ρ is introduced in the time complexity of Algorithm 9. However, as shown in our experiments (Table 5.2), ρ is very small in practice. On the other hand, due to LSD-Index, our approach avoids constructing and maintaining $2^{|\Sigma|}$ separate indices in the naive approach.

5.5.3 LSD-Index Construction

To construct the LSD-Index, a direct solution is based on Definition 19 as follows: we first conduct the tree decomposition on G, and then compute the LSDS for the vertices in each node according to Definition 18. In this approach, the time complexity for the LSDS computation is $O(n \cdot \omega \cdot ((2^{|\Sigma|})^2 + 2^{|\Sigma|} \cdot (m + n \log n)))$.

Algorithm 10: LSDS Operators

```
Procedure LSDSJoin($\mathcal{S}'_p$, $\mathcal{S}''_p$)
    $\mathcal{S}_p \leftarrow \emptyset$;
    foreach $(L', d') \in \mathcal{S}'_p$ do
        foreach $(L'', d'') \in \mathcal{S}''_p$ do
            $\mathcal{S}_p \leftarrow \mathcal{S}_p \cup \{(L' \cup L'', d' + d'')\}$;
    return $\mathcal{S}_p$;
Procedure LSDSPrune($\mathcal{S}_p$)
    foreach $(L, d) \in \mathcal{S}_p$ do
        foreach $(L', d') \in \mathcal{S}_p$ do
            if $L \subseteq L'$ and $d \leq d'$ then
                $\mathcal{S}_p \leftarrow \mathcal{S}_p \setminus \{(L'_c, d'_c)\}$;
    return $\mathcal{S}_p$;
```

Obviously, the cost of this part is prohibitive, which consequently makes this approach impractical.

To address this problem, we propose a new index construction algorithm. Instead of dividing the construction into two independent procedures, the new algorithm progressively maintains partial LSDS by coordinating the procedures of the tree decomposition and LSDS computation. Based on the partial LSDS, the new algorithm computes the complete LSDS in a top-down manner in which the computed complete LSDS can be re-used to accelerate the computation for those not-yet-computed complete LSDS. Before presenting the algorithm, we first introduce two operators on LSDS that are used in the index construction algorithm:

Definition 20. *(**Operator** LSDSJoin) Given two LSDS $\mathcal{S}'_p$ and $\mathcal{S}''_p$, operator LSDSJoin generates a new LSDS by joining the entities in $\mathcal{S}'_p$ and $\mathcal{S}''_p$, i.e.,* LSDSJoin$(\mathcal{S}'_p, \mathcal{S}''_p) = \{(L' \cup L'', d' + d'') \mid \forall (L', d') \in \mathcal{S}'_p \land (L'', d'') \in \mathcal{S}''_p\}$

Definition 21. *(**Operator** LSDSPrune) Given a LSDS $\mathcal{S}_p$, operator LSDSPrune removes (L_i, d_i) from $\mathcal{S}_p$, if $\exists$ $(L_j, d_j) \in \mathcal{S}_p$ such that $L_j \subseteq L_i \land d_j \leq d_i$, where*

$i \neq j$.

The procedures of these operators are shown in Algorithm 10.

Algorithm. With the above operators, our new index construction algorithm is shown in Algorithm 11. It contains two phases: in phase 1, it conducts tree decomposition in which partial LSDS are computed for vertex pairs incident to the involved edges (line 1-18); in phase 2, it computes the complete LSDS in a top-down manner based on the partial LSDS of phase 1 (line 19-24).

• *Phase 1: Partial* LSDS *maintained tree decomposition.* In phase 1, it conducts the tree decomposition following MDE and maintains the partial LSDS for the vertex pairs incident to edges involved in the decomposition. Specifically, it first initializes G_0 as G and T as an empty tree (line 1). For each edge (u, v), $\mathcal{S}_p(u, v)$ is initialized as $\{(\ell((u, v)), \phi((u, v)))\}$, where $\mathcal{S}_p(u, v)$ is used to store the partial LSDS (line 2-3). After that, it performs vertex elimination iteratively following the procedure of MDE (line 4-14). In the ith iteration, it eliminates the vertex v with minimum degree from G_{i-1} and assigns its $\pi(\cdot)$ as i, where $\pi(\cdot)$ records the elimination order (line 5-6). Then, for each vertex pair u, w in the $\mathsf{nbr}(v, G_{i-1})$, it first joins $\mathcal{S}_p(v, u)$ and $\mathcal{S}_p(v, w)$ by LSDSJoin and obtains $\mathcal{S}'$ (line 8). If G_{i-1} does not contain an edge (u, w), it adds an edge (u, w) into G_i and assigns $\mathcal{S}_p(v, w)$ as the result of LSDSPrune on $\mathcal{S}'$ (line 9-11); Otherwise, the $\mathcal{S}_p(u, w)$ is updated as the result of LSDSPrune on $\mathcal{S}_p(u, w) \cup \mathcal{S}'$ (line 13). In Algorithm 11, we assume $\pi(u) < \pi(w)$ for the clearness of the presentation. After the elimination of v, it adds a node $X(v)$ containing v and its neighbors $\mathsf{nbr}(v, G_{i-1})$ into T (line 14). After all vertices are eliminated, the parent-child relationships between nodes are generated (line 15-18). For a non-root vertex v, it selects the vertex $u \in X(v) \setminus \{v\}$ with smallest $\pi(\cdot)$ value (line 17) and sets $X(u)$ as the parent node of $X(v)$ (line 18).

Before presenting phase 2, we first introduce the label-constrained shortest distance (LSD) preserved graph $\mathcal{G}_i$, it's properties serve the theoretical bases of phase 2 and are also used for the proof of Algorithm 11.

Definition 22. *(LSD **Preserved Graph**) Given a graph G_i generated in phase 1, the LSD preserved graph of G_i, denoted by $\mathcal{G}_i$, is a labelled multigraph such that (1) $V(\mathcal{G}_i) = V(G_i)$; (2) if there is an edge $e = (u, v)$ in G_i, then, for each entity $(L, d) \in \mathcal{S}_p(u, v)$, there is an edge $e' = (u, v)$ with $\ell(e') = L$ and $\phi(e') = d$ in $\mathcal{G}_i$.*

Lemma 28. *Given two vertices u, v in $\mathcal{G}_i$, for any edge label set L, $\mathsf{dist}^L_{\mathcal{G}_i}(u, v) = \mathsf{dist}^L_G(u, v)$.*

Proof. This lemma can be directly proved based on Definition 20, Definition 21, and Definition 22. □

According to Lemma 28, the label-constrained shortest distance between any two vertices in G_i is preserved in $\mathcal{G}_i$. Moreover, we have the following lemma:

Lemma 29. *Given a vertex $v \in V(\mathcal{G}_{\pi(v)-1})$, for any edge label set L, the label-constrained shortest path from v to $u \in X(v) \setminus \{v\}$ regarding L in $\mathcal{G}_{\pi(v)-1}$ only contains v and u, or passes a vertex in $X(v) \setminus \{v, u\}$.*

Proof. According to the phase 1 of Algorithm 11, $X(v) \setminus \{v\}$ is the neighbors of v in $\mathcal{G}_{\pi(v)-1}$. Thus, the label-constrained shortest path regarding L from v to u either only contains v, u, or passes a vertex in $X(v) \setminus \{v, u\}$. □

Therefore, for a vertex v, if we have already known the complete LSDS $\mathcal{S}(u, w)$ for any two vertices $u, w \in X(v) \setminus \{v\}$, to compute the complete LSDS $\mathcal{S}(v, u)$, according to Definition 18, Lemma 28 and Lemma 29, we only need to join the partial LSDS $\mathcal{S}_p(v, w)$ with the complete $\mathcal{S}(u, w)$ for each $w \in X(v) \setminus \{v, u\}$ with $\mathsf{LSDSJoin}$, add the result to $\mathcal{S}_p(v, u)$ and remove redundant label-distance pair

