Efficient index structures for reachability and shortest path queries

Wang, Sibor

2016

Wang, S. (2016). Efficient index structures for reachability and shortest path queries.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/68898>

<https://doi.org/10.32657/10356/68898>

Efficient Index Structures for Reachability and Shortest Path Queries

A thesis submitted
for a degree of Doctor of Philosophy
by

Sibo WANG

School of Computer Engineering
Nanyang Technological University

Dec 2015

Abstract

Graphs are a fundamental data structure to represent objects and their relations in various domains, e.g., social science, citation analysis, web link analysis, and navigation systems. Reachability and shortest path queries are two types of primitive and well-studied graph queries. In this thesis, we investigate on how to build efficient index structures to answer the reachability queries and two variants of the shortest path queries. The first variant of the shortest path query considers the shortest path in timetable graphs, where it requires the consideration of spatiotemporal constraints between different edges. The second variant aims to provide the shortest path under an additional cost constraint. These problems find many real world applications, and are challenging to achieve high efficiency on sizable graphs.

First, we study the reachability queries on directed graphs. Given a directed graph G , a source s , and a target t , a reachability query asks whether there exists a path from s to t in G . The state-of-the-art solutions for answering reachability queries are a class of labelling schemes referred to as *Hierarchical Hub Labelings (HHL)*. One distinct characteristic of *HHL* is that any *HHL* index implicitly or explicitly defines a total order of vertices in the graph, and the performance of the index is solely decided by the ordering. Several heuristic approaches have been proposed for vertex ordering in *HHL*, but none of them provides any worst-case guarantee. We present a novel study on the vertex ordering in *HHL*, and devise a polynomial-time algorithm for vertex ordering that yields an *HHL* index whose size is at most $O(\sqrt{n})$ times the optimal one. Motivated by our theoretical study, we design a new heuristic for vertex ordering, and show that it leads to an *HHL* index with superior practical performance for reachability queries.

Besides, identifying fast routes in public transportation networks is an important problem with applications in map services and navigation systems. In public transportation networks, any traversal between different network locations relies on transportation services (e.g., buses, subways, and ferries.) that run on fixed routes with pre-defined schedules. A public transportation network can often be modeled as a timetable graph where (i) each node represents a station; and (ii) each directed edge $\langle u, v \rangle$ is associated with a timetable that records the departure (resp. arrival) time of each vehicle at station u (resp. v). There are several types of shortest path queries on timetable graphs. We study three types of shortest path queries for route planning that have been extensively studied on timetable graphs: *earliest arrival path (EAP)*, *latest departure path (LDP)*,

and *shortest duration path (SDP)* queries. These three types of queries aim to find a path that arrives at a place as early as possible, departs as late as possible but arrives on time, and has the shortest travel time, respectively. We present *Timetable Labelling (TTL)*, an efficient indexing technique for shortest path queries on timetable graphs. The basic idea of *TTL* is to associate each node u with a set of labels, each of which records the shortest travel time from u to some other node v given a certain departure time from u ; such labels would then be used during query processing to improve efficiency. In addition, we propose query algorithms that enable *TTL* to support EAP, LDP, and SDP queries, and investigate how we reduce the space consumption of *TTL* with advanced preprocessing and label compression methods. By conducting an extensive set of experiments on real world datasets, we demonstrate that *TTL* significantly outperforms the states of the art in terms of query efficiency, while incurring moderate preprocessing and space overheads.

Finally, *constrained shortest path (CSP)* is a variant of the shortest path problem with an additional total cost constraint. Specifically, in a CSP query, each edge in the road network is associated with both a length and a cost, e.g., travel distance and time. Given an origin s , a destination t , and a cost constraint θ , the goal is to find the path from s to t that minimizes its total length, while satisfying that its total cost does not exceed θ . Deriving the exact answer for a CSP query has been proven to be NP-hard, and the majority of previous work focuses on approximate solutions. We propose *COLA*, the first practical solution for approximate CSP processing on large road networks. *COLA* exploits the facts that a road network can be effectively partitioned, and that there exists a relatively small set of landmark vertices that commonly appear in CSP results. Meanwhile, *COLA* includes both an effective indexing scheme for partition boundary vertices, and an efficient on-the-fly algorithm called α -Dijk for path computation within a partition. Extensive experiments demonstrate that on continent-sized networks with tens of millions of vertices, *COLA* answers an approximate CSP query in sub-second time, whereas existing methods take hours. Interestingly, even without an index, the α -Dijk algorithm in *COLA* still outperforms previous solutions by more than an order of magnitude.

Acknowledgments

First and foremost, I would like to express the deepest gratitude to my advisor, Prof. Xiaokui Xiao, for supporting me throughout my PhD research. His continuous encouragement and limitless patience, along with his enthusiasm for research and knowledgeable insight, were irreplaceable in bringing this thesis to fruition. If I am an improved researcher in computer science and a better person through my PhD study, it is because of his positive influence on my life.

I would also like to thank all my collaborators: Prof. Yin Yang, Prof. Shuigeng Zhou, Dr. Andy Diwen Zhu, Dr. Wenqing Lin, Dr. Chun-Hee Lee, and Yi Yang. It is my great honor to work with them and learn from them. The suggestions from and discussions with them help me be better equipped for research.

Besides, I thank my colleges in Data Management and Analytics Graduate Lab (DMAL) for their companionship and insightful discussions. They include Jun Zhang, Youze Tang, Ning Chen, Boshen Zhang, Zongyang Ma, Yanchen Shi, and Prof. Gao Cong. I will always cherish my enjoyable experience with the DMAL group members.

Meanwhile, I thank my parents for their unconditional love and supports to me, which gives me endless energy and let me continue to advance.

Last but not the least, I thank my wife Wendy Lu Weng, who has been supportive, encouraging, and considerate while I have been trying to finish up my thesis.

Contents

Abstract	i
Acknowledgments	iii
List of figures	viii
List of Tables	x
1 Introduction	1
1.1 Problems and Motivations	2
1.2 Approaches and Contributions	4
1.3 Thesis Organization	6
2 Literature Survey	8
2.1 Reachability Query	8
2.1.1 2-Hop Labelling	9
2.1.2 Other Techniques for Reachability Queries	11
2.2 Conventional Shortest Path Queries	12
2.2.1 Dijkstra’s Traversal	12
2.2.2 Hierarchical Hub Labelling	13
2.2.3 Contraction Hierarchies	14
2.2.4 Graph Partitioning	16
2.3 Shortest Path Queries on Timetable Graphs	17
2.3.1 State of the Art	18
2.3.2 Other Techniques for Shortest Path Queries on Timetable Graphs	20
2.4 Constrained Shortest Path	21
2.4.1 State of the Art	21
2.4.2 Other Techniques for Constrained Shortest Path	24
2.5 Summary	25

3	Reachability Query	27
3.1	Preliminaries	28
3.1.1	Problem Definition	28
3.1.2	Hierarchical Hub Labelling Revisited	29
3.2	Deciding Level Order	30
3.2.1	Deriving Optimal <i>HHL</i> Index	31
3.2.2	Rationale and Challenge	33
3.2.3	Approximation Algorithm	35
3.2.4	Approximation Guarantee and Complexity Analysis	36
3.2.5	Heuristics	39
3.3	Experiments	43
3.3.1	Experimental Setup	43
3.3.2	Tuning k for BS	44
3.3.3	Evaluation of AC-Greedy and Sketch-based Heuristic	46
3.3.4	Evaluation on Large Datasets	47
3.4	Summary	49
4	Shortest Path Queries on Timetable Graphs	50
4.1	Problem Definition	51
4.2	Timetable Labelling	53
4.3	Query Processing	57
4.3.1	Candidate Generation and Refinement	57
4.3.2	Path Reconstruction	60
4.3.3	Summary and Query Extensions	61
4.4	TTL Implementation	63
4.4.1	Index Construction	64
4.4.2	Node Ordering	67
4.5	Label Compression	70
4.5.1	Route-Based Compression	70
4.5.2	Pivot-Based Compression	71
4.5.3	Query with Compressed Labels	72

4.6	Extensions	73
4.7	Experiments	74
4.7.1	Performance of Query Processing	75
4.7.2	Preprocessing and Space Overheads	77
4.7.3	Index Construction and Node Orders	79
4.8	Summary	80
5	Constrained Shortest Path on Road Networks	81
5.1	Background	83
5.2	COLA Framework	85
5.2.1	Overlay Graph	85
5.2.2	Constrained Labeling Index	87
5.2.3	Query Processing	90
5.2.4	Optimizations	91
5.3	COLA implementation	94
5.3.1	α -Dijk	94
5.3.2	Overlay Graph Construction	98
5.3.3	Labeling Index Construction	100
5.3.4	Vertex Ordering	102
5.4	Experiments	104
5.4.1	Experimental Settings	105
5.4.2	Query Efficiency Results	107
5.4.3	Index Size and Construction Time Results	108
5.4.4	Impact of α	109
5.5	Summary	111
6	Conclusions and Future Work	112
6.1	Conclusions	112
6.2	Future Directions	113
6.2.1	Variants of Reachability Queries	114
6.2.2	More Scenarios on Public Transportation Networks	115
6.2.3	Shortest Path Problems on Time Dependent Road Networks	115

A Proofs	117
Appendix - List of Publications	126
References	128

List of Figures

2.1	A DAG G	10
2.2	CH example.	15
2.3	A timetable graph G	19
2.4	An input graph G for CSP problem	22
3.1	Constructed graph G for an X3C instance.	33
3.2	A DAG G	35
3.3	Tuning k for BS on various datasets.	45
3.4	Index sizes on small datasets.	45
3.5	Total query time on small datasets.	46
3.6	Preprocessing time on small datasets.	46
3.7	Index sizes on large graphs.	47
3.8	Total query time on large graphs.	47
3.9	Preprocessing time on large graphs.	48
3.10	Total query time on large graphs for the random query set.	48
3.11	Total query time on large graphs for the equal query set.	48
4.1	A timetable graph G	51
4.2	An example of route-based compression.	67
4.3	Average query time for SDP queries.	76
4.4	Average query time for EAP queries.	76
4.5	Average query time for LDP queries.	76
4.6	Index size.	76
4.7	Total preprocessing time.	76
4.8	Total preprocessing time.	79

4.9	Index size.	79
5.1	Example of multiple criteria when choosing a route	81
5.2	Example of exact CSP and α -CSP	84
5.3	Example overlay graph ($\alpha = 1.2$).	86
5.4	Query efficiency vs. Query sets.	106
5.5	Space consumption.	107
5.6	Total preprocessing time.	107
5.7	Impact of α to α -Dijk.	110

List of Tables

2.1	Two <i>HHL</i> indices of G in Fig. 2.1.	10
3.1	Summary of Notations.	28
3.2	Datasets ($K = 10^3$, $M = 10^6$).	44
4.1	Table of notations.	53
4.2	The label sets of v_2 and v_4 in a <i>TTL</i> index for Figure 4.1, with a node order $o(v_i) = i$ for $i \in [1, 6]$	55
4.3	The label sets of v_1 , v_2 , and v_3 in a <i>TTL</i> index of G' in Figure 4.2a, with a node order $o(v_2) = 1$, $o(v_1) = 2$, and $o(v_3) = 3$	67
4.4	Public transportation networks ($K=10^3$).	75
4.5	Compression ratio.	78
5.1	List of notations.	84
5.2	Road networks ($K=10^3$, $M=10^6$).	104

Chapter 1

Introduction

Graphs are a fundamental data structure to represent objects and their relations in various domains, e.g., social science, citation analysis, web link analysis, and navigation systems. For example, Twitter can be mapped to a graph, where each person represents a vertex, and the “follow” relationship denotes an edge. Meanwhile, a road network can be easily represented as a graph, where each road junction indicates a vertex, and each road segment denotes an edge in the graph. The reachability and shortest path queries are two types of primitive queries that have been studied for decades. The reachability query asks whether there exists a path from a vertex s to another vertex t , and the shortest path query asks for the shortest one under a given metric among all possible paths from s to t . Both types of queries are fundamental to many applications. For example, consider the graph that represents Twitter. One may be interested in whether the posted content from a person A can be reposted and finally reach another person B , i.e., she wants to identify whether A can reach B in the Twitter graph, which finds important scenarios in viral marketing. In the meantime, given the graph that represents a road network, one may be interested in how to drive from a place to another with the least travel time, which finds many scenarios in navigation services.

In this thesis, we investigate on how to build efficient index structures to answer the reachability queries and two variants of the shortest path queries. The first variant of the shortest path query considers the shortest path problem on timetable graphs, where it requires the consideration of spatiotemporal constraints between different edges. The second variant aims to provide the shortest path under an additional cost constraint. These problems find many application scenarios, and are challenging to reach high query efficiency on sizable graphs. In what follows, this chapter will briefly introduce the problems we study and our proposed solutions. More details will be presented in later chapters respectively.

1.1 Problems and Motivations

Reachability queries. The first problem we studied is the reachability query. Given a directed graph G , a source vertex s , and a target vertex t , a reachability query asks whether there exists a path from s to t in G . It has many applications in social networks, transportation networks, semantic networks, etc. A classic approach to answer reachability queries is to preprocess the input graph G and construct a *2-hop labeling* (*2HL*) scheme, which assigns two label sets to each vertex in G and then utilizes the label sets to answer queries without inspecting the original graph G .

Cohen et al. [26] present a seminal work on *2HL* scheme, and show that it is NP-hard to minimize the size of the labeling (i.e., the total size of the label sets). As a solution, they propose an $O(\log n)$ -approximate algorithm that runs in $O(n^5)$ time. Since then, a plethora of labeling techniques have been proposed to improve upon Cohen et al.'s technique. Among them, the state of the art is a class of *2HL* schemes referred to as *Hierarchical Hub Labellings* (*HHL*). In particular, an *HHL* index (implicitly or explicitly) defines a total order o on the vertices in G , and constructs label sets in accordance to o . Since o uniquely decides the construction of label sets, it determines the total size of an *HHL* index as well as its query performance. Therefore, a good total order o is of utmost importance for an *HHL* index.

Existing work on *HHL* has developed several heuristics to obtain a satisfactory total order o , e.g., based on vertex degrees [60, 98], topological order of vertices [23]. However, none of these heuristics yields optimal or non-trivial asymptotic guarantees in terms of index size or query time. Therefore, the design of an optimal or worst-case efficient *HHL* index is currently an open problem. This motivates us to present a novel study on the vertex ordering in *HHL*.

Shortest path queries on timetable graphs. The second problem we studied is the shortest path queries on timetable graphs. A public transportation network can be modeled as a *timetable graph* [76], where (i) each node represents a station; and (ii) each directed edge $\langle u, v \rangle$ is associated with a timetable that records the departure (resp. arrival) time of each vehicle at station u (resp. v). Three types of shortest path queries for route planning on timetable graphs have been extensively studied:

- (i) *Earliest Arrival Path (EAP)*: If we are at station A at time t_d , what is the earliest time that we can arrive at station B , and which is the corresponding route?
- (ii) *Latest Departure Path (LDP)*: If we plan to arrive at station B no later than time t_a , what is the latest time that we should depart from station A , and which route should we take?

- (iii) *Shortest Duration Path (SDP)*: If we are to depart from station A no sooner than time t_d and arrive at station B no later than time t_a , which is the route that would minimize the duration of our trip from A to B ?

As shown by Cooke et al. in [27], each of the above three types of queries can be solved with a modified version of the Dijkstra’s algorithm [37] that exploits the temporal information in the timetable graph G . Cooke et al.’s method, however, is inefficient for the large transportation networks (with millions of temporal edges) commonly seen in practice. Therefore, numerous techniques (e.g., [13, 29, 36, 46]) are proposed to improve over Cooke et al.’s solution in terms of query efficiency. In particular, the state-of-the-art techniques either pre-arrange the temporal edges in G in a certain order [36] or augment G with information about certain important routes [46], and they accelerate query processing by utilizing the edge arrangement or the pre-computed information. Nevertheless, as we will show in our experiments, these techniques [36, 46] still incur considerable query overheads on sizable timetable graphs G , since they require traversing a substantial portion of the temporal edges in G (or its augmented version). This motivates us to develop a more efficient route planning scheme on transportation networks.

Constrained shortest path queries on road networks. The third problem we studied is the *constrained shortest path (CSP)* problem on road networks. Formally, CSP assigns each edge in the road network a length and a cost¹; given an origin s , a destination t , and a cost constraint θ , the result of the CSP query is the path from s to t that minimizes its total length, while satisfying that its total cost does not exceed θ . The CSP problem is known to be NP-hard [45, 54]. Hence, the majority of existing work, e.g., [17, 40, 49, 54, 70, 94], focuses on approximate solutions, which guarantees that the result path length is no longer than α times of the optimal path length (where α is the user-specified approximation ratio), subject to the cost constraint θ . Although there exist polynomial-time algorithms for approximate CSP (e.g. [40, 49, 54, 70, 94]), as we show in the experiments, the current state-of-the art solutions are still prohibitively expensive for large networks. There are two main reasons for their inefficiency. First, they aim at answering approximate CSPs on a general graphs, rather than specifically on road networks; consequently, they fail to utilize the latter’s special properties. Second and more importantly, most of them process each query on-the-fly, without the help of any precomputed index structure. The few known indices are all designed for exact CSPs; further, these indices consume large amounts of memory, and yet none of them succeeds at reducing query cost to a practical level. This motivates us to devise a novel and practical solution *COLA* for index-based approximate CSP processing on large road networks.

¹Note that the edge length and cost can be defined arbitrarily. Without loss of generality we assume that the objective criterion is the length of a path and the constraint is on the total cost of a path.

The aforementioned problems are closely related. First of all, all the three types of queries target to find a path that satisfy certain conditions. For example, the reachability query targets to find a path from a source node s to a target node t ; the constrained shortest path query asks for a path that has the minimum length without violating the cost constraint. Besides, all the paths for the three queries can be expressed through some tuples, dubbed as *labels*. In our three studied problems, the index structure are similar in a sense that each node v maintains an out-label (resp. in-label) set, which includes a set of labels corresponding to some paths from v (resp. to v). Then, the queries can be answered by a linear scan of the label set of the source and the target. Nevertheless, the labels for these three problems vary significantly, and it is non-trivial to propose efficient query algorithms for the latter two problems, i.e., the shortest path queries on timetable graphs and the constrained shortest path queries on road networks. Moreover, any technique for the constrained shortest path queries can be used to answer the reachability queries if we set all the length and cost of an edge to zero; any technique for the shortest path queries under timetable graphs can also be used to answer the reachability queries if we set all the departure time and arrival time to zero. Finally, road networks can be efficiently partitioned, and our proposed *COLA* index for constrained shortest path queries exploits this fact to improve the query efficiency on road networks. If there exist efficient partitioning algorithms for general graphs, then our techniques for *COLA* index can be similarly applied to speedup the query efficiency for the reachability queries and shortest path query variants on timetable graphs. Despite that fact that these three problems are highly closely related, when efficiency and scalability are taken into consideration, approaches for these three problems vary significantly, thus making these three problems worth studying.

1.2 Approaches and Contributions

Reachability queries. For reachability queries, we present a novel study on the derivation of a total order o for *HHL*, the state-of-the-art schemes for answering reachability queries. First, we prove that deriving an optimal vertex ordering that minimizes the index size is NP-hard. Afterwards, we present an algorithm for vertex ordering that yields an *HHL* index whose size is at most $O(\sqrt{n})$ times the optimal one, where n is the number of vertices in G . To our knowledge, this is the first *HHL* method with non-trivial worst-case guarantees. The algorithm, however, runs in $O(n^3)$ time and requires $O(n^2)$ space, which makes it unsuitable for sizable graphs. Motivated by this, we also investigate heuristic solutions for vertex ordering.

Specifically, we present a heuristic cost function for ordering vertices that is motivated by our theoretical study. We show that the cost function can be efficiently approximated

by utilizing reachability sketches [25]. With this, we obtain an *HHL* index that can scale to graphs with millions of vertices while still offering superior performance.

We experimentally evaluate our reachability solutions with a large variety of graphs. On 9 small graphs (where our $O(\sqrt{n})$ -approximate algorithm finishes preprocessing in reasonable time), we show that our heuristic method yields comparable index size and query performance to our $O(\sqrt{n})$ -approximate algorithm, which demonstrates the effectiveness of our heuristic. On large graphs, we show that our heuristic method consistently outperforms the state-of-the-art reachability indices in terms of space consumption and query time, in several cases by large margins. Furthermore, the preprocessing cost of our method is proportional to the index construction cost hence our heuristic solution can scale to sizeable graphs. This shows that our heuristic method is a favorable choice in scenarios where the reachability query time is more important than the preprocessing cost.

Shortest path queries on timetable graphs. We present *Timetable Labelling (TTL)*, an efficient indexing technique for the shortest path queries on timetable networks. The basic idea of *TTL* is to associate each node u with a set of *labels*, each of which records the shortest travel time from u to some other node v given a certain departure time from u . The labels are constructed in such a way that, for any EAP, LDP, or SDP query from a node u to another node v , we can efficiently retrieve a tiny summary of the query result by inspecting the label sets of u and v (without inspecting G), and we can easily convert the summary into the corresponding path in $O(k)$ time, where k is the number of nodes in the path. Compared with the existing techniques [27, 36, 46], *TTL* incurs lower query overheads as it completely avoids traversing G during query processing. In particular, in our experiments on 11 real transportation networks, *TTL* is shown to outperform the states of the art by up to 3 orders of magnitude in terms of query efficiency.

TTL is similar in spirit to *hierarchical hub labelling (HHL)* [6] (i.e., the state-of-the-art approach for reachability and distance queries on simple graphs), as both of them rely on pre-computed label sets for query processing. Nevertheless, *TTL* is not a straightforward extension of *HHL* that simply augments each label set with timetable information. Instead, *TTL* incorporates a set of non-trivial optimization techniques in its preprocessing and query algorithms, which exploit the characteristics of timetable graphs to achieve significant improvements in terms of both query performance and space consumptions.

Constrained shortest path queries on road networks. We propose a novel and practical solution *COLA* for index-based approximate CSP processing on large road networks. *COLA* mainly exploits two important properties of the road network. First, real road networks are often (roughly) planar, and, thus, can be effectively split into partitions, each of which contains only a relatively small number of boundary vertices. Accordingly,

COLA partitions the network, builds an overlay graph on the partitions, and indexes the skyline paths between pairs of boundary vertices. Second, in practice there often exist a relatively small number of landmark vertices in the road network that commonly appear in CSP results. Based on this property, we design an index-free algorithm α -*Dijk* as a component of *COLA* for path computation within a partition, i.e., between the origin (or destination) vertex and its corresponding partition boundaries. Extensive experiments using real continent-sized road networks containing tens of millions of vertices show that *COLA* answers an approximate CSP query within a second, whereas previous solutions need several hours. Further, even when an index is not available (e.g., when the edge lengths / costs change frequently), the α -*Dijk* module still outperforms existing methods by over an order of magnitude.

1.3 Thesis Organization

Chapter 2 conducts a detailed survey on the above studied problems. Firstly, we introduce previous work for reachability queries. Afterwards, we survey existing solutions for conventional shortest path queries. Then, we present the existing methods for shortest path queries on timetable graphs, and pinpoint the deficiencies of those solutions that target to answer the EAP, LDP, and SDP queries. Finally, we review existing techniques for both exact and approximate CSP queries. We first present two Dijkstra-like solution for exact CSP queries and approximate CSP queries. After that, we present an indexing technique for exact CSP queries, which pre-computes augmented information to reduce the Dijkstra-like traversal cost; we then review existing solutions for approximate CSP queries and pinpoint their limitations on large graphs.

Chapter 3 presents the study on node ordering in *HHL*. We first reduce a well-known NP-hard problem to the node ordering problem, and show that deriving an optimal *HHL* index is NP-hard. Subsequently, we provide the details of our approximate ordering algorithm, and give the formal proofs for the approximation guarantee, followed by the complexity analysis. As the approximate algorithm still incurs prohibitive preprocessing cost, we further present a heuristic ordering method motivated by our approximation algorithm. Finally, we experimentally evaluate our approximate and heuristic ordering algorithm. Extensive experimental results show that our approximate algorithm provides the smallest index size among all existing ordering methods, albeit at high preprocessing cost, which confirms that the proposed ordering method is worst case guaranteed. Meanwhile, our heuristic algorithm shows similar index size as the approximate one at a much lower preprocessing cost, which demonstrates the effectiveness of our heuristic ordering method.

Chapter 4 demonstrates *Timetable Labelling (TTL)*. We first describe the index structure of *TTL*, and then introduce a novel query algorithm which could efficiently answer

EAP, LDP, and SDP queries. After that, we demonstrate the index construction algorithm, that produces a correct *TTL* index. Next, we study label compression techniques to reduce the index size, without significantly affecting the query efficiency. Furthermore, we study some extensions for *TTL* that face real application scenarios. Finally, we experimentally evaluate our *TTL* against the state-of-the-art approaches to show the effectiveness of our proposed scheme.

Chapter 5 introduces *Constrained Labelling (COLA)*. We first present the *COLA* framework, and demonstrate how to construct an overlay graph, where *COLA* index is built on, by graph partition. Afterwards, we describe the index structure and query algorithms for *COLA* index. Next, we present the *COLA* implementation, including the α -*Dijk* algorithm for approximate CSP processing without an index. Finally, we experimentally evaluate our *COLA* index against the state-of-the-art algorithms and demonstrates that our *COLA* index outperforms the states-of-the-art by a large margin in terms of query efficiency.

Chapter 6 concludes the thesis and presents some promising future directions on the topic of reachability and shortest path queries.

Chapter 2

Literature Survey

In this chapter, we first review existing techniques to speed up the reachability queries. Next, we present existing solutions for the conventional shortest path problem. Afterwards, we survey previous work on the two variants of the shortest path problem: shortest path queries on timetable graphs, and the constrained shortest path problem. Finally, we summarize the approaches surveyed in this chapter.

2.1 Reachability Query

Let $G = (V, E)$ be a directed graph with a set V of vertices and a set E of edges. Given a source vertex s and a target vertex t in G , a reachability query asks whether there exists a path from s to t in G . The most intuitive approach is to proceed the reachability query with a traversal algorithm, e.g., breadth-first search or depth-first search, from the source vertex s , and see if t can be reached during the traversal. However, the query performance of this approach depends on the size of the graph, and is usually too slow for real applications when the graph is too large. Meanwhile, another straightforward strategy is to pre-compute the reachable vertices for each vertex v , i.e., the transitive closure of v , and store the transitive closure for each vertex in G . Then a reachability query can be answered with constant time by checking the stored transitive closures. However, this strategy incurs prohibitive space consumption to store the transitive closure for all vertices, and is impractical for large size graphs.

In this section, we first review *2-hop labelling (2HL)*, a labelling scheme that strikes a better trade-off between space consumption and query efficiency for sizable graphs. Then, we focus on a special case of the *2HL*: *Hierarchical Hub Labelling (HHL)*, which is currently the state-of-the-art scheme for reachability queries. Afterwards, we review other existing techniques for reachability queries.

2.1.1 2-Hop Labelling

In *2-hop labelling (2HL)* [26], each vertex contains an in-label set $L_{in}(v)$ and an out-label set $L_{out}(v)$. $L_{in}(v)$ (resp. $L_{out}(v)$) is a subset of vertices that can reach v (resp. v can reach). A *2HL* is computed in a manner such that, for any reachability query from s to t , there exists a common vertex in $L_{out}(s)$ and $L_{in}(t)$, and vice versa. Compared to graph traversal based solutions, *2HL* scheme requires only a linear scan of $L_{out}(s)$ and $L_{in}(t)$ for a reachability query from s to t , leading to a significantly improved query performance. Meanwhile, compared to the transitive closure based solutions, each label set in *2HL* only contains a selected subset of vertices by using both in-labels and out-labels, leading to a significantly reduced index size. This is akin to database normalization, where storing two separate base tables consumes less space than their join results. As a result, *2HL* strikes a better balance between query efficiency and space consumption.

In *2HL*, the total size of the label sets of a 2-hop labeling index $\mathcal{L}$, denoted as $|\mathcal{L}|$, is often used as an indicator of the quality of $\mathcal{L}$. The reason is that, with a smaller index size $|\mathcal{L}|$, the average label size for each vertex, i.e., $|\mathcal{L}|/n$, becomes smaller, indicating a smaller amortized label checking cost to answer the reachability queries. However, deriving a *2HL* with smallest index size is proven to be NP-hard [26]. As a result, most of the existing work turns to find approximate or heuristic solutions.

Cohen et al. [26] present an $O(\log(n))$ -approximate algorithm with a time complexity of $O(n^5)$, which is usually too slow for large graphs. To reduce the extremely high cost, a plethora of labeling approaches are proposed. Among the proposed approaches, the state-of-the-art solution is *Hierarchical Hub Labelling*, and we present the details in the following.

Hierarchical Hub Labelling. *Hierarchical Hub Labelling (HHL)* is a subclass of 2-hop labeling where a total order o is imposed on all the vertices, and the label set is then uniquely decided by the total order o . Let o be a total order on V , and $o(v)$ be an integer in $[1, n]$, such that $o(v) = i$ if and only if v is the i -th vertex in o . We refer to o as a *level order*, and $o(v)$ as the *level* of v in o . Meanwhile, we say u has a higher (resp. lower) level than v if $o(u) < o(v)$ (resp. $o(u) > o(v)$).

HHL [6, 101] is a subclass of 2-hop labeling indices that implicitly or explicitly defines a total order o on V , and satisfies the following constraints for $u \in L_{in}(v)$ (resp. $u \in L_{out}(v)$):

- *Reachability Constraint:* $u \rightarrow v$ (resp. $v \rightarrow u$);
- *Level Constraint:* $o(u) < o(v)$;
- *Path Constraint:* There does not exist a directed path from u to v (resp. from v to u) that contains a vertex w with $o(w) < o(u)$.

 Figure 2.1: A DAG G .

	<i>HHL</i> index $\mathcal{L}_1$			<i>HHL</i> index $\mathcal{L}_2$		
v	$o_1(v)$	$L_{in}(v)$	$L_{out}(v)$	$o_2(v)$	$L_{in}(v)$	$L_{out}(v)$
a	1	$\emptyset$	$\emptyset$	2	$\{d\}$	$\emptyset$
b	2	$\{a\}$	$\emptyset$	5	$\{a, d, e\}$	$\emptyset$
c	3	$\{a\}$	$\emptyset$	6	$\{a, d, f\}$	$\emptyset$
d	4	$\emptyset$	$\{a\}$	1	$\emptyset$	$\emptyset$
e	5	$\{d\}$	$\{b\}$	3	$\{d\}$	$\emptyset$
f	6	$\{d\}$	$\{c\}$	4	$\{d\}$	$\emptyset$
g	7	$\{a\}$	$\emptyset$	7	$\{a, d\}$	$\emptyset$
h	8	$\{a, b, c, g\}$	$\emptyset$	8	$\{a, b, c, d, e, f, g\}$	$\emptyset$

 Table 2.1: Two *HHL* indices of G in Fig. 2.1.

We illustrate the concept of *HHL* with an example.

Example 2.1. Given the DAG G shown in Fig. 2.1, Table 2.1 shows two *HHL* indices on G (namely $\mathcal{L}_1$ and $\mathcal{L}_2$) with level orders o_1 and o_2 respectively. Consider the in-label of vertex b in $\mathcal{L}_1$. It contains a since (i) a can reach b , (ii) $o_1(a) < o_1(b)$, and (iii) all paths from a to b do not contain any vertex with a higher level than a . Besides, b 's in-label set does not contain e since b has a higher level order than e (i.e., violating the Level Constraint). Furthermore, it also does not contain d since there exists a path from d to b that passes a whose level is higher than d 's (i.e., violating the Path Constraint).

On the other hand, consider b 's in-label set in $\mathcal{L}_2$. Clearly, a , d , and e all have a higher level than b , and all can reach b . Meanwhile, there does not exist any path from a to b passing through a higher level vertex other than a , and a satisfies Path Constraint. Similarly, d and e satisfy the Path Constraint. So b 's in-label set in $\mathcal{L}_2$ contains a , d , and e .

Note that, $|\mathcal{L}_1| = 12$, which is much smaller than $|\mathcal{L}_2| = 18$. As we can observe, the significant differences between $\mathcal{L}_1$ and $\mathcal{L}_2$ are due to that o_1 and o_2 are drastically different. $\square$

It is shown that the label sets in any *HHL* index $\mathcal{L}$ is *minimal* [101], i.e., removing any vertex in any label set of an *HHL* index will make the query result invalid for some vertices s and t with $s \rightarrow t$. Furthermore, given o , an *HHL* index is uniquely decided

by its level order o . Therefore, choosing a good o is essential to the construction of an efficient *HHL* index. This fact is illustrated in Example 2.1. In particular, given G in Fig. 2.1, the size of the *HHL* index with o_1 is $|\mathcal{L}_1| = 12$, whereas it is $|\mathcal{L}_2| = 18$ with o_2 . Existing *HHL* indices [23, 60, 98, 101] have adopted various heuristics to derive good level orders. For example, some solutions order the vertices based on their degree [60, 98], while some other solutions order the vertices based on a topological order [23]. However, none of those heuristics yields optimal or non-trivial asymptotic guarantees in terms of index size or query time. Therefore, the design of an optimal or worst-case efficient *HHL* index is currently an open problem.

2.1.2 Other Techniques for Reachability Queries

Besides 2-hop labelling, there exists another category of solutions [21, 89, 95, 99] that try to answer reachability queries with pruned depth-first search (DFS). To prune the DFS, they pre-compute some auxiliary information for index construction, which helps improve the query performance. *GRAIL* [99] is one of the examples, in which each vertex is assigned with k interval labels, where the interval labels are generated with k random DFS traversals. The intervals help reduce search space as follows: If the interval of u does not contain the interval of v , then u cannot reach v ; then, when a DFS is initiated to check the reachability query from s to t , as long as a vertex u has an interval that does not contain t 's, vertex u and its subtrees can be omitted during the DFS traversal. This helps to reduce the traversal cost, and help improve query efficiency. However, this technique has the benefit that they can be built in an efficient way, i.e, by k DFS traversals to generate the interval labels, but their query performance is often much worse than the other approaches, e.g., *HHL*.

There also exist a plethora of techniques that construct indices for processing reachability queries by utilizing a compressed version of the transitive closure of G [7, 22–24, 57, 62, 85, 91, 93]. Those methods, however, are limited to small size graphs because of the large space consumed when n is sufficiently large. Limiting the size of index with transitive closure is possible and is studied by Seufert et al. [85]. However, the saving of space is at the cost of degrading query performance.

Furthermore, Jin et al. [59, 61] develop the *3-hop approach* scheme, where each vertex contains a set of additional *important* vertices, besides the in-label set and out-label set. The in-label and out-label of a vertex u then can only contain vertices in the *important* vertices. This approach can further reduce the index size when compared to *2HL*. However, the construction and query time of *3-hop* degrade significantly, and is not a competitive approach on sizable graphs. Meanwhile, Jin et al. [59] propose a general framework called *SCARAB* to improve the query efficiency by introducing the concept of *reachability backbones*. However, the computation of reachability backbones itself is usually slow on large graphs as shown in [23], which renders *SCARAB* itself unscalable to large graphs.

2.2 Conventional Shortest Path Queries

Consider an edge weighted graph $G = (V, E)$, where each edge is associated with a positive weight $\ell(e)$, which is referred to as the length of edge e . The length of a path P , which consists of a sequence of edges $e_1, e_2, \dots, e_k$, is defined as $\ell(P) = \sum_{i=1}^k \ell(e_i)$. Given an origin s , and a destination t , a shortest path query from s to t asks for the path P from s to t with minimum length. A most straightforward variant of the shortest path query that also finds many application scenarios is the distance query. In particular, a distance query from s to t concerns only the length of the shortest path P , i.e., the value of $\ell(P)$. Due to the many application scenarios, these two problems have been studied for decades, and a plethora of techniques have been proposed to provide efficient query performance.

In this section, we review existing solutions for the conventional shortest path queries. We mainly focus on the solutions that are most relevant to our studied two problems and omit the discussion of other solutions for brevity. In particular, we first present the Dijkstra's algorithm, which is the classic solution for the shortest path problem. This technique could be extended to solve our studied two problems, and we give a general idea on how the Dijkstra's algorithm works. Afterwards, we present *HHL* for distance queries, which is the state-of-the-art technique for distance queries. Our indexing techniques share some similar spirit as *HHL*, and we explain the difference between our indexing technique and *HHL* when we explain *HHL*. Next, we explain *Contraction Hierarchies*, an efficient indexing technique for shortest path queries on road networks. This technique has also been extensively studied for different variants of shortest path queries, including our two studied problems. Finally, we review a notable class of techniques that speed up shortest path queries by graph partitioning. Our index structure for constrained shortest path queries on road network will also exploit this technique to speed up the query performance.