Algorithm 11: LSD-Index-Cons(G)

```
   // phase 1
1  $G_0 \leftarrow G$; $T \leftarrow \emptyset$ ;
2  foreach $(u, v) \in G$ do
3      $\mathcal{S}_p(u, v) \leftarrow \{(\ell((u, v)), \phi((u, v)))\}$;
4  for $i \leftarrow 1$ to $n$ do
5      $v \leftarrow$ vertex in $G_{i-1}$ with minimum degree;
6      $\pi(v) \leftarrow i$; $G_i \leftarrow G_{i-1} \setminus v$;
7      foreach $u, w \in \mathsf{nbr}(v, G_{i-1})$ do
8          $\mathcal{S}' \leftarrow \mathsf{LSDSJoin}(\mathcal{S}_p(v, u), \mathcal{S}_p(v, w))$;
9          if $(u, w) \notin G_{i-1}$ then
10             add an edge $(u, w)$ into $G_i$;
11             $\mathcal{S}_p(u, w) \leftarrow \mathsf{LSDSPrune}(\mathcal{S}')$;
12         else
13             $\mathcal{S}_p(u, w) \leftarrow \mathsf{LSDSPrune}(\mathcal{S}_p(u, w) \cup \mathcal{S}')$;
14     $X(v) \leftarrow \{v\} \cup \mathsf{nbr}(v, G_{i-1})$;
15 foreach $X(v) \in T$ do
16     if $\pi(v) < n$ then
17         $u \leftarrow$ the vertex in $X(v) \backslash \{v\}$ with smallest $\pi(\cdot)$ ;
18         add $X(v)$ as the child of $X(u)$;
   // phase 2
19 for $i \leftarrow n - 1$ to $1$ do
20     $v \leftarrow$ vertex with $\pi(\cdot) = i$ ;
21     for $u \in X(v) \setminus \{v\}$ do
22         for $w \in X(v) \setminus \{v, u\}$ do
23             $\mathcal{S}' \leftarrow \mathsf{LSDSJoin}(\mathcal{S}_p(v, w), \mathcal{S}_p(u, w))$;
24             $\mathcal{S}_p(v, u) \leftarrow \mathsf{LSDSPrune}(\mathcal{S}_p(v, u) \cup \mathcal{S}')$;
```

with LSDSPrune. Moreover, we can apply the above procedure recursively to compute the complete LSDS $\mathcal{S}(u, w)$. Following this idea, we can compute the complete LSDS in a top-down manner based on tree decomposition and use the computed complete LSDS to accelerate the computation of not-yet-computed complete LSDS.

• *Phase 2: Top-down complete* LSDS *computation.* The phase 2 of the construction algorithm is shown in line 19-24 of Algorithm 11. It processes the vertices in

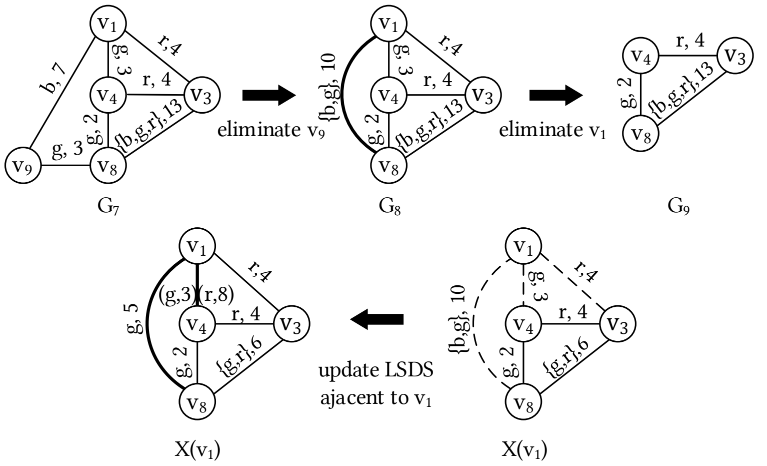


Figure 5.7: Procedure of Index Construction

the decreasing order of their $\pi(\cdot)$ value (line 19). For each vertex v, to compute the complete LSDS of v and u, where $u \in X(v) \setminus \{v\}$, it iterates the vertices $w \in X(v) \setminus \{v, u\}$, computes $\mathcal{S}'$ by LSDSJoin on $\mathcal{S}_p(v, w)$ and $\mathcal{S}_p(u, w)$, and removes the redundancy in $\mathcal{S}_p(v, u) \cup \mathcal{S}')$ with LSDSPrune (line 22-24). The construction finishes when all the vertices are processed.

Example 42. *In Figure 5.7, the upper (resp. lower) part illustrate some of key steps of phase 1 (resp. phase 2) during the construction of* LSD-Index *for* G *in Figure 5.1 (a), where the* LSDS *is shown near each edge. For example, in phase 1, when eliminating* v_9 *from* G_7*, a new edge* (v_1, v_8) *is added, and* $\mathcal{S}_p(v_1, v_8) = \{(\{b, g\}, 10)\}$ *is obtained by joining* $\mathcal{S}_p(v_9, v_8)$ *and* $\mathcal{S}_p(v_9, v_1))$*. In phase 2, for* $\mathcal{S}(v_1, v_8)$*, since* LSDSJoin$(\mathcal{S}_p(v_1, v_4), \mathcal{S}(v_8, v_4)) = \{(\{g\}, 5), (\{g, r\}, 10)\}$*,* $(\{g, r\}, 10)$ *and* $(\{b, g\}, 10) \in \mathcal{S}_p(v_1, v_8)$ *are redundant because of* $(\{g\}, 5)$*, thus,* $\mathcal{S}_p(v_1, v_8)$ *is updated to* $\{(\{g\}, 5)\}$*. Similarly,* $\mathcal{S}(v_1, v_4)$ *is updated to* $\{(\{g\}, 3), (\{r\}, 8)\}$*.*

Theorem 17. *Given a road network* G*, Algorithm 11 constructs* LSD-Index *correctly.*

Proof. It is clear that the tree decomposition is correctly conducted. Then, we prove all the complete LSDS are correctly computed by induction. Obviously,

$\mathcal{G}_{n-2}$ contains only two vertices v_{n-1} and v_n with $\pi(v_{n-1}) = n-1$ and $\pi(v_n) = n$. According to Definition 22 and Lemma 28, after phase 1 finishes, $\mathcal{S}_p(v_{n-1}, v_n)$ is a complete LSDS. In the phase 2, for a vertex v with $\pi(v) < n-2$, suppose for any $u, w \in X(v) \setminus \{v\}$ such that $u \neq w$, $\mathcal{S}_p(w, u)$ is a complete LSDS. According to Lemma 29, by checking all possible label-constrained shortest distances from v to $u \in X(v) \setminus \{v\}$ in line 21-24, the final $\mathcal{S}_p(v, u)$ is a complete LSDS. Thus, all the complete LSDS are correctly computed when phase 2 finishes. Therefore, the theorem holds. □

Theorem 18. *Given a road network G, the time complexity of Algorithm 11 to construct the index is $O(n \cdot \omega^2 \cdot \rho^2)$.*

Proof. The time complexity of two operators LSDSJoin and LSDSPrune can be bounded by $O(\rho^2)$. In phase 1, n vertices are eliminated. For each eliminated vertex, at most $O(\omega^2)$ edges are generated. For each edge, two operators are invoked once. Hence, the time complexity of phase 1 is $O(n \cdot \omega^2 \cdot \rho^2)$. In phase 2, for each node, we have to compute $O(\omega)$ LSDS and each needs to invoke two operators $O(\omega)$ times, phase 2 can be done in $O(n \cdot \omega^2 \cdot \rho^2)$. Thus, the overall time complexity of Algorithm 11 is $O(n \cdot \omega^2 \cdot \rho^2)$. □

5.5.4 Shortest Path Restoration

The algorithms described in the previous section focus on computing the label-constrained shortest distance. By slightly modifying the index structure and query processing algorithm, we can easily retrieve the corresponding label-constrained shortest path.

Augmented LSD-Index. According to Definition 18, each entity $(L, d) \in \mathcal{S}(u, v)$ in LSD-Index corresponds to a path $p(u, v)$ in G. To restore the shortest path for a query, we first need to restore the path represented by (L, d). Revisiting the construction procedures of LSD-Index shown in Algorithm 11, there are two cases in which $(L, d) \in \mathcal{S}(u, v)$ is generated: (1) the original edge (u, v) in G; (2) operator LSDSJoin is applied on $\mathcal{S}_p(w, u)$ and $\mathcal{S}_p(w, v)$ (line 8 or line 23). For case (1), we do not store any additional information in $\mathcal{S}(u, v)$. For case (2), we store (w, id_u, id_v) besides (L, d) in $\mathcal{S}(u, v)$, where id_u (resp. id_v) is the identification of (L_u, d_u) (resp. (L_v, d_v)) in $\mathcal{S}(w, u)$ (resp. $\mathcal{S}(w, v)$) that leads to the generation of (L, d). With this additional information, we can restore the path $p(u, v)$ represented by (L, d) in $\mathcal{S}(u, v)$ as follows: (1) $p(u, v)$ is the original edge (u, v) in G; or (2) $p(u, v)$ can be obtained by concatenating $p(u, w)$ and $p(w, v)$ represented by (L_u, d_u) in $\mathcal{S}(w, u)$ and (L_v, d_v) in $\mathcal{S}(w, v)$, respectively, while $p(u, w)$ and $p(w, v)$ can be obtained recursively in the same way. Clearly, as the size of added information for each entity is constant, the space complexity of the augmented LSD-Index and the time complexity of the corresponding construction algorithm keep the same as that for LSD-Index.

Query processing. For query processing, the general framework is similar to Algorithm 9 but with additional path information. Specifically, we keep the shortest paths p (resp. p') from s (resp. t) to the vertices in X_{lca} by storing the vertex v and the corresponding $(L, d) \in \mathcal{S}(v, u)$ leading to the final $d_s^{\mathcal{L}}(u)$ in line 4 and line 10 of Algorithm 9, and concatenate p and p' through $w \in X_{\mathsf{lca}}$ which leads to the final shortest distance in line 11 of Algorithm 9. For the edges in p (resp. p') that are not the original edges of G (represents by $(L, d) \in \mathcal{S}(v, u)$), they can be restored by the method as discussed above. Given a $q = (s, t, \mathcal{L})$, if the returned shortest path p has τ edges, then, the extra time complexity to restore the path can be bounded by $O(\tau)$. Since the lower bound to answer q is $\Omega(\tau)$ and τ is generally very small compared with $h \cdot \omega \cdot \rho$, the time complexity of the query processing is the same as that of Algorithm 9.

5.5.5 Extension for Directed Road Networks

In previous sections, we assume the road networks are undirected . Our techniques can be extended to support directed road networks.

Indexing. For the index structure, the LSD-Index for directed road networks is similar to that for undirected road network with two differences: (1) for the tree decomposition, we extend MDE for the directed road networks as follows: it iteratively eliminates the vertex v with the minimum degree and connects any pair u, w of v's neighbors with directed edges after the elimination of v, and the other parts are the same. (2) for LSDS stored in each node T_G, we trivially extend the label-constrained distances defined in Definition 18 for directed road networks by using the paths with direction. And for each node $X(v)$, we precompute and store $\mathcal{S}{<}v, u{>}$ and $\mathcal{S}{<}u, v{>}$ for any $u \in X(v) \setminus \{v\}$. Here, we use $\mathcal{S}{<}v, u{>}$ to represent the LSDS from v to u extended for directed road networks for distinction. For the index construction algorithm, the whole framework is the same as Algorithm 11 except the directions of edges/paths need to be considered.

Query processing. The query processing procedure for directed road networks is similar to Algorithm 9. Given a query $q = (s, t, \mathcal{L})$, we first compute the lowest common ancestor X_{lca} of $X(s)$ and $X(t)$. After that, we compute the label-constrained shortest distances from s to vertices in X_{lca} and from these vertices to t. Finally, we can obtain the label-constrained shortest distance from s to t and consequently restore the label-constrained shortest path from s to t in the same way as discussed for the undirected road networks.

5.5.6 Handling Large Σ

Although our indexing techniques can significantly reduce the index size, Σ might be very large in some scenarios, which makes the index size still very large. In this section, we introduce how to extend our techniques to address this issue.

It has been widely observed that the labels in real-life graphs usually follows the power-law distribution [70]. Therefore, we treat the high frequent labels and low frequent labels in different ways. Let Σ_f be the set of high frequent labels in G. We create a set of virtual labels Σ_v by evenly partitioning the labels in $\Sigma \setminus \Sigma_f$ into $|\Sigma_v|$ groups and each virtual label represents the labels in each group, where $|\Sigma_f| + |\Sigma_v| \ll |\Sigma|$. In G, we replace the real label for each edge with the corresponding virtual label and construct the LSD-Index regarding $\Sigma_f \cup \Sigma_v$.

Given a query $q = (s, t, \mathcal{L})$, if $\mathcal{L} \subseteq \Sigma_f$, we use Algorithm 9 to answer the query directly. Otherwise, let $\mathcal{L}_f$ be the label set $\mathcal{L} \cap \Sigma_f$ and $\mathcal{L}_v$ be the virtual label set representing $\mathcal{L} \cap \{\Sigma \setminus \Sigma_f\}$. We compute $\mathsf{dist}_G^{\mathcal{L}_f}(s,t)$ and $\mathsf{dist}_G^{\mathcal{L}_f \cup \mathcal{L}_v}(s,t)$ following Algorithm 9 based on the LSD-Index. Obviously, $\mathsf{dist}_G^{\mathcal{L}_f}(s,t) \geq \mathsf{dist}_G^{\mathcal{L}}(s,t)$ and $\mathsf{dist}_G^{\mathcal{L}}(s,t) \geq \mathsf{dist}_G^{\mathcal{L}_f \cup \mathcal{L}_v}(s,t)$. Therefore, if $\mathsf{dist}_G^{\mathcal{L}_f}(s,t) = \mathsf{dist}_G^{\mathcal{L}_f \cup \mathcal{L}_v}(s,t)$, we obtain $\mathsf{dist}_G^{\mathcal{L}}(s,t)$. Otherwise, the shortest path may involve some edges with virtual labels in $\mathcal{L}_v$ but real labels not in $\mathcal{L}$. In this case, for index entries $(L_v, d_v) \in \mathcal{S}(u,v)$ that are used for obtaining $\mathsf{dist}_G^{\mathcal{L}_f \cup \mathcal{L}_v}(s,t)$ and contain virtual labels, we need to further check whether $d_v = \mathsf{dist}_G^{\mathcal{L}}(u,v)$, this can be achieved by exploring the neighbors w of u connected with labels in $\mathcal{L}$ and recursively computing $\mathsf{dist}_G^{\mathcal{L}}(w,v)$. If $d_v \neq \mathsf{dist}_G^{\mathcal{L}}(u,v)$, we use the refined $\mathsf{dist}_G^{\mathcal{L}}(u,v)$ instead and the correct final result can be obtained.

5.6 Parallel Index Construction

Although Algorithm 11 significantly reduces the time cost to construct LSD-Index compared with building the index directly based on the definition, it is still expensive for large road networks due to the inevitable LSDSJoin and LSDSPrune operations during the LSDS computation. In this section, we further improve the construction efficiency by parallelizing the LSDS computation.

Recall that the computation of LSDS contains the partial LSDS maintenance in phase 1 and top-down complete LSDS computation in phase 2. For the partial LSDS maintenance in phase 1, the computation of $\mathcal{S}_p(v,u)$ in $X(v)$ only depends on $\mathcal{S}_p(w,v)$ and $\mathcal{S}_p(w,u)$ in $X(w)$, where $X(w)$ is a descendant of $X(v)$ in the tree decomposition. For the top-down complete LSDS computation in phase 2, the computation of $\mathcal{S}_p(v,u)$ in $X(v)$ only depends on $\mathcal{S}_p(v,w)$ and $\mathcal{S}_p(w,u)$ in $X(w)$, where $X(w)$ is an ancestor of $X(v)$ in the tree decomposition. Hence, we define:

Definition 23. ***(Tree Decomposition Level)*** *Given a tree decomposition T of G, for a node $X(v)$, the tree decomposition level of $X(v)$, denoted by $l(X(v))$, is defined as $l(X(v)) =$*

$$\begin{cases} \min\{l(X(u))|X(u) \in X(v).\mathsf{children}\} + 1, & X(v).\mathsf{children} \neq \emptyset \\ 1, & X(v).\mathsf{children} = \emptyset \end{cases}$$

where $X(v)$.children represents the children of $X(v)$ in T.