2.2.1 Dijkstra's Traversal

Dijkstra's algorithm [37] is the classic solution for shortest path and distance queries. Given an origin s , the Dijkstra's algorithm traverses the vertices in G in ascending order of the distance from s ; when the destination t is reached, it retrieves the shortest path for the shortest path query, or return the distance to t for a distance query.

The Dijkstra's algorithm maintains a heap H that stores the visited paths originating from s in ascending order of the length. More precisely, it stores a tuple (v, ℓ, v_{pred}) to summarize a path P , where v is the ending node of P , ℓ is the length of P , and v_{pred} is the predecessor of v in P . Through this summarized form, when the shortest path is found, it can recursively retrieve the nodes in the path through the predecessor, and

reconstructs the full path. For each vertex, it maintains the path that originates from s with least length during the traversal. In each iteration, the top path P in heap H is popped. Assume the ending node of P is w . Then, the Dijkstra's algorithm checks each outgoing edge $e = (w, v, \ell)$, and sees if $\ell(P) + \ell$ exceeds the length of the stored path for v . If the answer is yes, then the Dijkstra's algorithm updates the stored path for v , and pushes the new path into H . Otherwise, it discards the path and repeat the above process until H becomes empty.

To improve the query efficiency, an immediate optimization technique is the *bidirectional Dijkstra*, which applies a Dijkstra's traversal from s and a backward version of Dijkstra's traversal from t simultaneously, and the two traversals terminate when they meet at some common vertex in the graph. This optimization technique reduces the search cost for traversal. However, the time complexity of both algorithms are $O(m + n \cdot \log n)$, where n (resp. m) is the number of vertices (resp. edges) in G . In other words, the query performance of Dijkstra's and bidirectional Dijkstra's algorithm depend on the size of the graph, and will be too slow for real world applications when the input graph is large enough.

2.2.2 Hierarchical Hub Labelling

Hierarchical Hub Labelling (HHL) also stands as the state-of-the-art scheme for distance queries [6, 8, 30, 100]. Let $\text{dist}(u, v)$ denote the distance from u to v . For each vertex v , it contains an out-label set $L_{\text{out}}(v)$ and an in-label set $L_{\text{in}}(v)$. In particular $L_{\text{out}}(v)$ contains a set of pairs $(w, \text{dist}(v, w))$ with some vertices $w \in V$, and $L_{\text{in}}(v)$ contains a set of pairs $(u, \text{dist}(u, v))$ with some vertices $u \in V$. The label sets of the vertices are constructed in a way such that the following equation holds for any pair of vertices (u, w) :

$$\text{dist}(u, w) = \min_{v \in L_{\text{out}}(u), v \in L_{\text{in}}(w)} (\text{dist}(u, v) + \text{dist}(v, w)),$$

Similar to the case of *HHL* index for reachability queries, *HHL* index for distance queries is a subclass of 2-hop labeling indices that implicitly or explicitly defines a total order o on V , and the index is then solely decided by o . Then, in an *HHL* for distance queries, for $(u, \text{dist}(u, v)) \in L_{\text{in}}(v)$ (resp. $(u, \text{dist}(v, u)) \in L_{\text{out}}(v)$), it satisfies the following constraints:

- *Reachability Constraint*: $u \rightarrow v$ (resp. $v \rightarrow u$);
- *Level Constraint*: $o(u) < o(v)$;
- *Path Constraint*: There does not exist a shortest path from u to v (resp. from v to u) that contains a vertex w with $o(w) < o(u)$.

A distance query from s to t then can be answered by a linear scan of $L_{out}(s)$ and $L_{in}(t)$. As we have extensively discussed *HHL* in Section 2.1.1, we omit other details for brevity.

To answer a shortest path query, an *HHL* index can add additional path information, e.g., the predecessor or successor of a vertex in a shortest path. Then, the path can be fully reconstructed iteratively with the auxiliary vertex information stored in the label. However, the existing work [6, 8, 30, 100] on *HHL* mainly focuses on distance queries and does not study on how to efficiently retrieve the full path.

Remark. Our indexing techniques are similar in spirit to *HHL* [6] as both of them rely on pre-computed label sets for query processing. Nevertheless, our indexing techniques are not straightforward extensions of *HHL* that simply augment each label set with timetable information (for shortest path queries on timetable graphs) or cost information (for CSP problem). Instead, they include non-trivial optimizations and novel query algorithms that exploit the characteristics of the timetable graphs or road networks. We will see the details of our indexing techniques in Chapter 4 and Chapter 5. Meanwhile, in our studied two shortest path problems, we aim to provide full path, and will present details on how to efficiently reconstruct the path in our query algorithms.

2.2.3 Contraction Hierarchies

Contraction Hierarchies (CH) is a graph indexing technique that preprocesses G by constructing shortcuts (i.e. artificial edges) among vertices in G . During a shortest path /distance query processing from s to t , it employs a Dijkstra's algorithm from s and a backward version of Dijkstra's algorithm from t , simultaneously. During the traversal, it utilizes the shortcuts to reduce the number of vertices that need to be traversed, and hence improve the query efficiency. To construct a *CH* index, it imposes a total order o on the vertices, and then creates the index based on the total order o . In particular, an overlay graph G° is initialized to be a copy of the input graph G . Then, in the i -th iteration, the vertex v_i with i -th order in o is removed from the overlay graph G° , and some shortcut edges are added into G° in a manner that the distances between the remaining vertices in G° are still the same. To achieve this, before removing v_i from the overlay graph, the preprocessing algorithm considers all incoming edges, and outgoing edges of v_i in G° . For each incoming edge $e_i = (u, v_i)$ and each outgoing edge $e_i = (v_i, w)$ of v_i , it checks if (u, v_i, w) is the only shortest path in G° . If the answer is yes, then a shortcut (u, w) , with length equal to $\ell(e_i) + \ell(e_o)$, is added into G° . This process is referred to as the *contraction* of v_i . When all vertices are contracted, the added shortcuts are added into the original graph G . The newly created graph is then called the *augmented* graph, and is the final *CH* index. We illustrate this process with an example.

 Figure 2.2: CH example.

Example 2.2. Consider the input graph G as shown in Figure 2.2(a). Assume that each edge has a unit length, and the total order o is the alphabetical order of the vertices, i.e., $o(a) < o(b) \cdots < o(g)$. Then G° is initialized to be G , and the preprocessing algorithm inspects vertex a . For vertex a , there is only one outgoing edge, and will not affect the distance of other vertices if a is removed, so no shortcut is added. Similarly, when vertex b is contracted, no shortcut is added into G° . Meanwhile, when vertex c is being proceeded, note that there is one income edge (g, c) , and one outgoing edge (c, e) in G° . The removal of c will affect the distance from g to e . As a result, a shortcut edge (g, e) with length 2 is added into G° . Similarly, during the contraction of vertex d , a shortcut edge (g, f) with length 2 is added into G° . In the iteration contracting e , there is one incoming edge (g, e) , and two outgoing edges (e, f) , and (e, g) . The deletion of e will not affect the distance from g to f or from g to g . So no shortcut is added. Finally, in the contraction of f and g , no shortcut is added, and there are two shortcuts, i.e., (g, e) and (g, f) , added in total during preprocessing. The shortcuts then are added into the original graph G , and output as the CH index for the input graph G as shown in Figure 2.2(b). $\square$

Given the augmented graph, CH then proceeds a distance query from s to t with two modified version of Dijkstra's algorithm. In particular, when traversing vertices in ascending order of the distances from s , it only visits an edge (v, w) that satisfies $o(v) < o(w)$. Similarly, in the backward Dijkstra's traversal from t , it only visits an edge (v, w) where $o(v) > o(w)$. This approach guarantees that s and t will only visit vertices has higher level order than s or t , which reduces the traversal cost.

For example, consider a distance query from b to f . Then the Dijkstra's traversal from b visits g , and $dist(b, g) = 1$. Consider the backward Dijkstra's traversal from f . It will only visits vertices with higher level order than f , i.e., g , and $dist(g, f) = 2$. As the two traversal meets and there are no other paths with smaller distance, it returns $dist(b, g) + dist(g, f) = 3$ as the distance from b to f .

The aforementioned algorithm can also help return the shortest path from s to t on the augmented graph. However, the returned path P may contain shortcuts which

do not belong to the original graph, e.g., (g, f) in the above query. To return a full path in the original graph, *CH* examines whether the edge is a shortcut created during the contraction or an edge in the original graph. If the edge is a shortcut, then *CH* removes the edge from path P , and replace it with two edges in the augmented graph. For example, we replace (g, f) with (g, d) and (d, f) in path P . Afterwards, we examine whether the newly added edges are shortcuts created during contraction, and repeat the above process until all edges in path P are in the original graph.

Recall that *CH* requires a total order for the vertices. Similar to *HHL*, the total order has a great impact on the efficiency of *CH*. An inferior ordering may leads to $O(n^2)$ shortcuts, resulting in an $O(n^2 \cdot \log n)$ time complexity for a distance query. In [48], they propose several heuristics to derive a good ordering, and the ordering is shown to result in very efficient *CH* index for road networks. Nevertheless, the efficiency of the *CH* depends on the number of added shortcuts, and the efficiency degrades when the number of shortcuts is too large.

The most straightforward application for shortest path queries is route planning in navigation systems like Google Maps [5]. In this case, the road network is modeled as a graph, and the target is to find the path that has the least travel time / travel distance. The road network has several nice properties that can be further exploited to improve the query efficiency. Next, we present one class of the speedup techniques for road networks that is most relevant to our study.

2.2.4 Graph Partitioning

One notable class of solutions [31,63,65,82,83] employs graph partitioning to speed up the efficiency of shortest path queries on road networks. Road networks has an important property that they are often (roughly) planar, and, thus, can be effectively split into partitions, each of which contains only a relatively small number of boundary vertices. Based on this property, one can first design an efficient indexing methods to answer shortest path queries between boundary vertices, and then use the indexing to speed up shortest path query for any vertex pair in the graph.

Given an input graph G , a partitioning of G consists of a set $\mathcal{T} = \{G_1, G_2, \dots, G_{|\mathcal{T}|}\}$ of edge-disjoint subgraphs of G , such that the union of all G_i ($1 \leq i \leq |\mathcal{T}|$) equals G . Given $\mathcal{T}$, vertex v is called a *boundary vertex*, if it appears in more than one subgraphs in $\mathcal{T}$. Afterwards, the graph partitioning based solutions create an overlay graph G° that contains all boundary vertices. The edges in G° are created in a way that for each pair of boundary vertices (u, w) that there exists a shortest path P , an edge (u, w) with length $\ell(P)$ is added into G° . Meanwhile, the constructed overlay graph G° could be quite dense, and an edge elimination procedure is conducted. In particular, if an edge

is not a shortest path, or the deletion of this edge will not affect the distance between other vertices, this edge can be simply discarded.

Let G_s and G_t be the subgraph that contains s and t , respectively. To proceed a shortest path query, a Dijkstra's traversal from s and a backward Dijkstra's traversal from t are proceeded on $G^\circ \cup G_s \cup G_t$, simultaneously. It is shown in [31, 65] that, the graph partitioning methods can significantly reduce the query time. This is mainly due to the reduced search space, e.g. a small size of the overlay graph. Some other speed up techniques can be combined to improve the query efficiency. The representative of this class of solutions is MLD [31], which combines partitioning and *Contraction Hierarchies* that achieves a better query performance.

In the following, we omit further discussions on conventional shortest path processing for brevity and turn to the existing solutions for the two variants of the shortest path queries. A more detailed survey of the conventional shortest path can be found in [12].

2.3 Shortest Path Queries on Timetable Graphs

Let G be a timetable graph with n nodes, such that (i) each node represents a station in a public transportation network, and (ii) each edge e from a node u to another node v is associated with a vehicle b and two timestamps t_d and t_a , which indicate that there is a vehicle b that departs from u at t_d and arrives at v at t_a , without any stops in between. Each edge e is represented as a tuple $\langle u, v, t_d, t_a, b \rangle$, where b , t_d , and t_a are referred to as the vehicle, departure time, and arrival time of e , respectively. In a timetable graph, a path P must satisfy the following additional temporal-spatial constraint: (i) for any two consequent edges e_1 and e_2 in P , the arrival station of e_1 must be the same as the departure station of e_2 , and (ii) the arrival time of e_1 must be no larger than the departure time of e_2 .

In this thesis, we focus on three shortest path queries on timetable graphs, i.e., EAP, LDP, and SDP queries. The three queries ask for the path that arrives as early as possible, departs as late as possible but arrives on time, and has the least travel time, respectively. As shown by Cooke et al. in [27], EAP, LDP, and SDP queries can all be solved with a modified version of the Dijkstra's algorithm [37] that exploits the temporal information in the timetable graph G . Cooke et al.'s method, however, is inefficient for the large transportation networks (with millions of temporal edges) commonly seen in practice. Therefore, a plethora of techniques [13, 14, 27, 29, 36, 46, 50, 51, 68, 72, 76, 79, 84] have been proposed to improve over Cooke et al.'s solution in terms of query efficiency.

Next we first present the state-of-the-art techniques for these problems in Section 2.3.1. Then, we present other solutions in Section 2.3.2.

2.3.1 State of the Art

The states of the art include the *Connection Scan Algorithm (CSA)* [36, 96], the *Contraction Hierarchies for Timetables (CHT)* [46], and *T.Patterns* [11].

Rearrangement of edges. In particular, *CSA* represents the timetable graph G as two sequences of edges, such that the first (resp. second) sequence sorts edges in ascending order of their departure timestamps (resp. in descending order of their arrival timestamps). For any path query, *CSA* derives the query answer using one linear scan of one of the edge sequences. It can be guaranteed that through a linear scan, the *CSA* algorithm will not miss any shortest path for the EAP, LDP, and SDP queries. The main difference among the query algorithm for EAP, LDP, and SDP is which sequence of edges is scanned, and what additional information is stored during the linear scan.

In particular, for an EAP query from u to v with timestamp t_θ , *CSA* maintains the earliest arrival time $t(v)$ to each vertex v during the linear scan. The earliest arrival time for each vertex is initialized to be $+\infty$ except u , which is set to zero. The linear scan is proceeded on the sequence of edges that is sorted in ascending order of the departure times. It starts scanning from the edge whose starting timestamp is no smaller than t_θ . Then, when an edge $e = (v_i, v_j, t_d, t_a, b)$ is scanned, it inspects the stored earliest arrival time $t(v_i)$ for v_i , and check if $t(v_i) \leq t_d$ and $t_a \leq t(v_j)$. If the condition is satisfied, it infers that a path with an earlier arrival time is found, and *CSA* sets $t(v_j) = t_a$. The linear scan could terminate when $t_d \geq t(v)$ since any subsequent path that ends at v will incur a larger arrival time than $t(v)$. For an LDP query, the main difference is that it maintains the latest departure time for each vertex, and the linear scan is proceeded on the sequence of edges sorted in descending order of their arrival timestamps. Meanwhile, for an SDP query, *CSA* needs to maintain a list of timestamp pairs (t'_d, t'_a) to store the starting timestamp and ending timestamp for each candidate path. The candidate paths satisfy that no path has a departure time later than and arrival time earlier than another path. The linear scan is proceeded on the sequence of edges sorted in ascending order of their departure timestamps.

Next, we illustrate how *CSA* works for EAP queries with an example.

Example 2.3. Consider the timetable graph in Figure 2.3, where the ordered pair on an edge denotes the departure time and arrival time of the edge, respectively. Given an EAP query from v_5 to v_1 with a starting timestamp 5, the *CSA* algorithm then scans the sequence of edges in ascending order of the departure time. Before scanning the edges, $t(v_5)$ is set to zero, and $t(v_1), t(v_2), t(v_3), t(v_4), t(v_6)$ are set to $+\infty$. Afterwards, it inspects the edge with smallest departure time that is larger than 5, i.e., $(v_4, v_3, 5, 7, b_3)$. However, as $t(v_4)$ is $+\infty$, $t(v_3)$ is not updated. Then, it turns to the next edge, i.e. $(v_5, v_6, 6, 7, b_3)$. As $t(v_5) = 0$ and the arrival timestamp $t_a = 7$ is smaller than $t(v_6), t(v_6)$

Figure 2.3: A timetable graph G .

is updated to 7. The next edge with smallest departure time is $(v_5, v_2, 6, 8, b_1)$, and $t(v_2)$ is updated to 8. Next, the algorithm inspects the edge $(v_6, v_1, 7, 8, b_3)$, as $t(v_6) = 7$, and $t(v_6) \leq 7$. Since 8 is smaller than $t(v_1)$, and $t(v_1)$ is updated to 8. Afterwards, it inspects the edge from v_3 to v_1 . As $t(v_3)$ is $+\infty$, no update is proceeded on $t(v_1)$. When *CSA* further inspects the next edge in the sequence, it finds that the edge $(v_2, v_1, 8, 10, b_1)$ has a departure time no smaller than $t(v_1)$, and *CSA* terminates the linear scan and return the path that goes through v_5 , v_6 , and v_1 in order as the answer of the query. $\square$

This technique incurs very small preprocessing overheads, and has very good cache locality, which helps it outperform the temporal Dijkstra's algorithm [27] in terms of query time. However, the query efficiency still depends on the size of the graph, and is usually limited to medium size graphs.

Query processing with augmented graph. Meanwhile, *CHT* [46] shares the similar spirit as *CH*, and preprocesses G by constructing *shortcuts* (i.e., artificial edges) among the nodes in G , such that each shortcut captures a fastest route between the two nodes that it connects. During query processing, *CHT* employs a bidirectional Dijkstra-like search from the source and destination nodes simultaneously, and it utilizes the pre-computed shortcuts to reduce the number of nodes that need to be traversed. However, this *CH* based optimization technique is not as efficient on public transportation networks as on road networks. There are mainly two reasons. The first reason is that, in public transportation networks, it usually needs to add many shortcuts between two nodes, e.g., edges with different departure time. This makes the overlay graph too large, and affects the query performance. The second reason is that, in the bidirectional Dijkstra-like search, the arrival time (resp. departure time) is unknown for EAP (resp. LDP) queries. In their implementation, the backward traversal from the destination t is mainly used to mark the vertices that can reach t with higher level orders. On the contrary, on conventional shortest path queries, the backward traversal can be also used to help prune the search space.

T.Patterns [11] exploits a similar idea to *CHT*, but instead of maintaining shortcuts, it pre-computes a set of fastest paths in G and record them in a set S . Then, when processing queries, *T.Patterns* utilizes the pre-computed paths to construct the major

parts of the query results, which improves query performance. Nevertheless, *T.Patterns* does not guarantee exact query results, e.g., its answer for an EAP query might not be an actual EAP. In other words, *T.Pattern* trades query accuracy for efficiency. In our study, we aim to provide exact answers for the path queries. As a consequence, in our experimental study in Chapter 4, we compare our indexing technique with *CSA* and *CHT* instead of *T.Patterns*.

2.3.2 Other Techniques for Shortest Path Queries on Timetable Graphs

Apart from *CSA*, *CHT*, and *T.Patterns*, other techniques for timetable graphs can be divided into two categories. Techniques in the first category [50, 51, 68, 72, 84] convert G into a *time-expanded graph* G' , such that each spatiotemporal event in G (e.g., arrival of a vehicle b at a station u) is mapped to a node in G' , and two consecutive events (e.g., arrival of b at u , followed by its departure) is mapped to a directed edge in G' . For example, if there exists an edge e that departs from v_1 at time t_d and arrives at v_2 at time t_a , then this edge is first mapped to two nodes (v_1, t_d) and (v_2, t_a) in the time-expanded graph G' ; as these two events are consecutive, then an edge is added from (v_1, t_d) to (v_2, t_a) in G' . Based on G' , various techniques [50, 51, 68, 72, 84] have been proposed to accelerate path queries. However, as pointed out in [12, 14], such techniques are generally not comparable to the state-of-the-art methods that process queries on G instead of G' . The main reason is that they do not consider the special properties of the timetable graphs, and simply adopt the existing optimization techniques on G' , which actually complicates the problem, and hence is less effective than the methods that directly process queries in the timetable graphs. On the other hand, techniques in the second category [28, 32, 34, 38, 74, 75, 77, 80] considers query types that are more sophisticated than EAP, LDP, SDP, in that they take into account advanced query constraints, such as the cost of making a transfer, the fare of a ride, etc.

In addition, there is a line of research [27, 29, 33, 39, 42, 55, 66, 68, 72, 78] on route planning on *time-dependent graphs*, i.e., simple graphs where the length of each edge is a function of time. Such graphs are typically used to model road networks where the traveling time of an edge is a certain pre-defined function of time. For example, assume that the travel time of an edge e is 2 minutes during peak hours from 7:00am to 9:00am and 5:00pm to 7:00pm per day, while the travel time of e is 1 minute during the remaining off-peak hours. Then the travel time of the edge can be modeled as a piece-wise linear function $f(t)$ where t is the current time.

Techniques developed for time-dependent graphs can also be applied on timetable graphs (via a sophisticated mapping from timetables to time-dependent functions), but as shown in [13, 29, 46], they are inferior to *CHT* in terms of query performance on

timetable graphs. The main reason is similar to that in the first category, i.e. they ignore the characteristics of the timetable graphs, and the optimization techniques are not as effective as the ones devised directly on the timetable graph.

2.4 Constrained Shortest Path

Let $G = (V, E)$ be a directed graph with a vertex set V and an edge set E . Each edge $e \in E$ is associated with a *length* $\ell(e) \geq 0$ and a *cost* $c(e) \geq 0$. For a path $P = \langle e_1, e_2, \dots, e_k \rangle$ in G , the *length* and *cost* of P are defined as $\ell(P) = \sum_{i=1}^k \ell(e_i)$ and $c(P) = \sum_{i=1}^k c(e_i)$, respectively.

Given an origin vertex $s \in V$, a destination vertex $t \in V$, and a cost constraint θ , a *constrained shortest path (CSP)* query asks for the shortest path P among all paths from s to t with costs no more than θ . Let ℓ_{max} (resp. c_{max}) be the maximum length (resp. maximum cost) of an edge in G . The CSP problem has been proven to be NP-hard, if both ℓ_{max} and c_{max} can be arbitrarily large [45, 54]. On the other hand, if either ℓ_{max} or c_{max} is polynomial to the number of vertices, there exist polynomial-time solutions for CSP, e.g., [54, 64]. Nevertheless, these algorithms incur tremendous costs for large graphs, and, thus, are far from practical. As such, recently much effort has been devoted to solving approximate versions of the CSP problem. In our thesis, we follow a popular definition called α -CSP, which asks for a path with length no larger than α ($\alpha \geq 1$) times that of the optimal one, while the cost does not exceed θ . For the ease of exposition, we define the α -dominance relation as follows.

Definition 2.1 (α -dominance). Let P_1 and P_2 be two paths connecting the same origin and destination vertices. P_1 α -dominates P_2 iff $c(P_1) \leq c(P_2)$ and $\ell(P_1) \leq \alpha \cdot \ell(P_2)$. $\square$

Next, we present the state-of-the-art solutions for exact and approximate CSP queries in Section 2.4.1, and point out their drawbacks that render them inefficient for answering CSP queries. Other existing methods for exact CSP and approximate CSP queries are reviewed in Section 2.4.2.

2.4.1 State of the Art

Exact CSP without index. The state-of-the-art for index-free exact CSP processing is a classic algorithm in [53], which we call *Sky-Dijk* because it follows the general idea of Dijkstra’s algorithm [37]. The main difference between *Sky-Dijk* and Dijkstra’s algorithm is that the former incrementally maintains a set of paths at each vertex, rather than a single shortest path. Specifically, *Sky-Dijk* maintains a label set $L(v)$ for each vertex v , which contains the current set of skyline paths from the origin s to v , i.e., those

 Figure 2.4: An input graph G for CSP problem

not dominated by another path in $L(v)$. Similar to Dijkstra's algorithm, each $L(v)$ is initialized to empty and updated iteratively.

Meanwhile, akin to Dijkstra's algorithm, *Sky-Dijk* maintains a heap H of paths originating from s , in ascending order of their costs¹. Initially, H contains only one trivial path with no edge, which both starts and ends at the origin vertex s . Then, in each iteration, *Sky-Dijk* pops the top path P from H . Let v be the last vertex in P . If $v \neq t$, i.e., P has not reached the query destination, the algorithm enumerates each path P' that can be obtained by appending an edge (v, v') at the end of P , and checks whether P' exceeds the cost limit θ or is dominated by any path in $L(v')$. If so, P' is simply discarded; otherwise, *Sky-Dijk* adds P' to both H and $L(v')$, and updates $L(v')$ to eliminate paths dominated by P' . The algorithm terminates when H is empty, and returns the path in $L(t)$ with the minimum length, where t is the destination vertex.

Example 2.4. Consider the example in Figure 2.4 with an exact CSP query with origin $s = v_1$, destination $t = v_5$, and cost constraint $\theta = 6$. *Sky-Dijk* initializes the heap H with a trivial path P_0 from v_1 to v_1 with no edge, and zero cost / length. Then, it pops P_0 from H , and extends it to obtain paths $P_1 = \{(v_1, v_2)\}$ with cost $c_1 = 1$ and length $\ell_1 = 2$, and $P_2 = \{(v_1, v_3)\}$ with $c_2 = 3$ and $\ell_2 = 4$. The algorithm adds P_1 , P_2 to the label set $L(v_2)$ and $L(v_3)$ respectively, and both of them to H .

Next, P_1 is popped from H , and *Sky-Dijk* extends it to obtain paths $P_4 = \{(v_1, v_2), (v_2, v_3)\}$, $P_5 = \{(v_1, v_2), (v_2, v_4)\}$ and $P_6 = \{(v_1, v_2), (v_2, v_5)\}$, which are added to $L(v_3)$, $L(v_4)$ and $L(v_5)$ respectively. Note that now $L(v_3)$ contains two paths P_2 ($c_2 = 3, \ell_2 = 4$) and P_4 ($c_2 = 4, \ell_2 = 3$); neither dominates the other. Meanwhile, note that P_6 does not need to be inserted to the H , as it has reached the destination $t = v_5$, and, thus, cannot be extended further. After that, P_2 is popped from H ; extending P_2 generates $P_7 = \{(v_1, v_3), (v_3, v_5)\}$, which is eventually returned as the CSP result. The algorithm continues until H becomes empty, at which time it inspects $L(v_5)$, and returns P_7 , which is pinpointed with bold face as shown in Figure 2.4.

¹Note that, similar to the Dijkstra's algorithm, it suffices to store in H only the length, cost and the last two vertices for each path. For ease of presentation we assume H contains full paths.

According to [53], the worst-case time complexity of *Sky-Dijk* is $O(\ell_{max} \cdot m \cdot n \cdot \log(\ell_{max} \cdot n))$, where m (resp. n) is the number of edges (resp. vertices) in the graph, and ℓ_{max} is the maximum edge length. Clearly, *Sky-Dijk* is a polynomial-time algorithm when ℓ_{max} is polynomial to n , but its complexity is rather high due to the $\ell_{max} \cdot n \cdot m$ term. On real road networks, the performance of *Sky-Dijk* is also poor, since it does not optimize for such datasets at all. As shown in [87] and also in our experiments, *Sky-Dijk* incurs enormous costs for large graphs, and is clearly impractical.

α -CSP without index. The current state-of-the-art solution for α -CSP is Breugem’s method [20] (called *CP-Dijk* in the following), which resembles *Sky-Dijk* with a *conservative pruning (CP)* technique originally developed by Tsaggouris and Zaroliagis [90]. Specifically, given an α -CSP with origin s , destination t , cost constraint θ , and approximation ratio α , *CP-Dijk* applies the same data structures and follows the same steps as *Sky-Dijk*, with a single modification: that each label set $L(v)$ maintains the set of paths that are not $\sqrt[n]{\alpha}$ -dominated (Definition 5.2) by another path in $L(v)$, where n is the total number of vertices in the input graph G . Because $\sqrt[n]{\alpha}$ -dominance is a relaxed condition of exact (i.e., 1-) dominance, this modification leads to faster query processing. However, for a large graph, $\sqrt[n]{\alpha}$ is very close to 1 even for a large α . Consequently, the performance improvement of *CP-Dijk* over *Sky-Dijk* is often negligible, as shown in our experiments.

Unlike *Sky-Dijk*, *CP-Dijk* is a polynomial-time algorithm as long as each edge length is represented by an integer. Specifically, the time complexity of *CP-Dijk* is $O(\kappa \cdot m \cdot n \cdot \log(\kappa \cdot n))$, where $\kappa = \log(n \cdot \ell_{max}/\ell_{min})/(\alpha - 1)$ [20], ℓ_{max}, ℓ_{min} being the minimum and maximum values of an edge length, respectively. Note that κ is logarithmic to ℓ_{max} and ℓ_{min} , and, thus, is polynomial to the size of the problem description, since each edge length is represented by an integer, whose size is logarithmic to its value.

As explained before, in terms of practical performance, *CP-Dijk* obtains only marginal improvement over *Sky-Dijk*; nevertheless, *CP-Dijk* is at least no worse than *Sky-Dijk*. According to Section 2.4.2, other polynomial-time solutions for α -CSP can be far more costly. The fact that *CP-Dijk* is the state-of-the-art for α -CSP processing reveals that previous research focuses mostly on asymptotic complexity, not practical performance. Finally, if we know in advance the maximum number of nodes (say, k) on any path, *CP-Dijk* could use $\sqrt[k]{\alpha}$ in place of $\sqrt[n]{\alpha}$ to obtain higher pruning power [20]. Unfortunately, in practice it is usually hard to determine an appropriate value for k in advance.

Exact CSP with index. The current state-of-the-art for indexed CSP processing is *CSP-CH* [87], which accelerates *Sky-Dijk* through contraction hierarchies [48], an indexing technique that has been shown to be effective for accelerating conventional shortest path processing on road networks [97]. Similar to [48], in each iteration *CSP-CH* removes a vertex from the graph, and substitutes it with new shortcut edges for the remaining

vertices. Each shortcut (u, w) created during the removal of vertex v represents a path $\langle (u, v), (v, w) \rangle$ that is not dominated by any other path from u to w . After that, *CSP-CH* answers query with a bidirectional *Sky-Dijk* search from both the origin s and the destination t simultaneously, utilizing the shortcuts to reduce the number of nodes to be traversed.

The problem of *CSP-CH* is that unlike conventional shortest path search, in CSP there can be numerous shortcuts (i.e., multiple skyline sets) for each removed vertex, leading to a prohibitively large index size. *CSP-CH* uses heuristics to alleviate this problem, e.g., by adding only a selected set of shortcuts, and at the same time keeping the vertex in the graph instead of removing it. Such compromises, however, dramatically decrease the effectiveness of the index. Consequently, its query processing cost is still impractically high.

α -CSP with index. To our knowledge, we are not aware of any indexed solution for α -CSP processing. In sum, none of the state-of-the-art methods optimizes for road networks, applies indexing effectively, or obtains acceptable query time for large networks.

2.4.2 Other Techniques for Constrained Shortest Path

Joksche [64] first studies the CSP problem, and proposes a dynamic programming algorithm for exact CSP. Subsequently, Handler and Zang [52] propose two methods for exact CSP processing: one method formulates CSP as an integer linear programming (ILP) problem, and solves it with a standard ILP solver. This same methodology is used in Beasley and Christofides [15] and in Mehlhorn and Ziegelmann [71]. However, as shown in [71], these ILP-based solutions scale poorly, and incur tremendous processing costs on large road networks. The other solution in [52] reduces CSP to a *k-shortest path* problem, and repeatedly computes the next shortest path (in terms of total length) until reaching one that satisfies the cost constraint. Afterwards, Hansen [53] proposes an augmented Dijkstra’s algorithm [37] for exact CSP queries, and Skriver et al. [86] extend the Bellman-Ford algorithm [16, 41] to CSP queries; according to their experiments, their solution and Hansen’s [53] show similar performance, and they both outperform the *k-shortest-path* based solution in [52]. However, as shown in the experimental result in [86], these methods are still too slow for large road networks. The state-of-the-art methods for exact CSP are described in Section 2.4.1, for index-free and indexed processing, respectively.

Regarding α -CSP, Hansen [53] proposes the first solution, which runs in polynomial time but with a high complexity: $O(m^2 \cdot \frac{n^2}{\alpha-1} \log \frac{n}{\alpha-1})$. Following a similar idea as in [41], Warburton [94] obtains a lower time complexity; however, it is limited to acyclic graphs. This algorithm is later improved by Hassin [54] and Ergün et al. [40], but these methods are still limited to acyclic graphs. Lorenz and Raz. [70] extend Hassin’s method [54] to general graphs. However, this solution is orders of magnitude slower than an exact

CSP algorithm based on k-shortest path, as shown in [67]. Later on, Tsaggouris and Zaroliagis [90] propose a method with improved performance based on the conservative pruning technique. Breugem [20] refines the method in [90], leading to *CP-Dijk*, i.e., the current state-of-the-art for α -CSP as described in Section 2.4.1.

Delling et al. [35] study a related query, which returns the entire set of skyline paths between two given vertices. Their solution creates shortcuts similarly as *CSP-CH* [87], and can be adapted to answer CSP queries. However, this method is not scalable to large graphs, as shown in [44, 87]. In order to reach an acceptable processing time, [35] proposes to modify the problem setting by relaxing the definition of dominance. However, with this relaxation, the method can no longer be used to answer CSP or α -CSP queries. Another related query type is to find the shortest path in terms of a weighted sum of edge costs [43, 47]. These methods, however, cannot be used to answer CSP or α -CSP queries.

Besides, there also exist a line of research work on other constrained route planning queries [9, 18, 69, 81]. Among them, Ref. [18, 69, 81] study the shortest path queries that include label constraints. In this problem, each edge is associated with a label. Let U be the set of all labels in the graph. Given a source s , a target t , and a set $T \subseteq U$ of labels, the query then asks the shortest path that only goes through labels in T . Aljubayrin et al. [9] study the safest path via safe zones. In their problem, several safe zones are given, and the query asks the path from a source to a target that minimizes the travel time outside the safe zones.

2.5 Summary

In this chapter, we first review the approaches for reachability queries. We present *2-hop labelling* (*2HL*), an efficient index for reachability queries that strikes a good balance between space consumption and query performance. However, the proposed preprocessing algorithm on constructing *2HL* index incurs $O(n^5)$ time complexity and $O(n^2)$ space consumption, which renders it inapplicable for large graphs. We then review the existing state-of-the-art scheme for reachability queries: *Hierarchical Hub Labelling* (*HHL*), a subclass of *2HL*, that exploits a total order o on the vertices, and then construct the index based on the total order o . Afterwards, we review other existing solutions for the reachability queries, and pinpoint their deficiencies and limitations.

Then, we review the existing optimization techniques for the conventional shortest path and distance queries. We mainly present the techniques that are relevant to our studied two shortest path variants. In particular, we first review the Dijkstra's algorithm, which is the classic solution for the shortest path queries. Afterwards, we review *HHL*, which is the state-of-the-art solution for not only reachability queries, but also distance

queries. Meanwhile, road networks has some nice properties and we present two categories that make full use of the properties to improve the query efficiency. We first review *Contraction Hierarchies*, an efficient index scheme for shortest path queries on road networks. We demonstrate the details, and give examples to show how it works. Then, we review a notable class of solutions that uses graph partitioning to speed up the query performance; we present how the query algorithm works and explain why it leads to improved query efficiency.

Afterwards, we demonstrate the state-of-the-art methods for shortest path queries on timetable graphs. We first review *CSA*, a method that answer EAP, LDP, and SDP queries through a careful arrangement of edges by departure timestamps or arrival timestamps. The algorithm requires only a linear scan of the edges, which benefits from a good cache locality, and outperforms the Dijkstra's based solution. We explain the algorithm in details, and give examples to show how the query algorithm works. However, the query performance of *CSA* depends on the size of the graph, and is not scalable to large graphs. Then, we present *CHT*, a solution that adopts the technique of *CH* into timetable graphs. However, this algorithm still incurs considerable overheads for query, and we explain why this optimization technique is not as efficient on transportation networks as on road networks. The other solution *T.Patterns* does not guarantee correctness of the query result, and does not fit to our studied problem. After that, we briefly review some other existing solutions for shortest path queries on timetable graphs. However, such solutions does not consider the characteristics of the transportation networks, and is less efficient than the state-of-the-art methods.

Finally, we present a detailed survey on existing methods for the constrained shortest path queries. We present two Dijkstra-like algorithm to answer exact CSP queries and approximate CSP queries, respectively. These two solutions are also the state-of-the-art methods for index-free methods. However, such solutions are still too slow, and we present a solution that adopts the technique of *CH* into the CSP problem. The *CH* based solution, however, is still inefficient due to the large number of shortcuts added during the contraction. We further review some other existing work that solves CSP or approximate CSP queries. However, such solutions are generally too slow, and are not practical for real world applications. Among the existing works, none of them has studied how to produce an efficient index to answer approximate CSP queries or use the characteristic of road network to optimize the query performance. As we will see in Chapter 5, our indexing method make full use of the characteristic of the road network, and can provide significant better query performance for approximate CSP queries.

Chapter 3

Reachability Query

Given a directed graph G , an origin s and a target t , a reachability query asks whether there exists a path from s to t . The state of the art is a class of 2-hop labeling schemes [26] referred to as *Hierarchical Hub Labelings (HHL)*. In particular, an *HHL* index (implicitly or explicitly) defines a total order o on the vertices in G , and constructs label sets in accordance to o . Since o uniquely decides the construction of label sets, it determines the total size of an *HHL* index as well as its query performance. Therefore, a good total order o is of utmost importance for an *HHL* index.

This chapter presents a study on the derivation of a total order o for *HHL*. First, we formally prove that deriving an optimal o with minimum index size is NP-hard. Afterwards, we present an algorithm for vertex ordering that yields an *HHL* index whose size is at most $O(\sqrt{n})$ times the optimal one, where n is the number of vertices in G . To our knowledge, this is the first *HHL* method with non-trivial worst-case guarantees. Noteworthy, the $O(\sqrt{n})$ approximation guarantee is also the best approximation we can achieve as shown in an independent research work by Babenko et al. [10]. The algorithm, however, runs in $O(n^3)$ time and requires $O(n^2)$ space, which makes unsuitable for sizable graphs. Motivated by this, we also investigate heuristic solutions for vertex ordering. Specifically, we focus on reachability queries, and present a heuristic cost function for ordering vertex that is motivated from our theoretical study. We show that the cost function can be efficiently approximated by utilizing *reachability sketches* [25]. Finally, we experimentally evaluate our reachability solutions with a large variety of graphs. We show that our heuristic method yields comparable index size and query performance to our proposed approximate algorithm, and outperforms the other competitors.

The remainder of the chapter is organized as follows. We present the problem definition and revisit *HHL* in Section 3.1. Afterwards, we formally prove that deriving an optimal total order for *HHL* with minimum index size is NP-hard, and propose the greedy algorithm to compute a vertex ordering that achieves non-trivial performance guarantee in Section 3.2. Motivated by the theoretical study in Section 3.2, we develop a new

Table 3.1: Summary of Notations.

notation	description
$G = (V, E)$	the input DAG
n, m	the number of vertices and edges in G
$o(v)$	the level order of v
$\mathcal{L}$	an <i>HHL</i> index of G
$L_{in}(v), L_{out}(v)$	the in-label and out-label of vertex v
$\mathcal{B}_{in}(v), \mathcal{B}_{out}(v)$	the set of vertices that can reach v and v can reach in G
$\mathcal{R}$	the set of all pairs (u, v) such that u can reach v

heuristic technique for vertex ordering in Section 3.2.5. Then, we demonstrate the efficiency of our approaches on various large datasets, compared with several state-of-the-art techniques. Finally, we summarize this chapter in Section 3.4.

3.1 Preliminaries

This section defines the problem that we study in this chapter, and revisits *HHL*.

3.1.1 Problem Definition

Let $G = (V, E)$ be a directed graph with a set V of vertices and a set E of edges. Let $|V| = n$ and $|E| = m$. For any two vertices s and t in V , if there exists a directed path in G that starts from s and ends at t , then we say s can *reach* t , denoted by $s \rightarrow t$. If s cannot reach t , we denote it as $s \nrightarrow t$. A *reachability query* from s to t returns *true* if s can reach t , and *false* otherwise.

We say that s and t are *strongly connected* if both $s \rightarrow t$ and $t \rightarrow s$ holds. A *strongly connected component (SCC)* of G is defined as a *maximal* subset of V , such that any two vertices in the SCC are strongly connected. Note that all SCCs of G are disjoint, i.e., there does not exist a vertex that belongs to more than one SCC.

Observe that $s \rightarrow t$ holds if and only if one of the following conditions holds: (i) s and t are in the same SCC, or (ii) there exists a directed path from the SCC containing s to the SCC containing t . Therefore, for the purpose of reachability queries, we can simplify G into a *directed acyclic graph (DAG)* as follows. First, we map each SCC C of G to a vertex $f(C)$ in the DAG. After that, for any two SCCs C and C' , if there exists a directed edge in G that starts from a vertex in C and ends at a vertex in another SCC C' , we create a directed edge from $f(C)$ to $f(C')$ in the DAG. As such, for any two vertices s

and t , we can decide whether $s \rightarrow t$ holds, by first retrieving the vertices $f'(s)$ and $f'(t)$ in the DAG that s and t are mapped to, respectively, and then checking whether any of the following conditions holds: $f'(s) = f'(t)$, or $f'(s) \rightarrow f'(t)$ in the DAG.

In the remainder of this chapter, we assume that G has been processed into a DAG with the above method, which can be done in $O(|V| + |E|)$ time [88]. For convenience, we abuse notation and use G to denote the DAG. Table 3.1 lists the notations that we will frequently use.

3.1.2 Hierarchical Hub Labelling Revisited

As mentioned earlier, *Hierarchical Hub Labelling (HHL)* is a special case of the 2-hop labeling scheme proposed by Cohen et al. [25]. In a 2-hop labelling index, each vertex $v \in V$ is associated with an *in-label* set $L_{in}(v)$ and an *out-label* set $L_{out}(v)$. In particular, $L_{in}(v)$ contains some vertices in G that can reach v , while $L_{out}(v)$ consists of some vertices in G that v can reach. The label sets of the vertices are constructed in such a way that, for any two vertices s and t , we have $s \rightarrow t$ if and only if the following equation holds:

$$(L_{out}(s) \cup \{s\}) \cap (L_{in}(t) \cup \{t\}) \neq \emptyset \quad (3.1)$$

In other words, if we are to answer a reachability query from s to t , it suffices to scan the out-label set of s and the in-label set of t to see if they contain a common vertex. As such, the total size of the label sets of a 2-hop labelling index $\mathcal{L}$, denoted as $|\mathcal{L}|$, is often used as an indicator of the quality of $\mathcal{L}$. Formally,

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

Let o be a total order on V , and $o(v)$ be an integer in $[1, n]$, such that $o(v) = i$ if and only if v is the i -th vertex in o . We refer to o as a *level order*, and $o(v)$ as the *level* of v in o . *Hierarchical Hub Labelling (HHL)* [6, 101] is a subclass of 2-hop labelling indices that implicitly or explicitly defines a total order o on V , and satisfies the following constraints for $u \in L_{in}(v)$ (resp. $u \in L_{out}(v)$):

- *Reachability Constraint*: $u \rightarrow v$ (resp. $v \rightarrow u$);
- *Level Constraint*: $o(u) < o(v)$;
- *Path Constraint*: There does not exist a directed path from u to v (resp. from v to u) that contains a vertex w with $o(w) < o(u)$.

Given a total order o , an *HHL* index can be easily constructed using Algorithm 1. In particular, given G and o , we first initialize the label sets of all vertices to $\emptyset$ (Line 1). After that, we retrieve the vertex $v \in V$ with $o(v) = 1$, and identify the set $B_{in}(v)$ of vertices that can reach v in G (Lines 3-4). Then, for each vertex $u \in B_{in}(v)$, if v 's in-label set and u 's out-label contains no common vertex, then we add v into u 's out-label set (Lines 5-7). Next, we inspect each vertex w that v can reach in G , and insert v into w 's in-label set if v 's out-label set and w 's in-label set do not overlap (Lines 8-11). Finally, we remove v from G . The vertices in G with levels $2, 3, \dots, n$ are processed iteratively in a similar manner.

From Algorithm 1, it can be observed that an *HHL* index is uniquely decided by its level order o . Therefore, choosing a good o is essential to the construction of an efficient *HHL* index. The following example illustrates how o may affect $|\mathcal{L}|$:

Example 3.1. Consider a graph G that contains n nodes $v_1, v_2, \dots, v_n$, and contains a path P that traverses v_i in ascending order of i . For simplicity, assume that G contains no other edges except for those in P , and that n is a power of 2. Suppose that $o(v_i) = i$ for $1 \leq i \leq n$. Then, the size of the corresponding $\mathcal{L}$ is $O(n^2)$, since each v_i would be added into the label set $L_{in}(v_j)$ for all v_j with $j > i$.

Now consider that we construct the index with a different node order o , in n iterations as follows. In the first iteration, we set $o(v_{n/2}) = 1$, and then remove $v_{n/2}$ from P , obtaining two sub-paths of P . After that, in the j -th iteration ($j > 1$), we select the longest sub-path of P in our collection, and identify the node v' exactly in the middle of the sub-path; then, we set $o(v') = j$ and remove v' from the sub-path, which breaks the sub-path into two smaller sub-paths. With such an ordering, each node v' is added only into the label sets of the nodes on the sub-path from which v' is removed, and hence, the resulting $\mathcal{L}$ has a size of $O(n \log n)$. $\square$

As a consequence, choosing a good order o for *HHL* index is of utmost importance. Existing *HHL* indices [23, 60, 98, 101] have adopted various heuristics to derive good level orders. However, they all fail to provide non-trivial performance guarantees. In the next section, we will show that deriving an optimal o with minimum size is NP-hard. Then, we present an algorithm for selecting o that yields an $O(\sqrt{n})$ -approximation in terms of the size of the *HHL* index decided by o .

3.2 Deciding Level Order

This section presents a study on how to decide the level order o for the vertices. First, we reduce a well-known NP-hard problem to the problem of deriving an optimal *HHL* index, and prove it to be NP-hard in Section 3.2.1. As a result, deriving an optimal order

Algorithm 1: Preprocessing Algorithm for *HHL*

input : A DAG $G = (V, E)$ and a level order o
output: An *HHL* index $\mathcal{L}$

```

1 set the in-label sets and out-labels of all vertices to  $\emptyset$ ;
2 for  $i = 1$  to  $n$  do
3   let  $v$  be the vertex with  $o(v) = i$ ;
4   let  $B_{in}(v)$  be the set of vertices that can reach  $v$  in  $G$ ;
5   for any vertex  $u \in B_{in}(v)$  do
6     if  $L_{out}(u) \cap L_{in}(v) = \emptyset$  then
7        $\quad$  insert  $v$  into  $L_{out}(u)$ ;
8   let  $B_{out}(v)$  be the set of vertices that  $v$  can reach in  $G$ ;
9   for any vertex  $w \in B_{out}(v)$  do
10    if  $L_{out}(v) \cap L_{in}(w) = \emptyset$  then
11       $\quad$  insert  $v$  into  $L_{in}(w)$ ;
12  remove  $v$  from  $G$ ;
13 return all label sets constructed;
    
```

o that minimizes the *HHL* index size will be inherently NP-hard. Afterwards, we clarify the rationale and challenge in deriving an approximate ordering method in Section 3.2.2. Next, we present our approximation algorithm in Section 3.2.3. Finally, we prove the approximation guarantee of our algorithm and analyze the complexity in Section 3.2.4.

3.2.1 Deriving Optimal *HHL* Index

As shown in Section 3.1, an *HHL* index is uniquely decided by the given level ordering. As such, the choice of o has a profound effect on the size of the *HHL* index $\mathcal{L}$. In particular, the average query performance of *HHL* is solely decided by the index size. As a consequence, obtaining an *HHL* index with minimum size is of utmost importance. Nevertheless, deriving an optimal *HHL* index is challenging. In the following we prove that deriving an optimal *HHL* index is NP-hard. This indicates that it will be in vain to derive a polynomial time ordering algorithm that minimizes the *HHL* index. We present with the following theorem.

Theorem 3.1. *Given a directed graph $G = (V, E)$, obtaining an optimal *HHL* index with minimum size for G is NP-hard.*

Proof. We make a reduction from Exact Cover by 3-Set (X3C). Formally, given a finite set U , with $3 \cdot n$ elements, and a collection $\mathcal{C}$ of three-element subset of U , the X3C problem asks if there exists a subcollection $\mathcal{C}'$ of $\mathcal{C}$ such that every element of U occurs

in exactly one set in $\mathcal{C}'$. This problem has been shown to be NP-hard [56]. Next, we explain how to construct an instance of *HHL* given an arbitrary X3C problem instance.

Let X be an instance of X3C with $U = \{x_1, x_2, \dots, x_{3n}\}$, and let $\mathcal{C} = \{C_1, C_2, \dots, C_m\}$ ($m > n, C_1 \cup C_2 \cup \dots \cup C_m = U$) be a set of 3-sets, i.e., each C_i is a set consisting of three different variables in U . We construct a graph G as follows:

- For each 3-set C_i , we construct a vertex c_i ;
- For each variable x_i , we construct two vertices $v_1^{x_i}, v_2^{x_i}$.
- We add two auxiliary vertices a_1 and a_2 . In addition, we add directed edges (a_1, a_2) and (a_2, c_i) for all c_i .
- We add an edge $(c_i, v_1^{x_i})$ if and only if $x_i \in C_i$.
- We add an edge $(v_1^{x_i}, v_2^{x_i})$ for all x_i .

For example, given an X3C instance with $U = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and $\mathcal{C} = \{C_1, C_2, C_3\}$ where $C_1 = \{x_1, x_2, x_3\}, C_2 = \{x_2, x_3, x_4\}$ and $C_3 = \{x_4, x_5, x_6\}$, the constructed graph G as shown in Figure 3.1.

We claim that for any X3C instance X , there is an exact cover $\{C'_1, C'_2, \dots, C'_n\}$ for X if and only if the *HHL* index for its corresponding G has a minimal index size $1 + 4m + 7n$.

We prove the claim as follows. Assume that there is an exact cover $\{C'_1, C'_2, \dots, C'_n\}$ for X . Denote their corresponding vertices in G as $c'_1, c'_2, \dots, c'_n$, respectively. We define a total order ℓ for all vertices in G : for $c'_1, c'_2, \dots, c'_n$, we set their order as $1, 2, \dots, n$, respectively; for a_2 , we set its order as $n + 1$; for $v_1^{x_i}$, we set its order as $1 + n + i$; for $v_2^{x_i}$, we set its order as $1 + 4n + i$; for vertices c_i not in the exact cover solution, we first sort them based on i and denote them as $c_{1'}, c_{2'}, \dots, c_{(m-n)'}$, and then set their order as $1 + 7n + 1, 1 + 7n + 2, \dots, 1 + 6n + m$; finally, we set a_1 's order as $2 + 6n + m$. Given the above order, it can be verified that the *HHL* index size is $1 + 4m + 7n$. Next we prove that the optimal index size is $1 + 4m + 7n$.

First of all, for each directed edge (u, v) , either u is in v 's in-label or v is in u 's out-label, which contributes to $1 + 4m + 3n$ labels. Next it is easy to verify that a_2 can reach $v_2^{x_i}$ for any i . To derive correct query result, *HHL* has to add at least $3n$ labels, e.g., by adding c'_i to $v_2^{x_k}$'s out-label, where $C'_i \cap \{x_k\} \neq \emptyset$. Besides, a_1 can reach $v_1^{x_i}$ for any i , which requires to add at least n labels to correctly answer the reachability query, e.g., by adding c'_i to a_1 's out-label. So the total label size is at least $1 + 4m + 7n$. As we can obtain an *HHL* index size of $1 + 4m + 7n$, the optimal index size is $1 + 4m + 7n$.

Next we prove that given an *HHL* index of minimal size $1 + 4m + 7n$ for G , we can obtain an exact cover for X . We denote an ordered pair (u, v) as a *reachable pair* if u

 Figure 3.1: Constructed graph G for an X3C instance.

can reach v . We say that (u, w) is covered by a vertex v if $v \in L_{in}(w) \vee (v = w)$ and $v \in L_{out}(u) \vee (v = u)$. As discussed above, all directed edges are required to add a label into HHL index, which requires $1 + 4m + 3n$ labels. Besides, for the reachable pairs $(a_2, v_2^{x_i})$, it requires HHL to add $3n$ labels to cover them. Since G has HHL index of minimal size $1 + 4m + 7n$, the reachable pairs $(a_1, v_1^{x_i})$ can be covered with a minimal n labels. However, it is easy to verify that there are at least $3n$ paths of the form $(a_1, v_1^{x_i})$. This can be satisfied if and only if there exist n 3-sets $C'_1, C'_2, \dots, C'_n$ which covers all $3n$ elements in U , which indicates them as an exact cover for X .

This finishes the proof that minimizing HHL index size is NP-hard. $\square$

In the literature, some research work on HHL has developed several heuristics to obtain a satisfactory total order o , e.g., based on vertex degrees [60, 98], topological order of vertices [23]. However, none of those heuristics yields non-trivial asymptotic guarantees in terms of index size or query time. Next, we present our approximate solution for HHL index, which provides $(\sqrt{n})$ -approximate guarantee in terms of index size.

3.2.2 Rationale and Challenge

Recall that every HHL index is a 2-hop labelling index, and hence, it must ensure that Equation 3.1 holds for any vertices $s, t \in G$ such that $s \rightarrow t$. That is, $L_{out}(s) \cup \{s\}$ and $L_{in}(t) \cup \{t\}$ must contain at least one common vertex. We refer to this requirement as the *2-hop requirement*. In the following, we first present an intuitive way to understand the 2-hop requirement, and then explain the rationale behind our approximate algorithm, as well as the challenge that we face.

First, recall that a *reachable pair* (u, w) is an ordered pair of vertices $u, w \in V$, such that $u \rightarrow w$. Also (u, w) is *covered* by a vertex v , if $v \in L_{out}(u) \vee u = v$ and $v \in L_{in}(w) \vee v = w$. Let $\mathcal{R}$ be the set of all reachable pairs in G . Then, imposing the

2-hop requirement is equivalent to requiring every reachable pair in $\mathcal{R}$ to be covered by at least one vertex.

Now consider the i -th vertex v_i with $o(v_i) = i$ that need to be inserted into, intuitively, the insertion of v_i into other vertices' label sets can be regarded as using v_i to cover some reachability pairs in $\mathcal{R}$. Accordingly, we can regard *HHL* as using n vertices to cover all reachability pairs in $\mathcal{R}$.

The i -th vertex v_i incurs a space cost of $O(c_i)$, where c_i denotes the number of label sets where v_i is inserted into. In other words, there is a cost of using v_i to cover reachability pairs, and the cost equals c_i . Therefore, the minimizing of the *HHL* index size is equivalent to minimizing the cost incurred by using $v_1, v_2, \dots, v_n$ in turn to cover reachability pairs.

One may observe that the above coverage problem is somewhat similar to the problem of *weighted set cover* [92]. In particular, in the weighted set cover problem, we are given a collection Λ of elements and a collection Θ of sets, such that each set $\theta \in \Theta$ contains a number of elements in Λ and is associated with a cost $c(\theta)$. The objective is to identify a subset of Θ with the minimum total cost that cover all elements in Λ . This problem is known to be NP-hard, but there exists polynomial-time algorithms that return $\log(\sum_{\theta \in \Theta} c(\theta))$ -approximations [92].

The weighted set cover problem is related to ours because, intuitively, we may regard each reachable pair as an element in Λ , and each vertex v_i as a set in Θ that contains the reachable pairs covered by v_i . However, there exists a crucial difference between the two problems: in weighted set cover, each $\theta \in \Theta$ has a fixed cost, whereas in our problem, the cost c_i of a vertex v_i depends on the vertices $v_1, v_2, \dots, v_{i-1}$ processed before v_i . In particular, based on the three constraints, v_i is inserted into the out-label set of a vertex u , if and only if there does not exist a vertex v' satisfying $o(v') < o(v_i)$ and $u \rightarrow v' \rightarrow v_i$. In addition, v_i is added into the in-label set of a vertex w , if and only if there does not exist a vertex v' satisfying $o(v') < o(v_i)$ and $v_i \rightarrow v' \rightarrow w$. As such, the approximate solution for weighted set cover does not work for our problem.

One may also consider applying Cohen et al.'s approximate solution for 2-hop labeling [26] to address our problem. In particular, Cohen et al. utilizes a mapping to the weighted set cover problem, such that each reachable pair is mapped to an element in Λ . Meanwhile, each vertex v is mapped to multiple sets in Θ , such that each set contains a subset S of the reachability pairs that v may cover, and the cost of the set is decided by S . Given this mapping, Cohen et al.'s solution applies a greedy approach to iterative select sets in Θ to cover all reachable pairs. (Note that the greedy approach may select multiple sets that correspond to the same vertex v , in an arbitrary order.) The resulting 2-hop labeling index, however, may not be an *HHL* index since the Cohen et al.'s greedy approach does not take into account any total ordering of vertices. In general, we find that it is rather difficult modifying Cohen et al.'s approach to accommodate the constraints imposed by *HHL*.

 Figure 3.2: A DAG G .

3.2.3 Approximation Algorithm

Intuitively, if a vertex v covers a large number of reachable pairs, v is likely to be an important vertex for reachability queries. We use $\mathcal{R}(v)$ to denote the reachable pairs covered by v . Let $v_1, v_2, \dots, v_{i-1}$ to denote the order node with the 1st, 2nd, $\dots$, $(i-1)$ -th rank in our level order o . We further denote the reachable pairs covered by node v but not by $v_1, v_2, \dots, v_{i-1}$ as $C_i(v)$. More formally:

$$C_i(v) = \mathcal{R}(v) \setminus \bigcup_{j=1}^{i-1} \mathcal{R}(v_j)$$

Meanwhile, as we target to minimize the index size, the number of labels added required by v to cover the reachable pairs will also have a great impact. Suppose, we order v as the i -th vertex in the total order o , then the set of reachable pairs that it needs to cover is exactly C_i . Based on Algorithm 1, it requires to add v into the labels of the remaining vertices to cover the reachable pairs in $C_i(v)$. Define the influence set of a vertex v (denoted as $I_i(v)$) as the set of reachable pairs of the form (u, w) that either $u = v$ or $w = v$ in $C_i(v)$. Then the increased label size is at least $|I_i(v)|$. Meanwhile, observe that if a reachable pair $(u, w) \in C_i(v)$ can be covered by v , then (u, v) and (v, w) will also be covered by v . In other words, it is enough to add v into the out-label or in-label of the vertices in $I_i(v)$. As a result, the number of labels required for v is exactly $|I_i(v)|$.

Our approximate solution targets to find a balance between the importance of a vertex v , i.e., the number of covered reachable pairs, and the space consumption incurred by v . In particular, denote $|C_i(v)|/|I_i(v)|$ as the *cost coverage ratio* of a vertex v . We first choose v_1 to be the vertex with minimum cost coverage ratio. Afterwards, we decide v_i in ascending order of i , and we set v_i to the vertex that has the minimum cost coverage ratio, i.e.,

$$v_i = \arg \min_v \frac{|C_i(v)|}{|I_i(v)|}$$

We illustrate how vertices are ordered with an example.

Example 3.2. Consider the DAG shown in Figure 3.2. Initially, vertex a covers 9 reachable pairs, i.e., $C(a) = \{(a, b), (a, c), (a, g), (a, h), (d, a), (d, b), (d, c), (d, g), (d, h)\}$.

The influence set of a is $I(a) = \{(a, b), (a, c), (a, g), (a, h), (d, a)\}$, which indicates that if a is ordered as the first vertex, it requires to add 5 labels to $\mathcal{L}$. The cost coverage ratio of a is $\frac{5}{9}$. It is easy to verify that a has the lowest cost coverage ratio, and a is chosen as the first node ranked in o , i.e., $o(a) = 1$.

To order the next vertex, it further checks the cost coverage ratio of other vertices. For example, $C_1(b) = \{(b, h), (e, b), (e, h)\}$, and it does not include (d, b) since it has been covered by vertex a . Meanwhile, $I_1(b) = \{(b, h), (e, b)\}$. The cost coverage ratio of b is $\frac{2}{3}$. As b has the lowest cost coverage ratio, b is ranked as the second vertex in o , i.e., $o(b) = 2$. It repeats this process, and finally outputs the total order for the vertices, i.e., $o(a) = 1, o(b) = 2, \dots, o(h) = 8$. $\square$

Algorithm 2 presents the pseudo-code of the approximated algorithm, dubbed *Average-Cost-Greedy (AC-Greedy)*. Note that AC-greedy returns the level order of the vertices, which uniquely decides the *HHL* index. Next, we present the analysis for AC-Greedy algorithm in terms of approximation guarantee, and complexity.

3.2.4 Approximation Guarantee and Complexity Analysis

Let o^{ac} be the level order generated with AC-greedy algorithm. Let o^{opt} be a level order that produces the minimal *HHL* index size. Let $\mathcal{L}^{ac}$ and $\mathcal{L}^{opt}$ denote the *HHL* corresponding to o^{ac} and o^{opt} , respectively. The approximation ratio is defined as $\frac{|o^{ac}|}{|o^{opt}|}$. In the following, we use n to denote the number of vertices and m to denote the number of edges in G . We then have Theorem 3.2.

Theorem 3.2. *The approximation ratio of the AC-Greedy algorithm is $2\sqrt{n}$ in the worst case.*

Before proving Theorem 3.2, we first show a lower bound and an upper bound of $I_i(v)$ based on $C_i(v)$ and n with the following lemma¹.

Lemma 3.1. $\sqrt{|C_i(v)|} \leq |I_i(v)| \leq \sqrt{n} \cdot \sqrt{|C_i(v)|}$.

Lemma 3.1 is important in that it provides us a means to bound the number of labels added due to vertex v , using only the number of reachable pairs newly covered by v , without looking into the actual number of label sets that v is inserted into. With this lemma, we are ready to prove Theorem 3.2.

Proof of Theorem 3.2. Let v_i be the i -th vertex in o^{ac} . Recall that $C_i(v_i)$ is the set of reachable pairs that is covered by v_i . For each reachable pair r in $C_i(v_i)$, we define

¹All omitted proofs in this chapter can be found in Appendix A.

$c(r) = \frac{|C_i(v_i)|}{|L_i(v_i)|}$ as the cost of r that contributes to the label size of $\mathcal{L}^{ac}$. Recall that AC-Greedy chooses the vertex with the minimum cost coverage ratio. With the definition of cost of a reachable pair, AC-Greedy can be also interpreted as choosing the vertex that makes its covered reachable pairs incurs the least cost.

Then, the label size can be reformulated as:

$$|\mathcal{L}^{ac}| = \sum_{r \in \mathcal{R}} c(r). \quad (3.2)$$

Next, consider the optimal ordering o^{opt} , with v'_i being the i -th vertex in o^{opt} . Let $C'_i(v'_i)$ be the reachable pairs covered by v'_i but not by $v'_1, v'_2, \dots, v'_{i-1}$. Let $I'_i(v'_i)$ be v'_i influence set on $C'_i(v'_i)$. It is easy to verify that $C'_i(v'_i)$ and $I'_i(v'_i)$ also satisfy Lemma 3.1.

Let $r_{i,j}$ be the j -th tuple in $C'_i(v'_i)$ that will be covered if we choose the vertex with the order given by AC-Greedy algorithm. Suppose $(k-1)$ vertices has been ordered by AC-Greedy before it covers $r_{i,j}$. Then if we choose v'_i as the k -th vertex, the number of reachable pairs that it will cover is $C_k(v'_i)$, and the influence set will be $I_k(v'_i)$. By how AC-Greedy algorithm works, it guarantees that the cost of $r_{i,j}$ will not exceed the cost it bears by choosing v'_i to cover $r_{i,j}$. Meanwhile, observe that if we choose v'_i to cover $r_{i,j}$, the number of reachable pairs it covers is at least $|C'_i(v'_i)| - j + 1$, i.e., $|C_k(v'_i)| \geq |C'_i(v'_i)| - j + 1$.

Then we have:

$$\begin{aligned} c(r_{i,j}) &\leq \frac{|I_k(v'_i)|}{|C_k(v'_i)|} \\ &\leq \frac{\sqrt{n}}{\sqrt{|C_k(v'_i)|}} \quad (\text{by lemma 3.1: } |I'_i(v)| \leq \sqrt{n} \cdot \sqrt{|C'_i(v)|}) \\ &\leq \frac{\sqrt{n}}{\sqrt{|C'_i(v'_i)| - j + 1}} \quad (\text{by } |C_k(v'_i)| \geq |C'_i(v'_i)| - j + 1) \\ &\leq \frac{\sqrt{n}}{\sqrt{|C'_i(v'_i)|}} \cdot \frac{I'_i(v'_i)}{\sqrt{|C'_i(v'_i)| - j + 1}} \\ &\quad (\text{by lemma 3.1: } \sqrt{|C'_i(v)|} \leq |I'_i(v)|) \end{aligned} \quad (3.3)$$

On the other hand, we have

$$\frac{1}{\sqrt{1}} + \frac{1}{\sqrt{2}} + \dots + \frac{1}{\sqrt{|C'_i(v'_i)|}} \leq 2\sqrt{|C'_i(v'_i)|}. \quad (3.4)$$

Algorithm 2: AC-Greedy Algorithm

input : A DAG $G = (V, E)$
output: A level order o for V

- 1 let $G_1 = G$;
- 2 **for** $k = 1, \dots, |V|$ **do**
- 3 choose the vertex v in G_k that minimizes $\frac{|C_k(v)|}{|I_k(v)|}$ and set $o(v) = k$;
- 4 remove v from G_k , denote resulting graph as G_{k+1} ;
- 5 update $\frac{|C_k(v)|}{|I_k(v)|}$ with G_{k+1} for vertices v that are not ordered yet.
- 6 **return** o ;

Combining Equations 3.2-3.4, we have the following:

$$\begin{aligned}
 |\mathcal{L}^{ac}| &= \sum_{r \in \mathcal{R}} c(r) \\
 &= \sum_{i=1}^n \sum_{j=1}^{|C'_i(v'_i)|} c(r_{i,j}) \\
 &\leq \sum_{i=1}^n \sum_{j=1}^{|C'_i(v'_i)|} \frac{\sqrt{n}}{\sqrt{|C'_i(v'_i)|}} \cdot \frac{I'_i(v'_i)}{\sqrt{|C'_i(v'_i)|} - j + 1} \\
 &\leq \sum_{i=1}^n \frac{\sqrt{n}}{\sqrt{|C'_i(v'_i)|}} \cdot I'_i(v'_i) \cdot \left(\frac{1}{\sqrt{1}} + \frac{1}{\sqrt{2}} + \dots + \frac{1}{\sqrt{|C'_i(v'_i)|}} \right) \\
 &\leq \sum_{i=1}^n \frac{\sqrt{n}}{\sqrt{|C'_i(v'_i)|}} \cdot I'_i(v'_i) \cdot 2\sqrt{|C'_i(v'_i)|} \\
 &= 2\sqrt{n} \cdot \sum_{i=1}^n I'_i(v'_i) = 2\sqrt{n} \cdot |\mathcal{L}^{opt}|
 \end{aligned} \tag{3.5}$$

This finishes the proof of Theorem 3.2. □

Noteworthily, the $(\sqrt{n})$ -approximation is actually the best approximation guarantee that we can achieve as shown later in an independent research work by Babenko et al. [10]. We refer interested readers to [10] for the detailed examples.

Meanwhile, the following theorem establishes the complexity of AC-Greedy.

Theorem 3.3. *AC-Greedy runs in $O(n^3)$ time and takes $O(n^2)$ space.*

3.2.5 Heuristics

While AC-Greedy provides a non-trivial performance guarantee in terms of index size, its $O(n^3)$ time complexity and $O(n^2)$ space complexity renders it inapplicable for large graphs. The high time and space complexity of AC-Greedy is mainly caused by the fact that it requires computing the cost coverage ratio for each vertex v and update the cost coverage ratio when a vertex is ordered, which is difficult to derive. To address this issue, we propose to heuristically approximate the cost coverage ratio with a measure that can be efficiently computed.

Let v_i be the vertex with $o(v_i) = i$, and G_i be the graph after removing $v_1, v_2, \dots, v_{i-1}$ (and incident edges) from G . Denote $S_{in}(v_i, G_i)$ be the set of vertices that can reach v_i in G_i , and $S_{out}(v_i, G_i)$ be the set of vertices that v_i can reach in G_i ; $|S_{in}(v_i, G_i)|$ and $|S_{out}(v_i, G_i)|$ as the *in-score* and *out-score* of v_i in G_i , respectively. Then, it can be verified that the following equation holds:

$$C_i(v_i) \leq |S_{in}(v_i, G_i)| \cdot |S_{out}(v_i, G_i)| + |S_{in}(v_i, G_i)| + |S_{out}(v_i, G_i)| \quad (3.6)$$

In addition, the inequality is tight when every reachable pairs (s, t) covered by v_i is not covered by $v_1, v_2, \dots, v_{i-1}$. We use $\mathcal{C}(v_i)$ to denote the right hand side of Equation 3.6.

Similarly, the following inequality holds for $I_i(v_i)$:

$$\begin{aligned} I_i(v_i) &= \left| \{s \mid (s, t) \in r(v_i)\} \setminus \{v_i\} \right| + \left| \{t \mid (s, t) \in r(v_i)\} \setminus \{v_i\} \right| \\ &\leq |S_{in}(v_i, G_i)| + |S_{out}(v_i, G_i)|. \end{aligned} \quad (3.7)$$

This inequality is also tight when every reachable pair (s, t) covered by v_i is not covered by $v_1, v_2, \dots, v_{i-1}$, and satisfies that $s \in S_{in}(v_i, G_i) \cup \{v_i\}$ and $t \in S_{out}(v_i, G_i) \cup \{v_i\}$. We use $\mathcal{I}(v_i)$ to denote the right hand side of Equation 3.7. Intuitively, given Equations 3.6 and 3.7, we can use $\mathcal{I}(v_i)/\mathcal{C}(v_i)$ as a heuristic approximation of $\frac{|I_i(v_i)|}{|C_i(v_i)|}$.

However, deriving the exact values of $\mathcal{I}(v_i)$ and $\mathcal{C}(v_i)$ is still time consuming, because (i) the computation of $\mathcal{I}(v_i)/\mathcal{C}(v_i)$ requires us to derive the in-score and out-score of v using BFS (or DFS) on G , and (ii) the in-score and out-score of each vertex need to be recomputed whenever another vertex is removed from G . To address this issue, we propose to approximate in-score and out-score of each remaining vertex v in G by using k-min sketch [25].

Remarks. Our previous work [101] considers using a function similar to $\mathcal{I}(v_i)/\mathcal{C}(v_i)$ for deriving level orders, but does not provide a strong motivation; it also points out the large overheads incurred by the function, and resorts to simple heuristics instead. We provide a strong motivation here by establishing its connection to $\frac{|I_i(v_i)|}{|C_i(v_i)|}$, which is theoretically justified function for ranking vertices in *HHL*. Meanwhile, the previous proposed two

heuristics only provide coarse estimations on the function, while the proposed heuristic method is based on advanced reachability sketches, which will be more accurate. As we will see in our experiment, our sketch based method consistently provides better ordering, and outperforms those in [101] in terms of index size and query time.

Next we present an estimation approach by using k -min reachability sketch [25]. For ease of exposition, we will focus on the estimation of out-score, but the same method can be applied to estimate in-score. We first briefly review how k -min sketch works, and then present how to update the sketch efficiently when vertices are removed.

3.2.5.1 Estimation with k -min sketch

The k -min reachability sketch is constructed in k iterations, where k is a user-specified parameter. In the i -th iteration, we generate a random key X_v^i for each vertex v , by sampling from a uniform distribution on $[0, 1]$. Then, we compute, for each vertex v , the minimum value of the random keys of the vertices it can reach in G . We denote this minimum value as $O_{min}^i(v)$. More formally,

$$O_{min}^i(v) = \min_{u \in S_{out}(v, G)} X_u^i.$$

If $S_{out}(v, G) = \emptyset$, we define $O_{min}^i(v) = 1$. As shown in [25], we can compute $O_{min}^i(v)$ for all v simultaneously, in $O(m + n)$ time. Given that we have k iterations, the total time overhead to construct the k -min sketch is $O(k \cdot (m + n))$. The sketch then consists of k values $O_{min}^i(v)$ ($i \in [1, k]$) for each vertex v . Once the sketch is constructed, we have an unbiased estimator of out-score for each vertex v as follows:

$$\hat{S}_{out}(v) = \frac{k}{\sum_{i=1}^k O_{min}^i(v)} - 1 \quad (3.8)$$

The rationale behind this estimator is that given a uniform random sample of N values from $[0, 1]$, the expected minimal value is $1/(N + 1)$; accordingly, given a minimal value O_{min} with N unknown, the expected value of N is $1/O_{min} - 1$. The larger the sketch parameter k is, the more accurate the estimation will be. The estimation has been shown to have a coefficient of variation (CV) no more than $\sqrt{1/(k - 2)}$, where CV denotes the standard deviation of a random variable divided by its mean.