As discussed above, if we compute the LSDS level by level based on Definition 23 (from bottom level to top level in phase 1 while from top level to bottom level in phase 2), then the LSDS computations related to the nodes at the same level has no dependence with each other, which means we can process the computations related to these nodes simultaneously with any extra costs.

Algorithm. Following the above idea, the parallel construction algorithm, LSD-Index-ParCons, is shown in Algorithm 12. LSD-Index-ParCons follows a similar framework to Algorithm 11. It first conducts the tree decomposition following MDE (line 1-8). During the decomposition, instead of maintaining the partial LSDS, it only records the vertex v leading to the update of $\mathcal{S}_p(u,w)$ in $\mathcal{D}(u,w)$

Algorithm 12: $\mathsf{LSD\text{-}Index\text{-}ParCons}(G)$

```
1  $G_0 \leftarrow G$; $T \leftarrow \emptyset$;
2  for $i \leftarrow 1$ to $n$ do
3      line 5-6 of Algorithm 11;
4      foreach $u, w \in \mathsf{nbr}(v, G_{i-1})$ do
5          insert $v$ into $\mathcal{D}(u, w)$;
6          line 9-10 of Algorithm 11;
7      $X(v) \leftarrow \{v\} \cup \mathsf{nbr}(v, G_{i-1})$;
8  line 15-18 of Algorithm 11;
9  for $i \leftarrow 1$ to $n$ do
10     $v \leftarrow$vertex with $\pi(\cdot) = i$;
11     if $X(v)$.children $= \emptyset$ then $l(X(v)) \leftarrow 1$;
12     else $l(X(v)) \leftarrow \min_{X(u) \in X(v).\mathsf{children}} l(X(u)) + 1$ ;
13 $l_{max} \leftarrow \max_{X(v) \in T} l(X(v))$;
   // Partial LSDS computation
14 for $i \leftarrow 1$ to $l_{max}$ do
15     for $X(v) \in T$ with $l(X(v)) = i$ in parallel do
16         for $u \in X(v)$ in parallel do
17             if $(v, u) \in G$ then
18                 $\mathcal{S}_p(v, u) \leftarrow \{(\ell((v, u)), \phi((v, u)))\}$;
19             else $\mathcal{S}_p(v, u) \leftarrow \emptyset$ ;
20             for $w \in \mathcal{D}(v, u)$ do
21                 $\mathcal{S}' \leftarrow \mathsf{LSDSJoin}(\mathcal{S}_p(w, v), \mathcal{S}_p(w, u))$;
22                 $\mathcal{S}_p(v, u) \leftarrow \mathsf{LSDSPrune}(\mathcal{S}_p(v, u) \cup \mathcal{S}')$;
   // Complete LSDS computation
23 for $i \leftarrow l_{max}$ to 1 do
24     for $X(v) \in T$ with $l(X(v)) = i$ in parallel do
25         for $u \in X(v) \setminus \{v\}$ in parallel do
26             for $w \in X(v) \setminus \{v, u\}$ do
27                 $\mathcal{S}' \leftarrow \mathsf{LSDSJoin}(\mathcal{S}_p(v, w), \mathcal{S}_p(w, u))$;
28                 $\mathcal{S}_p(v, u) \leftarrow \mathsf{LSDSPrune}(\mathcal{S}' \cup \mathcal{S}_p(v, u))$;
```

(line 5). After finishing the tree decomposition, it computes the tree decomposition level for each nodes following Definition 23 (line 9-13). Then, it conducts partial LSDS computation in a bottom-up manner (line 14-22) and the complete LSDS computation in a top-down manner (line 23-28). For the nodes at a specific level, they are processed simultaneously (line 15-16, line 24-25). When the algorithm finishes, LSD-Index is correctly constructed, which can be proved similar to Algorithm 11.

5.7 Experiments

In this section, we compare our algorithms with the state-of-the-art methods for label-constrained shortest path queries. All experiments are conducted on a machine with an Intel Xeon 2.5GHz CPU (40 cores) and 256 GB main memory running Linux.

Datasets. We use eight publicly available real road networks from DIMACS [1]. In each road network, vertices represent intersections between roads, edges correspond to roads or road segments, the weight of an edge is the physical distance between two vertices, and the label of an edge represents its road types. The road types of these road netowrks can be divided into four main categories: (1) A1, Primary Highway With Limited Access; (2) A2, Primary Road Without Limited Access; (3) A3, Secondary and Connecting Road; (4) A4, Local, Neighborhood, and Rural Road. The road types follows the power-law distribution. Since different datasets contain different number of labels (from 18 $\sim$ 32), in our experiments (except Exp-6), for the purpose of controlling variables and keeping the distribution of labels as same as possible, we refine the labels of each dataset and make each dataset contain 10 labels using the following method: for the labels in each main category, we sort the labels in the increasing order of their frequency and merge two labels with similar frequency as one, we continue this process until only 10 labels remains. Table 5.2 provides the details about these datasets. Table 5.2 shows the value of h and ω of the tree decomposition for

[1] http://users.diag.uniroma1.it/challenge9/download.shtml

Table 5.2: Datasets used in Experiments

Dataset	Description	n	m	$\|\Sigma\|$	h	ω	ρ	ρ_{avg}	Indexing Time (S)	Indexing Time (P)	Index Size
NY	New York City	264,346	733,846	10	717	126	28	1.29	36.74s	6.97s	34.52 MB
COL	Colorado	435,666	1,057,066	10	477	133	32	1.12	24.27s	4.81s	36.45 MB
FLA	Florida	1,070,376	2,712,798	10	643	82	38	1.19	34.03s	5.76s	95.91 MB
CAL	California	1,890,815	4,657,742	10	834	177	31	1.11	92.04s	15.93s	161.21 MB
EST	Eastern USA	3,598,623	8,778,114	10	1,366	240	28	1.17	258.63s	39.79s	327.17 MB
WST	Western USA	6,262,104	15,248,146	10	1,450	299	35	1.11	343.51s	45.18s	546.95 MB
CTR	Central USA	14,081,816	34,292,496	10	2,342	540	94	1.31	5,959.00s	741.12s	1.45 GB
USA	Full USA	23,947,347	58,333,344	10	2,886	570	136	1.27	7,152.18s	903.94s	2.37 GB