When another vertex is removed, $O_{min}^i(v)$ needs to be revised accordingly. To obtain the new $O_{min}^i(v)$ values, a naive approach is to reconstruct the sketch on the version of G where ordered vertices are removed. This, however, incurs significant computation overheads. To circumvent this problem, we observe that the keys X_u^i for all vertices are randomly generated, and hence, we can still use them and only need to update the O_{min} values for the remaining vertices.

Algorithm 3: $score(v)$ for sketch-based estimation

```

1 Let  $\hat{S}_{in}(v)$  be  $k / \sum_{i=1}^k I_{min}^i(v) - 1$ ;
2 Let  $\hat{S}_{out}(v)$  be  $k / \sum_{i=1}^k O_{min}^i(v) - 1$ ;
3 return  $(\hat{S}_{in}(v) + \hat{S}_{out}(v)) / (\hat{S}_{in}(v) * \hat{S}_{out}(v) + \hat{S}_{in}(v) + \hat{S}_{out}(v))$ ;
```

Algorithm 4: initialization()

```

1 for  $i$  from 1 to  $k$  do
2   generate a uniform random key  $X_v^i \in [0, 1]$  for all vertex  $v \in V$ ;
3 for  $v$  in  $V$  do
4   initialize  $I_{min}^i(v)$  and  $O_{min}^i(v)$  for  $i = 1, 2, \dots, k$ ;
5   compute the score function of each vertex  $v$  with  $score(v)$ ;
```

In addition, after a vertex v has been ordered and removed from G , not all O_{min} values need to be updated. In particular, when we remove v from G , for all the vertices that cannot reach v in G , their O_{min} values will not be affected. The only vertices that will be affected is the vertices in $S_{in}(v, G)$. Let $\mathcal{N}_{out}(u)$ be the out-neighbor of vertex u . Based on the definition of O_{min} , we have another observation:

$$O_{min}(u) = \min_{w \in \mathcal{N}_{out}(u)} \min\{X_w, O_{min}(w)\}.$$

Thus, we can update the O_{min} values safely based on the reversed topological order of the vertices with a backward BFS from v . In addition, during the backward BFS, if a vertex u' with $O_{min}(u') < O_{min}(v)$ is visited, we can omit all incoming edges of u' as the removal of vertex v will not affect O_{min} of u' and the vertices that can reach u' .

3.2.5.2 Implementation and Complexity Analysis

Algorithm 3 presents the score function we used for a given vertex v to estimate $\frac{|I_i(v)|}{|C_i(v)|}$. We use $\hat{S}_{out}(v)$ to denote the estimated $S_{out}(v, G)$ and $\hat{S}_{in}(v)$ to denote the estimated $S_{in}(v, G)$, the score for any vertex v is as shown in Line 3 of Algorithm 3.

Algorithm 4 presents the initialization phase of our heuristic algorithm. As shown in Lines 1-2 of Algorithm 4, it first generates the random keys for all vertices. After that it initializes $I_{min}^i(v)$ and $O_{min}^i(v)$ for all the vertices $v \in V$ and for each $i \in \{1, 2, \dots, k\}$ (Lines 4-6). Here I_{min} and O_{min} are the reachability sketches that are used to estimate $|S_{in}(v, G)|$ and $|S_{out}(v, G)|$, respectively.

After the initialization phase, the heuristic algorithm chooses an order for vertices based on the score function. Each time it selects a vertex v , it needs to perform an update procedure with two BFSs, as shown in Lines 5-6 of Algorithm 5. Afterwards, the

Algorithm 5: Sketch-based Ordering Algorithm

input : A DAG $G = (V, E)$, an integer k
output: A level order ℓ for V

```

1 initialization();
2 let  $G_1 = G$ ;
3 for  $i = 1, \dots, |V|$  do
4   find the vertex  $v$  with the lowest score function value;
5   set  $\ell(v) = i$ ;
6   do backward BFS from  $v$  and update  $O_{min}$  for the vertices that can reach  $v$  in
   reversed topological order;
7   do forward BFS from  $v$  and update  $I_{min}$  for the vertices that can be reached by
    $v$  in topological order;
8   remove  $v$  from  $G_i$ , denote resulting graph as  $G_{i+1}$ ;
9   recalculate the score function values based on  $G_{i+1}$ ;
10 return  $\ell$ ;

```

score function for all vertices are updated, and the algorithm proceeds to choose the next vertex with the smallest score, until the order of all vertices is decided.

Lazy Updates. To find the vertex with the minimum score function efficiently, we use a min heap to store the vertices and its score function. After the initialization, the top of the heap is the vertex with the minimum score function, which can be retrieved in constant time and removed in $O(\log n)$ time. However, when the score of a vertex is updated, we need $O(\log n)$ time to maintain the heap. To reduce the time required in heap maintenance, we adopt a lazy update strategy as follows. We use a hash table to record the latest score of each vertex $v \in V$, but when the score of v changes, we do not immediately update the score of v in the heap. Instead, whenever we pop a vertex w from the top of the heap, we check the hash table to see whether w 's score in the heap is identical to its newest score. If they are not identical, then we re-insert w back into the heap with its latest score; otherwise, we take w as the vertex with the current minimum score. It still remains to prove that by exploiting lazy update, we can still correctly produce the correct results. The following lemma establishes the correctness by exploiting lazy update.

Lemma 3.2. *The lazy update approach can always guarantee to provide the correct score values for every remaining vertex.*

Complexity Analysis. Our heuristic algorithm runs in $O(n^2 \log n + knm)$ time and $O(kn + m)$ space. In particular, in each step we need to pop at most n vertices from the heap, update their scores, and then re-insert them back to the heap. The heap operations

take $O(n \log n)$ time, and we need $O(k(n+m))$ time to initialize or update I_{min} , O_{min} and $score$. Summing up the costs of n steps leads to an $O(n^2 \log n + knm)$ time complexity. In addition, we need $O(n+m)$ space to store the graph G and $O(nk)$ space for I_{min} and O_{min} , and hence, the space complexity of the algorithm is $O(nk + m)$. However, as we will show in Sections 3.3.3 and 3.3.4, the practical performance of our heuristic is much better than the worst case bound, and is scalable for handling large graphs.

3.3 Experiments

This section experimentally evaluates our reachability methods, namely, the AC-Greedy algorithm in Section 3.2 (denoted as AC) and the heuristic algorithm in Section 3.2.5 (denoted as *Butterfly-Sketch* (BS)). We compare our methods with all existing *HHL* schemes that use different node orderings: *Butterfly-U* (BU) [101], *Butterfly-L* (BL) [101] *TF-labeling* (TF) [23], and *Distribution Labeling* (DL) [60]. We implement our methods in C++, and adopt the C++ implementation of all our competitors provided by their inventors. All of our experiments are conducted on a machine running CentOS 5.0, with 64GB main memory, Intel Xeon E5-2650 v2 CPU. If a method requires more than 24 hours (except AC-Greedy) or more than 64GB RAM to preprocess a dataset D , we omit the method from the experiments on D .

3.3.1 Experimental Setup

We use a large variety of datasets from the literature of reachability queries, as shown in Table 3.2. The table presents the statistics of the **coalesced DAG** for each dataset. Among them, *RG5*, *RG10*, *RG20*, and *RG40* are synthetic DAGs generated using the technique in [23] by varying the average degree of vertices from 5 to 40, and they are used in [101]. The other 11 datasets are the largest DAGs that have been adopted in the literature. In particular, *uni-22m*, *uni-100m*, *uni-150m*, *wiki*, *citeseerx*, *go-uniprot*, and *patents* are from [23, 60], while *GovWild*, *Yago2*, *Twitter*, and *Web-UK* are used in [85]. To test our heuristic solution compared to AC-Greedy, we adopt all the small datasets used in [61], i.e., *arxiv*, *Yago-sub*, *go-sub*, *pubmed*, and *citeseer*. Besides the aforementioned five datasets, we also use four small real datasets: *web-Stanford*, *p2p*, *web-BerkStan* that are public available in Stanford Large Network Dataset Collection.²

On each dataset G , we generate a set of 10^6 reachability queries. In particular, we first derive a topological order on the vertices in G . Then, for each query q , we randomly select two vertices from G , and we choose the vertex with lower (resp. higher) topological rank as the source (resp. terminal) vertex s (resp. t). This method of query generation ensures that none of the queries can be answered by trivially checking whether the terminal vertex has a lower topological rank than the source vertex.

²<http://snap.stanford.edu>

dataset	$ V $	$ E $	avg. deg.
arxiv	6.00K	66.71K	11.12
Yago-sub	6.64K	42.39K	6.38
go-sub	6.79K	13.36K	1.97
pubmed	9.00K	40.03K	4.48
citeseer	10.72K	44.26K	4.13
web-Sanford (web-ST)	29.92K	53.01	1.77
p2p	48.41K	55.34K	1.14
web-BerkStan (web-BS)	109.41K	583.77K	5.27
Pokec	325.90K	379.63K	1.15
RG5	1.0M	5.0M	5.00
RG10	1.00M	10.00M	10.00
RG20	1.00M	20.00M	20.00
RG40	1.00M	40.00M	40.00
uniprot22m (uni-22m)	1.60M	1.60M	1.00
uniprot100m (uni-100m)	16.09M	16.09M	1.00
uniprot150m (uni-150m)	25.04M	25.04M	1.00
wiki	2.28M	2.28M	1.01
Twitter	16.62M	18.36M	1.10
Yago2	16.13M	25.68M	1.59
Web-UK	20.44M	37.80M	1.85
citeseerx	6.26M	14.77M	2.36
GovWild	8.02M	23.65M	2.95
patent	3.68M	15.69M	4.27
go-uniprot	6.97M	34.77M	4.99

Table 3.2: Datasets ($K=10^3$, $M=10^6$).

3.3.2 Tuning k for BS

Recall that BS has a parameter k , i.e., the number of sketch values kept for each vertex in G . The impact of k is two-fold. On one hand, an increased k can contribute to a more accurate estimation, hence provides better level ordering and index size. On the other hand, an increased k can incur much higher preprocessing cost for maintaining and updating the k copies of sketch values. In this set of experiments, we study the effect of k on the performance of BS regarding these two aspects. Figure 3.3 illustrates the index size and preprocessing time of BS on various datasets. (We only include a representative subset of the datasets to avoid cluttering the figures.) Observe that, when

 Figure 3.3: Tuning k for BS on various datasets.

Figure 3.4: Index sizes on small datasets.

k increases, the index size of BS decreases. This is because a larger k leads to more accurate estimation with the reachability sketch, which in turn results in improved level orderings and index size. However, the preprocessing time of BS tends to increase with k , since a larger k requires more sketch values to be processed. Interestingly, on some datasets, when k increases from 1 to 4, the preprocessing cost of BS actually decreases. The reason is that an increased k leads to a better vertex ordering o , which may reduce the time required to construct an index given o , thus mitigating the increased overheads in generating o itself. Based on the results in Figure 3.3, we choose $k = 8$ for BS as it strikes a good balance between index size and preprocessing time.

Figure 3.5: Total query time on small datasets.

Figure 3.6: Preprocessing time on small datasets.

3.3.3 Evaluation of AC-Greedy and Sketch-based Heuristic

In this set of experiments, we evaluate the effectiveness of our AC-Greedy algorithm (AC) in terms of index size and query performance. Meanwhile, we evaluate our sketch-based heuristic (BS) against AC and evaluate its estimation accuracy. We compare AC and BS against all existing ordering methods, i.e., BU, BL, TF, and DL. We focus on 9 small dataset, since AC incurs prohibitive preprocessing overheads on other larger graphs.

Figures 3.4 and 3.5 show the index size and query performance of all methods, correspondingly. Observe that, the index size of AC is up to two orders of magnitude smaller than that of TF, and up to one order of magnitude smaller than that of BU and BL. Meanwhile, the index size of AC is up to 5 times smaller than that of DL, and marginal smaller than that of BS. In addition, the query performance shows a similar trend, and AC yields the least query time in all cases. This is consistent with the fact that AC is worst-case efficient, while the other methods are not. Figure 3.6 reports the preprocessing time of all methods. Note that AC is missing on *Pokec* dataset since it requires more than 64G memory. The preprocessing time of AC is prohibitively high due to its $O(n^3)$ time complexity and $O(n^2)$ space consumption, which renders it inapplicable on large graphs.

Meanwhile, the index size (resp. query time) of BS is highly comparable to that of AC, which indicates that the sketch-based estimation adopted in BS leads to a high-quality vertex ordering that is almost as good as the one from AC. Besides, the preprocessing time of BS is significantly less than that of AC, which renders the effectiveness of our sketch-based heuristic. The index sizes (resp. query time) of BU, BL, TF, and DL are never better than that of BS, and are much worse in some cases, which indicates that

Figure 3.7: Index sizes on large graphs.

Figure 3.8: Total query time on large graphs.

their ordering heuristic is inferior than ours. Finally, the preprocessing time of our BS is generally comparable to that of other heuristics. However, as the preprocessing happens off-line infrequently, the slightly increased overhead in preprocessing can be compensated by the improved query performance.

In summary, our AC provides the *HHL* index with smallest index and least query time among all methods. However, the prohibitive preprocessing cost makes it inapplicable on sizable graphs. Our sketch-based heuristic BS significantly reduces the preprocessing cost, while produces *HHL* indices highly comparable to that of BS, and still outperforms the existing solutions by a large margin, with a moderate preprocessing cost.

3.3.4 Evaluation on Large Datasets

To further demonstrate the effectiveness and scalability of our BS algorithm, we evaluate all methods except AC on 15 large real datasets. We first report the results on index size in Figure 3.7. Observe that BS outperforms the competitors in all cases. For example, on *RG10*, BS's index is around 60 times smaller than that of TF, and 6 times smaller than

Figure 3.9: Preprocessing time on large graphs.

Figure 3.10: Total query time on large graphs for the **random** query set. Figure 3.11: Total query time on large graphs for the **equal** query set.

that of DL, and 2 times smaller than that of BL. Besides, BS's index size is around 2 times smaller than that of BU on *Web-UK* and *citeseerx* datasets. This further confirms that the node ordering provided by our BS is superior to the other alternatives.

Meanwhile, Figure 3.8 shows the total query processing time for all approaches on

the large datasets. Again, BS outperforms all competitors on all datasets, which is consistent with the results in Figure 3.7. Finally, Figure 3.9 shows the preprocessing time of all approaches on various large datasets. Note that on *RG20* (resp. *RG40*), DL and TF (resp. DL, TF and BL) are missing due to their excessive memory consumption ($> 64GB$). Meanwhile, the preprocessing time of our BS is 10 times smaller than that of TF and DL on *RG10*, and 5 times smaller than that of BL on *RG20*. On other datasets, our BS is generally comparable with all other techniques in preprocessing cost. This reflects that our BS is scalable on sizable graphs, and can be applied on large networks in real world scenarios, e.g., social networks, the Web.

Figures 3.10 and 3.11 show the total query time of each method for the **random** and **equal** query sets, respectively. On **random** query set, we randomly select the source vertex and target vertex, while on **equal** queries, we generate half reachable queries, and half unreachable queries. Observe that, even though the query efficiency of each method varies with the query workload, the relative performance of the methods is similar to that shown in Figure 3.8. In particular, BS is still the most efficient technique in all cases, and it significantly outperforms BL, TF, and DL on *RG10*, and BU on *Web-UK* and *citeseerx*.

From the above experiments, we can see that our sketch-based heuristic solution provides the smallest index and least query time compared to existing states of the art. Meanwhile, extensive experiments shown that BS is scalable to sizable graphs. So our ordering method is clearly the choice for *HHL* reachability index.

3.4 Summary

This chapter presents a novel study on the vertex ordering in *Hierarchical Hub Labelling* (*HHL*). We first prove that deriving an optimal vertex ordering for *HHL* is NP-hard. Afterwards, we present an algorithm for vertex ordering that yields an *HHL* index whose size is at most $O(\sqrt{n})$ times the optimal one, where n is the number of vertices in G . To our knowledge, this is the first theoretical study for the approximation of *HHL* index and we show that our *AC-greedy* is an *HHL* method with non-trivial performance guarantee. Then, motivated by our theoretical study, we design a new heuristic for vertex ordering, and show that it leads to an *HHL* index with superior practical performance for reachability queries. Extensive experiments on real data demonstrate the efficiency of our solutions.

Chapter 4

Shortest Path Queries on Timetable Graphs

Identifying fast routes in transportation networks is an important problem with applications in map services and navigation systems. This problem has been extensively studied in the past decades, yielding a plethora of indexing techniques that aim to improve query efficiency at reasonable costs of pre-computation and space. Most of the existing techniques, however, consider *private transportation*, i.e., they assume that the traversal from one network location to another is carried out with a private vehicle without any service time constraint. In contrast, relatively less effort has been made for the (equally important) case of *public transportation*, where any traversal between different network locations relies on transportation services (e.g., buses, subways, ferries, etc.) that run on fixed routes with pre-defined schedules.

Route planning for public transportation drastically differs from that for private transportation, as the former needs to take into account additional spatiotemporal constraints that do not apply to the latter, such as the time required to wait for a bus at a station, or the feasibility of transferring from one bus to another given their respective schedules. Towards addressing the problem, a common approach is to model a public transportation network as a *timetable graph* [76], where (i) each node represents a station; and (ii) each directed edge $\langle u, v \rangle$ is associated with a timetable that records the departure (resp. arrival) time of each vehicle at station u (resp. v). Three types of shortest path queries for route planning on timetable graphs have been extensively studied, i.e., the *earliest arrival path (EAP)*, the *latest departure path (LDP)*, and the *shortest duration path (SDP)*. These three types of queries aim to find a path that arrives at a place as early as possible, departs as late as possible but arrives on time, and has the shortest travel time, respectively.

In this chapter, we present *Timetable Labelling (TTL)*, an efficient indexing technique to answer the three shortest path queries on timetable networks. The basic idea of *TTL*

 Figure 4.1: A timetable graph G .

is to associate each node u with a set of *labels*, each of which records the shortest travel time from u to some other nodes v given a certain departure time from u . The labels are constructed in such a way that, for any EAP, LDP, or SDP query from a node u to another node v , we can efficient retrieve a tiny summary of the query result by inspecting the label sets of u and v (without inspecting G), and we can easily convert the summary into the corresponding path in $O(k)$ time, where k is the number of nodes in the path. Compared with the existing techniques [27, 36, 46], *TTL* incurs lower query overheads as it completely avoids traversing G during query processing. In particular, in our experiments on 11 real transportation networks, *TTL* is shown to outperform the state of the art by up to 3 orders of magnitude in terms of query efficiency.

In the following, we first provides the problem definition in Section 4.1. Then we present our *TTL* framework in Section 4.2. Afterwards, we present the query processing algorithm and *TTL* implementation in Section 4.3 and Section 4.4, respectively. Next, we present our label compression techniques and the query algorithm under compressed scheme in Section 4.5. Furthermore, we present some extensions for *TTL* in Section 4.6. Finally, we conduct extensive set of experiment to evaluate our *TTL* in Section 4.7 and summarize this chapter in Section 4.8.

4.1 Problem Definition

Let G be a multigraph with n nodes, such that (i) each node represents a station in a public transportation network, and (ii) each edge e from a node u to another node v is associated with a vehicle b and two timestamps t_d and t_a , which indicate that there is a vehicle b that departs from u at t_d and arrives at v at t_a , without any stops in between. (Note that there may exist multiple edges from u to v .) For convenience, we represent e as a tuple $\langle u, v, t_d, t_a, b \rangle$. We refer to b , t_d , and t_a as the vehicle, departure time, and arrival time of e , respectively, and we define the *duration* of e as $t_a - t_d$. In addition, we say that e is an *outgoing edge* of u and an *incoming edge* of v . We define the *out-degree* (resp. *in-degree*) of v as the number of its outgoing (reps. incoming) edges, denoted by $d_{out}(v)$ (resp. $d_{in}(v)$). We define the paths in G as follows:

Definition 4.1 (Paths). A **path** P in G is a sequence $\langle e_1, e_2, \dots, e_k \rangle$ of edges in G , such that for any $i \in [1, k - 1]$, the starting node of e_{i+1} equals the ending node of e_i , and the departure time of e_{i+1} is after the arrival time of e_i . $\square$

We define the starting node (resp. departure time) of P as that of e_1 , the ending node (resp. arrival time) of P as that of e_k , and the duration of P as the difference between P 's starting and arrival time. In addition, if all edges on P have the same vehicle b (i.e., P does not contain a transfer), then we refer to b as the vehicle of P ; otherwise, we define the vehicle of P as *null*.

We consider three types of route planning queries on G , as defined in the following.

Definition 4.2 (Earliest Arrival Path Query). Given two nodes u and v in G and a starting timestamp t , an **earliest arrival path (EAP)** query asks for the path with the earliest arrival time among those that (i) start from u no sooner than t and (ii) end at v . $\square$

Definition 4.3 (Latest Departure Path Query). Given two nodes u and v in G and an ending timestamp t' , a **latest departure path (LDP)** query asks for the path with the latest departure time among those that (i) start from u and (ii) end at v no later than t' . $\square$

Definition 4.4 (Shortest Duration Path Query). Given two nodes u and v in G , a starting timestamp t , and an ending timestamp t' , a **shortest duration path (SDP)** query asks for the path with the shortest duration those that (i) start from u no sooner than t and (ii) end at v no later than t' . $\square$

For simplicity, we use the term *path queries* to refer to EAP, LDP, and SDP queries. We illustrate Definitions 4.2-4.4 with an example.

Example 4.1. Figure 4.1 shows a timetable graph G containing 6 nodes $v_1, v_2, \dots, v_6$, and 14 edges. The ordered pair $\langle t_d, t_a \rangle$ on each edge e indicates that the departure (resp. arrival) time of the edge equals t_d (resp. t_a). We use solid, dashed, and dotted lines to represent the edges correspond to vehicles b_1 , b_2 , and b_3 , respectively. Suppose that we are at v_5 at time $t = 5$, and plan to travel to v_1 . In that case, the earliest time that we can arrive at v_1 is $t' = 8$, for which we need to take vehicle b_3 from v_5 at time 6, and stay on the vehicle until it arrives at v_1 . In other words, if we have an EAP query from v_5 to v_1 with a starting timestamp $t = 5$, then the answer to the query is a path via vehicle b_3 that starts from v_5 at time 6 and ends at v_1 at time 8. On the other hand, for an LDP query from v_1 to v_4 with an ending timestamp $t' = 13$, then the answer is a path that (i) starts from v_1 at time 10 via vehicle b_1 , (ii) transfers to vehicle b_2 at v_2 , and then (iii) arrives at v_4 at time 13 on b_2 . Finally, for an SDP query from v_5 to v_1 with starting time $t = 6$ and ending time $t' = 10$, the query result is identical to the EAP query mentioned above. $\square$

Notation	Description
G	a timetable graph
n	the number of nodes in G
$\langle v, u, t_d, t_a, b \rangle$	an edge in E
$P = \langle e_1, \dots, e_k \rangle$	a path in G (see Section 4.1)
$o(v)$	the rank of v in the node order o (see Section 4.2)
$\mathcal{L}$	the TTL on G (see Definition 4.7)
$\langle v, t_d, t_a, b, p \rangle$	a label in $\mathcal{L}$ (see Definition 4.7)
$L_{in}(v)$	the in-label set of v (see Definition 4.7)
$L_{out}(v)$	the out-label set of v (see Definition 4.7)
$\cdot$	a placeholder that denotes irrelevant information in the context

Table 4.1: Table of notations.

Given G , our objective is to construct an index structure on G that can efficiently answer path queries. Table 4.1 lists the notations that we will frequently use in this chapter. In particular, we will use “ $\cdot$ ” as a placeholder to denote some information that is irrelevant to the discussions in the context.

4.2 Timetable Labelling

This section presents *timetable labelling* (TTL), our index structure for route planning on timetable graphs. At a high level, a TTL index pre-computes two sets of *labels*, $L_{in}(v)$ and $L_{out}(v)$, for each node v in G , such that each label in $L_{in}(v)$ (resp. $L_{out}(v)$) is a tuple concerning a “fast” path that ends at (starts from) v . The construction of the label sets is governed by a strict total order o on the nodes in G , which, intuitively, defines the relative importance of each node with respect to the others. We refer to o as the *node order*, and use $o(v)$ to denote the rank of a node v in o . Without loss of generality, we assume that $o(v)$ is an integer in $[1, n]$, and say that v ranks lower than another node u if $o(v) > o(u)$. Given a timetable graph G and a node order o , the TTL index can be uniquely constructed. Specifically, TTL is defined based on the concepts of *canonical paths*.

Definition 4.5 (Canonical Paths). *Given a node order o and two nodes u and v in G , a path P from u to v is a **canonical path**, if it satisfies the following constraints:*

- (i) **Dominance Constraint:** *There does not exist a path from u to v with (i) departure time later than P ’s and arrival time no later than that of P , or (ii) departure time no earlier than P ’s and arrival time earlier than that of P .*

(ii) **Rank Constraint:** Among all nodes on P , either u or v has the highest rank. $\square$

Intuitively, the dominance constraint in Definition 4.5 requires that a canonical path to be both the earliest arrival path and the latest departure path from u to v at a certain timestamp. Meanwhile, the rank constraint is crucial in proving the correctness of *TTL*'s query algorithm, which will be discussed in Section 4.3. We illustrate Definition 4.5 with an example.

Example 4.2. Consider the timetable graph in Figure 4.1. Assume that we use a node order $o(v_1) < o(v_2) < \dots < o(v_6)$. The path P from v_5 to v_1 taking vehicle b_3 at time $t = 6$ is a canonical path, since (i) no path from v_5 to v_1 after time $t = 6$ can arrive at v_1 earlier than P does or arrive v_1 before time $t = 8$ with departure time later than P 's, and (ii) the rank of v_1 is the highest among all nodes on P . In contrast, the path from v_6 to v_3 taking b_3 via v_1 at time $t = 7$ is not a canonical path, since the rank of v_1 is higher than those of v_3 and v_6 . $\square$

In relation to canonical paths, we also introduce the concepts of *intermediate nodes* and *pivots*.

Definition 4.6 (Pivots). An **intermediate node** of a path P refers to any node on P , except for P 's starting and ending nodes. The **pivot** of a canonical path is null if the path contains only two nodes; otherwise, it is the intermediate node of the path with the highest rank. $\square$

Given Definition 4.5 and 4.6, a *TTL* index on G is *uniquely* determined by a node order o as follows:

Definition 4.7 (Timetable Labelling). Given a node order o , let $\mathcal{C}$ be the set of all canonical paths on G . For each $P \in \mathcal{C}$, let u , v , t_d , t_a , b , and p be P 's starting node, ending node, departure time, arrival time, vehicle, and pivot, respectively. A **Timetable Labelling (TTL)** consists of label sets where P is mapped to a label $l = \langle x, t_d, t_a, b, p \rangle$ in either $L_{out}(u)$ or $L_{in}(v)$ as follows:

- (i) If $o(u) > o(v)$ (i.e., u ranks lower than v), then $x = v$ and $l \in L_{out}(u)$.
- (ii) If $o(u) < o(v)$, then $x = u$ and $l \in L_{in}(v)$. $\square$

In short, in a *TTL* index, the label sets $L_{in}(v)$ and $L_{out}(v)$ for a node v only contain the information about the canonical paths between v and the nodes that rank higher than v . For convenience, we refer to $L_{in}(v)$ (resp. $L_{out}(v)$) as the *in-label set* (resp. *out-label set*) of v , and each label in $L_{in}(v)$ (resp. $L_{out}(v)$) as an *in-label* (resp. *out-label*) of v . As an example, Table 4.2 shows the label sets of v_2 and v_4 in Figure 4.1, assuming a node order $o(v_1) < o(v_2) < \dots < o(v_6)$.

v	$L_{in}(v)$	$L_{out}(v)$
v_2	$\langle v_1, 9, 10, b_2, null \rangle$ $\langle v_1, 10, 11, b_1, null \rangle$	$\langle v_1, 8, 10, b_1, null \rangle$
v_4	$\langle v_1, 10, 13, null, v_2 \rangle$ $\langle v_2, 11, 13, b_2, v_6 \rangle$ $\langle v_3, 10, 13, b_3, null \rangle$	$\langle v_1, 5, 9, b_2, v_3 \rangle$ $\langle v_3, 5, 7, b_2, null \rangle$

Table 4.2: The label sets of v_2 and v_4 in a *TTL* index for Figure 4.1, with a node order $o(v_i) = i$ for $i \in [1, 6]$.

The label sets of a *TTL* index have an important property: for any path query from a node u to another node v , we can retrieve essential information about the query result from the label sets of u and v , as shown in the following lemma¹.

Lemma 4.1. *Let q be a path query from a node u to another v , and P be the result of q . Then, $L_{out}(u)$ contains a label $\langle w, t_d, t_a, b, p \rangle$, and $L_{in}(v)$ contains a label $\langle w', t'_d, t'_a, b', p' \rangle$, such that one of the following conditions holds:*

- (i) $w = v$, and t_d, t_a, b, p equal P 's departure time, arrival time, vehicle, and pivot, respectively. That is, $L_{out}(u)$ contains a label pertinent to P .
- (ii) $w' = u$, and t'_d, t'_a, b', p' equal P 's departure time, arrival time, vehicle, and pivot, respectively. That is, $L_{in}(v)$ contains a label pertinent to P .
- (iii) $w = w', t_a \leq t'_d$, and t_d (resp. t'_a) equals P 's departure (resp. arrival) time. That is, P contains a node w , such that $L_{out}(u)$ (resp. $L_{in}(v)$) contains a label pertinent to the sub-path of P from u to w (resp. from w to v). $\square$

We demonstrate Lemma 4.1 with the following example.

Example 4.3. Consider the label sets shown in Table 4.2 for the timetable graph in Figure 4.1. For an EAP query that from v_4 to v_2 with a starting timestamp $t = 4$, the result is a path P via vehicle b_2 that starts from v_4 at time 5, passes through v_3 and v_1 , and ends at v_2 at time 10. Observe that the out-label set $L_{out}(v_4)$ of v_4 contains a label $\langle v_1, 5, 9, b_2, v_3 \rangle$, which captures the sub-path of P from v_4 to v_1 . Meanwhile, the in-label set $L_{in}(v_2)$ of v_2 contains a label $\langle v_1, 9, 10, b_2, null \rangle$, which captures the sub-path of P from v_1 to v_2 . $\square$

Lemma 4.1 is the basis of the query algorithms for *TTL*, we clarify in Section 4.3. Meanwhile, the following lemma shows that the label sets of a *TTL* index is *minimal*, in the sense that if we remove any label from the label sets, we can no longer answer queries based on Lemma 4.2.

¹All the proofs in this chapter can be found in Appendix A.

Algorithm 6: *SketchGen*

input : $L_{out}(u), L_{in}(v), t, t'$
output: a set S_{uv} of path sketches

```

1 initialize  $i = 1, j = 1$ , and  $C = \emptyset$ ;
2 while  $i \leq |L_{out}(u)|$  or  $j \leq |L_{in}(v)|$  do
3   let  $l_u = \langle w, t_d, t_a, \cdot, \cdot \rangle$  be the  $i$ -th label of  $L_{out}(v)$ ;
4   if  $[t_d, t_a] \not\subseteq [t, t']$  then
5     | increase  $i$  by one, and then goto Line 4;
6   else if  $w = v$  then
7     | insert  $\langle l_u, null \rangle$  into  $S_{uv}$ ;
8     | increase  $i$  by one, and then goto Line 4;
9   let  $l_v = \langle w', t'_d, t'_a, \cdot, \cdot \rangle$  be the  $j$ -th tuple of  $L_{in}(u)$ ;
10  if  $[t'_d, t'_a] \not\subseteq [t, t']$  then
11    | increase  $j$  by one, and then goto Line 4;
12  else if  $w' = u$  then
13    | insert  $\langle null, l_v \rangle$  into  $S_{uv}$ ;
14    | increase  $j$  by one, and then goto Line 4;
15  if  $w = w'$  then
16    | if  $t_a \leq t'_d$  then
17      | add  $\langle l_u, l_v \rangle$  into  $S_{uv}$ ;
18      | increase  $i$  by one;
19    | else
20      | increase  $j$  by one;
21  else
22    | if  $o(w) < o(w')$  then
23      | increase  $i$  by one;
24    | else
25      | increase  $j$  by one;
26 return  $S_{uv}$ ;
    
```

Lemma 4.2. *For any node v , if we remove any label from $L_{in}(v)$ or $L_{out}(v)$, then Lemma 4.1 no longer holds.* $\square$

In subsequent sections, we first introduce *TTL*'s query processing procedure (in Section 4.3), followed by its preprocessing algorithm (in Section 4.4) and several optimizations in terms of index space and query time (in Section 4.5 and 4.6). This is for the ease of exposition, since the preprocessing method of *TTL* is considerably more sophisticated than its query algorithm.

Remark. *TTL* can be regarded as a non-trivial extension of *Hierarchical Hub Labelling*

(*HHL*) [6], which is the state-of-the-art indexing approach for answering shortest path queries on simple graphs. In particular, *HHL* also constructs label sets based on a total order of nodes, and it utilizes the label sets for query processing; nevertheless, the label sets of *HHL* do not contain any temporal information. Compared with *HHL*, *TTL*'s preprocessing and query algorithms are much more sophisticated, due to the complexity induced by the spatiotemporal constraints in timetable graphs. Furthermore, *TTL* features several novel optimization methods (for reducing space consumption and improving query efficiency) that are unique to timetable graphs, as we clarify in Sections 4.5 and 4.6.

4.3 Query Processing

This section presents the query algorithm for *TTL*. In a nutshell, *TTL*'s query algorithm consists of three phases:

- (i) *Candidate Generation*: This phase inspects the label sets of *TTL* to retrieve a set of *path sketches*, each of which corresponds to a *candidate* path that could be the answer for the given query.
- (ii) *Refinement*: This phase inspects the path sketches produced by the candidate generation phase, and then pinpoints the sketch s_{ans} corresponds to the query answer.
- (iii) *Path Reconstruction*: This phase converts the sketch s_{ans} back to a path on G , and returns the path as the query result.

In what follows, we elaborate each phase of *TTL*'s query algorithm, assuming that we are given an SDP query from a node u to another node v , with a starting timestamp t and an ending timestamp t' . We will clarify how the algorithm can be modified to handle EAP and LDP queries in Section 4.3.3.

4.3.1 Candidate Generation and Refinement

Consider the out-label set $L_{out}(u)$ of u and the in-label set $L_{in}(v)$ of v . By Definition 4.7, the labels in $L_{out}(u)$ and $L_{in}(v)$ capture the information about a set $\mathcal{P}_{uv}$ of paths from u to v as follows. First, if $L_{out}(u)$ contains a label $l_1 = \langle v, t_d, t_a, \cdot, \cdot \rangle$, then there exists a path P_1 from u to v with departure time t_d and arrival time t_a . In this case, we define the ordered pair $\langle l_1, null \rangle$ as a *sketch* of P_1 . Second, if $L_{in}(v)$ contains a label $l_2 = \langle u, t'_d, t'_a, \cdot, \cdot \rangle$, then there is a path P_2 from u to v with departure time t'_d and arrival time t'_a . Accordingly, we refer to $\langle null, l_2 \rangle$ as a sketch of P_2 . Third, if $L_{out}(u)$ and $L_{in}(v)$ contains labels $l_3 = \langle w, t_d^*, t_a^*, \cdot, \cdot \rangle$ and $l_4 = \langle w, t_d^\circ, t_a^\circ, \cdot, \cdot \rangle$, respectively, such that $t_a^* \leq t_d^\circ$, then there exists a path P_3 from u to v with departure time t_d^* and t_a° . In that case,

we say that $\langle l_3, l_4 \rangle$ is a sketch of P_3 . In summary, based on $L_{out}(u)$ and $L_{in}(v)$, we can construct the sketches for a set $\mathcal{P}_{uv}$ of paths from u to v , and each sketch records the departure and arrival time of the corresponding path.

By Lemma 4.1, $\mathcal{P}_{uv}$ must contain the answer P_{ans} to an SDP query from u to v with a starting timestamp t and an ending timestamp t' . If we are given the sketches of all paths in S_{uv} , then we can easily identify one corresponding to P_{ans} , by searching for the sketch with the shortest duration among those with departure time no sooner than t and arrival time no later than t' . However, constructing sketches for all paths in $\mathcal{P}_{uv}$ is costly, since $|\mathcal{P}_{uv}| = |L_{out}(u)| \cdot |L_{in}(v)|$ in the worst case. (This worst case occurs when each label in $L_{out}(u)$ can be combined with any label in $L_{in}(v)$ to form a path sketch.) Nevertheless, many of the sketches could have been omitted without affecting the correctness of the query result. For example, consider the label sets $L_{out}(v_4)$ and $L_{in}(v_2)$. Observe that $L_{out}(v_4)$ contains a label $l_1 = \langle v_1, 5, 9, b_2, v_3 \rangle$, which can form two path sketches when combined with the labels $l_2 = \langle v_1, 9, 10, b_2, null \rangle$ and $l_3 = \langle v_1, 10, 11, b_1, null \rangle$ in $L_{in}(v_2)$. However, the sketch $\langle l_1, l_3 \rangle$ is *dominated* by the sketch $\langle l_1, l_2 \rangle$, in the sense that $\langle l_1, l_2 \rangle$ corresponds to a path that departs from u at time 5 and arrives at v at time 10, which is strictly better than the path corresponding to $\langle l_1, l_3 \rangle$, as the latter has departure time 5 and arrival time 11. In other words, the path pertinent to $\langle l_1, l_3 \rangle$ can never be the result for any path query, and hence, we can omit $\langle l_1, l_3 \rangle$ during query processing.