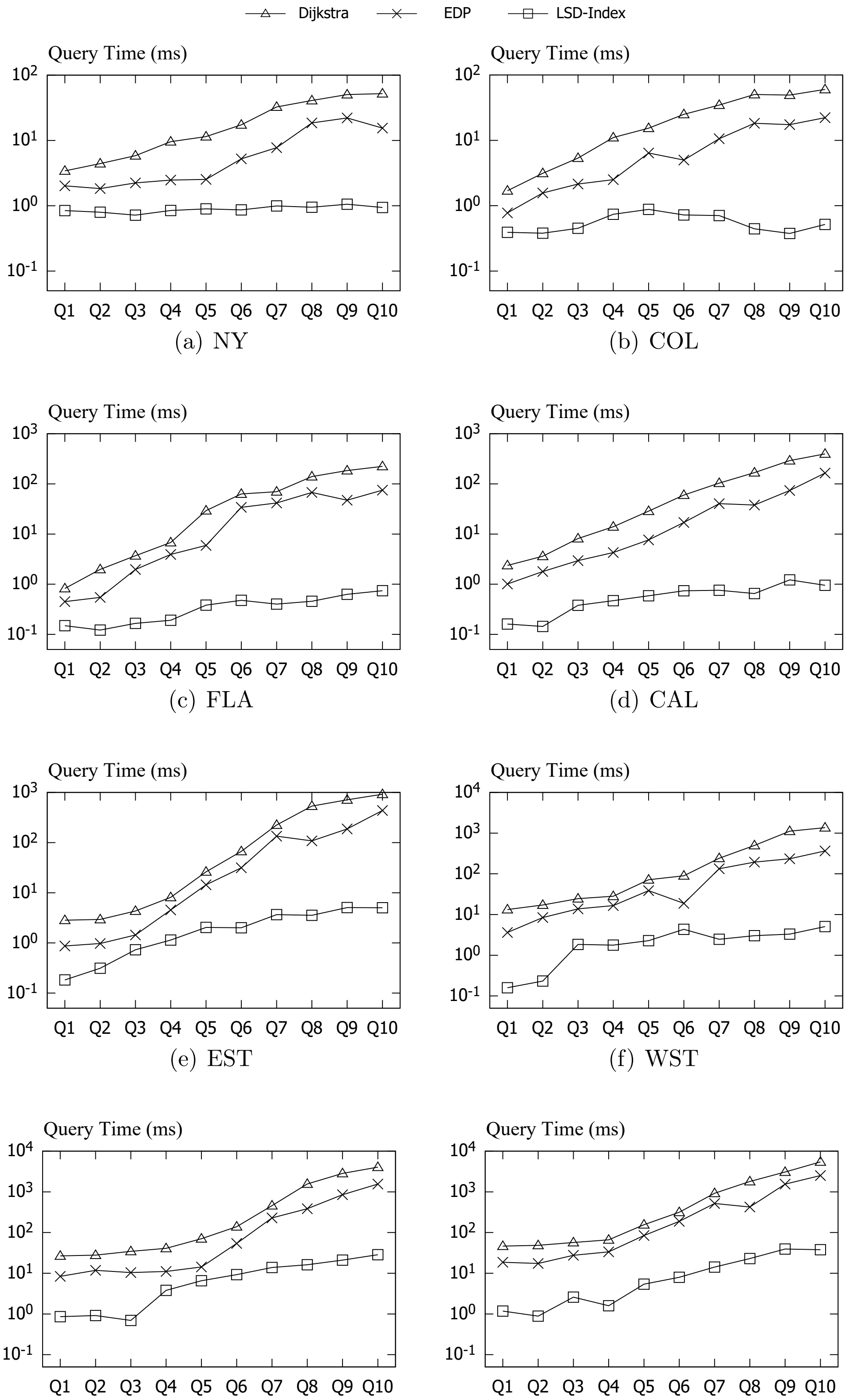
each road network and it is clear that h and ω are small in practice. Table 5.2 also shows the value of ρ and ρ_{avg} of LSD-Index for each road network, where ρ_{avg} represents the average size of LCDS in LSD-Index. It is clear that ρ and ρ_{avg} are much smaller than h and ω in practice.

Algorithms. We implement and compare tne following algorithms. All the algorithms are implemented in C++ and compiled in GCC 8.3.1 with -O3 flag. We adopt OpenMP to implement our parallel algorithm.

- Dijkstra: direct online search algorithm using the Dijkstra's algorithm following the edges with labels in given $\mathcal{L}$.
- EDP: The state-of-the-art algorithm for label-constrained shortest path queries, which is introduced in Section 5.3.
- LSD-Index: Our proposed algorithms include query processing algorithm (Algorithm 9), index construction algorithm (Algorithm 11), and parallel index construction algorithm (Algorithm 12).

For EDP, we implement all the optimization techniques mentioned in [41]. Since EDP builds its index gradually during the query processing, for fairness, we generate random queries to warm up EDP as [41] until its cache size becomes stable or reaches the memory limit (20GB) before our experiments.

Exp-1: Efficiency when varying query distance. In this experiment, we evaluate the query efficiency of the algorithms by varying the label-constrained shortest distance between the source vertex and target vertex in the query. We randomly generate 10 groups of queries $Q_1, \ldots, Q_{10}$ and each group contains 1000 queries. For each query $q = (s, t, \mathcal{L})$ in group i, the label-constrained distance between s and t regarding $\mathcal{L}$ ranging from $(\frac{\delta}{1\text{ km}})^{\frac{i-1}{10}}$ to $(\frac{\delta}{1\text{ km}})^{\frac{i}{10}}$ kilometers, where δ is the longest distance between any two vertices in the road network. And $\mathcal{L}$



(a) NY

(b) COL

(c) FLA

(d) CAL

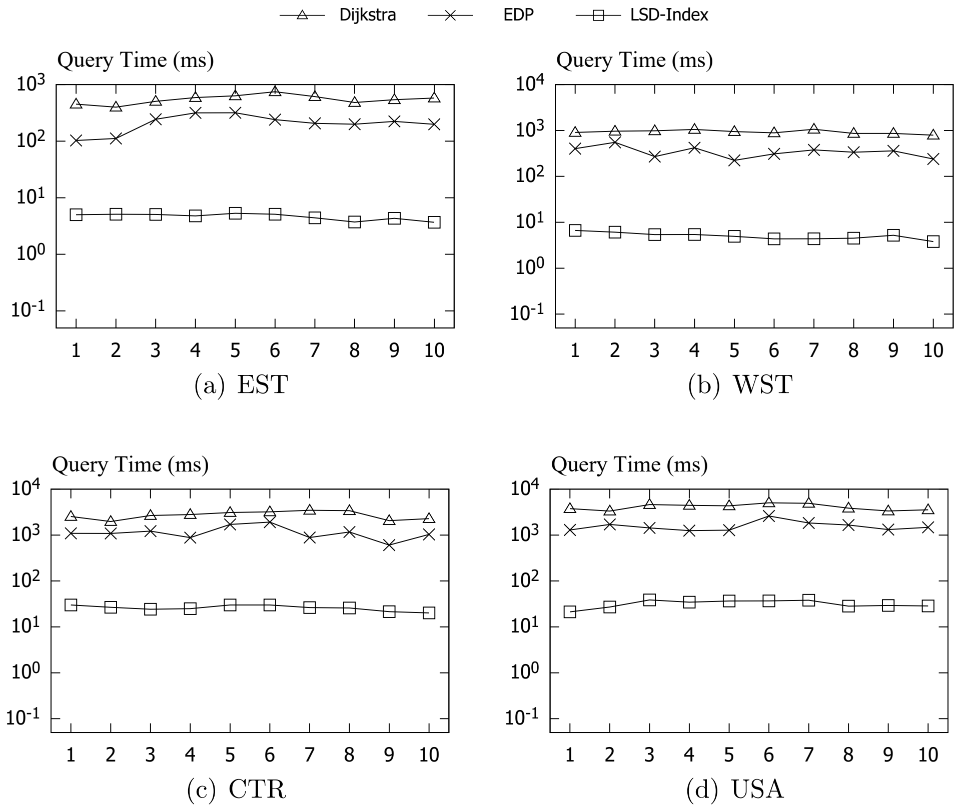
(e) EST

(f) WST

(g) CTR

(h) USA

Figure 5.8: Query Processing Time (Varying Query Distance)



(a) EST (b) WST

(c) CTR (d) USA

Figure 5.9: Query Processing Time (Varying $|\mathcal{L}|$)

is set as minimum edge label set which can make the label-constrained distance between s and t satisfy the above condition. Figure 5.8 shows the average query processing time for queries in each group on the eight datasets.

As shown in Figure 5.8, the query processing time of all the algorithms increases when the distance increases. This is because as the distance between s and t increases, more vertices or nodes have to be explored. Moreover, EDP is always faster than Dijkstra while LSD-Index is much faster than EDP and the performance gap enlarges as the distance increases. For example, on dataset FLA (Figure 5.8 (c)), the query processing times for Q_1 of all three methods are within 1ms, while for Q_{10}, the query processing time of Dijkstra, EDP and our algorithm are 223.6ms, 74.94ms and 0.74ms, respectively, which means LSD-Index achieves 2 order of magnitude speedup compared with EDP. The reasons are Dijkstra and EDP have to explore many vertices in the road networks while LSD-Index only needs to visit vertices in the nodes along the tree decomposition, which is much less than that of Dijkstra and EDP.

Exp-2: Efficiency when varying $|\mathcal{L}|$. In this experiment, we evaluate the query efficiency of the algorithms by varying $|\mathcal{L}|$ of the queries. To do this, we randomly generate 10 groups of queries and each group contains $1,000$ queries. For each query $q = (s, t, \mathcal{L})$ in group i, $\mathcal{L}$ is set as an edge label set with $|\mathcal{L}| = i$ such that s can reach t following the edges with edge label in $\mathcal{L}$. We record the average query processing time for queries in each group and the results for the four large datasets is demonstrated in Figure 5.9, the results on the remaining datasets show similar trends.