In general, when *TTL* inspects label sets in its candidate generation phase, it only constructs path sketches that are not dominated by others, so as to improve query efficiency. To facilitate this, the preprocessing algorithm of *TTL* sorts the labels l in each label set in ascending order of $f(l)$, where $f(l)$ is a total order of labels as follows: for any two labels $l_1 = \langle v, t_d, t_a, \cdot, \cdot \rangle$ and $l_2 = \langle v', t'_d, t'_a, \cdot, \cdot \rangle$ in the same label set, we have $f(l_1) < f(l_2)$ if and only if

- (i) $o(v) < o(v')$ (i.e., v ranks higher than v'), or
- (ii) $o(v) = o(v')$ and $t_d < t'_d$, or
- (iii) $o(v) = o(v')$, $t_d = t'_d$, and $t_a < t'_a$.

With the above ordering of labels, *TTL* generates path sketches in $O(|L_{out}(u)| + |L_{in}(v)|)$ time using the *SketchGen* method in Algorithm 6. The input to the algorithm includes $L_{out}(u)$ and $L_{in}(v)$, as well as the SDP query's starting timestamp t and ending timestamp t' ; its output is a set S_{uv} of path sketches corresponding to the candidate answers for the query. The basic idea of the algorithm is to generate path sketches with two concurrent linear scans on $L_{out}(u)$ and $L_{in}(v)$, respectively.

Specifically, *SketchGen* first scans the labels in $L_{out}(u)$, until it finds a label $l_u = \langle w, t_d, t_a, \cdot, \cdot \rangle$ with $t_d \geq t$ and $t_a \leq t'$ (Lines 1-5). If $w = v$, then *SketchGen* inserts l_u into S_{uv} as a path sketch, since l_u captures a candidate path from u to v with departure time

t_d and arrival time t_a ; after that, *SketchGen* moves on to the next label in $L_{out}(u)$ (Lines 6-8).

On the other hand, if $w \neq v$, then *SketchGen* turns its attention to $L_{in}(v)$, and scans the labels in $L_{in}(v)$ until it counters a label $l_v = \langle w', t'_d, t'_a, \cdot, \cdot \rangle$ with $t'_d \geq t$ and $t'_a \leq t'$ (Lines 9-11). If $w' = u$, then *SketchGen* adds l_v into S_{uv} as a path sketch, and moves on to the next label in $L_{in}(v)$ (Lines 12-14).

Meanwhile, if $w' \neq u$, then *SketchGen* considers whether l_u and l_v can be combined to form a path sketch $\langle l_u, l_v \rangle$. In particular, if $w = w'$ and $t_a \leq t'_d$, then *SketchGen* inserts $\langle l_u, l_v \rangle$ into S_{uv} , and move on to the next label in $L_{out}(u)$ (Lines 15-18). However, if $w \neq w'$ or $t_a > t'_d$, then *SketchGen* would move on to the next label in either $L_{out}(u)$ or $L_{in}(v)$ to continue the construction of path sketches. Here, *SketchGen* differentiates two cases.

Case 1: $w = w'$ but $t_a > t'_d$. In this case, *SketchGen* proceeds to the next label in $L_{in}(v)$ (Lines 19-20). The rationale is that, due to the way that we sort the labels in $L_{in}(v)$, the next label in $L_{in}(v)$ could be $l_v^* = \langle w, t_d^*, \cdot, \cdot, \cdot \rangle$ with $t_a \leq t_d^*$, in which case l_u and l_v^* may form a path sketch.

Case 2: $w \neq w'$. In this case, *SketchGen* checks whether $o(w) < o(w')$. If $o(w) < o(w')$ holds, then *SketchGen* moves on to the next label in $L_{out}(u)$ (Lines 21-23), since the label could be $l_u^* = \langle w', \cdot, t_a^*, \cdot, \cdot \rangle$, in which case l_u^* and l_v may form a path sketch. On the other hand, if $o(w) > o(w')$, then the next label in $L_{out}(u)$ cannot be combined with l_v as a path sketch, and hence, *SketchGen* would proceed to the next label in $L_{in}(v)$ instead.

In general, our discussions above are applicable for any pair of labels $l_u \in L_{out}(u)$ and $l_v \in L_{in}(v)$ encountered by *SketchGen* during its linear scans of $L_{out}(u)$ and $L_{in}(v)$. In particular, any labels with starting time before t or ending time after t' are omitted (Lines 4-5 and 10-11), while any remaining $l_u \in L_{out}(u)$ that concerns v and any remaining $l_v \in L_{in}(u)$ that concerns u are inserted into S_{uv} as path sketches (Lines 6-8 and 12-14). For the other labels, *SketchGen* carefully decides (i) whether a pair of labels can form a path sketch, and (ii) whether it should proceed to the next label in $L_{out}(v)$ or $L_{in}(v)$ (Lines 15-25).

It can be verified that *SketchGen* runs in $O(|L_{out}(u)| + |L_{out}(v)|)$ time and generates at most $O(|L_{out}(u)| + |L_{out}(v)|)$ path sketches in S_{uv} , since it linearly scans $L_{out}(u)$ and $L_{out}(v)$. In addition, S_{uv} always contains a sketch that corresponds to the answer of the given SDP query, as shown in Lemma 4.3. The rationale is that, due to the order in which *TTL* sorts the labels in each label set, the linear scans employed by *SketchGen* only omit the label pairs that either (i) cannot form path sketches or (ii) correspond to paths that are dominated by others.

Lemma 4.3. *Let S_{uv} be the set of sketches that SketchGen returns for an SDP query, and P_{ans} be the answer of the query. Then, one of the sketches in S_{uv} corresponds to P_{ans} .* $\square$

Once S_{uv} is obtained, it is straightforward to identify the sketch in S_{uv} that is pertinent to the SDP query result P_{ans} . In particular, for each sketch in S_{uv} , we can calculate the duration of the corresponding path based on the departure and arrival time recorded in the sketch; based on that, we can pinpoint the sketch with the shortest duration. In Section 4.3.2, we will explain how we can convert this sketch back into P_{ans} .

Let s_{ans} be the sketch in S_{uv} that corresponds to the result P_{ans} of the given SDP query. Then, $s_{ans} = \langle l_u, l_v \rangle$, where l_u (resp. l_v) is either *null* or a label in $L_{out}(u)$ (resp. $L_{in}(v)$). To transform s_{ans} back to P_{ans} , it suffices to convert $l_u \neq \text{null}$ (resp. $l_v \neq \text{null}$) to the canonical path P_u (resp. P_v) in G that correspond to l_u (resp. l_v). Such conversions can be performed by exploiting the pivot of P_u (resp. P_v), which is recorded in l_u (resp. l_v). To explain, we first present a lemma about pivots.

Lemma 4.4. *Let P be a canonical path from u to v , and p be the pivot of P . Then, the sub-path of P from u to p (resp. from p to v) is also a canonical path.* $\square$

Suppose that the pivot of P_u is a node p , and $l_u = \langle w, \cdot, \cdot, \cdot, p \rangle$. Let P_1 (resp. P_2) be the sub-path of P from u to p (resp. from p to w). By Lemma 4.4, both P_1 and P_2 are canonical paths. Then, by Definition 4.7, there should exist two labels l_1 and l_2 (in TTL) that correspond to P_1 and P_2 , respectively. In other words, we can “unfold” l_u into a sequence of labels $S = \langle l_1, l_2 \rangle$, such that the canonical path pertinent to l_u equals a concatenation of the canonical paths corresponding to l_1 and l_2 . We refer to l_1 (resp. l_2) as the left (resp. right) child of l_u .

4.3.2 Path Reconstruction

In general, the above unfolding method can be applied on any label $\langle \cdot, \cdot, \cdot, \cdot, p \rangle$ where $p \neq \text{null}$ (i.e., the label corresponds to a canonical path that contains at least two edges). Therefore, given the label sequence $S = \langle l_1, l_2 \rangle$ mentioned above, we can recursively modify S by replacing each label with its children, until every label in S has a *null* pivot. In that case, each label in S represents an edge in G . Then, by replacing each label in S with its corresponding edge, we can transform S into the canonical path P_u pertinent to l_u . Using the same method, we can unfold l_v into its corresponding canonical path P_v , after which we can concatenate P_u and P_v to obtain the answer P_{ans} of the given SDP query.

Algorithm 7 shows the pseudo-code of the above unfolding approach, which takes as input a label l and returns the canonical path P corresponding to l . The most crucial step of the algorithm is the identification of l ’s children. To ensure the efficiency of

Algorithm 7: *PathUnfold*

input : a label $l = \langle \cdot, \cdot, \cdot, \cdot, p \rangle$ in *TTL*
output: the canonical path corresponding to l

```

1 if  $p = \text{null}$  then
2   return a path that only contains the edge in  $G$  corresponding to  $l$ ;
3 else
4    $l_1 =$  the left child of  $l$ ;
5    $l_2 =$  the right child of  $l$ ;
6    $P_1 = \text{PathUnfold}(l_1)$ ;
7    $P_2 = \text{PathUnfold}(l_2)$ ;
8   set  $P =$  a concatenation of  $P_1$  and  $P_2$ , with  $P_1$  proceeding  $P_2$ ;
9   return  $P$ ;

```

this step, *TTL* pre-computes two pointers from each label to its left and right children, respectively, so that it takes $O(1)$ time to pinpoint the children of any label. As such, the running time of Algorithm 7 is linear to the number of nodes on P .

Example 4.4. Consider an SDP query from v_2 to v_4 in Figure 4.1, with starting time $t = 8$ and ending time $t' = 13$. By a linear scan of the labels in $L_{out}(v_2)$ and $L_{in}(v_4)$ (in Table 4.2), we obtain two path sketches: (i) $\langle l_1, l_2 \rangle$, with $l_1 = \langle v_1, 8, 10, b_1, \text{null} \rangle$ in $L_{out}(v_2)$ and the label $l_2 = \langle v_1, 10, 13, \text{null}, v_2 \rangle$ in $L_{in}(v_4)$, (ii) $\langle \text{null}, l_3 \rangle$, with $l_3 = \langle v_2, 11, 13, b_2, v_6 \rangle$ in $L_{in}(v_4)$. Then, we set $s_{ans} = \langle \text{null}, l_3 \rangle$, since $\langle \text{null}, l_3 \rangle$ has a shorter duration than $\langle l_1, l_2 \rangle$. After that, we unfold l_3 by recursively reconstructing the SDPs from v_2 to v_6 and v_6 to v_4 . In the end, we obtain the path $P_{ans} = \langle \langle v_2, v_6, 11, 12, b_2 \rangle, \langle v_6, v_4, 12, 13, b_2 \rangle \rangle$. $\square$

The following corollary shows the correctness of Algorithm 7.

Corollary 4.1. *Given a label l in *TTL*, Algorithm 7 returns the canonical path corresponding to l .* $\square$

4.3.3 Summary and Query Extensions

Summing up the discussions in Sections 4.3.1 and 4.3.2, *TTL* answers any SDP query from u to v in $O(|L_{out}(u)| + |L_{in}(v)| + k)$ time, where k is the number of nodes on the query result. Furthermore, *TTL*'s query algorithm for SDP queries can be easily extended to support EAP and LDP queries. In particular, the extension is based on the following lemma about *SketchGen*.

Algorithm 8: *IndexBuild*

input : G and a node order o
output: a *TTL* index $\mathcal{L}$

```

1  let  $G_1 = G$ ;
2  for  $i = 1, 2, \dots, n$  do
3      let  $u_i$  be the node with  $o(u_i) = i$ ;
4      let  $T_d$  be a set containing the departure timestamps of all outgoing edges of  $u$  in  $G_i$ ;
5      for each  $t_d \in T_d$  in descending order do
6          create a hash table  $A$  that maps each node  $v$  to a tuple  $\langle t_v, b_v, p_v \rangle$ , where  $t_v$  is
              the earliest arrival time at  $v$  from  $u$  (initialized as  $+\infty$ ),  $b_v$  is a vehicle
              (initialized as a placeholder  $\star$ ), and  $p_v$  is a pivot node (initialized as null);
7          create a min-heap  $H$  that (i) accepts entries of the form  $\langle e, p \rangle$ , where  $e$  is an
              edge, and  $p$  is a node, and (ii) sorts entries in ascending order of the arrival time
              of their edges;
8          for each outgoing edge  $e'$  of  $u$  with departure time  $t_d$  do
9              insert an entry  $\langle e', \text{null} \rangle$  into  $H$ ;
10         while  $H$  is not empty do
11             remove the top entry  $\langle e, p \rangle$  from  $H$ ;
12             let  $\langle x, y, t_d, t_a, b \rangle$  denote  $e$ ;
13             retrieve the tuple  $\langle t_x, b_x, p_x \rangle$  associated with  $x$  in  $A$ ;
14             retrieve the tuple  $\langle t_y, b_y, p_y \rangle$  associated with  $y$  in  $A$ ;
15             if  $t_x = +\infty$  or  $t_d \geq t_x$  and  $t_a < t_y$  then
16                 set  $t_y = t_a$  and  $p_y = p$ ;
17                 if  $b_y = \star$  then
18                     set  $b_y = b$ ;
19                 else if  $b_y \neq b$  then
20                     set  $b_y = \text{null}$ ;
21                 for each outgoing edge  $e^*$  of  $y$  do
22                     let  $\langle \cdot, \cdot, \cdot, \cdot, b^* \rangle$  denote  $e^*$ ;
23                     if  $b^* \neq b$  then
24                         set  $b^* = \text{null}$ ;
25                     if  $p = \text{null}$  or  $o(p) < o(y)$  then
26                         insert  $\langle e^*, p \rangle$  into  $H$ ;
27                     else
28                         insert  $\langle e^*, y \rangle$  into  $H$ ;
29         for each node  $v$  in  $G_i$  do
30             let  $\langle t_v, b_v, p_v \rangle$  be the tuple in  $A$  corresponding to  $v$ ;
31             if there is no  $\langle u, \cdot, t_a, \cdot, \cdot \rangle \in L_{in}(v)$  with  $t_a \leq t_v$  then
32                 if there does not exist  $\langle u_j, t'_d, t'_a, \cdot, \cdot \rangle \in L_{out}(u_i)$  and
                     $\langle u_j, t_d^*, t_a^*, \cdot, \cdot \rangle \in L_{in}(v)$ , such that  $t'_a < t_d^*$  and  $[t'_d, t_a^*] \subset [t_d, t_v]$  then
33                     insert into  $L_{in}(v)$  a label  $\langle u_i, t_d, t_v, b_v, p_v \rangle$ ;
34     repeat Lines 4-33 to construct labels in the out-label set of each node, using the
        reverse version of temporal Dijkstra's algorithm;
35     remove  $v$  from  $G_i$ , and let  $G_{i+1}$  denote the resulting graph;
36 return the collection of all label sets as a TTL index  $\mathcal{L}$ ;

```

Lemma 4.5. *Let $L_{out}(u)$, $L_{in}(v)$, t , and t' be the input to *SketchGen* (as in Algorithm 6), and S_{uv} be its output. When $t' = +\infty$, one of the sketches in S_{uv} corresponds to the answer of an EAP query from u to v with starting time t . On the other hand, if $t = -\infty$, then S_{uv} contains a sketch corresponding to the answer of an LDP query from u to v with ending time t' . $\square$*

By Lemma 4.5, if we are to answer an EAP query from u to v with starting time t , we can first invoke *SketchGen*, setting $t' = +\infty$. Then, we examine the path sketches returned by *SketchGen* to identify the one s_{ans} with the earliest arrival time. After that, we unfold s_{ans} into a path using Algorithm 7, and return the path as the query result.

On the other hand, if we are given an LDP query from u to v with ending time t , we can apply *SketchGen* with $t = -\infty$, and obtain a set of sketches S_{uv} . After that, we inspect S_{uv} to pinpoint the sketch s_{ans} with the latest departure time. Finally, we use Algorithm 7 to convert s_{ans} into the query answer. It can be verified that the above query algorithms for EAP and LDP queries have the same time complexity as the one for SDP queries.

4.4 TTL Implementation

This section first presents the algorithms for constructing *TTL* indices in Section 4.4.1, assuming that the node order o is given. Afterwards, we discuss how to derive a good ordering in *TTL* in Section 4.4.2. By Definition 4.7, each label in *TTL* corresponds to a canonical path in G , and vice versa. Therefore, if we are to build a *TTL* index $\mathcal{L}$ based on a node order o , then conceptually, we should identify all canonical paths in G , and convert each of them into a label in $\mathcal{L}$. But how can we derive all canonical paths in G ?

A straightforward approach is to run the temporal Dijkstra's algorithm to retrieve all temporal shortest paths in G , and then examine the paths obtained to identify the canonical paths. This approach, however, incurs prohibitive preprocessing costs due to the large number of temporal shortest paths that it retrieves. To address this issue, we will first present an algorithm for computing paths in G that satisfy the Dominance Constraint (referred to as the *non-dominated* paths) in Section 4.4.1.1. Then, we explain how the algorithm can be used to avoid computing the paths that violate Rank Constraint, which allows us to construct a *TTL* index $\mathcal{L}$ in a more efficient manner. Note that, even though our index construction algorithm has the same worst-case complexity as the straightforward approach, the former's practical efficiency is significantly higher than that of the latter, as we show in Section 4.7.

4.4.1 Index Construction

4.4.1.1 Computing Non-Dominated Paths

Let u be a node in G , and T_d be a set that contains the departure timestamps of all outgoing edges of u . Suppose that we are to compute all non-dominated paths that start from a node u in G . We first make an interesting observation as follows.

Observation 4.1. *Let t_{-1} be the largest timestamp in T_d . Then, any earliest arrival path (EAP) P from u to a node v with departure time t_{-1} is a non-dominated path.* $\square$

Based on Observation 4.1, we can compute all non-dominated paths that start from u at time t_{-1} , by deriving all EAPs from u at time t_{-1} . This derivation can be performed with a modified version of Dijkstra's algorithm [27], referred to as *temporal Dijkstra's algorithm*. First, we create a hash table A that maps each node v to the earliest arrival time (EAT) t_v from u to v . We initialize $t_v = +\infty$ for any $v \neq u$, and $t_v = 0$ otherwise. Then, we collect all outgoing edges of u with departure time t_{-1} , and insert them into a min-heap H that sorts edges in ascending order of their arrival timestamps. After that, we iteratively remove the top edge $e = \langle x, y, t_d, t_a, \cdot \rangle$ in H , and process it as follows. We first retrieve the current EAT of x and y (denoted as t_x and t_y , respectively) from the hash table A . If $t_d \geq t_x$ and $t_a < t_y$, we set $t_y = t_a$, and insert into H all outgoing edges of y with departure time no sooner than t_a . Finally, we proceed to pop the next edge in H . When H becomes empty, we terminate the algorithm. Then, by Observation 4.1, each final EAT t_v recorded in H corresponds to a non-dominated path (from u to v with starting time t_{-1}).

Now suppose that we apply the temporal Dijkstra's algorithm to compute all EAPs that start from u at time t_{-1}^* , where t_{-1}^* denotes the second largest timestamp in T_d . We have another observation as follows.

Observation 4.2. *Then, any EAP P from u with departure time t_{-1}^* is a non-dominated path, if it is not dominated by any EAP from u with departure time t_{-1} .* $\square$

Given any EAP from u to v with departure time t_{-1}^* , we can decide whether it is a non-dominated path, by comparing its arrival time with that of the EAP from u to v with departure time t_{-1} . In general, an EAP from u with departure time t is a non-dominated path, if it is not dominated by any EAP from u with departure time $t^* > t$. Therefore, if we are to compute all non-dominated paths from u , we can examine the timestamps in T_d in descending order, and apply the modified Dijkstra's algorithm once for each timestamp $t \in T_d$, to identify the non-dominated EAPs from u with departure t . The following lemma shows that our approach does not miss any non-dominated path that starts from u .

Lemma 4.6. *For any non-dominated path P starting from u , there exists a timestamp $t \in T_d$, such that P is an EAP starting from u at time t .* $\square$

Finally, let us consider a “backward” version of the temporal Dijkstra’s algorithm, such that (i) the min-heap H sorts edges based on arrival time, (ii) the hash table A records the latest departure time of each node, and (iii) we start by inserting into H all incoming edges of u with a given arrival time t' . It can be verified that such an algorithm computes the latest departure paths (LDP) that arrive at u at time t' . In addition, we can observe that an LDP to v with arrival time t' is non-dominated, if it is not dominated by any LDP to v with arrival time $t < t'$. Furthermore, we have the following lemma.

Lemma 4.7. *Let T_a be a set that contains the arrival timestamps of all incoming edges of u . For any non-dominated path P ending at u , there exists a timestamp $t' \in T_a$, such that P is an LDP ending at u at time t' .* $\square$

Therefore, we can compute all non-dominated paths ending at u , by enumerating the timestamps in T_a in ascending order, and by applying the reverse version of our Dijkstra’s algorithm once for each timestamp. In Section 4.4.1.2, we will utilize our modified version of the Dijkstra’s algorithm to identify canonical paths and construct label sets.

4.4.1.2 Construction of Label Sets

Algorithm 8 shows the preprocessing method of *TTL*, referred to as *IndexBuild*. Given G and a node order o , *IndexBuild* first identifies the node u_1 with the highest rank, as well as the latest departure time t_d among the outgoing edges of u_1 (Lines 1-4). After that, it invokes the temporal Dijkstra’s algorithm to compute the EAPs from u starting at t_d (Lines 5-28), with some simple bookkeeping to keep track of the vehicle b of each EAP P (Lines 17-20 and 23-24), as well as the intermediate node p on P with the highest rank (Lines 25-28). For each EAP P ending at a node v at time t_a , *IndexBuild* inserts a label $\langle u_1, t_d, t_a, b, p \rangle$ into $L_{in}(v)$ (Lines 30-34). The reason is as follows. First, based on our discussions in Section 4.4.1.1, P satisfies the Dominance Constraint in Definition 4.5. Second, given that u has a higher rank than all other nodes, P also satisfies the Rank Constraint in Definition 4.5. Therefore, P is a canonical path, and hence, we add into $L_{in}(v)$ a label corresponding to P (see Definition 4.7).

After that, *IndexBuild* examines the other departure time of u_1 ’s outgoing edges in descending order (Line 5). For each departure time t_d , *IndexBuild* derives the EAPs starting from u_1 at time t_d , using the temporal Dijkstra’s algorithm (6-8). For each of the EAP P derived, if P ends at a node v at time t_v , then *IndexBuild* examines whether the current $L_{in}(v)$ contains a label $l = \langle u_1, \cdot, t_a, \cdot, \cdot \rangle$ with $t_a \leq t_v$ (Line 31). If such a label exists, then P is dominated by an EAP that *IndexBuild* previously derived, in which case P will be omitted; otherwise, P must be a canonical path, and hence, *IndexBuild* inserts into $L_{in}(v)$ a label corresponding to P .

Once all departure timestamps from u_1 are enumerated, *IndexBuild* employs the reverse version of the temporal Dijkstra's algorithm to derive all canonical paths ending at u_1 (Line 34). In particular, *IndexBuild* examines the arrival timestamps at v in ascending order and, for each timestamp t_a , uses the backward Dijkstra's algorithm to compute the LDPs that end at u_1 at time t_a . Based on the LDPs computed, *IndexBuild* identifies all canonical paths ending at u_1 , and constructs labels in other nodes' out-label sets, in a manner similar to the processing of canonical paths starting from u . After that, *IndexBuild* removes u_1 from G (Line 35), and proceeds to process the remaining nodes.

Let u_i denote the node with $o(u_i) = i$ ($i \in [2, n]$). The subsequent execution of *IndexBuild* inspects u_i in ascending order of i . For each u_i , *IndexBuild* examines a modified version of G (denoted as G_i) where all nodes with higher rank than u_i are removed. In particular, *IndexBuild* computes the all non-dominated paths in G_i that start from u_i , using the same approach as in the case of u_1 (Lines 5-28). The set $\mathcal{S}$ of non-dominated paths thus derived would satisfy the Rank Constraint in Definition 4.5, since they do not contain any node that ranks higher than u_i . However, as they are derived on G_i instead of G , they may not all be non-dominated paths in G . To identify the paths in $\mathcal{S}$ that are non-dominated in G , *IndexBuild* processes each path $P \in \mathcal{S}$ as follows. Assume that P starts from u_i at time t_d and ends at a node v at time t_a . If $L_{out}(u_i)$ contains a label $l_1 = \langle u_j, t'_d, t'_a, \cdot, \cdot \rangle$ and $L_{in}(v)$ has a label $l_2 = \langle u_j, t_d^*, t_a^*, \cdot, \cdot \rangle$, such that $t'_a < t_d^*$ and $[t'_d, t'_a] \subset [t_d, t_a]$, then P is dominated by a path in G that goes from u_i to v via u_j . (Note that $j < i$ always holds, due to the order in which *IndexBuild* processes nodes.) In that case, *IndexBuild* omits P (Line 32). On the other hand, if such labels l_1 and l_2 do not exist, then P is not dominated by any path in G that passes through a node u_k with $k < i$. In that case, P must be a non-dominated path in G . Then, given that P satisfies the Rank Constraint in Definition 4.5, it is a canonical path on G ; as such, *IndexBuild* would insert into $L_{in}(v)$ a label pertinent to P (Line 33). In a similar fashion, *IndexBuild* would also apply the backward Dijkstra's algorithm to compute all canonical paths that ends at u_i , and create labels in the out-label sets accordingly. We illustrate the execution of *IndexBuild* with the following example.

Example 4.5. Consider the timetable graph G' in Figure 4.2a with a node order $o(v_2) = 1$, $o(v_1) = 2$, and $o(v_3) = 3$. The index construction algorithm first selects v_2 , and performs a temporal Dijkstra traversal from v_2 with the largest departure timestamp, i.e., 4. Then it obtains one EAP, based on which *IndexBuild* inserts a label $\langle v_2, 4, 5, b_3, null \rangle$ into $L_{in}(v_3)$. Next, it performs a temporal Dijkstra traversal from v_2 at timestamp 3 and timestamp 2 in sequence. These two traversals lead to two additional labels. After that, a backward Dijkstra is performed from v_2 with the smallest arrival timestamp, i.e., 2. Then it obtains one LDP, and *IndexBuild* inserts a label $\langle v_2, 1, 2, b_1, null \rangle$ into $L_{out}(v_1)$ corresponding to this LDP. Afterwards, two more backward Dijkstra traversals

v	$L_{in}(v)$	$L_{out}(v)$
v_2	$\emptyset$	$\emptyset$
v_1	$\emptyset$	$\langle v_2, 1, 2, b_1, null \rangle$ $\langle v_2, 2, 3, b_2, null \rangle$ $\langle v_2, 3, 4, b_3, null \rangle$
v_3	$\langle v_2, 2, 3, b_1, null \rangle$ $\langle v_2, 3, 4, b_2, null \rangle$ $\langle v_2, 4, 5, b_3, null \rangle$	$\emptyset$

Table 4.3: The label sets of v_1 , v_2 , and v_3 in a *TTL* index of G' in Figure 4.2a, with a node order $o(v_2) = 1, o(v_1) = 2$, and $o(v_3) = 3$.

Figure 4.2: An example of route-based compression.

are performed from v_2 at timestamp 3 and 4 in a row, resulting in two more labels. This finishes the traversal from v_2 , and v_2 is removed from the graph. Then v_1 and v_3 have no incoming and outgoing edges, and both incur no label cost during the traversal. This finishes the index construction, ending up with 6 labels as shown in Table 4.3. $\square$

Once all u_i are processed, *IndexBuild* returns the collection of all label sets constructed as a *TTL* index $\mathcal{L}$. The following Lemma 4.8 shows the correctness and time complexity of *IndexBuild*.

Lemma 4.8. *Given G and a node order o , *IndexBuild* returns a *TTL* index in $O(n \cdot m + n^2 \cdot \log n)$ time, where m is the number of edges in G .* $\square$

4.4.2 Node Ordering

As shown in Sections 4.1 and 4.4.1, a *TTL* index $\mathcal{L}$ on G is uniquely decided by the given node ordering o . As such, the choice of o has a profound effect on the *size* of $\mathcal{L}$, defined

as

$$|\mathcal{L}| = \sum_{v \in V} (|L_{in}(v)| + |L_{out}(v)|). \quad (4.1)$$

Node ordering has been extensively studied on *HHL* for reachability and distance queries. Following our study in Chapter 3, we extend our AC-Greedy algorithm to *TTL* to derive the node ordering. In Section 4.4.2.1, we present how to extend our AC-Greedy algorithm under timetable graphs. Afterwards, we propose an heuristic solution based on the approximation algorithm in Section 4.4.2.2.

4.4.2.1 Approximation Algorithm

Let $\mathcal{N}$ be the set of all non-dominated paths on G . We first introduce a few notations. We say that a node v *covers* a non-dominated path $P \in \mathcal{N}$, if v is on P . We use $\mathcal{N}(v)$ to denote the set of non-dominated paths in $\mathcal{N}$ that are covered by v , and we refer to $|\mathcal{N}(v)|$ as the *coverage* of v . Let u_i ($i \in [1, n]$) denote the node with the i -th highest rank in our node order o . We define $R_i(v)$ as follows:

$$R_i(v) = |\mathcal{N}(v) \setminus \cup_{j=1}^{i-1} \mathcal{N}(u_j)|.$$

Meanwhile, we define the *influence set* of u_i (denoted as $I(u_i)$) as the set of non-dominated paths in $R_i(u_i)$ that either start from u_i or end at u_i . Observe that each path P in $I(u_i)$ is a canonical path with respect to our node order. Furthermore, if we construct a *TTL* based on our node order, then P would be mapped into a label $\langle u_i, \cdot, \cdot, \cdot, \cdot \rangle$. In other words, $|I(u_i)|$ equals the number of labels that are due to u_i , and we have $|\mathcal{L}| = \sum_{i=1}^n |I(u_i)|$. Intuitively, $|I(u_i)|$ captures the space overhead incurred by u_i in the *TTL* index. Similar to AC-Greedy algorithm, we define $\frac{|I(u_i)|}{|R_i(u_i)|}$ as the *cost coverage ratio*.

To order the nodes, we first choose u_1 to be the node with the minimum cost coverage ratio. After that, for $i \in [2, n]$, we decide u_i in ascending order of i , and we set u_i to the node that has the smallest cost coverage ratio.

For any node v in G , let $T_d(v)$ (resp. $T_a(v)$) be the set that contains the departure (resp. arrival) timestamps of all outgoing (resp. incoming) edges of v , and

$$\alpha = \sum_{v \text{ in } G} (|T_d(v)| + |T_a(v)|). \quad (4.2)$$

The following lemma shows an upper-bound of $|I(u_i)|$ based on $|R_i(u_i)|$ and α .

Lemma 4.9. $|I(u_i)| \leq \sqrt{\alpha} \cdot \sqrt{|R_i(u_i)|}$.

Theorem 4.1 shows the approximation ratio of our ordering method.

Theorem 4.1. *A TTL index based on a node order $u_1, u_2, \dots, u_n$ has a size that is at most $2\sqrt{\alpha}$ times of that with the optimal node ordering. To derive such a node order, it takes $O(n^2 \cdot m)$ time and $O(n \cdot m)$ space, where m is the number of edges in the timetable graph.* $\square$

4.4.2.2 Heuristic Approach

As shown in Theorem 4.1, the approximation algorithm in Section 4.4.2.1 incurs tremendous overheads, which renders it inapplicable in sizable graphs. To remedy this issue, we devise a heuristic approach that has the same spirit of the approximation algorithm, but entails much smaller computation costs. First, we randomly sample a number of edges from G . After that, for each edge $e = \langle u, v, t_d, t_a \rangle$, we employ the temporal Dijkstra's algorithm to identify all EAPs starting from u at time t_d , and employ the backward version of temporal Dijkstra's algorithm to identify all LDPs ending at v at time t_a . Notice that all these EAPs (resp. LDPs) form a tree T rooted at u (resp. v), such that the directed path from the root (resp. any leaf) to any leaf (resp. the root) is an EAP from u (resp. an LDP to v).

Furthermore, for any node w in T , the number of nodes in the subtree under w (including w itself) equals the number of EAPs from u (resp. LDPs to v) that are covered by w . We refer to this number as the *coverage* of w in T , and refer to T as the *EAP tree* from u at time t_d (resp. *LDP tree* to v at time t_a). Note that the coverage of all nodes in T can be computed in $O(n)$ time using a bottom-up scan of T . Meanwhile, for any node w in an EAP tree T rooted at u (resp. an LDP tree T rooted at v), if w is ordered before u (resp. v), then the path from u to w (resp. w to v) on T belongs to the influence set of w . So we refer to $I(w)$ as the *influence* of w in T , which is defined as the following:

$$I(w) = \begin{cases} 1 & \text{if } w \in T \\ 0 & \text{otherwise} \end{cases}$$

Let $\mathcal{T}$ denote the set of all EAP and LDP trees that we have sampled, and $u_1, u_2, \dots, u_n$ denote the node order that we chose. To decide our node order, we maintain an array that records, for each node w , the sum of the coverage of w in all trees in $\mathcal{T}$, and the sum of the influence of w in all trees in $\mathcal{T}$. We refer to this sum as the *coverage sum* and *influence sum* of w , respectively. Afterwards, we use the score obtained through dividing the influence sum by the coverage sum as the estimated cost coverage ratio ρ of node w . We first identify u_1 as the node with the minimum ρ value. Then, we remove, from each EAP or LDP tree $T \in \mathcal{T}$, the subtree under u_1 (including u_1), and we update the coverage sum, influence sum, and ρ value of each node accordingly. Subsequently, for each $i \in [2, n]$ in ascending order, we select u_i to be the node with the minimum ρ value in the modified $\mathcal{T}$ with the subtrees under $u_1, \dots, u_{i-1}$ removed.

The above heuristic approach runs in $O(n^2 \cdot |\mathcal{T}|)$ time, where $|\mathcal{T}|$ denotes the number of EAP and LDP trees in $\mathcal{T}$. To explain, observe that for any $i \in [1, n]$, we take $O(n \cdot |\mathcal{T}|)$ time to identify u_i , remove it and its subtrees from all EAP or LDP trees, and update the remaining nodes' coverage sums, influence sum, and ρ value. Therefore, the total

running time of the heuristic approach equals $O(n^2 \cdot |\mathcal{T}|)$. Apparently, a larger $|\mathcal{T}|$ leads to a better node ordering but a larger computation overheads. In our implementation of *TTL*, we start from $\mathcal{T} = \emptyset$, then iteratively sample EAP and LDP trees from G , until the total number of edges in the sampled trees reaches $x \cdot m$, where x is a constant and m is the number of edges in G . That is, we restrict the total size of the EAP and LDP trees. Our experiments show that $x = 8$ strikes a good tradeoff between index size and preprocessing time.

4.5 Label Compression

Even with a good node order o , a *TTL* index $\mathcal{L}$ may still have a sizable number of labels and incur considerable space overheads. To reduce the space consumption of $\mathcal{L}$, we propose to *compress* the labels in $\mathcal{L}$ by exploiting their similarities. In particular, we present a *route-based* compression approach in Section 4.5.1, followed by a *pivot-based* compression method in Section 4.5.2. Afterwards, we present the query algorithms under the compressed label scheme in Section 4.5.3.

4.5.1 Route-Based Compression

Consider the example in Figure 4.2a, where we have three vehicles b_1 , b_2 , and b_3 serving the same route $r_1 = \langle v_1, v_2, v_3 \rangle$, i.e., they all start from node v_1 , pass by v_2 , and then arrive at v_3 . Figure 4.2b shows the label sets of v_3 , assuming that $o(v_1) < o(v_2) < o(v_3)$. Observe that the labels in $L_{in}(v_3)$ are rather redundant, as they repeatedly record the arrival time of b_1 , b_2 , and b_3 . To eliminate such redundancy, we propose to replace the labels in $L_{in}(v_3)$ with two labels $\langle v_1, \text{null}, \text{null}, r_1, v_2 \rangle$ and $\langle v_2, \text{null}, \text{null}, r_1, \text{null} \rangle$, as shown in Figure 4.2c. That is, we *combine* the labels in $L_{in}(v_3)$ that share the same starting node, ending node, and pivot. To ensure lossless decompression, we create a time table that records, for v_1 , v_2 , and v_3 , the departure and arrival time of each bus serving route r_1 , as shown in Figure 4.2d. During query processing, if we encounter the label $\langle v_1, \text{null}, \text{null}, r_1, v_2 \rangle$ in $L_{in}(v_3)$, then we inspect the timetable entries associated with v_1 and v_3 to reconstruct three original labels concerning b_1 , b_2 , and b_3 . The label $\langle v_2, \text{null}, \text{null}, r_1, \text{null} \rangle$ can be decompressed in the same way.