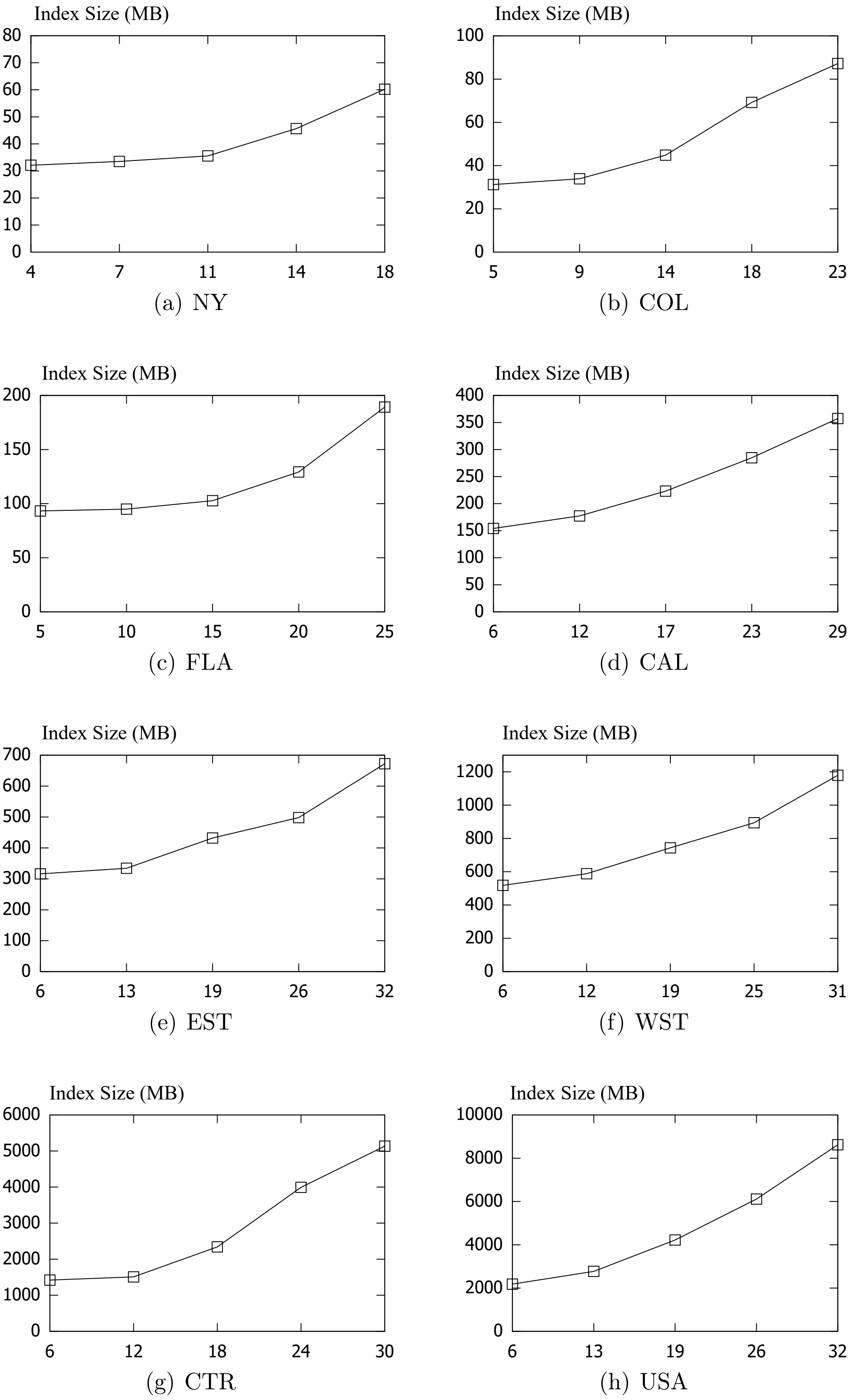
Based on the results, we can observe that: (1) LSD-Index always outperforms Dijkstra and EDP by at least an order of magnitude. The reasons are the same as discussed in Exp-1. (2) the average processing time of all the algorithms keeps stable when we vary $|\mathcal{L}|$. For Dijkstra and EDP, when $|\mathcal{L}|$ is small, the label-constrained shortest distance between s and t regarding $\mathcal{L}$ is large generally, which implies that the traversal on the road network is long. As $|\mathcal{L}|$ increases, the label-constrained shortest distance between s and t regarding $\mathcal{L}$ becomes small, but the number of edges with edge label in $\mathcal{L}$ increases as well. As a result, the number of explored vertices and edges during the query processing keep similar, which explains why the query processing time keep the stable for Dijkstra and EDP. For LSD-Index, because LSD-Index processes the queries based on the tree decomposition, the whole processing is nearly independent with $\mathcal{L}$. Therefore, the query processing time of LSD-Index keeps stable when we vary $|\mathcal{L}|$ as well.

Exp-3: Indexing time. Table 5.2 presents the time to construct LSD-Index for each dataset, including the sequential construction algorithm and the parallel construction algorithm (running with 32 threads). For the first six road networks, the index can be constructed within 6 minutes even for the sequential construction algorithm. However, the sequential construction algorithm needs 1.5–2 hours to complete the index construction for CTR and USA. Considering the size of these two datasets, the indexing time is acceptable but not highly satisfactory. On the other hand, for the parallel construction algorithm, it takes less than 60 seconds to construct the index for the first six datasets and less than $1,000$ seconds to construct the index for the USA dataset. As shown in the results, our proposed algorithms can efficiently construct LSD-Index in practice, especially the parallel construction algorithm.

Exp-4: Index size. The size of LSD-Index for each road network is shown in Table 5.2. As shown in Table 5.2, the index sizes of the first six road networks are within 1 GB, and even for the whole USA road network, the index size is only 2.37 GB. Considering USA dataset is around 0.8 GB in size, 2.37 GB is still small. We omit the index size of EDP because its index size varies by different cache strategies. In our experimental setting, we set the index size limit for EDP to 20 GB and the index sizes for most of the large road networks (WST, CTR, USA) in our setting are beyond 10 GB. From the results, it is clear that LSD-Index is a compact index structure.

Exp-5: Index size when varying $|\Sigma|$. In this experiments, we evaluate the index size when varying $|\Sigma|$. For each datasets, we set the number of labels from $\lceil \frac{|\Sigma|}{5} \rceil$ to $|\Sigma|$. For the smaller label set, we generate them using the similar method mentioned before: we sort the original labels in each main category according to their frequency and merge label labels with similar frequency until the number of labels reach the required size.

As shown in Figure 5.10, as $|\Sigma|$ increases, the index sizes increases as well. This is because that the larger $|\Sigma|$ is, the more information needs to be stored



(a) NY

(b) COL

(c) FLA

(d) CAL

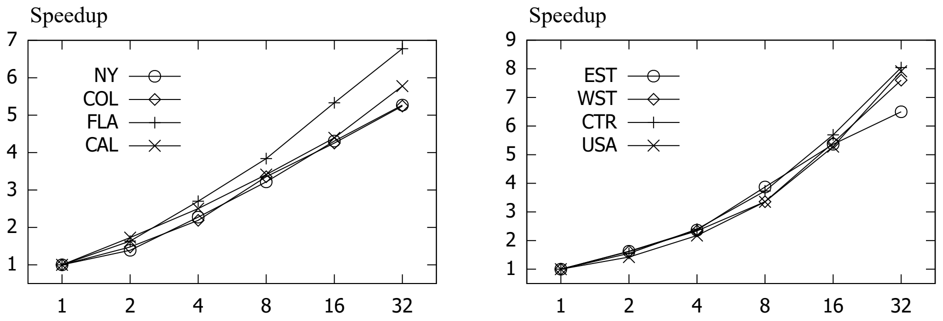
(e) EST

(f) WST

(g) CTR

(h) USA

Figure 5.10: Index Size (Varying $|\Sigma|$)