In general, in the in-label set $L_{in}(v)$ of a node v , if all labels from a node u are associated with vehicles serving the same route r (hence the same pivot p), then we compress the labels into a single label $\langle u, \text{null}, \text{null}, r, p \rangle$, and create the timetables of r associated with u and v . The out-label sets in $\mathcal{L}$ can be compressed in the same manner. We observe that this *route-based* compression approach is particularly effective on a path that is only served by buses from one route, as in the case of Figure 4.2a.

4.5.2 Pivot-Based Compression

The route-based compression approach is applicable on a label l , only if the vehicle of l is not *null*, i.e., the canonical path corresponding to l is carried out by one vehicle without transfer. Nonetheless, we may still compress labels pertinent to paths with vehicle transfers, by exploiting the pivots recorded in the labels. For example, consider an in-label set $L_{in}(v)$ of a node v , where there are three labels from a node u : $l_1 = \langle u, t_d, t_a, null, p \rangle$, $l_2 = \langle u, t'_d, t'_a, null, p \rangle$, and $l_3 = \langle u, t_d^*, t_a^*, null, p \rangle$. Although the vehicles of l_1 , l_2 , and l_3 equal *null*, they all have the same pivot p , i.e., the canonical paths corresponding to l_1 , l_2 , and l_3 all pass through p . In that case, we compress l_1 , l_2 , and l_3 into a single label $l_c = \langle u, null, null, null, p \rangle$.

After that, whenever we encounter l_c during query processing, we decompress it by reconstructing l_1 , l_2 , and l_3 using their children (see Section 4.3.2). For instance, suppose that $o(u) < o(p) < o(v)$. Then, $L_{in}(p)$ must contain three labels $\langle u, t_d, \cdot, \cdot, \cdot \rangle$, $\langle u, t'_d, \cdot, \cdot, \cdot \rangle$, $\langle u, t_d^*, \cdot, \cdot, \cdot \rangle$, which are the left children of l_1 , l_2 , and l_3 , respectively. In addition, $L_{in}(v)$ should have three labels $\langle p, \cdot, t_a, \cdot, \cdot \rangle$, $\langle p, \cdot, t'_a, \cdot, \cdot \rangle$, $\langle p, \cdot, t_a^*, \cdot, \cdot \rangle$ that are the right children of l_1 , l_2 , and l_3 , respectively. Therefore, we can easily reconstruct l_1 , l_2 , and l_3 , by inspecting $L_{in}(p)$ and $L_{in}(v)$, based on the pivot information in l_c .

In general, in an in-label (resp. out-label) set, if all in-labels from (resp. out-labels to) a node u' have the same pivot p' , then we may replace them with a compressed label $\langle u', null, null, null, p' \rangle$. There is one caveat, though: if a label l and its left (resp. right) child are both compressed, then the reconstruction of l would also require reconstruction of its left (resp. right) child. Such recursive reconstruction incurs considerable overheads, which degrades query efficiency on compressed label sets. To avoid such recursions, we impose a constraint in label compression: if a label is compressed, then neither of its children should be compressed. An immediate question is, how should we choose the labels to compress, so as to minimize space consumption without violating the compression constraint?

To answer the above question, we construct a *dependency graph* as follows. First, in the in-label set $L_{in}(v)$ (resp. out-label set $L_{out}(v)$) of a node v , if we may generate a compressed label $\langle u, null, null, null, p \rangle$, then we create a tuple $\langle u, v, p \rangle$ (resp. $\langle v, u, p \rangle$). After that, we map each tuple into a node, and draw an undirected edge between any two nodes with a parent-child relationship, e.g., $\langle u, v, p \rangle$ and $\langle u, p, \cdot \rangle$. In addition, we assign to each node $\langle u, v, p \rangle$ a weight that equals $c - 1$, where c equals the number of labels compressed into $\langle u, v, p \rangle$. In other words, c captures the reduction of space due to $\langle u, v, p \rangle$. We refer to the resulting graph as a dependency graph D .

Observe that, if we are to create a set S of compressed labels under our compression constraints, then the labels in S correspond to an independent set on D . Accordingly, we can maximize the effect of compression by identifying the maximum-weight independent

set on D , which is NP-hard. We address the problem by adopting an approximation algorithm [73] for maximum-weight independent set, which runs in $O(n' \log n' + m')$ time, where n' and m' are the numbers of nodes and edges in D , respectively.

Finally, we note that the above *pivot-based* compression method can be combined with the route-based compression approach in Section 4.5.1. Specifically, we first apply route-based compression on the label sets, and then, employ pivot-based compression but exclude those labels that have been compressed using the route-based approach. As such, we benefit from both types of compression. Next, we present a more detailed description of *TTL*'s query algorithm under a compressed index structure.

4.5.3 Query with Compressed Labels

In above sections, we have explained how we can reconstruct original labels from the ones generated by our compression methods. In what follows, we clarify how we can process queries on the compressed index. In particular, we first explain the candidate generation and refinement process on compressed labels, and then introduce the algorithm for path reconstruction. For ease of exposition, we focus on SDP queries, but our solution can be easily extended to EAP and LDP queries.

Candidate Generation and Refinement. Given a query asking for the SDP from u to v with a starting timestamp t and an ending timestamp t' , we generate the set S_{uv} of path sketches by linearly scanning $L_{out}(u)$ and $L_{in}(v)$ using Algorithm 6, with one modification: whenever we encounter a compressed label l , we decompress it based on the compression technique used. In particular, if l is produced by the route-based compression, then l should be in the form of $\langle \cdot, null, null, r, \cdot \rangle$, i.e., the route information r of l is available. Then, as explained in Section 4.5.1, the original labels associated with l can be easily retrieved from the timetable associated with r . Next, consider that l is generated by the pivot-based compression. In that case, l should be in the form of $\langle \cdot, null, null, null, p \rangle$. Let $l_1, l_2, \dots, l_j$ ($j \geq 1$) be the original labels associated with l . Without loss of generality, assume that $o(u) < o(p) < o(v)$. Then, as explained in Section 4.5.2, the left (resp. right) children of $l_1, l_2, \dots, l_j$ are stored in $L_{in}(p)$ (resp. $L_{in}(v)$). As such, we can recover $l_1, l_2, \dots, l_j$ by inspecting the labels in $L_{in}(p)$ and $L_{in}(v)$ based on the pivot information of l . After that, we can replace l with $l_1, l_2, \dots, l_j$ in the linear scan.

After the linear scan, the set S_{uv} of all the path sketches with respect to the SDP query can be obtained. Then, we identify the sketch s_{ans} in S_{uv} that is pertinent to the SDP query result, using the same approach as described in Section 4.3.1. Note that all labels in s_{ans} are not compressed, due to the decompression operations performed during the linear scan.

Algorithm 9: *CPathUnfold*

```

input  : a label  $l = \langle \cdot, \cdot, \cdot, \cdot, p \rangle$  in TTL
output: the canonical path corresponding to  $l$ 

1 if  $p = \text{null}$  then
2   return a path that only contains the edge in  $G$  corresponding to  $l$ ;
3 else
4    $l_1$  = the left child of  $l$ ;
5    $l_2$  = the right child of  $l$ ;
6   let  $l'_1$  be the original label of  $l_1$  to be computed;
7   if  $l_1$  is a label with route-based compression then
8     obtain  $l'_1$  by inspecting the timetable of the route associated with  $l_1$ ;
9   else if  $l_1$  is a label with pivot-based compression then
10    obtain  $l'_1$  by inspecting the left and right children of  $l_1$ ;
11    $P_1 = \text{CPathUnfold}(l'_1)$ ;
12   repeat Lines 7-14 by replacing  $l_1$  with  $l_2$ ,  $l'_1$  with  $l'_2$ , and  $P_1$  with  $P_2$ ;
13   set  $P$  = a concatenation of  $P_1$  and  $P_2$ , with  $P_1$  preceding  $P_2$ ;
14   return  $P$ ;

```

Path Reconstruction. Once s_{ans} is identified, we proceed to reconstruct the path corresponding to s_{ans} . Algorithm 9 presents the pseudo-code of *CPathUnfold*, namely, the method for path reconstruction with the compressed index. *CPathUnfold* takes as input a label l (in s_{ans}), which, as mentioned, is guaranteed to be uncompressed due to the way s_{ans} is generated. Without loss of generality, assume that the left child l' of l is compressed. Then, the departure timestamp of l' is the same as that of l , and the starting node and ending node of l' can be learnt from l . Therefore, to decompress l' , it suffices to identify the arrival timestamp and vehicle of l' . If l' is produced from the route-based compression, then we can pinpoint the arrival timestamp and vehicle of l' by inspecting the timetable of the route associated with l' . Meanwhile, if l' is generated by the pivot-based compression, then l' can be reconstructed by inspecting the children of l' and the pivot information in l' (as explained in Section 4.5.2). The case that the right child of l is compressed can be addressed in a similar manner.

4.6 Extensions

Concise Representation of Query Results. Recall that our query algorithms always return a path P represented as a sequence of edges $\langle e_1, e_2, \dots, e_k \rangle$ in G . Although such an edge sequence provides very detailed information about P , users in practice may prefer a more concise representation of P as the query result. For example, suppose that $e_1, e_2, \dots, e_j$ ($j < k$) have the same vehicle b_1 , while $e_{j+1}, \dots, e_k$ all concern the same

vehicle b_2 . In other words, the path P requires the user to (i) take vehicle b_1 from node u at time t_d (where u and t_d are the starting node and departure time of e_1 , respectively), and then (ii) get off b_1 at node w and transfer to vehicle b_2 at time t'_d (where t'_d is the departure time of e_{j+1}), and stay on b_2 until she reaches her destination. In that case, instead of presenting P to the user, we could provide her with a concise description of P_c , which is a sequence of two tuples: $\langle u, b_1, t_d \rangle, \langle w, b_2, t'_d \rangle$. Such a *concise path* is more user-friendly as it is easier to comprehend and remember than P . Moreover, the concise representation of query results is favorable to the compression techniques in Section 4.5, since it reduces the amount of label decompression required during query processing.

To generate a concise path P_c for a query, a straightforward approach is to first derive the corresponding path P , and then convert P into P_c by identifying the edges on P where there are changes of vehicles. Interestingly, our query algorithm can be easily modified to return P_c without even deriving P . In particular, given a path query q , we first invoke our candidate generation and refinement algorithms (in Section 4.3.1) to identify the path sketch $s_{ans} = \langle l_u, l_v \rangle$ of the query result. After that, we invoke the *PathUnfold* algorithm to unfold the labels in s_{ans} , with one slight modification: during the unfolding process, whenever we encounter a label l whose vehicle does not equal *null* (i.e., the path corresponding to l does not involve a transfer), then we do not unfold l . The output of modified algorithm would then be a *partially unfolded* path, which is a sequence where each element is either an edge in G or a label whose corresponding path is carried out by one vehicle. After that, it is straightforward to construct P_c from the partially unfolded paths. This partial unfolding approach leads to higher query efficiency since it not only reduces the number of paths to unfold, but also accelerates the construction of P_c .

Extended Timetables. So far we have assumed that we are given a timetable graph G with a finite set E of temporal edges. For example, E corresponds to the predefined vehicles routes on a weekday. However, if we construct a *TTL* index on such a timetable graph G , then the index can only return paths whose starting and ending time are within the same day, e.g., we will miss paths that start on late Monday night and ends on early Tuesday morning. To address this issue, we can extend E to include the timetables of two consecutive weekdays, and construct a *TTL* index accordingly. Then, the index could return any path whose total duration does not exceed 24 hours. In case that the weekend timetable is different from that of a weekday, we can create three additional *TTL* indices, each of which incorporates the timetables of two consecutive days from Friday to Monday. We note that such *indexing partitioning* approach is widely adopted in spatial-temporal indexing (e.g., [58]).

4.7 Experiments

This section experimentally evaluates *TTL* on a machine with an Intel E5-2650V2 2.60GHz CPU and 64GB RAM, running Ubuntu 14.04.1. All methods tested are implemented in

dataset	$ V $	$ E $	avg degree
Austin	2.7K	317.5K	118.7
Madrid	4.6K	1912.2K	412.5
Budapest	5.5K	1375.2K	251.5
Salt Lake City	6.3K	329.0K	52.6
Rome	8.8K	2267.7K	258.3
Denver	9.5K	708.7K	74.5
Houston	9.8K	1111.8K	112.9
Toronto	10.8K	3295.1K	305.4
Berlin	12.8K	1965.8K	153.0
Los Angeles	15.0K	1903.2K	126.6
Sweden	51.4K	3926.5K	76.4

Table 4.4: Public transportation networks ($K=10^3$).

C++ and compiled with GCC 4.8.2. In each experiment, we repeat each method 5 times, and report the average measurement.

Datasets. We use 11 publicly available datasets from [1] as our datasets. Each dataset records the timetable of the public transportation network of a major city or country on a weekday. Table 4.4 shows the characteristics of the data.

Query Sets. For each dataset, we generate 10^5 queries for EAP, LDP, and SDP, respectively, and we report the average query time of each method for each query set. Each query is generated by selecting source and destination node uniformly at random. In addition, the starting and ending timestamps of each query are also randomly generated, except that the starting (resp. ending) timestamp of an LDP (resp. EAP) query is always set to $-\infty$ (resp. $+\infty$).

Methods. We evaluate two versions of *TTL*: one with the compression methods in Section 4.5 applied (referred to as *C-TTL*), and one without (referred to as *TTL*). For each version, we test its query performance under two scenarios: when it returns a complete path in answering a query, and when it returns a concise representation of the path. We compare all versions of *TTL* with the state-of-the-art indexing methods on timetable graphs, namely, *CHT* [46] and *CSA* [36]. Unless otherwise specified, we use heuristic algorithm in Section 4.4.2.2 to decide the node ordering used in *TTL* and *C-TTL*.

4.7.1 Performance of Query Processing

In the first set of experiments, we evaluate the query efficiency of all methods. Figure 4.3 shows the average query time of each method for SDP queries in log-scale. Observe that, *TTL* and *C-TTL* significantly outperform *CSA* and *CHT* in all cases. In particular, the

Figure 4.3: Average query time for SDP queries.

Figure 4.4: Average query time for EAP queries.

Figure 4.5: Average query time for LDP queries.

Figure 4.6: Index size.

Figure 4.7: Total preprocessing time.

average query time of *TTL* is always below $30\mu s$, and is at least three (resp. two) orders lower than that of *CSA* (resp. *CHT*) in almost all cases. *C-TTL* is slightly slower than *TTL* (as it needs to decompress labels during query processing), but is still orders of magnitude faster than *CSA* and *CHT*. Meanwhile, *CHT* is superior to *CSA*, as it adopts

a more advanced index structure than the latter.

Interestingly, the query time of *TTL* and *C-TTL* does not always increase with the size of the input dataset. To explain, observe that the average query cost of *TTL* and *C-TTL* is decided by two factors: (i) the average number of labels in each label set, denoted as l_{avg} , and (ii) the average number of path unfolding operations required during path reconstructions, which is roughly comparable to the average number of nodes on the query result path (denoted as n_{avg}). We observe that neither l_{avg} nor n_{avg} necessarily increases with the dataset size. For example, on the *Austin* dataset, $l_{avg} \approx 1600$, and $n_{avg} \approx 39$ for SDP queries; in contrast, on *Sweden*, we have $l_{avg} \approx 775$ and $n_{avg} \approx 19$ for SDP queries, even though *Sweden* is more than ten times larger than *Austin*. In addition, we note that a similar phenomenon is also demonstrated in previous work [6,8] on labelling indices for conventional graphs, i.e., the query cost of a labelling approach depends more on the input data's topology than its size.

The above results concern the case when we return a detailed path for each SDP query. In addition to that, Figure 4.3 also plots the performance of *TTL* and *C-TTL* when they return concise paths as query results. Observe that the query efficiency of both *TTL* and *C-TTL* improves noticeably when concise paths are adopted. This is because when *TTL* and *C-TTL* return concise paths instead of complete paths, their costs of path reconstruction during query processing are significantly reduced, due to the decreased amount of path unfolding operations, as mentioned in Section 4.6. In addition, the performance gap between *TTL* and *C-TTL* is reduced in the case of concise paths, because *C-TTL* performs a smaller number of label decompressions when constructing concise paths.

Apart from SDP queries, we also evaluate the average query time of each method for EAP and LDP queries, respectively, and we observe that *TTL* and *C-TTL* consistently outperform *CSA* and *CHT* by large margins. In particular, Figure 4.4 and Figure 4.5 report the average query time for EAP and LDP queries, respectively. Observe that the performance of each method for EAP and LDP queries on each dataset is generally similar. However, *CSA* and *CHT* incur several times less query time for EAP and LDP queries than for SDP queries. This is because, to answer EAP (resp. LDP) queries, they only maintain the earliest arrival time to (resp. latest departure time from) each node that is traversed. However, to answer an SDP query, they both need to maintain a list to record the departure time and arrival time of each non-dominated path that ends at a certain traversed node, which can be excessively large. Overall, *TTL* and *C-TTL* still outperform *CSA* and *CHT* by a large margin.

4.7.2 Preprocessing and Space Overheads

Our second set of experiments evaluates the space overheads of each method. Figure 4.6 shows the results in log-scale. The index size of *CSA* is rather small, since it only requires

dataset	$\Delta_1/ \mathcal{L} $	$\Delta_2/ \mathcal{L} $	$\Delta_3/ \mathcal{L} $
Austin	20.25%	33.66%	46.20%
Madrid	16.52%	30.23%	45.61%
Budapest	10.41%	35.45%	41.71%
Salt Lake City	38.00%	38.78%	61.43%
Rome	14.42%	31.85%	43.87%
Denver	23.84%	34.92%	49.23%
Houston	28.77%	32.62%	49.54%
Toronto	19.95%	32.77%	44.78%
Berlin	6.70%	23.51%	27.58%
Los Angeles	16.78%	31.68%	41.66%
Sweden	12.74%	28.52%	35.92%

Table 4.5: Compression ratio.

storing two copies of G (in the form of sorted edge sequences). The space consumption of CHT is comparable to that of CSA , which indicates that the shortcuts that it creates take roughly the same space as the input graph G . On the other hand, the space costs of TTL and $C-TTL$ are considerably larger than those of CSA and CHT . However, the relatively large space overhead of TTL and $C-TTL$ is justified by their superior query performance against CSA and CHT . Furthermore, even on the country-scale transportation network *Sweden*, the index size of TTL (resp. $C-TTL$) is only a few hundred MB, which can easily fit in the main memory of a commodity machine. In other words, the space overheads of TTL and $C-TTL$ are still acceptable for practical applications.

In the next set of experiments, we evaluate the effectiveness of label compression schemes. In particular, on each dataset, we construct four TTL indices: (i) one without compression, denoted as $\mathcal{L}$; (ii) one with route-based compression, denoted as $\mathcal{L}_1$; (iii) one with pivot-based compression, denoted as $\mathcal{L}_2$, and (iv) one with both route-based and pivot-based compressions, denoted as $\mathcal{L}_3$. We define $\Delta_i = |\mathcal{L}| - |\mathcal{L}_i|$ ($i = 1, 2, 3$). Table 4.5 shows the values of $\Delta_1/|\mathcal{L}|$, $\Delta_2/|\mathcal{L}|$, and $\Delta_3/|\mathcal{L}|$ on each dataset. Observe that, by using route-based compression and pivot-based compression separately, our index size can be reduced by up to 38.00% and 38.78%, respectively. When both compression techniques are applied, the size of our TTL indices can be reduced by up to 61.43%.

This reduction in space consumption comes at a moderate cost of query efficiency, though. For instance, on the *Budapest* dataset, our compression techniques reduce the size of $C-TTL$ by 41.71%, but increase the query time (when returning concise paths) from $10\mu s$ to $12\mu s$. Nonetheless, even with compression applied, the query performance of $C-TTL$ is still significantly better than that of CSA and CHT , as shown in Figure 4.3. As such, $C-TTL$ is a competitive method compared to TTL , especially when the index size is a concern (e.g., when the index is to be loaded in a mobile device).

Figure 4.8: Total preprocessing time.

Figure 4.9: Index size.

Finally, Figure 4.7 shows the preprocessing time of all methods. *CSA* incurs the smallest pre-computation overheads as it only requires sorting the edges in G twice. *CHT* has a much larger preprocessing cost, since it needs to construct a sizable number of shortcuts on G . The pre-computation time of *TTL* and *C-TTL* are larger than that of *CSA* and *CHT*, but is still below 17 minutes even on the largest dataset *Sweden*. Furthermore, *TTL* and *C-TTL* have similar pre-processing overheads, which indicates that the compression algorithms adopted by *C-TTL* entail negligible additional costs.

4.7.3 Index Construction and Node Orders

In the last set of experiments, we evaluate the efficiency of our index construction algorithm and the effectiveness of our ordering algorithms. In particular, we first compare our index construction algorithm with a baseline approach that (i) runs the temporal Dijkstra’s algorithm $O(n)$ times to compute all temporal shortest paths in G , and then (ii) examines the paths obtained to identify the canonical paths. We denote the baseline approach as *brute-force Dijkstra*. Figure 4.8 reports the index construction time of Algorithm 8 (referred to as *H-Order*) and *brute-force Dijkstra*, both of which adopt our heuristic ordering algorithm mentioned in Section 4.4.2.2 as the ordering methods. Observe that Algorithm 8 is at least two orders of magnitude faster than *brute-force Dijkstra*, even though both algorithms have the same worst-case time complexity. The main reason is that Algorithm 8 performs a more advanced version of the Dijkstra’s search from v_i , with various heuristics adopted to prune the search space (see Section 4.4). For instance, whenever it encounters a node v_j with $o(v_j) < o(v_i)$ during the search, it would omit all edges pertinent to v_j , since any path that contains those edges would violate the Rank Constraint in Definition 4.5 (and hence, cannot be canonical). This demonstrates that

our index construction algorithm is highly superior to the *brute-force Dijkstra* approach in terms of practical efficiency.

Next, we compare our heuristic ordering algorithm (referred to as *H-Order*) and our approximate ordering algorithm (referred to as *A-Order*), with a baseline approach that orders nodes randomly (denoted by *Rand-Order*). Figure 4.9 reports the index size of *TTL* (in log-scale) for each of the three node ordering methods. The results of *A-Order* are omitted on some datasets due to its prohibitive memory consumption ($> 64GB$). Observe that, both *A-Order* and *H-Order* outperform the baseline approach by large margins in terms of index size, which demonstrates the importance of node ordering in index construction and the superiority of our approaches. In particular, the index size of *H-Order* is more than one order of magnitude smaller than that of *Rand-Order* on *Berlin* dataset. Besides, the index size of *H-Order* is comparable to that of *A-Order*, which shows the effectiveness of our heuristic ordering approach.

Figure 4.8 presents the total processing time of *TTL* for each ordering methods. The total processing time of *A-Order* is at least 2 orders of magnitude larger than that of *H-Order*, since *A-Order* incurs significant overheads in computing all non-dominated paths and updating the marginal coverage of each node. In addition, *Rand-Order* has a total processing time several times larger than that of *H-Order*, since the former generates an enormous number of labels (see Figure 4.9). In summary, our heuristic ordering method yields similar index size as the approximation algorithm, but incurs much less preprocessing cost.

4.8 Summary

This chapter presents *Timetable Labelling (TTL)*, an efficient indexing technique for route planning on public transportation networks. The basic idea of *TTL* is to associate each node u with a set of *labels*, each of which records the shortest travel time from u to some other node v given a certain departure time from u ; such labels would then be used during query processing to improve efficiency. We investigate various aspects in the design of *TTL* indices, and propose advanced algorithms for query processing, index construction, and label compression. With extensive experiments on real datasets, we show that *TTL* significantly outperforms the state-the-art in terms of query efficiency.

Chapter 5

Constrained Shortest Path on Road Networks

Nowadays route planning via online mapping/navigation services has become an essential part of driving in many places. Most popular online maps today, including Google Maps [5], Bing Maps [4] and iOS Maps [3], compute routes based on a single criterion, which is usually either the total route length or the total travel time. In practice, however, the user may have multiple criteria in mind when selecting the best route, and the relative importance of these criteria may change depending on the nature of the trip. Figure 5.1 shows such an example with two possible paths from an origin s to a destination t . Path 1 is shorter, but slower as it goes through a university campus with many pedestrians and speed bumps. In contrast, Path 2 is longer and yet faster as it contains mainly highway segments. Clearly, the user should choose Path 2 if she needs to rush to the destination as quickly as possible. On the other hand, if the trip is not time-critical, Path 1 is probably more pleasant and less stressful, compared to Path 2 which takes a long detour. Hence, in this example the choice of the best path depends on two different criteria (total path length and total travel time), as well as their relative importance, which varies in different trips.

Figure 5.1: Example of multiple criteria when choosing a route

One way to handle multiple criteria is to present to the user a set of results, and let the user choose the best one manually. For example, the system could return the set of

skyline paths [19], in which no result is *dominated* (i.e., no better in all criteria and strictly worse in one) by another. However, this approach leaves the burden of choice to the user and increases the complexity for user-system interactions; the problem is exacerbated when there are a large number of result paths in the skyline set, e.g., for a long trip on a dense road network. Such complex interactions can be a dangerous distraction for the user driving a vehicle.

In this chapter we focus on a different approach: *constrained shortest path (CSP)* [54, 70], which optimizes based on one criterion and treats all other criteria as constraints. For instance, in the example of Figure 5.1, we can present a slider bar to the user which controls the maximum total travel time, e.g., as a percentage of the shortest trip duration. Then, the system minimizes the total travel distance while satisfying the time constraint specified by the user. In particular, we focus on two criteria (i.e., one objective and one constraint) for selecting the best path, for two reasons. First, in terms of effectiveness, the complexity of the user-system interaction significantly increases with each additional constraint, e.g., fine tuning with two slider bars is far more complex than with one. Second, in terms of efficiency, answering a CSP query with a single constraint is already rather expensive as we explain soon; processing CSPs with multiple constraints may not be feasible for large networks even with the proposed methods.

Formally, CSP assigns each edge in the road network a length and a cost¹; given an origin s , a destination t , and a cost constraint θ , the result of the CSP query is the path from s to t that minimizes its total length, while satisfying that its total cost does not exceed θ . The CSP problem is known to be NP-hard [45, 54]. Hence, the majority of existing work, e.g., [17, 40, 49, 54, 70, 94], focuses on approximate solutions, which guarantees that the result path length is no longer than α times of the optimal path length (where α is the user-specified approximation ratio), subject to the cost constraint θ . Although there exist polynomial-time algorithms for approximate CSP (e.g. [40, 49, 54, 70, 94]), as we show in the experiments, the current state-of-the-art solutions are still prohibitively expensive for large networks. There are two main reasons for their inefficiency. First, they aim at answering approximate CSPs on a general graphs, rather than specifically on road networks; consequently, they fail to utilize the latter's special properties. Second and more importantly, most of them process each query on-the-fly, without the help of any precomputed index structure. The few known indices are all designed for exact CSPs; further, these indices consume large amounts of memory, and yet none of them succeeds at reducing query cost to a practical level.

We thus propose a novel and practical solution *COLA* for index-based approximate CSP processing on large road networks. *COLA* mainly exploits two important properties of the road network. First, real road networks are often (roughly) planar, and, thus, can

¹Note that the edge length and cost can be defined arbitrarily. Without loss of generality we assume that the objective criterion is the length of a path and the constraint is on the total cost of a path.

be effectively split into partitions, each of which contains only a relatively small number of boundary vertices. Accordingly, *COLA* partitions the network, builds an overlay graph on the partitions, and indexes the skyline paths between pairs of boundary vertices. Second, in practice there often exist a relatively small number of landmark vertices in the road network that commonly appear in CSP results. Based on this property, we design an index-free algorithm α -*Dijk* as a component of *COLA* for path computation within a partition, i.e., between the origin (or destination) vertex and its corresponding partition boundaries. Extensive experiments using real continent-sized road networks containing tens of millions of vertices show that *COLA* answers an approximate CSP query within a second, whereas previous solutions need several hours. Further, even when an index is not available (e.g., when the edge lengths / costs change frequently), the α -*Dijk* module still outperforms existing methods by over an order of magnitude.

In the following, Section 5.1 provides necessary background of the problem, which paves the way to present the *COLA* framework in Section 5.2. After that, Section 5.3 describes our *COLA* implementation, including the α -*Dijk* algorithm for approximate CSP processing without an index. Finally, Section 5.4 contains an extensive set of experiments, and Section 5.5 gives a summary of this chapter.

5.1 Background

Let $G = (V, E)$ be a directed road network with a vertex set V and an edge set E . Each edge $e \in E$ is associated with a *length* $\ell(e) \geq 0$ and a *cost* $c(e) \geq 0$. For a path $P = \langle e_1, e_2, \dots, e_k \rangle$ in G , the *length* and *cost* of P are defined as $\ell(P) = \sum_{i=1}^k \ell(e_i)$ and $c(P) = \sum_{i=1}^k c(e_i)$, respectively. Following previous work [17, 40, 49, 54, 70, 94], we assume that the length of each edge is an integer. In practice this can be done by measuring the edge length in a sufficiently small unit, e.g., foot or meter, if the edge length represents travel distance. Similarly, we assume that the cost of each edge is also an integer. We use ℓ_{max} (resp. c_{max}) to denote the maximum length (resp. cost) of an edge in G . Meanwhile, we assume that ℓ_{max} (reps. c_{max}) is non-zero; otherwise, the problem can be trivially regarded as conventional shortest path problem.

Given an origin vertex $s \in V$, a destination vertex $t \in V$, and a cost constraint θ , a *constrained shortest path (CSP)* query asks for the shortest path P among all paths from s to t with costs no more than θ . If there exist multiple CSPs with the same length, we break ties by the cost of the paths. The CSP problem has been proven to be NP-hard, if both ℓ_{max} and c_{max} can be arbitrarily large [45, 54]. On the other hand, if either ℓ_{max} or c_{max} is polynomial to the number of vertices, there exist polynomial-time solutions for CSP, e.g., [54, 64]. Nevertheless, as we have reviewed in Section 2.4, these algorithms incur tremendous costs for large graphs, and, thus, are far from practical. As such, recently much effort has been devoted to solving approximate versions of the CSP problem. This chapter follows a popular definition called α -CSP, defined as follows.

Table 5.1: List of notations.

Symbol	Meaning
$G = (V, E)$	Input graph
n, m	Numbers of vertices and edges in G
$\ell(e), c(e)$	Length and cost of an edge e
ℓ_{max}, c_{max}	Maximum length and cost for any edge in G
α	Approximation ratio in α -CSP
s, t	Query origin and destination vertices
$\mathcal{T}$	A partitioning of graph G
G_s, G_t	Subgraph in $\mathcal{T}$ containing s and t , respectively
$G^\circ = (V^\circ, E^\circ)$	Overlay graph of G (refer to Section 5.2.1)

Definition 5.1 (α -CSP query). *Given an origin s , a destination t , a cost constraint θ , and an approximation ratio α , an α -CSP query returns a path P , such that $c(P) \leq \theta$, and $\ell(P) \leq \alpha \cdot \ell(P_{opt})$, where P_{opt} is the optimal answer to the exact CSP query with the origin s , the destination t , and the cost constraint θ . $\square$*

Example 5.1. Figure 5.2 illustrates an example of exact- and α -CSP on a graph with 5 vertices $v_1, v_2, \dots, v_5$. The length and cost for each edge are also shown in the figure. For example, the edge from v_1 to v_2 has cost $c = 1$ and length $\ell = 2$. Given origin $s = v_1$, destination $t = v_5$, and cost constraint $\theta = 6$, the CSP query returns the path $P_{opt} = \langle (v_1, v_3), (v_3, v_5) \rangle$, since (i) $c(P_{opt}) = 6 \leq \theta$ and (ii) the length $\ell(P_{opt}) = 5$ is the smallest among all paths from v_1 to v_5 with a cost no more than θ . Meanwhile, for $\alpha = 1.2$, a valid result for the α -CSP query with the same parameters $s = v_1, t = v_5$ and $\theta = 6$ is $P_\alpha = \langle (v_1, v_2), (v_2, v_5) \rangle$, since $c(P_\alpha) = 5 \leq \theta$, and $\ell(P_\alpha) = 6 \leq \alpha \cdot \ell(P_{opt}) = 6$. $\square$

 Figure 5.2: Example of exact CSP and α -CSP

Two important concepts in solving α -CSP are *dominance relationship* and *skyline*. Recall from Section 2.4 that α -CSP is defined as follow.

Definition 5.2 (α -dominance). *Let P_1 and P_2 be two paths connecting the same origin and destination vertices. P_1 α -dominates P_2 iff $c(P_1) \leq c(P_2)$ and $\ell(P_1) \leq \alpha \cdot \ell(P_2)$. $\square$*

For instance, consider paths $P_1 = \langle (v_1, v_2), (v_2, v_5) \rangle$ and $P_2 = \langle (v_1, v_3), (v_3, v_5) \rangle$ in the above example. When $\alpha = 1.2$, P_1 α -dominate P_2 , since (i) the cost of P_1 is $c(P_1) = 5 \leq c(P_2) = 6$, and (ii) the length of P_1 is $\ell(P_1) = 6 \leq \alpha \cdot \ell(P_2) = 1.2 \times 5 = 6$.

Based on the above definition, a set of paths S is called a *skyline set*, iff no path in S is α -dominated by another in the same set S . We say that a path P is a *skyline path* if P is in a skyline set. Note that if two paths P_1 and P_2 have the same cost, it is possible that they α -dominate each other, in which case we can put either one of them in a skyline set. For exact CSP, we define dominance relationship and skyline in the same way, by simply fixing $\alpha = 1$. Table 5.1 summarizes common symbols throughout this chapter.

5.2 COLA Framework

This section presents the general framework of our proposed solution *constrained labeling (COLA)*. The implementation of several important components in *COLA* is described in Section 5.3. Basically, *COLA* partitions the road network and constructs an *overlay graph* on top of the partitions. The index structure in *COLA* is then built on the overlay graph, whose size is much smaller than the original graph, leading to much less query processing costs. In the following, Section 5.2.1 describes the overlay graph; Section 5.2.2 explains the index structure of *COLA*; Section 5.2.3 elaborates on query processing based on the *COLA* index; Section 5.2.4 presents several sophisticated optimizations that significantly reduces query costs.

5.2.1 Overlay Graph

Given an input graph G , a partitioning of G consists of a set $\mathcal{T} = \{G_1, G_2, \dots, G_{|\mathcal{T}|}\}$ of edge-disjoint subgraphs of G , such that the union of all G_i ($1 \leq i \leq |\mathcal{T}|$) equals G . Given $\mathcal{T}$, we say that a vertex v is a *boundary vertex*, if v appears in more than one subgraphs in $\mathcal{T}$. Graph partitioning is a well-studied problem, and *COLA* could use any of the existing solutions, e.g., [31, 63, 65, 82, 83]. In our implementation, we use a state-of-the-art approach for road networks by Delling et al. [31].

We formally define an overlay graph as follows.

Definition 5.3 (Overlay graph). *Given an input graph G , a partitioning $\mathcal{T}$ of G , and the query approximation ratio α^2 , a graph $G^\circ = (V^\circ, E^\circ)$ is an overlay graph of G with respect to $\mathcal{T}$, if it satisfies the following three conditions:*

²Note that the overlay graph and the *COLA* index both require the knowledge of α , which we consider as a system parameter. The choice of α is discussed further in Section 5.4.4.

 Figure 5.3: Example overlay graph ($\alpha = 1.2$).

- (i) V° consists of all boundary vertices with respect to $\mathcal{T}$;
- (ii) For each edge $e^\circ \in E^\circ$ that starts at vertex v and ends at vertex v' , there exists a path P in G that goes from v to v' , such that $c(P) = c(e^\circ)$ and $\ell(P) = \ell(e^\circ)$;
- (iii) For any pair of origin and destination vertices $s, t \in G^\circ$, and any path P in G from s to t , there exists a path P° in G° that goes from s to t that α -dominates P , i.e., $c(P^\circ) \leq c(P)$ and $\ell(P^\circ) \leq \alpha \cdot \ell(P)$.