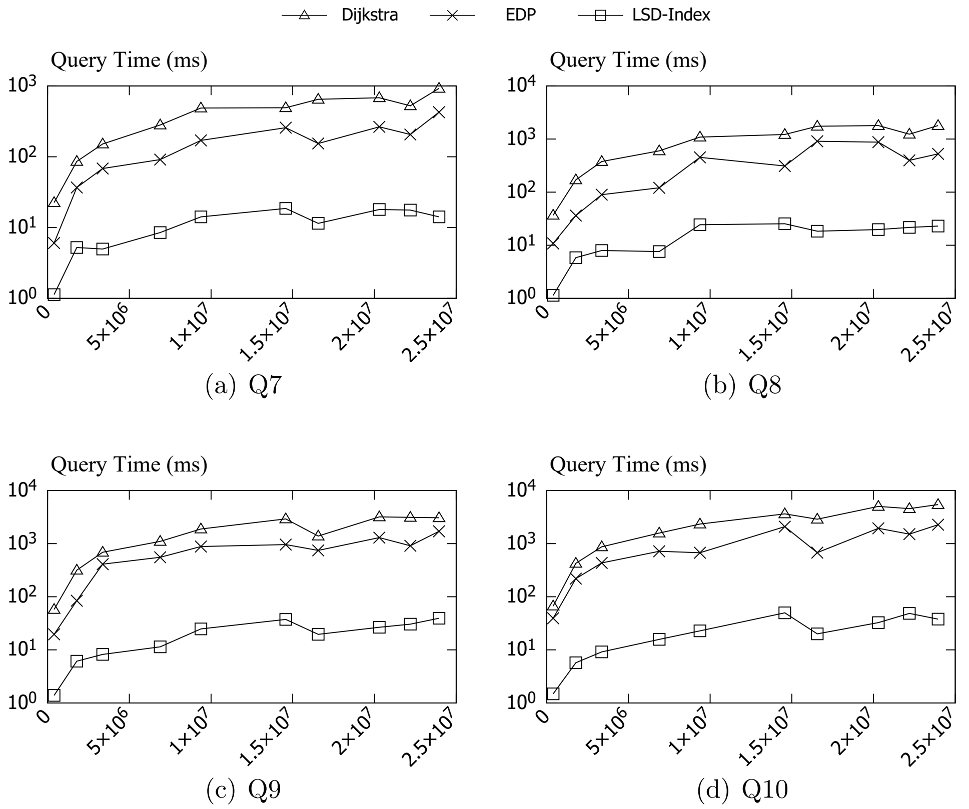
(a) #thread (b) #thread

Figure 5.11: Parallel Indexing Speedup (Varying #Thread)

in the index. However, even for the largest road network USA, the largest index size is 8.6GB when $|\Sigma| = 32$, which is only 10 times the size of the dataset (around 0.8GB). The experimental results confirm that LSD-Index is a compact index structure.

Exp-6: Scalability of parallel indexing algorithm. In this experiment, we evaluate the scalability of the parallel index construction algorithm (Algorithm 12) by varying the number of available threads from 1 to 32. The results are shown in Figure 5.11. As shown in Figure 5.11, the speedup increases nearly linearly as the number of available threads increases. For the small road networks, the algorithm achieves 5x-7x speedup when #thread is 32 (Figure 5.11 (a)). For the large road networks, it achieves 6x-8x speedup when #thread is 32 (Figure 5.11 (b)). The results demonstrate that our parallel index construction algorithm scales well in practice.

Exp-7: Scalability when varying dataset size. In this experiment, we evaluate the scalability of the three algorithms. To achieve this goal, we divide the USA dataset into 10 × 10 grids and generate 10 sub road networks $G_1 \subseteq G_2 \subseteq ... \subseteq G_{10}$ by choosing the 1 × 1, 2 × 2,...,10 × 10 grids in the middle of the USA dataset. For each sub road network, we generate 10 groups of queries and each group contains 1,000 queries using the same method used in Exp-1. We



(a) Q7 (b) Q8

(c) Q9 (d) Q10

Figure 5.12: Query Processing Time (Varying Dataset Size)

report the average query processing time for the queries in each group and show the results of Q_7–Q_{10} when varying the dataset size in Figure 5.12. The results of $Q_1 - Q_6$ show similar trends but are omitted. The x-axis for each figure is the number of vertices for each dataset.

Figure 5.12 shows that EDP is more efficient than Dijkstra while LSD-Index is much faster than EDP when varying the dataset size, which is consistent with the results shown in Exp-1 and Exp-2. Moreover, as the dataset size increases, the query processing time for Dijkstra and EDP has an obviously increasing trend while the query processing time for LSD-Index is relatively more scale-independent to the data size. This is because Dijkstra and EDP have to traverse the road network to answer a query. As the dataset size increases, more vertices and edges have to be explored generally. On the other hand, LSD-Index processes the queries based on the tree decomposition and the processing involves only a few nodes.

5.8 Limitation of LSD-Index

Since the LSD-Index is based on tree decomposition, its size is linearly proportional to the treewidth ω and treeheight ρ according to Theorem 15. For the road networks studied in this chapter, their treewidth and treeheight are typically small. However, for some other types of graphs, the treewidth and treeheight could be very large. Maniu et al. [65] have done an experimental study on the real-world graph datasets. From their study, infrastructure networks, including road networks, public transportation networks, and power grids, have small treewidth values, with upper bounds smaller than 600. Conversely, other types of graphs may have large tree width. For example, scale free networks like social networks and web graphs are known to have a large dense core and a sparse tree-like fringe [64]. The core part is hard to decompose. LSD-Index may not be suitable for these graphs.

5.9 Chapter Summary

In this Chapter, we study the label-constrained shortest path query problem on road networks. We devise a novel index structure named LSD-Index based on the tree decomposition. With LSD-Index, we propose an efficient query processing algorithm to answer the queries. Moreover, we also present efficient index construction algorithms. The experimental results demonstrate the efficiency of our proposed algorithms. For future work, we are interested in extending our work to dynamic graphs by devising efficient index maintenance algorithm for graph label/vertex/edge updates.

Chapter 6

EPILOGUE

In this thesis we study the efficient computation of paths in massive graphs. The main contributions of this thesis are

- *Scalable Indexing Schema.* We develop novel labeling methods to produce the same indexes as TOL on distributed graphs. To overcome the limitation that TOL cannot be executed in parallel, we resort to finding the backward label set of each vertex. We propose to use a filtering-and-refinement framework to find backward label sets. Using this framework, we design new labeling algorithms and further improve the efficiency with batch labeling optimization.

- *Extensive Studies on Shortest-Path Queries.* We conduct extensive studies on shortest-path queries on complex networks. To support shortest-path queries, we extend the distance query processing methods PLL and CTL for shortest-distance queries. To reduce the space cost required for extensions, we propose MLL and extend it for weighted and directed graphs.

- *Compact Index.* We study the label-constrained shortest path query problem on road networks. We devise a novel index structure LSD-Index based on the tree decomposition. With LSD-Index, we propose an efficient query processing algorithm to answer the queries. Moreover, we also present efficient index construction algorithms.

The following are the problems that need further studies and are the focus of our future research.

- *Maintain Index for Dynamic Graphs.* Real-world graphs are typically dynamic. An important yet well-studied problem is how to efficiently maintain the index when the graph changes. Hence, one future research direction is to study the properties of the index approaches and design efficient index maintenance algorithms for path computation.
- *Exploit Multi-Core and Distributed System.* In this thesis, we use distributed and multi-core systems to accelerate index construction. The computation nodes in modern distributed systems usually contain multiples core. Utilizing the characteristics of both distributed and multi-core systems to accelerate index construction is still an open problem.
- *Exploit Heuristic Methods for Path Computation.* The index-based methods incur high indexing costs for some query problems, e.g., label-constrained shortest path queries on complex networks. As the size of real-world graphs grows, heuristic methods, like the pruned landmark, bi-directional traversal and A*, could be more suitable and is a promising future research direction.

BIBLIOGRAPHY

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

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of SEA*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011.

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

[4] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *ACM SIGMOD Record*, volume 18, pages 253–262. ACM, 1989.

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

[6] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance

queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2013.

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

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

[9] B. Awerbuch. A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3):147–150, 1985.

[10] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. 2016.

[11] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2012.

[12] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In G. H. Gonnet, D. Panario, and A. Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[13] A. Berry, P. Heggernes, and G. Simonet. The minimum degree heuristic and the minimal triangulation process. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 58–70. Springer, 2003.

[14] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[15] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4-5):175–308, 2006.