□

Intuitively, an overlay graph compresses the input graph by (i) including only the boundary vertices of the partitions and removing all other vertices, (ii) using edges to represent paths in G , and (iii) reducing the number of edges by removing paths that are α -dominated by others. For any two vertices v and v' in the overlay graph G° , there could be multiple edges from v to v' , when there are multiple paths from v to v' in G and no single path can α -dominate all others. Note that the above definition does not require the overlay graph to be minimal, i.e., for the same graph G and partitioning $\mathcal{T}$, there may be another possible overlay graph with few edges. Hence, there can be different ways to build an overlay graph, and we explain one such algorithm later in Section 5.3.2. Note that,

Example 5.2. Consider again graph G in Figure 5.2. Figures 5.3(a) shows a partitioning $\mathcal{T}$ of G consisting of three subgraphs G_1 - G_3 . Specifically, G_1 contains vertex set $V_1 = \{v_1, v_2, v_3\}$, and edges in G between these three vertices. Similarly, G_2 and G_3 involve vertex sets $V_2 = \{v_3, v_5\}$ and $V_3 = \{v_2, v_4, v_5\}$, respectively, as well as the corresponding edges in G . $\mathcal{T}$ is a partitioning of G , since (i) the three subgraphs contain disjoint sets of edges and (ii) the union of their edges constitute the set of all edges in G .

Figure 5.3(b) shows the overlay graph G° corresponding to $\mathcal{T}$ for $\alpha = 1.2$. The vertex set of G° is $V^\circ = \{v_2, v_3, v_5\}$, which are the boundary vertices between the three subgraphs. To determine the edge set E° of G° , we examine each pair of vertices in V° . Between vertices v_2 and v_3 , there is only one path in G , i.e., the edge (v_2, v_3) . Hence,

E° includes this path. Similarly, E° also includes the only path between v_3 and v_5 . Between v_2 and v_5 , there are three paths: $P_1 = \{(v_2, v_5)\}$, $P_2 = \{(v_2, v_4), (v_4, v_5)\}$ and $P_3 = \{(v_2, v_3), (v_3, v_5)\}$. According to the edge lengths and costs shown in Figure 5.2, P_1 α -dominates P_2 ; thus we eliminate P_2 from E° . Meanwhile, observe that path P_3 already exists in G° , since E° contains both of its two segments (v_2, v_3) and (v_3, v_5) . Hence there is only one edge between v_2 and v_5 in E° , which corresponds to P_1 . $\square$

Since the overlay graph can be pre-computed and has a smaller size than the original graph, it can be used as a low-cost index to accelerate α -CSP processing, as follows. Given an α -CSP query q on G with an origin s , a destination t , and a cost threshold θ , we first identify the subgraphs G_s and G_t in $\mathcal{T}$ that contain s and t , respectively. Then, we construct graph G^q (which we call an *extended graph*) by merging G_s , G_t and G° , i.e., $G^q = (V_s \cup V_t \cup V^\circ, E_s \cup E_t \cup E^\circ)$, where V_s (resp. E_s) and V_t (resp. E_t) are the vertex (resp. edge) sets of G_s and G_t , respectively. After that, we run an α -CSP algorithm on G^q ; its result corresponds to a result for the original α -CSP query, according to the following lemma³.

Lemma 5.1. *Let P_{opt} be a result of the exact CSP query on graph G with origin s , destination t , and cost constraint θ . Meanwhile, let $G_s = (V_s, E_s)$, $G_t = (V_t, E_t)$ be the subgraphs in a partitioning $\mathcal{T}$ that contain s and t , respectively. Then, any result P of an α -CSP with parameters s, t, α on the extended graph $G^q = (V_s \cup V_t \cup V^\circ, E_s \cup E_t \cup E^\circ)$ satisfies $c(P) \leq \theta$ and $\ell(P) \leq \alpha \cdot \ell(P_{opt})$. $\square$*

To translate an α -CSP on G^q to an α -CSP on G , we “unfold” each edge in E° into a path in G , which can be done according to Condition 2 in Definition 5.3. Because G^q can be viewed as a compressed version of G with significantly fewer vertices and edges, searching for an α -CSP on G^q is expected to be faster than doing so on G . On the other hand, the speedup using an overlay graph is limited, since the query processing algorithm is the same, albeit on a smaller graph. Next we introduce a much more powerful index structure.

5.2.2 Constrained Labeling Index

The main *COLA* index is constructed on the overlay graph $G^\circ = (V^\circ, E^\circ)$ described in the previous subsection. For each vertex $v^\circ \in G^\circ$, the index contains two label sets for v° : an *in-label set* $B_{in}(v^\circ)$ and an *out-label set* $B_{out}(v^\circ)$. Each entry in $B_{out}(v^\circ)$ corresponds to

³All proofs in this chapter can be found in the Appendix A.

a path⁴ from v° to another vertex in G° . Symmetrically, each label in $B_{in}(v)$ corresponds to a path from another vertex in G° to v° . The paths in the label sets are carefully chosen such that given any pair of origin and destination vertices $s^\circ, t^\circ \in G^\circ$, and a cost constraint θ , we can construct the α -CSP from s° to t° subject to θ using only the paths in $B_{out}(s^\circ)$ and $B_{in}(t^\circ)$. In other words, with the *COLA* index we do not need to search for the α -CSP result; instead, we simply combine pre-computed paths in the label sets to form a result.

Formally, we define the *COLA* index as follows.

Definition 5.4 (*COLA index*). *Given an overlay graph G° , a COLA index contains label sets $B_{in}(v^\circ)$ and $B_{out}(v^\circ)$ for each vertex $v^\circ \in G^\circ$ satisfying the following conditions:*

- (i) *Each entry in $B_{in}(v^\circ)$ corresponds to a path from another vertex in G° to v° ;*
- (ii) *Each entry in $B_{out}(v^\circ)$ corresponds to a path from v° to another vertex in G° ;*
- (iii) *For any path P between any two vertices $s^\circ, t^\circ \in V^\circ$ (P may contain vertices in $V \setminus V^\circ$), the COLA index contains both an out-label in $B_{out}(s^\circ)$ with cost c_o and length ℓ_o and an in-label in $B_{in}(t^\circ)$ with cost c_i and length ℓ_i such that $c_o + c_i \leq c(P)$ and $\ell_o + \ell_i \leq \alpha \cdot \ell(P)$.*

□

Condition 3 in the above definition indicates that for any path P connecting two vertices s° and t° in the overlay graph, we can derive another path P' by concatenating two paths P_o and P_i from the out-label set of s° and in-label set of t° respectively, such that P' α -dominates P . Therefore, according to the definition of α -CSP (Definition 5.1) and α -dominance (Definition 5.2), we can obtain an α -CSP result between s° and t° by joining the paths from their label sets, without searching for the result from scratch.

Example 5.3. Continuing the example in Example 5.2 with $\alpha = 1.2$, let $P_1^\circ = \{(v_2, v_3)\}$, $P_2^\circ = \{(v_3, v_5)\}$ and $P_3^\circ = \{(v_2, v_5)\}$. Meanwhile, let P'_1, P'_2 and P'_3 be three trivial paths from v_2 to v_2 , v_3 to v_3 , and v_5 to v_5 , respectively, with zero cost and length. Then, one possible *COLA* index include the following labels: $B_{out}(v_2) = \{P_1^\circ, P'_1\}$, $B_{in}(v_2) = \{P'_1\}$, $B_{out}(v_3) = \{P'_2\}$, $B_{in}(v_3) = \{P'_2\}$, $B_{out}(v_5) = \{P'_3\}$ and $B_{in}(v_5) = \{P_2^\circ, P_3^\circ, P'_3\}$. Clearly, these labels satisfy the first two conditions of Definition 5.4. To show that the index also satisfies the third condition, observe that there are in total 5 paths in G connecting the three nodes in G° : $P_1 = \{(v_2, v_5)\}$, $P_2 = \{(v_2, v_4), (v_4, v_5)\}$, $P_3 = \{(v_2, v_3), (v_3, v_5)\}$, $P_4 = \{(v_2, v_3)\}$ and $P_5 = \{(v_3, v_5)\}$. Clearly, each of P_1, P_3, P_4 and P_5 can be represented by combining a pair of out- and in- labels above; P_2 is α -dominated by P_1 as explained in Example 5.2.

Algorithm 10: COLA

input : $s, t, \theta, \alpha, G, G^\circ, B_{out}(s)$, and $B_{in}(t)$
output: A path for the α -CSP query with the origin s , the destination t , and cost threshold θ on G

- 1 Initialize both $B_{out}(s)$ and $B_{in}(t)$ to empty;
- 2 Perform a Dijkstra search from s within its partition G_s to obtain the set of skyline paths $L(v^o)$ from s to each of the boundary vertex v^o in G_s ;
- 3 **for each** boundary vertex $v^o \in G_s \cap G^\circ$ **do**
- 4 Join $L(v^o)$ and $B_{out}(v^o)$; // optimized in Section 5.2.4
- 5 **for each** joined path P from s to $w^o \in G^\circ$ **do**
- 6 **if** $c(P) > \theta$ or P is α -dominated by a path in $B_{out}(s)$ to w^o **then**
- 7 Discard P ;
- 8 **else**
- 9 Add P to $B_{out}(s)$;
- 10 Delete all paths in $B_{out}(s)$ from s to w^o that are α -dominated by P ;
- 11 Compute $B_{in}(t)$ similarly as Lines 2-9 (see Section 5.2.3);
- 12 Join $B_{out}(s)$ and $B_{in}(t)$; // optimized in Section 5.2.4
- 13 **return** the α -SCPs from the above join result;

One may wonder why we need both an in-label set and an out-label set, instead of just one of them. For example, given a pair of origin and destination vertices s° and t° , if the out-label set $B_{out}(s^\circ)$ of s° contains a path that ends at t° and satisfies the cost constraint, we could simply return this path as the α -CSP result, without checking the in-labels of t° . The problem with having only out- (or in-) labels is that we must store for each vertex the complete set of labels containing skyline paths to (or from) every other vertex in G° , leading to a prohibitively large index size. In contrast, by using both in-labels and out-labels, each label set only contains path to (or from) a selected subset of vertices, leading to a significantly reduced index size. This is akin to database normalization, where storing two separate base tables consumes less space than their join results.

There are various ways to build the *COLA* index, and we elaborate our method in Section 5.3.3. Here we mention one important detail in our implementation that is critical for reducing index size. Note that this implementation technique does not affect the *COLA* framework or the query processing algorithm. Specifically, we apply a standard technique in the literature of *Hierarchical Hub Labelling* [6, 8, 23, 30, 60, 98, 100, 101] for reachability and conventional shortest paths, which introduces a ranking function r of

⁴Note that, similar to *Sky-Dijk* and *CP-Dijk*, it suffices to store the important path parameters such as its length, cost and last two vertices, instead of the entire path. These details will be clarified in Section 5.3.3; for now, we assume that each entry in a label set is a path for simplicity.

all vertices in G° , whose values reflect the relative importance of the vertices. Then, for each $v^\circ \in G^\circ$, we require that (i) each entry in $B_{in}(v^\circ)$ is a path P_i originating at a vertex that has the highest rank among all vertices in P_i , and symmetrically, and (ii) each entry in $B_{out}(v^\circ)$ is a path P_o ending at a vertex that has the highest rank among all vertices in P_o . Similar to the situation in *HHL* [6, 8, 23, 30, 60, 98, 100, 101], the choice of the ordering plays an important role for the effectiveness of the index. As we have extensively studied node ordering in Chapter 3 and Chapter 4, we follow a similar spirit of AC-Greedy algorithm (ref. Section 3.2), and present the details of our ordering method in Section 5.3.4.

5.2.3 Query Processing

This subsection clarifies the processing of an α -CSP query with a pair of origin and destination vertices $s, t \in G$ and a cost constraint θ , using the overlay graph and the *COLA* index described in previous subsections. Note that if both s and t belong to the overlay graph G° , we can simply join $B_{out}(s)$ and $B_{in}(t)$, and select the α -CSP result by concatenating a path from $B_{out}(s)$ and another from $B_{in}(t)$, according to Condition 3 in Definition 5.4. However, either s or t may not appear in the overlay graph, where the α -CSP queries with s and t cannot be answered purely by the *COLA* index which is built on the overlay graph. Note that we could build the *COLA* index on the original graph G instead of the overlay graph G° . Nevertheless, doing so may lead to a prohibitively large index size, since G is far larger than G° .

The main idea of *COLA* query processing is to build $B_{out}(s)$ and $B_{in}(t)$ during query time, using the *COLA* index as well as subgraphs $G_s, G_t \in \mathcal{T}$ containing s and t , respectively. In particular, $B_{out}(s)$ and $B_{in}(t)$ must satisfy that the α -CSP result can be obtained by concatenating a path from $B_{out}(s)$ and another from $B_{in}(t)$. Formally, for any path P between s and t , there must exist paths P_o and P_i in $B_{out}(s)$ and $B_{in}(t)$ respectively, such that the concatenation of P_o and P_i α -dominates P . Given this property, the α -CSP result can be obtained by joining $B_{out}(s)$ and $B_{in}(t)$ similarly as the case when s and t are boundary vertices.

The main challenge thus lies in the computation of $B_{out}(s)$ and $B_{in}(t)$. We first focus on the former, initialized to empty. Let $G_s \in \mathcal{T}$ be the sub-graph containing s . We perform a Dijkstra-like search from vertex s to every boundary vertex of G_s . This can be done, for example, using a slightly modified version (i.e., with multiple destinations) of the *CP-Dijk* algorithm described in Section 2.4.1. In our implementation, we use a novel algorithm α -*Dijk*, detailed in Section 5.3.1, which is significantly more efficient than *CP-Dijk*.

After we finish the Dijkstra search, we extract the set of skyline paths (c.f. Section 4.1) $L(v^\circ)$ for each boundary vertex $v^\circ \in G_s \cap G^\circ$. Then, we join $L(v^\circ)$ with $B_{out}(v^\circ)$ and add

the results to $B_{out}(s)$. Specifically, for each path P_1 in $L(v^\circ)$ from s to v° , and each path P_2 in $B_{out}(v^\circ)$ from v° to another boundary vertex (say, $w^\circ \in G^\circ$), we concatenate P_1 and P_2 into a path P from s to w° , and insert P to $B_{out}(s)$ if the latter does not contain a path that α -dominates P . After that, we purge from $B_{out}(s)$ all paths from s to w° that are α -dominated by P . The computation for $B_{in}(t)$ is symmetric and omitted for brevity. Algorithm 10 summarizes the *COLA* query processing algorithm.

Example 5.4. Consider the overlay graph in Example 5.2, and the *COLA* index $\mathcal{L}$ in Example 5.3 with $\alpha = 1.2$. Given an α -CSP query from v_1 to v_5 with cost threshold $\theta = 7$, the query processing first identifies if v_1 and v_5 are boundary vertices.

As v_5 is a boundary vertex, $B_{in}(v_5) = \{P_2^\circ, P_3^\circ, P_3'\}$ can be directly obtained. However, as v_1 is not a boundary vertex, its out-label $B_{out}(v_1)$ needs to be retrieved on the fly. A Dijkstra-like search is initiated from v_1 , and the set of skyline paths from v_1 to the boundary nodes in v_1 's subgraph is extracted. In particular, it retrieves the skyline set from v_1 to v_2 , which contains only $P_1 = \{(v_1, v_2)\}$. Then, it join the skyline set with $B_{out}(v_2)$, and add the joined paths into $B_{out}(v_1)$ if they are not α -dominated by any path in $B_{out}(v_1)$. Afterwards, $B_{out}(v_1) = \{P_1, P_1 \cdot P_1^\circ\}$. Similarly, we retrieve the skyline set from v_1 to v_3 , and joined paths in the skyline set with $B_{out}(v_3)$. The paths are $P_2 = \{(v_1, v_3)\}$ and $P_3 = \{(v_1, v_2), (v_2, v_3)\}$. Note that P_3 is α -dominated by $P_1 \cdot P_1^\circ$, and P_3 is not added into $B_{out}(v_1)$, ending up with $B_{out}(v_1) = \{P_1, P_1 \cdot P_1^\circ, P_2\}$. By joining $B_{out}(v_1)$ and $B_{in}(v_5)$, a skyline set for all paths from v_1 to v_5 is retrieved. Finally, It inspects the results, unfolds edges in G° if necessary, and returns path $P = \{(v_1, v_2), (v_2, v_3), (v_3, v_5)\}$. $\square$

The query processing algorithm described so far contains several nested-loop join operations, which can be rather expensive for large label / skyline sets. Next we present effective optimizations that reduce the cost of such joins.

5.2.4 Optimizations

First we optimize the join between $B_{out}(s)$ and $B_{in}(t)$, which produces the α -CSP result. This is based on the following observation.

Observation 5.1. Let P_o and P_i be two arbitrary paths in $B_{out}(S)$ and $B_{in}(t)$ respectively that can be joined, i.e., P_o ends at the starting vertex of P_i . We have the following:

- (i) If $c(P_o) + c(P_i) > \theta$, then joining P_o (resp. P_i) with any path in $B_{in}(t)$ (resp. $B_{out}(s)$) with cost higher than P_i (resp. P_o) cannot lead to an α -CSP result;
- (ii) If $c(P_o) + c(P_i) \leq \theta$, then joining P_o (resp. P_i) with any path in $B_{in}(t)$ (resp. $B_{out}(s)$) with length longer than P_i (resp. P_o) cannot lead to an α -CSP result.

Algorithm 11: LabelJoin

```

input :  $\theta, \alpha, B_{out}(s)$ , and  $B_{in}(t)$ 
output: An  $\alpha$ -CSP  $P^*$  with a cost not larger than  $\theta$  from  $s$  to  $t$ 

1 Sort paths in  $B_{out}(s)$  firstly in ascending order of end vertex ID, and secondly in
  ascending order of cost (which is also descending order of length);
2 Sort paths in  $B_{in}(t)$  firstly in ascending order of end vertex ID, and secondly in
  ascending order of length (which is also descending order of cost);
3 Initialize  $\alpha$ -CSP result  $P^*$  to empty;
4 repeat
5   Scan both  $B_{out}(s)$  and  $B_{in}(t)$  simultaneously, until reaching a matching pair
      $P_o \in B_{out}(s)$  and  $P_i \in B_{in}(t)$ ;
6   while  $P_o$  matches  $P_i$ , i.e.,  $P_o$  ends at the origin vertex of  $P_i$  do
7     if  $c(P_o) + c(P_i) > \theta$  then
8       Set  $P_i$  to the next entry in  $B_{in}(t)$ ;
9     else
10      Update  $P^*$  to the concatenation of  $P_o$  and  $P_i$  if the combination of  $P_o$  and  $P_i$ 
        has length smaller than  $P^*$ ;
11      Set  $P_o$  to next entry in  $B_{out}(s)$ ;
12 until reaching the end of either  $B_{out}(s)$  or  $B_{in}(t)$ ;
13 return  $P^*$ ;
    
```

□

Based on the above observation, we accelerate the join between $B_{out}(s)$ and $B_{in}(t)$ through a careful ordering of the labels. Specifically, *COLA* sorts paths in $B_{out}(s)$ by the ID of their end vertices, breaking ties by total cost (in ascending order). Note that $B_{out}(s)$ is a skyline set, meaning that paths with the same end vertex are also automatically sorted in descending order of their lengths (otherwise one path would dominate another). Similarly, *COLA* sorts paths in $B_{in}(t)$ firstly by the ID of their origin vertices, and secondly in ascending of their lengths / descending order of their costs. With such ordering, we propose a novel algorithm *LabelJoin* (as shown in Algorithm 11), which joins $B_{out}(s)$ and $B_{in}(t)$ with a *linear scan of each set*.

Lemma 5.2. *Algorithm 11 correctly computes the α -CSP result from $B_{out}(s)$ and $B_{in}(t)$ in linear time.* □

Example 5.5. Consider an α -CSP query from s to t with cost threshold $\theta = 13$ and $\alpha = 1.2$ under *COLA*. Assume that $B_{out}(s) = \{P_1, P_2\}$ where both P_1 and P_2 ends at w , with $c(P_1) = 4$, $\ell(P_1) = 7$, $c(P_2) = 7$, and $\ell(P_2) = 4$; $B_{in}(t) = \{P_3, P_4, P_5\}$ where all P_3 , P_4 , and P_5 starts at w , with $c(P_3) = 7$, $\ell(P_3) = 5$, $c(P_4) = 6$, $\ell(P_4) = 6$, $c(P_5) = 5$, and

$\ell(P_5) = 7$. Clearly, $B_{out}(s)$ is sorted by ascending order of the cost, and $B_{in}(t)$ is sorted in descending order of the cost.

It can be verified that the skyline set of the paths from s to t is $|B_{out}(s) \cdot B_{in}(t)| = 6$. By Algorithm 11, we first check P_1 and P_3 , to see if the concatenated path $P' = P_1 \cdot P_3$ satisfies the cost constraint. As $c(P') = 11 \leq \theta = 12$ and P^* is empty, the currently found shortest path under const constraint is hence updated to $P^* = P'$. Afterwards, the *LabelJoin* algorithm will proceed to the next path P_2 in $B_{out}(s)$, and concatenate it with P_3 . Here, P_1 is not further concatenated with P_4 and P_5 since based on Observation 5.1.2, i.e., by concatenating P_1 with P_4 or P_5 will produce a path with a larger length than P' .

Consider the concatenated path $P'_1 = P_2 \cdot P_3$. Note that the cost of P'_1 exceeds the cost threshold, and based on Algorithm 11, $B_{in}(t)$ will proceed to the next path, i.e., P_4 . Consider $P'_2 = P_2 \cdot P_4$, the cost of P'_2 is 13, which satisfies the cost constraint, and the length of P'_2 is 10, smaller than P^* . Hence P^* is updated to P'_2 . Meanwhile, *LabelJoin* further try to proceed the next path in $B_{out}(s)$, and find that there is no more path, which stops the label checking, and returns P^* . $\square$

Next, we focus on the join between a skyline set $\mathcal{L}(v^\circ)$ and a label set $B_{out}(v^\circ)$ in the *COLA* index (Line 4 of Algorithm 10). The case for joining $L(v^\circ)$ with $B_{in}(v^\circ)$ is symmetric and omitted for brevity. The basic idea for the optimization is not to compute the complete join results, but only those results that can possibly lead to an α -CSP result. Specifically, we avoid generating certain join results based on the following observation.

Observation 5.2. Let P be a path from s to w° from the join result of $L(v^\circ)$ and $B_{out}(v^\circ)$. P cannot possibly lead to an α -CSP result, if any of the following is true.

- (i) w° cannot possibly reach t on the input graph G ;
- (ii) The minimum cost for any path from w° to t exceeds $\theta - c(P)$;
- (iii) There exist a path P' from s to t satisfying the cost constraint, such that the minimum length of any path from w° to t exceeds $\ell(P')/\alpha - \ell(P)$.

$\square$

According to the above observation, before performing any join operation, we first select a set of end vertices W for the join results that can possibly lead to an α -CSP result, which is incrementally pruned using Observation 5.2. Then, we filter the *COLA* index, and use only the labels that reach a vertex in W . Algorithm 12 shows the algorithm for computing W . Filtering the *COLA* index before joining it with the skyline sets can be understood as using a semi-join to improve join performance in database systems. Note that the *COLA* index was constructed before we know the query parameters, hence, it usually contains large number of labels not needed for answering the query at hand.

Algorithm 12 shows the pseudo-code for computing W . W is initialized with all boundary vertices reachable from both s and t . Then, those vertices are pruned that cannot lead to a path from s to t within cost threshold θ . After that, the algorithm computes an upper bound for the length of a path from s to t , and uses it to prune more vertices in W based on the third condition in Observation 5.2.

5.3 COLA implementation

This section details the implementation of *COLA*. Most importantly, *COLA* includes an algorithm α -*Dijk* which has three main uses: (i) for performing the intra-partition search during query processing, (ii) for building the *COLA* index, and (iii) standalone as an index-free algorithm for α -CSP. In the following, Section 5.3.1 presents α -*Dijk*. Section 5.3.2 clarifies the algorithm for building the overlay graph. Section 5.3.3 describes the construction of the *COLA* index. Section 5.3.4 explains how we determine the vertex order, which affects the efficiency of *COLA*.

5.3.1 α -Dijk

Similar to *CP-Dijk* (refer to Section 2.4), α -*Dijk* is based on *Sky-Dijk* with enhanced pruning based on the relaxed α -dominance definition. On the other hand, the pruning strategy of α -*Dijk* is radically different from that in *CP-Dijk*. The intuition is as follows. Imagine that we have a total “budget” for pruning along a path; the higher the budget allocated to a node, the stronger pruning power it is allowed to apply to reduce the size of its set of associated paths. *CP-Dijk* simply distributes this budget equally to each node on the path. Since there can be a large number of nodes on a path, each of them only receives little pruning power, leading to ineffective pruning everywhere. In contrast, α -*Dijk* concentrates the pruning power to nodes that are associated with a large number of paths, and does not allocate pruning power at all to nodes that have relatively few paths. In a real road network, there are usually a small number of landmark vertices that appear frequently in CSPs, which tend to accumulate a large number of paths. As a result, concentrating the pruning power to such nodes leads to effective reduction of the total number of paths to be examined, and, thus, accelerates query processing.

Formally, α -*Dijk* applies α_i -dominance to prune entries in $L(v_i)$, where the values of $\alpha_i \in [1, \alpha]$ depends on v_i . In other words, the amount of pruning applied in each $L(v_i)$ is *adaptive*, rather than fixed as in *CP-Dijk*. Let P_{opt} be the CSP from an origin vertex s to a destination vertex t under a cost threshold θ . Let k be the number of edges in P_{opt} , ℓ_i ($i \in [1, k]$) be the length of the i -th edge in P_{opt} , and v_j be the vertex that the i -th

edge points to. By adopting α_i -dominance into the construction of $L(v_i)$, α -Dijk always returns a path P from s to t , such that $c(P) \leq \theta$ and

$$\begin{aligned} \ell(P) &\leq \ell_1 \prod_{i=1}^k \alpha_i + \ell_2 \prod_{i=2}^k \alpha_i + \ell_3 \prod_{i=3}^k \alpha_i + \dots + \ell_k \prod_{i=k}^k \alpha_i \\ &= \sum_{j=1}^k \left(\ell_j \cdot \prod_{i=j}^k \alpha_i \right). \end{aligned}$$

Therefore, P can be an α -CSP if each α_j is carefully selected to ensure that

$$\sum_{j=1}^k \left(\ell_j \cdot \prod_{i=j}^k \alpha_i \right) \leq \sum_{j=1}^k (\ell_j \cdot \alpha). \quad (5.1)$$

A straightforward approach to enforce Inequality (5.1) is to set $\alpha_i = \alpha^{\frac{1}{n}}$ (as in the case of CP -Dijk), but it leads to ineffective pruning in each $L(v_i)$, as we discuss in Section 2.4.1. Instead, we propose to set $\alpha_i = 1$ for some vertices v_i but $\alpha_j > \alpha^{\frac{1}{n}}$ for the others v_j . That is, we give up pruning in some entry lists, in exchange for more effective optimization in the other entry lists. Intuitively, this allows us to focus our pruning effort on the entry lists that are important for the given α -CSP query to achieve higher efficiency. To implement this idea, however, we have to address two crucial issues. First, on which entry lists $L(v_i)$ should we have $\alpha_i > \alpha^{\frac{1}{n}}$? Second, given that $\ell_1, \ell_2, \dots, \ell_k$ are unknown in advance, how can we appropriately select α_i without invalidating Equation (5.1)? In the following, we clarify these issues by elaborating the details of our α -Dijk algorithm.

Detailed Algorithm. Algorithm 13 shows the pseudo-code of α -Dijk. Given an α -CSP query with an origin vertex s , a destination vertex t , and a cost threshold θ , α -Dijk first invokes the vanilla Dijkstra's algorithm to compute, for each vertex v , the minimum-length path $P_\ell(v)$ from s to v (Line 1). Let $\ell^\perp$ and $c^\top$ be the length and cost of $P_\ell(v)$, respectively. Observe that, if $c^\top(t) \leq \theta$, then $P_\ell(t)$ is the CSP from s to t under cost threshold θ . In that case, α -Dijk directly returns $P_\ell(t)$ as the query answer (Line 3-4). Otherwise, α -Dijk proceeds to apply the vanilla Dijkstra's algorithm once again, and derives the minimum-cost path $P_c(v)$ from s to each vertex v (Line 5). Then, it records the length of $P_c(v)$ as $\ell^\top(v)$.

As a next step, α -Dijk creates a min-heap H similarly as in *Sky-Dijk* and *CP-Dijk*, except that each heap entry in α -Dijk stores more information to allow the adaptive allocation of pruning budget. Specifically, each entry $\rho = \langle v, c, \ell, \tau, v_{pred} \rangle$ in H stores (i) information about a path P from s to a vertex v with cost c , length ℓ , and predecessor (i.e., the vertex in P before its v) v_{pred} , and also (ii) an additional *surrogate value* $\tau \in [\ell/\alpha, \ell]$ which, as we clarify shortly, is used to decide whether and how ρ can prune other entries

may observe that ℓ and ℓ' are respectively available from ρ and ρ' , and wonder why α -Dijk does not check $\ell' \leq \alpha\ell$, but uses τ and τ' as surrogates for ℓ and ℓ' , respectively. The reason is that τ and τ' provide a means for us to control the amount of pruning to be performed in $L(v)$. In particular, if we set $\tau' = \ell'/\alpha$, then $\tau = \ell$, then ρ' can prune ρ iff ρ represents a path that is α -dominated by the path associated with ρ' ; in contrast, if we set $\tau' > \ell'/\alpha$ or $\tau < \ell$, then the “pruning power” of ρ' is *weakened*, in the sense that it may not prune ρ even if the path associated with ρ is α -dominated by the one associated with ρ' . As we mention previously, such deliberate weakening of pruning power in $L(v)$ could allow us to apply more aggressive pruning in the other entry lists, which helps reduce query overhead.

Here, a natural question arises: how does α -Dijk decide the surrogate value of an entry? This question is answered in Lines 16-18 of Algorithm 13. In particular, if ρ is not pruned by ρ' , then α -Dijk would insert ρ into $L(v)$; but before the insertion, α -Dijk heuristically decides the surrogate value τ of ρ based on the number of entries in $L(v)$. If $|L(v)| > \log_{\alpha} \frac{\ell^{\top}(v)}{\ell^{\perp}(v)}$, then α -Dijk sets $\tau = \ell/\alpha$, which grants ρ the maximum pruning power; otherwise, α -Dijk sets $\tau = \ell$, minimizing the pruning capability of ρ . To explain, observe that $\log_{\alpha} \frac{\ell^{\top}(v)}{\ell^{\perp}(v)}$ is an upper bound of $|L(v)|$ when we apply α -dominance in the construction of $L(v)$. Intuitively, if the size of the current $L(v)$ exceeds this upper bound, then applying aggressive pruning in $L(v)$ is likely to reduce $|L(v)|$ and help improve query efficiency. On the other hand, if $|L(v)| \leq \log_{\alpha} \frac{\ell^{\top}(v)}{\ell^{\perp}(v)}$, then pruning entries in $L(v)$ tends to be ineffective, in which case it is more preferable to omit pruning in $L(v)$.

After ρ is inserted into $L(v)$, α -Dijk inspects each outgoing edge e of v (Line 18). If $c + c(e) \leq \theta$, then α -Dijk pushes an entry $\langle v^*, c + c(e), \ell + \ell(e), \tau + \tau(e), v \rangle$ into H , where v^* is the vertex that e points to, and $\tau(e)$ is defined as $\ell(e)$ (Lines 19-21). This is to capture the path that goes from s to v^* , with v being the predecessor of v^* . The above iterative process terminates when H becomes empty. In that case, α -Dijk retrieves the entry in $L(t)$ with the smallest length, and then reconstructs the path P represented by the entry. Finally, P is returned as the answer to the α -CSP query.

Theoretical Analysis. We first present a theorem that shows the correctness of α -Dijk.

Theorem 5.1. *Suppose that $c^{\top}(t) \leq \theta$ does not hold in Line 3 of Algorithm 13. For each vertex v , let $L(v)$ be the entry list of v when α -Dijk terminates. Then, for any path P' from s to v with a cost smaller than θ , there exists an entry ρ' in $L(v)$ with a cost at most $c(P')$ and a length at most $\alpha \cdot \ell(P')$. $\square$*

Next, we discuss the time complexity of our α -Dijk algorithm. For each vertex v , the number of stored entries in $L(v)$ is bounded by $\ell_{\max} \cdot n$. When adding labels for outgoing edges of each vertex v , it incurs at most $d_v \cdot \ell_{\max} \cdot n$ labels, where d_v is the out-degree of v . To sum up with, there are at most $\ell_{\max} \cdot n \cdot m$ labels added in the whole procedure. To insert $\ell_{\max} \cdot n \cdot m$ entries into the heap, it requires $\mathcal{O}(\log(\ell_{\max} n))$ time for each operation, ending up with $\mathcal{O}(\ell_{\max} \cdot m \cdot n \cdot \log(\ell_{\max} \cdot n))$ time complexity.

Algorithm 13: α -Dijk

input : An α -CSP query on G with an origin s , a destination t , and a cost threshold θ
output: An answer P to the α -CSP query

- 1 Apply the Dijkstra's algorithm once to compute the minimum-length path $P_\ell(v)$ from s to each vertex v ;
- 2 Let $\ell^\perp(v)$ and $c^\top(v)$ be the length and cost of $P_\ell(v)$, respectively;
- 3 **if** $c^\top(t) \leq \theta$ **then**
- 4 **return** $P_\ell(v)$
- 5 Apply the Dijkstra's algorithm once to compute the minimum-cost path $P_c(v)$ from s to each vertex v ;
- 6 Let $\ell^\top(v)$ be the length of $P_c(v)$;
- 7 Creates a min-heap H that (i) stores entries in the form of $\langle v, c, \ell, \tau, v_{pred} \rangle$, and (ii) sorts the entries in ascending order of their c values, with ties broken by their ℓ values;
- 8 Create an entry list $L(v)$ for each vertex v in G ;
- 9 Insert an entry $\langle s, 0, 0, 0, null \rangle$ into H ;
- 10 **while** H is not empty **do**
- 11 Pop the top entry $\rho = \langle v, c, \ell, \tau, v_{pred} \rangle$ in H ;
- 12 Let $\langle v, c', \ell', \tau', * \rangle$ be the entry in $L(v)$ with the largest cost;
- 13 **if** $\tau' \leq \tau$ **then**
- 14 **continue**;
- 15 **if** $|L(v)| > \log_\alpha \frac{\ell^\top(v)}{\ell^\perp(v)}$ **then**
- 16 Modify ρ to set $\tau = \ell/\alpha$;
- 17 Insert ρ into $L(v)$;
- 18 **for each** outgoing edge e of v **do**
- 19 Let v^* be vertex that e points to;
- 20 **if** $c + c(e) \leq \theta$ **then**
- 21 Push $\langle v^*, c + c(e), \ell + \ell(e), \tau + \tau(e), v \rangle$ into H ;
- 22 Identify the entry in $L(t)$ with the smallest length, and then reconstruct the path P represented by the entry;
- 23 **return** P ;

5.3.2 Overlay Graph Construction

Given a partition $\mathcal{T} = \{G_1, G_2, \dots, G_k\}$ of G and the set V° of boundary vertices defined by $\mathcal{T}$, we construct an overlay graph $G^\circ = (V^\circ, E^\circ)$ by creating a number of edges between any two vertices $v, v' \in V^\circ$ that appear in the same subgraph in $\mathcal{T}$. These edges are computed using a slightly modified version of α -Dijk that incorporates the following changes to Algorithm 13:

- (i) The input to the algorithm includes only a subgraph $G_i \in \mathcal{T}$ and an origin vertex s in G_i .

- (ii) Lines 3, 4, 21, and 23 of Algorithm 13 are omitted, due to changes in the algorithm's input. (Note that Line 22 remains.)
- (iii) In Line 24, the algorithm outputs the entry list $L(v)$ constructed for each vertex v in G_i .

We refer to the above algorithm as the *single-source α -Dijk*. In a nutshell, the algorithm takes as input an origin vertex s in G_i , and returns an entry list $L(v)$ for each vertex v in G_i , such that each entry $\rho = \langle v, c, \ell, \tau, v_{pred} \rangle$ in $L(v)$ represents a path in G_i that goes from s to v , with cost c , length ℓ , and v_{pred} being the predecessor of v in the path. For convenience, we use $c(\rho)$, $\ell(\rho)$, and $\tau(\rho)$ to denote the cost, length, and surrogate value recorded in ρ , respectively. The following lemma shows the important properties of the single-source α -Dijk.