[16] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[17] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[18] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[19] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[20] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

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

[22] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st international conference on Very large data bases*, pages 493–504. VLDB Endowment, 2005.

[23] X. Chen, Y. Peng, S. Wang, and J. X. Yu. Dlcr: efficient indexing for label-constrained reachability queries on large dynamic graphs. *Proceedings of the VLDB Endowment*, 15(8):1645–1657, 2022.

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

[25] Z. Chen, A. W.-C. Fu, M. Jiang, E. Lo, and P. Zhang. P2h: efficient distance querying on road networks by projected vertex separators. In *Proceedings of the 2021 International Conference on Management of Data*, pages 313–325, 2021.

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

[27] J. Cheng, J. X. Yu, X. Lin, H. Wang, and S. Y. Philip. Fast computation of reachability labeling for large graphs. In *International Conference on Extending Database Technology*, pages 961–979. Springer, 2006.

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

[29] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT press, 2009.

[31] L. d. F. Costa, F. A. Rodrigues, G. Travieso, and P. R. Villas Boas. Characterization of complex networks: A survey of measurements. *Advances in physics*, 56(1):167–242, 2007.

[32] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[33] E. Djikstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[34] W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. *Proceedings of the VLDB Endowment*, 5(11):1304–1316, 2012.

[35] M. Farhan, Q. Wang, Y. Lin, and B. Mckay. A highly scalable labelling approach for exact distance queries in complex networks. *arXiv preprint arXiv:1812.02363*, 2018.

[36] X. Feng, L. Chang, X. Lin, L. Qin, and W. Zhang. Computing connected components with linear communication cost in pregel-like systems. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 85–96. IEEE, 2016.

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

[38] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, volume 5, pages 156–165. Citeseer, 2005.

[39] R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. *ALENEX/ANALC*, 4:100–111, 2004.

[40] R. Halin. S-functions for graphs. *Journal of geometry*, 8(1-2):171–186, 1976.

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

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

[43] H. Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems (TODS)*, 15(4):558–598, 1990.

[44] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *arXiv preprint arXiv:1403.0779*, 2014.

[45] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *Proceedings of SIGMOD*, pages 123–134. ACM, 2010.

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

[47] R. Jin, N. Ruan, B. You, and H. Wang. Hub-accelerator: Fast and exact shortest path computation in large social networks. *arXiv preprint arXiv:1305.0507*, 2013.

[48] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *Proceedings of the VLDB Endowment*, 6(14):1978–1989, 2013.

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

[50] D. B. Johnson. A note on dijkstra's shortest path algorithm. *Journal of the ACM (JACM)*, 20(3):385–388, 1973.

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

[52] J. Kim and T. Wilhelm. What is a complex graph? *Physica A: Statistical Mechanics and its Applications*, 387(11):2637–2652, 2008.

[53] A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. *Electron. Notes Discret. Math.*, 8:54–57, 2001.

[54] J. Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[55] K. Lakhotia, R. Kannan, Q. Dong, and V. Prasanna. Planting trees for scalable and efficient canonical hub labeling. *Proceedings of the VLDB Endowment*, 13(4).

[56] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187, 2005.

[57] J. Leskovec and A. Krevl. Snap datasets: Stanford large network dataset collection, 2014.

[58] H. Li, Y. Ge, R. Hong, and H. Zhu. Point-of-interest recommendations: Learning potential check-ins from friends. In *Proceedings of SIGKDD*, pages 975–984. ACM, 2016.

[59] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin. Scaling distance labeling on small-world networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1060–1077, 2019.

[60] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin. Scaling up distance labeling on graphs with core-periphery properties. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1367–1381, 2020.

[61] Y. Li, S. George, C. Apfelbeck, A. M. Hendawi, D. Hazel, A. Teredesai, and M. H. Ali. Routing service with real world severe weather. In Y. Huang, M. Schneider, M. Gertz, J. Krumm, and J. Sankaranarayanan, editors, *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, pages 585–588. ACM, 2014.

[62] Y. Li, M. L. Yiu, N. M. Kou, et al. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11(4):445–457, 2017.

[63] Y. Liu, T. Pham, G. Cong, and Q. Yuan. An experimental evaluation of point-of-interest recommendation in location-based social networks. *Proc. VLDB Endow.*, 10(10):1010–1021, 2017.

[64] T. Maehara, T. Akiba, Y. Iwata, and K.-i. Kawarabayashi. Computing personalized pagerank quickly by exploiting graph structures. *Proceedings of the VLDB Endowment*, 7(12):1023–1034, 2014.

[65] S. Maniu, P. Senellart, and S. Jog. An experimental study of the treewidth of real-world graph data. In *22nd International Conference on Database Theory (ICDT 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[66] J. Maue, P. Sanders, and D. Matijevic. Goal-directed shortest-path queries

using precomputed cluster distances. *Journal of Experimental Algorithmics (JEA)*, 14:3–2, 2010.

[67] E. Nuutila. Efficient transitive closure computation in large digraphs. 1998.

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

[69] D. Ouyang, L. Yuan, L. Qin, L. Chang, Y. Zhang, and X. Lin. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *Proc. VLDB Endow.*, 13(5):602–615, 2020.

[70] Y. Peng, Y. Zhang, X. Lin, L. Qin, and W. Zhang. Answering billion-scale label-constrained reachability queries within microsecond. *Proc. VLDB Endow.*, 13(6):812–825, 2020.

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

[72] N. Robertson and P. D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[73] N. Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[74] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[75] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal*, 29(2):595–618, 2020.

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

[77] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *International Conference on Extending Database Technology*, pages 237–255. Springer, 2004.

[78] C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)*, 46(4):1–31, 2014.

[79] J. Su, Q. Zhu, H. Wei, and J. X. Yu. Reachability querying: Can it be even faster? *IEEE Transactions on Knowledge and Data Engineering*, 29(3):683–697, 2016.

[80] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[81] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 845–856, 2007.

[82] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[83] L. D. J. Valstar, G. H. L. Fletcher, and Y. Yoshida. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of SIGMOD*, pages 345–358. ACM, 2017.

[84] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 ACM*

SIGMOD International Conference on Management of data, pages 913–924. ACM, 2011.

[85] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 75–75. IEEE, 2006.

[86] H. Wang, J. Li, J. Luo, and H. Gao. Hash-base subgraph query processing method for graph-structured xml documents. *Proceedings of the VLDB Endowment*, 1(1):478–489, 2008.

[87] Y. Wang, Q. Wang, H. Koehler, and Y. Lin. Query-by-sketch: Scaling shortest path graph queries on very large networks. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1946–1958, 2021.

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

[89] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: An independent permutation labeling approach. *Proceedings of the VLDB Endowment*, 7(12):1191–1202, 2014.

[90] Wikipedia. Expressways of china, 2021.

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

[92] G. Xu and Y. Xu. *GPS: theory, algorithms and applications*. Springer, 2016.

[93] J. Xu, F. Jiao, and B. Berger. A tree-decomposition approach to protein structure prediction. In *2005 IEEE Computational Systems Bioinformatics Conference (CSB'05)*, pages 247–256. IEEE, 2005.

[94] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In A. Gangemi, S. Leonardi, and A. Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 1307–1317. ACM, 2015.

[95] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1601–1606. ACM, 2013.

[96] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010.

[97] J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. Springer, 2010.

[98] T. Zhang, Y. Gao, L. Chen, W. Guo, S. Pu, B. Zheng, and C. S. Jensen. Efficient distributed reachability querying of massive temporal graphs. *The VLDB Journal*, 28(6):871–896, 2019.

[99] T. Zhang, Y. Gao, C. Li, C. Ge, W. Guo, and Q. Zhou. Distributed reachability queries on massive graphs. In *International Conference on Database Systems for Advanced Applications*, pages 406–410. Springer, 2019.

[100] J. Zhou, S. Zhou, J. X. Yu, H. Wei, Z. Chen, and X. Tang. Dag reduction: Fast answering reachability queries. In *Proceedings of the 2017 ACM*

International Conference on Management of Data, pages 375–390. ACM, 2017.

[101] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1323–1334. ACM, 2014.

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

[103] L. Zou, K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. Efficient processing of label-constraint reachability queries in large graphs. *Inf. Syst.*, 40:47–66, 2014.