Lemma 5.3. *Let s and v be two vertices in a graph G_i , and $L(v)$ be the entry list of v constructed by the single-source α -Dijk given G_i and s as input. Let $\mathcal{P}$ be the set of all paths from s to v in G_i . Then, for any entry $\rho \in L(v)$, there exists a path $P \in \mathcal{P}$ with $c(P) = c(\rho)$, $\ell(P) = \ell(\rho)$, and $\tau(\rho) \in [\frac{1}{\alpha}\ell(P), \ell(P)]$. Furthermore, for any path $P' \in \mathcal{P}$, there exists an entry $\rho \in L(v)$ with $c(\rho) \leq c(P')$ and $\tau(\rho) \leq \ell(P')$. $\square$*

Algorithm 14 shows the pseudo-code of our algorithm for constructing an overlay graph $G^\circ = (V^\circ, E^\circ)$ based on a partition $\mathcal{T} = \{G_1, G_2, \dots, G_k\}$ of G . The algorithm first initializes V° as the set of all boundary vertices defined by $\mathcal{T}$, and sets $E^\circ = \emptyset$ (Line 1). After that, it processes each subgraph G_i in turn (Lines 2-13). In particular, for each boundary vertex v° in G_i , it invokes the single-source α -Dijk to derive an entry list $L(v)$ for each vertex v in G_i (Line 4). If v is also a boundary vertex in G_i , then some of the entries in $L(v)$ may be converted into edges between v° and v in E° (Lines 5-13). The conversion process is as follows. First, the algorithm creates a set $S = \emptyset$ for storing entries (Line 6). After that, it inspects the entries in $L(v)$ in ascending order of their costs, and compares each entry ρ with the last entry ρ' inserted into S . If ρ' does not exist (i.e., S is empty) or $c(\rho') \leq c(\rho)$ or $\ell(\rho') \leq \alpha \cdot \tau(\rho)$, then the algorithm inserts ρ into S as an entry to be converted into an edge in E° (Lines 9-10). Otherwise, the path represented by ρ must be α -dominated by ρ' , and hence, the algorithm omits ρ , and modifies ρ' to set $\tau(\rho') = \min\{\tau(\rho'), \tau(\rho)\}$ (Lines 11-12). The change of $\tau(\rho')$ is intended to ensure certain properties in the resulting overlay graph G° , which will be clarified in Corollary 5.1. After all entries in $L(v)$ are examined, the algorithm retrieves each entry ρ that has been inserted in S , and converts it into an edge $e^\circ \in E^\circ$ with $c(e^\circ) = c(\rho)$ and $\ell(e^\circ) = \ell(\rho)$. In addition, we define the surrogate value of e° as $\tau(e^\circ) = \tau(\rho)$. (This surrogate value is useful in our COLA index construction to guarantee its correctness, as we will clarify in Section 5.3.3.) Once all subgraphs in $\mathcal{T}$ are processed, the algorithm terminates and returns $G^\circ = (V^\circ, E^\circ)$.

The following corollary follows from Lemma 5.3 and establishes a few properties of the overlay graph constructed from Algorithm 14.

Algorithm 14: Overlay Graph Construction

input : A partition $\mathcal{T} = \{G_1, G_2, \dots, G_k\}$ of G
output: An overlay graph $G^\circ = (V^\circ, E^\circ)$ of G

```

1  Let  $E^\circ = \emptyset$ , and  $V^\circ$  be the set of boundary vertices defined by  $\mathcal{T}$ ;
2  for each subgraph  $G_i \in \mathcal{T}$  do
3      for each boundary vertex  $v^\circ$  in  $G_i$  do
4          Feed  $v^\circ$  and  $G_i$  as input to the single-source  $\alpha$ -Dijk, which outputs an entry list
               $L(v)$  for each vertex  $v$  in  $G_i$ ;
5          for each boundary vertex  $v$  in  $G_i$  do
6              Let  $S = \emptyset$ ;
7              for each entry  $\rho$  in  $L(v)$  in ascending order of  $c(\rho)$  do
8                  Let  $\rho'$  be the last entry inserted into  $S$ ;
9                  if  $S$  is empty or  $c(\rho') > c(\rho)$  or  $\ell(\rho') > \alpha \cdot \tau(\rho)$  then
10                     Insert  $\rho$  into  $S$ ;
11                 else
12                      $\tau(\rho') = \min\{\tau(\rho'), \tau(\rho)\}$ ;
13                 for each entry  $\rho$  in  $S$  do
14                     Insert an edge  $e^\circ$  into  $E^\circ$ , such that  $e^\circ$  starts from  $v^\circ$  and ends at  $v$ , with
                         $c(e^\circ) = c(\rho)$ ,  $\ell(e^\circ) = \ell(\rho)$ , and  $\tau(e^\circ) = \tau(\rho)$ ;
15 return  $G^\circ = (V^\circ, E^\circ)$ ;
    
```

Corollary 5.1. Let $G^\circ = (V^\circ, E^\circ)$ be the overlay graph returned by Algorithm 14 given a partition $\mathcal{T}$ of G . Let E^* be the set of all edges in E° that are constructed from a subgraph $G_i \in \mathcal{T}$, such that they all start from a vertex v_1 and end at a vertex v_2 with $v_1, v_2 \in V^\circ$. Let $\mathcal{P}$ be the set of all paths from v_1 to v_2 in G_i . For any $e^\circ \in E^*$, there exist a path $P \in \mathcal{P}$ with $c(P) = c(e^\circ)$, $\ell(P) = \ell(e^\circ)$, and $\tau(e^\circ) \in [\frac{1}{\alpha}\ell(P), \ell(P)]$. Furthermore, for any path $P' \in \mathcal{P}$, there exists an edge $e' \in E^\circ$ with $c(e') \leq c(P')$ and $\tau(e') \leq \ell(P')$. $\square$

5.3.3 Labeling Index Construction

This subsection details the construction of the *COLA* index. Each label entry in $B_{out}(v)$ contains a tuple $\langle w, c, \ell, v_{succ} \rangle$, mapped from a path P from v to w , with cost c and length ℓ . Meanwhile, v_{succ} is the successor of u in the P . Each label entry in $B_{in}(v)$ contains a tuple $\langle w, c, \ell, v_{succ} \rangle$, mapped from a path P' from w to v , with cost c , length ℓ , and v_{succ} being the predecessor of v in P' . Recall from Section 5.2.2 that in our implementation of the *COLA* index, there is a ranking function r that introduces a total order of the vertices in G° . Accordingly, we define the rank of a path as follows.

Definition 5.5 (Rank of a Path). *Let P be an arbitrary path in G° . We define the rank of a path $r(P)$ to be the highest rank of the vertices that appear in P . We further define $\mathcal{P}_i$ to be the set of paths P whose rank is no more than i , i.e., $r(P) \leq i$. $\square$*

Note that when $i = n$, $\mathcal{P}_i$ denotes the set of all paths in G° . Our index construction algorithm runs in an iterative manner. After finishing i iterations, the labels constructed in B_{out} and B_{in} guarantees that, for any path P from u to v ($u, v \in G^\circ$) whose rank is no more than i , there exists a label entry $\rho_u = \langle w, c_u, \ell_u, v_{succ} \rangle$ in $B_{out}(u)$ corresponding to a path P_1 , and a label entry $\rho_v = \langle w, c_v, \ell_v, v_{pred} \rangle$ in $B_{in}(v)$ concerning a path P_2 in G° such that $c_u + c_v \leq c(P)$ and $\ell_u + \ell_v \leq \alpha \cdot \ell(P)$. In other words, by concatenating P_1 and P_2 , we find a path that can α -dominate P .

Algorithm 15 shows the procedure for the index construction. It produces indices in n iterations. We explain how labels in B_{in} are computed, and B_{out} can be explained similarly. In the i -th iteration, the vertex u_i with $r(u_i) = i$ is selected. A modified version of α -Dijk is invoked to produce a set of label lists $L(v)$ for $v \in G^\circ$. The main modification is that: when deriving the lower bound of the length $\ell_{bot}(v)$ from u_i to v , it requires to use the τ value of a path instead of the length of the path. The main reason is that G° is a crippled version of G to capture the paths from u_i to v , and the minimum length path P in G might not be preserved in G° ; however, the τ value of a path denotes the lower bound of the minimum length path that it has pruned, indicating the τ value of the minimum- τ path is a lower bound of the length from u_i to v .

Meanwhile, when a label entry $\rho = \langle v, c, \ell, \tau, v_{pred} \rangle$ is popped from the heap H . A dominance checking procedure is proceeded, which checks if a path with cost c and length τ will be dominated by existing labels. In particular, it uses the labels constructed in the previous $i - 1$ iterations to do a dominance checking. Specifically, it checks if there exists a label $\rho^+ \in B_{out}(u_i)$ and a label $\rho^- \in B_{in}(v)$ such that $c(\rho^+) + c(\rho^-) \leq c$ and $\ell(\rho^+) + \ell(\rho^-) \leq \tau$. If the answer is yes, this label entry can be skipped in this iteration without affecting the query correctness as will be proved in Lemma 5.4.

When the α -Dijk's algorithm finishes, it outputs the label list into L . For each vertex v , the most simple approach is to subsequently select the label from $L(v)$ with the smallest cost and cannot be α -dominated by previous selected labels. This approach, however, cannot efficiently help path reconstruction. In particular, given an in-label entry $\rho^- = \langle w, c, \ell, v_{pred} \rangle$, and denote the path pertinent to this label as P , the path reconstruction requires the knowledge of the label pertinent to the sub-path of P that starts from v_{pred} to u_i . We denote this label as the *child* of ρ^- . This requirement, however, cannot be guaranteed by the simple selection strategy. Instead, we adopt a more advanced approach to guarantee that, for each out-label in B_{out} (resp. in-label in B_{in}), its child is always in B_{out} (resp. B_{in}).

Observe that, in the i -th iteration, if a label $\langle w, c, \ell, \tau, v_{pred} \rangle$ is stored in L , there must exist a label $\langle v_{pred}, c', \ell', \tau' v'_{pred} \rangle$ stored in L . This is due to the traversing order

of a path in α -Dijk. If a path P from u to w has been traversed, its sub-path from u to w 's predecessor must also be traversed. Note that $c' \leq c$ and $\ell' \leq \ell$. By storing the labels in descending order of the cost, with ties broken by the length, we can guarantee that each label is ordered before its child. Denote the sorted label as L_i . This procedure is proceeded as shown in Lines 8-9 of Algorithm 15. After that, we maintain a hash-table T to record the child of the added labels. Besides, we record the last pruned label concerning v , and denote it as $\rho_{prune}(v)$. Note that $\rho_{prune}(v)$ is initially set to the first label concerning v in L_i . Then in order of the sorted list L_i , we inspect whether (i) the current label $\rho = \langle w, c, \ell, \tau \rangle$ satisfies that it is a child of a previous stored label, and (ii) whether the next label ρ_p concerning w satisfies that $\ell(\rho_p) \leq \tau(\rho_{prune}) \cdot \alpha$. If both conditions are not satisfied (Line 13-15), this label can be pruned safely. Otherwise, a label $\langle u_i, c, \ell, v_{pred} \rangle$ is inserted into $B_{in}(w)$; its child is inserted into the hash-table T ; and ρ_{prune} is updated to ρ_p as shown in Lines 17-20.

When the label entries are updated, a backward version of α -Dijk is proceeded, and entries are added in to B_{out} . After that, the input graph of the next iteration is updated by removing vertex u_i and its incident edges. After n iteration, the algorithm finishes and ends up with two label sets B_{out} and B_{in} that guarantees the query correctness as shown in Lemma 5.4.

Lemma 5.4. *Given a COLA index produced by Algorithm 15, it is guaranteed that for any path P from s to t ($s, t \in G^\circ$), there exists a label entry ρ_o in $B_{out}(s)$ and a label entry ρ_t in $B_{in}(t)$ such that, $c(\rho_o) + c(\rho_t) \leq c(P)$ and $\ell(\rho_o) + \ell(\rho_t) \leq \alpha \cdot \ell(P)$. $\square$*

5.3.4 Vertex Ordering

Recall that every in-label (resp. out-label) of each vertex v in the COLA index is a skyline path P originating (reps. ending) at a vertex v' that has the highest rank among all vertices in P , i.e., the rank of v' is higher than the rank of v . In other words, the ordering of vertices would affect the total number of labels in COLA, which is similar to the situation in HHL reachability and distance indices [6, 8, 23, 30, 60, 98, 100, 101]. Recall that, computing an HHL index with minimal size is NP-hard. To produce a COLA index with small size and also overcome the computational difficulty, we present in this section a sampling based approach that shares the similar spirit to our AC-Greedy algorithm proposed in Section 3.2.4. We first define the *coverage* of v as the number of skyline paths P in G_r° that goes through v , denoted by $C(v)$. Meanwhile, we define the *influence* of v as the number of skyline paths P in G_r° that starts or ends with v , denoted by $I(v)$. Then, we define the coverage cost ratio of v as $\frac{I(v)}{C(v)}$. In each iteration, we select v with the smallest coverage cost ratio, and remove v from G° , and continue to order the rest vertices in residual graph G_r° until G_r° is empty.

Algorithm 15: *Index Construction*

input : An overlay graph G° , and a rank for all vertices in G°
output: a COLA index $\mathcal{L}$

- 1 Creates a min-heap H that (i) stores entries in the form of $\langle v, c, \ell, \tau, v_{pred} \rangle$, and (ii) sorts the entries in ascending order of their c values, with ties broken by their ℓ values;
- 2 Let $G_i^\circ = G^\circ$;
- 3 **for** $i = 1, 2, \dots, n$ **do**
- 4 Let u_i be the vertex whose rank is i ;
- 5 Invoke Algorithm 13 on G_i° except that (i) to derive $\ell^\perp(v)$, it requires to compute the path P with minimum τ value instead of minimum-length; (ii) when a label entry $\rho = \langle w, c, \ell, \tau, v_{pred} \rangle$ is popped from the heap, it invoke a dominance checking procedure to see if a path with cost c and length τ , will be dominated by existing labels with a linear label scanning similar to Algorithm 11; (iii) output the label list L ;
- 6 Let L_i be an empty list;
- 7 **for** $v \in V^\circ$ whose $L(v)$ is not empty **do**
- 8 Insert all entries in $L(v)$ into L_i ;
- 9 Sort all entries in L_i in descending order of cost, with ties broken by length;
- 10 Initiate a hash-table T to record preserved labels, and the key is of the form $\langle w, c, \ell \rangle$;
- 11 Record the last label $\rho_{prune}(v)$ pruned with α -dominance, initially it is set to the first label in L_i concerning v ;
- 12 **for** each label entry $\rho = \langle w, c, \ell, \tau, v_{pred} \rangle$ in L_i **do**
- 13 Let $\rho_p \in L_i$ be the next label concerning w after ρ ;
- 14 **if** $\langle w, c, \ell \rangle \notin T$ **and** $\ell(\rho_p) \leq \tau(\rho_{prune}(w)) \cdot \alpha$ **then**
- 15 **continue**;
- 16 **else**
- 17 Add $\rho' = \langle u_i, c, \ell, v_{pred} \rangle$ into $B_{in}(w)$;
- 18 Let P be the path pertinent to ρ' , and let e be the edge that points from v_{pred} to w ;
- 19 Insert an entry $\langle v_{pred}, c - c(e), \ell - \ell(e) \rangle$ into T ;
- 20 Update ρ_{prune} to be ρ_p ;
- 21 Repeat the above procedure with a backward modified α -Dijk and add entries into B_{out} ;
- 22 Update G_{i+1}° by removing the incident edges to u_i ;
- 23 **return** B_{in}, B_{out} ;

However, it is highly expensive to compute all skyline paths in G_r° , especially when G_r° is sufficiently large. To avoid the deficiency, we still use a sampling based approach to estimate the coverage and influence of v in G_r° . To explain, let u be a randomly sampled node, and β be a randomly sampled number in $[0, 1]$. We define $c(e) \cdot \beta + \ell(e) \cdot (1 - \beta)$ as the *weight* of an edge e . Then we sample from u two spanning tree with the minimum

dataset	$ V $	$ E $	$ V^\circ $	$ G^\circ $
New York City (NY)	0.3M	0.7M	4.8K	0.2M
Florida (FLA)	1.1M	2.7M	4.2K	0.2M
Northwest USA (NW)	1.2M	2.8M	4.2K	0.2M
Northeast USA (NE)	1.5M	3.9M	7.0K	0.7M
Great Lakes (LKS)	2.8M	6.9M	10.9K	1.8M
Western USA (W)	6.3M	15.2M	7.8K	1.2M
Central USA (CTR)	14.1M	34.3M	16.8K	6.9M
Full USA (USA)	23.9M	58.3M	17.7K	9.5M

Table 5.2: Road networks ($K=10^3$, $M=10^6$).

total weight using a forward and backward version of Dijkstra’s traversal. The rationale is that, if a path is a skyline path, then there exists a β in $[0, 1]$ such that the path has the minimal weight. Hence, we sample $2k$ trees $T_1, T_2, \dots, T_{2k}$ using the above strategy, where $2k$ is sufficiently large. Let $C_j(v)$ be the size of subtree with root v in T_j ($1 \leq j \leq 2k$). Then, we use the *coverage sum* of the sampled $2k$ trees, i.e., $\sum_{j=1}^{2k} C_j(v)$ to estimate the coverage, and the *influence sum* of the sample $2k$ trees, i.e., $\sum_{j=1}^{2k} I_j(v)$ to estimate the influence. Meanwhile, we use the score obtained through dividing the influence sum by the coverage sum to denote the estimated cost coverage ratio of a node v , denote as $\rho(v)$. Then, we order the vertex v with the smallest ρ score among all vertices in G_r° . After that, we remove all subtrees rooted at v from $T_1, T_2, \dots, T_{2k}$ and update the estimated cost coverage ratio for the rest vertices respectively.

A larger sample size $2k$ tends to provide a more accurate estimated cost coverage ratio for each vertex. However, it comes at a higher computational cost. To explain, let n_o be the number vertices in G° , then the number of nodes in a sampled tree is at most n_o . Therefore, the time complexity in computing the sampled trees by the above method is $O(k \cdot n_o^2)$. Meanwhile, the updates of coverage sum, influence sum, and ρ can be finished with $O(k \cdot n_o)$ time through a back tracking of the trees that contains the ordering vertex, and the total update cost is $O(k \cdot n_o^2)$. As a result, the preprocessing complexity is $O(kn_o^2)$. Let m_o be the number of edges in G° . To reduce the cost, we constrain the total number of edges sampled from all trees to be at most $h \cdot m_o$, where $h = 8$ following the setting in Section 4.4.2.2.

5.4 Experiments

This section experimentally evaluates *COLA* against the current state-of-the-art methods. Section 5.4.1 explains the settings of our experiments. Section 5.4.2 and 5.4.3 present evaluation results in terms of query efficiency and overhead of indexing, respectively.

5.4.1 Experimental Settings

All methods are implemented in C++, compiled with full optimization, and tested on a Linux machine with an Intel Xeon 2.6GHz CPU and 64GB RAM. We repeat each experiment 5 times and report the average results.

Datasets. We use 8 real road networks of various sizes from the 9th DIMACS Implementation Challenge [2]. Table 5.2 summarizes the properties of the datasets, where $|V|$, $|E|$, $|V^\circ|$ and $|E^\circ|$ are the cardinalities of vertices in the road network, edges in the road network, vertices in the overlay graph and edges in the overlay graph, respectively. The dataset sizes vary from small cities to the full USA road networks. Each edge in the network concerns the distance and the travel time cost from one vertex to another.

Query sets. For each dataset, we generate 8 query sets Q1-Q8, each containing 100 queries as follows. For each query set, we report the average performance of all 100 queries. Q1-Q8 are generated as follows. First, we randomly choose the query origin s and destination t among the vertices of the road networks. Then, we classify the query into one of the 8 query sets, based on the longitude and latitude of s and t . Specifically, we compute the L_∞ distance between s and t on the 2-dimensional space defined by their longitudes and latitudes. Let λ_{max} denote the maximum value of this L_∞ distance. If the distance between s and t lies between $(\lambda_{max}/2, \lambda_{max}]$, we classify the query into Q8; if it is between $(\lambda_{max}/4, \lambda_{max}/2]$, it goes to Q7; if it is between $(\lambda_{max}/8, \lambda_{max}/4]$, it goes to Q6, etc. If the distance is too small to fit in any query set, we add the query to Q1. Clearly, the distance between the origin and the destination increases with the query set ID. Hence, the differences among results for different query sets reflect the impact of the travel distance.

Next, we clarify the generation of the cost constraint θ . For each query, we first compute c_{min} , the minimum cost of any path, and c_{max} , the cost of the minimum-length path from s to t ; then, we select θ uniformly at random from $[c_{min}, c_{max}]$. Note that if $\theta < c_{min}$, the query cannot return any result; if $\theta > c_{max}$, the minimum-length path is always a valid result to the CSP query.

Another important parameter is α , which determines the approximation guarantee of α -CSP queries. We view α as a system parameter rather than part of the query, since α controls the tradeoff between query accuracy and system efficiency, which is more relevant to the system environment than individual queries. In order to find a suitable value for α , we performed a set of experiments with varying α , whose results can be found in Section 5.4.4. The conclusion is that the value $\alpha = 1.1$ leads to a good balance between the query efficiency and accuracy. Hence, in the rest of our experiments, we fix $\alpha = 1.1$.

Methods. We compare *COLA* against three state-of-the-art methods: *Sky-Dijk* [53], *CP-Dijk* [20] and *CSP-CH* [87], described in Section 2.4.1. In fact, among existing methods

Figure 5.4: Query efficiency vs. Query sets.

these are also the only methods that can answer exact CSP or α -CSP queries within reasonable time (up a few hours for each query) on the datasets used in our experiments. Our implementations of *COLA* and *CSP-CH* keep the index in main memory in order to

Figure 5.5: Space consumption.

Figure 5.6: Total preprocessing time.

obtain fast query response time. In addition, we also include α -Dijk (see Section 5.3.1) in our comparisons, which answers α -CSP queries without an index.

5.4.2 Query Efficiency Results

Figure 5.4 plots the average query execution time (in seconds) for all methods, on all 8 query sets Q1-Q8 and all 8 road network datasets. Note that the vertical axis is in logarithmic scale. The most apparent observation is that regardless of the dataset or query set, our main proposal *COLA* consistently outperforms all other methods. The performance gap is several orders of magnitude, except for one setting (query set Q1 on the smallest NY dataset), where all methods take negligible time to finish.

More importantly, the *COLA* query time is always within one second, even on the largest network covering the entire USA. For state-sized networks, *COLA* always finishes within milliseconds. In contrast, the rest of the methods require at least 1 second finish, except for a few settings that involve both a small dataset and a query set with very close query origin and destination. For queries with relatively far apart origin / destination pairs, these methods (other than *COLA*) can take several hours to process one query. From these observations, we conclude that *COLA* not only outperforms its competitors, but brings down the query processing cost from prohibitively high to a practically low. In other words, *COLA* makes α -CSP feasible on current hardware.

Besides *COLA*, the next fastest method is our index-free algorithm α -Dijk, again in all settings. The performance gap between α -Dijk and previous methods is also notable, often more than an order or magnitude except for the cases with short queries and / or small road networks. Interestingly, index-free α -Dijk outperforms the indexed method

CSP-CH, which is given the advantage of large amounts of pre-computations. This suggests that exact CSP as a query type may not be practical for large networks; in such situations, it is necessary to relax the query definition.

Comparing the three existing methods, we observe that the performance improvement of *CP-Dijk* (for α -CSP) over *Sky-Dijk* (for exact CSP) is negligible, in all settings. This confirms that the *CP-Dijk* (and also other previous work on α -CSP, as *CP-Dijk* is the current state of the art) focuses on theoretical improvement in terms of asymptotic complexity rather than practical performance. *CSP-CH* shows better performance, thanks to its pre-computed index. Nevertheless, the performance of all three methods is so poor that they often need hours to process a single query. Further, on the USA dataset, these methods cannot answer queries in Q7 and Q8 with our hardware, since they require more than 64GB memory. Note that *Sky-Dijk* and *CP-Dijk* do not store any index structure in memory, and yet they require far more memory than our index-based method *COLA*.

The impact of the distance between the origin s and destination t becomes apparent, once we compare results for different query sets. When they are close to each other (e.g., Q1 and Q2), the query cost for methods are small, especially on small networks. As the s and t become farther apart, query costs increases quickly. This is expected, as the longer the trip, the more vertices are involved during CSP processing. An important observation is that *COLA*'s performance is robust against varying query sets. This is mainly due to its use of the overlay network. Specifically, *COLA* utilizes the precomputed distances between boundary vertices of different partitions, and the number of boundary vertices involved slowly increases with the distance between s and t , compared to the number of vertices in the network.

Finally, we compare results on different datasets. As expected, the larger the network, the more expensive it is to process a CSP query. Once again, unlike its competitors, the query cost of *COLA* grows slowly with the network size, thanks to its use of the overlay network and the constrained labeling index, whose effects become more pronounced as the dataset becomes larger.

5.4.3 Index Size and Construction Time Results

Figure 5.5 illustrates the memory consumption for all methods, before running any query. The results for the non-indexed approaches *Sky-Dijk*, *CP-Dijk* and α -*Dijk* are simply the size of the road network. On the other hand, the results for *CSP-CH* and *COLA* indicate their respective index sizes. In addition, we also show the size of the overlay graph in *COLA* in the same figure.

The most important observation is that the index structures of *CSP-CH* and *COLA* fit into the size of available memory in our setting, i.e., 64 GB, which is commonplace among modern servers. Comparing different methods, the non-indexed methods of course

consume the smallest amount of memory, which is simply the size of the input graph. However, note that the results here are memory usage before executing a query, rather than peak memory consumption. For instance, as explained in Section 5.4.2, all three existing methods runs out of memory while executing a query in Q7 or Q8 on the USA dataset, which suggest that they consume vast amounts of memory during query execution.

Comparing *CSP-CH* and *COLA*, the former uses smaller amounts of memory than the latter, and yet the index size of *CSP-CH* is considerable compared to non-indexed methods. Although *COLA* requires a larger index, its memory overhead is affordable, and is more than compensated by its query performance as shown in Figure 5.5. The results also show that the size of the overlay graphs is negligible, indicating that the space consumption of *COLA* is mainly attributed to its constrained labeling index.

Figure 5.6 presents the total pre-processing time of *CSP-CH* and *COLA*. Note that the non-indexed methods are not present as they do not need pre-processing. In general, pre-processing time is not critical, for two reasons. First, pre-processing happens off-line, and it is usually performed infrequently. Second, even if we do need to re-compute the index often (e.g., when the path lengths / costs are defined in such a way that they change over time), pre-processing time can be effectively reduced through parallel computations. For example, in *COLA*, the construction of overlay graph and the labeling index involve multiple independent path computations (explained in Sections 5.3.2 and 5.3.3), and, thus, can be performed in parallel on multiple servers. According to the experimental results, *COLA* incurs a higher pre-processing time than *CSP-CH*, and the costs of both methods increase with the size of the road network, as expected.

Summarizing the experiments, *COLA* effectively reduces the processing time of α -CSP queries from hours to sub-second, using no more than 64GB memory for all settings. Hence, it is clearly the method of choice for α -CSP processing. When an index is not available, we recommend the α -*Dijk*, which might be suitable for applications that do not require fast response. Since all previous methods are prohibitively expensive and α -CSP has promising use cases, the proposed methods might become key enablers for new online navigation services based on α -CSP.

5.4.4 Impact of α

Recall that our overlay graph construction and index construction algorithm both require the knowledge of the parameter α . Meanwhile, the key ingredient of the overlay graph construction and index construction is the α -*Dijk* algorithm. The preprocessing efficiency and query accuracy of our *COLA* index is hence mainly decided by the query efficiency and accuracy of our α -*Dijk*, respectively. So, in this set of experiments, we evaluate the impact of α to the query performance and query accuracy of α -*Dijk*.

 Figure 5.7: Impact of α to α -Dijk.

For each dataset, we consider the CSPs of setting a threshold θ to limit the travel time cost, while minimizing the distance. Each query is generated by randomly selecting an origin s and a destination t . The generation of cost is similar to that in Section 5.4. To evaluate the query performance, we randomly generate 1000 queries and report the average query time. To inspect the accuracy, we inspect the relative error ratio of the length of the α -CSP P' returned by α -Dijk compared to the exact CSP P returned by *Sky-Dijk* given the same origin s , destination t , and cost threshold θ . Specifically, the relative error ratio is calculated as $(\ell(P') - \ell(P)) / \ell(P)$. Generally, a smaller relative error ratio indicates that the α -CSP query returns a more accurate result, e.g., when the relative error ratio is zero, it means α -Dijk returns the answer for the exact CSP query, without loss of the accuracy. We report the average error ratio on the 1000 queries to evaluate the accuracy. Note that CTR and USA are missing in Figure 5.7 since *Sky-Dijk* incurs prohibitive memory consumption ($>64\text{GB}$).

In our experiment, we vary α from 1 to 1.4, and present the results on the 6 datasets where the exact CSP can finish processing. Figures 5.7(a) and (b) present the query time and relative error on NY, NW, and LKS datasets, respectively. As we can observe, an increased α can help to significantly reduce the query time. When α changes from 1 to 1.1, the improvement of query efficiency is more than one order of magnitude. When the size of the dataset increases, this improvement becomes more significant, e.g., on

LKS, the improvement is over two order of magnitude. Meanwhile, Figure 5.7 shows the relative error of the queries in average. As we can observe, an increased α leads to a larger relative error ratio, which is as expected. However, the error ratio is general smaller than the worst case bound. For example, the relative error of α -Dijk for $\alpha = 1.1$, is around 4.5% on NY dataset, which is only half that of the worst case bound, i.e., 10%. Figures 5.7(c) and (d) report the query time and relative error on FLA, NE, and W datasets, respectively. We observe a similar phenomena when we change α . We set α to 1.1 for α -Dijk since it strikes a good balance between the query performance and query accuracy.

5.5 Summary

In this chapter, we present *COLA*, a novel and practical solution for approximate constrained shortest path processing, which can be applied to navigation services where a single criterion on the path is not sufficient to capture the user's requirement. Existing methods for CSP processing incur prohibitively high costs, and to our knowledge *COLA* is the first solution that brings down the cost of answering an α -CSP query to a practical level. *COLA* mainly exploits two important properties of the road network. First, real road networks are often (roughly) planar, and, thus, can be effectively split into partitions, each of which contains only a relatively small number of boundary vertices. Accordingly, *COLA* partitions the network, builds an overlay graph on the partitions, and indexes the skyline paths between pairs of boundary vertices. Second, in practice there often exist a relatively small number of landmark vertices in the road network that commonly appear in CSP results. Meanwhile, *COLA* also includes an algorithm, α -Dijk, which significantly outperforms existing techniques for α -CSP processing without an index. Experiments using large-scale real data confirm our performance claims.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we study three problems on reachability and shortest path queries, each of which finds enormous real-world applications.

First, we study the problem of reachability queries. We present a novel study on the vertex ordering in *hierarchical hub labellings (HHL)*, the state-of-the-art labelling scheme for reachability queries. We formally prove that deriving an optimal vertex ordering that minimizes the index size is NP-hard. Our proposed *AC-greedy* algorithm, with $O(n^3)$ complexity can provide an *HHL* index whose size is at most $O(\sqrt{n})$ times the optimal one, where n is the number of vertices in the input graph. To our knowledge, this is the first theoretical study for the approximation of *HHL* index and we show that *AC-greedy* is an *HHL* method with non-trivial performance guarantee. Second, motivated by our theoretical study, we design a new heuristic for vertex ordering, and show that it leads to an *HHL* index with superior practical performance for reachability queries. Extensive experiments on real data demonstrate the efficiency of our solutions.

Next, we investigate on the problem of shortest path queries on timetable graphs. We present *Timetable Labelling (TTL)*, an efficient indexing technique for route planning on public transportation networks. In particular, we formalize the conceptual framework of *TTL* and establish a set of constraints to avoid redundant labels in *TTL*. We devise an efficient query processing algorithm for *TTL*, which can answer any route planning query with a linear scan of the labels sets associated with the query nodes.

Besides, to compress the labels of *TTL*, we propose two novel techniques that significantly reduce the space consumption of *TTL* without substantial effects on query performance. In addition, we develop a simple and intuitive method to concisely present the query results, which not only is user friendly but also leads to higher query efficiency in *TTL*. Finally, we experimental evaluate *TTL* against several state-of-the-art

techniques on 11 real transportation networks with up to 4 million edges, and demonstrate that *TTL* significantly outperforms the competitors in terms of query efficiency, while incurring moderate preprocessing and space overheads.

For the constrained shortest path queries on road networks, we present *COLA*, a novel and practical solution for approximate constrained shortest path processing, which can be applied to navigation services where a single criterion on the path is not sufficient to capture the user's requirement. Existing methods for CSP processing incur prohibitively high costs, and to our knowledge *COLA* is the first solution that brings down the cost of answering an α -CSP query to a practical level. *COLA* utilizes important properties for real road networks, and applies effective indexing which leads to orders of magnitude reduction in query execution time. Meanwhile, *COLA* also includes an algorithm, α -*Dijk*, which significantly outperforms existing techniques for α -CSP processing without an index. Experiments using large-scale real data confirm our performance claims.

6.2 Future Directions

In our three studied problems, we assume that the graphs are static. However, in many scenarios, the graph might be dynamically changing. For example, in social networks, one could make new friends; in public transportation networks, the buses or subways may get delayed due to traffic conditions; in road networks, road might be blocked due to maintenance. It is desirable to reflect the updates real time. It is true that our indices can be reconstructed from scratch in several minutes or hours. However, note that the changes may happen one after another, and it is highly possible that the subsequent changes of the graphs are not reflected into the updated index. So it will be interesting to study how to efficiently update the indices, e.g., in milliseconds, instead of reconstructing them from scratch. This guarantees that the changes of the graphs can be almost reflected instantly into the proposed indices, so that our indices can provide the correct answer. This could be very useful in systems that require real-time query answers. In addition, our current proposed solutions are all devised on single machines. In many cases, one might be able to use many cores of a machine or many machines from a cloud to preprocess the indices or handle the queries. It would be interesting to study how to devise efficient parallelized algorithms that can help improve the preprocessing and query efficiency.

Besides the above problems, we further have the following three future directions that are related to our three studied problems, respectively.

6.2.1 Variants of Reachability Queries

In many real applications, each edge is usually associated with a label to denote certain relationship between two vertices. The reachability queries on such graphs usually involve some label constraints on the path connecting the two vertices. Such a reachability query, denoted as *label constrained reachability* query, is used to verify certain relationships between two vertices, and finds many real world applications, e.g., Semantic Webs, social networks. More formally, let L be the set of labels that appear on the edges of the input graph. Given a set of labels L_c as constraint, a label constrained reachability query asks whether an origin s can reach a destination t , only through the edges with labels $l \in L_c$.

Another variant is the *k-hop reachability* queries, which asks whether there exists a path with no more than k edges from an origin to a destination. This variant also finds application scenarios in sensor networks, the World Wide Web, and social networks, etc. Existing solutions for conventional reachability queries cannot be used to solve these problems. Meanwhile, as *HHL* is the state-of-the-art scheme for conventional reachability queries. It is interesting to consider whether *HHL* can be applied to such problems.

However, it is challenging to design an efficient *HHL* index for these two variants of reachability queries. In particular, the nature of the two variants of reachability queries make the SCC mapping method (mentioned in Section 3.1.1) inapplicable. On one hand, the superb efficiency of reachability queries heavily relies on the fact that, for reachability queries, a graph of billion nodes can be shrunk to a DAG with only million nodes using the SCC mapping method. So to devise an efficient *HHL* index structure, the first key is to find a way to reduce the size of the graph for the query processing. For example, for the label constrained reachability queries, if the relationships do not hold for any vertices in a subgraph G_1 to any vertices in a subgraph G_2 , then we can handle any query for vertices in $G_1 \cup G_2$ by only considering on subgraph G_1 and subgraph G_2 . For the *k-hop* reachability queries, we can also try to find disjoint subgraphs such that for any vertices in different subgraphs, they are not *k-hop* reachable. Then we can handle the queries in different subgraphs.

The other problem is how to devise an efficient preprocessing and query algorithm for their *HHL* indices. For example, if one directly adopts the index construction algorithm for reachability *HHL* index, and applies it to the label constrained reachability queries, she will find it too costly. In particular, the out-label (resp. in-label) each vertex v maintained not only need to include the vertices u that it can reach (resp. can be reached), but also the labels of the different paths from v to u (resp. u to v). This incurs prohibitive preprocessing cost, as well as large space consumption, resulting in degraded query performance. So it will be challenging but also interesting to devise better preprocessing and query algorithms that are specifically designed for these two problems.

6.2.2 More Scenarios on Public Transportation Networks

In public transportation networks, the users may concern more aspects, besides the travel time. For instance, one may take into account fares, convenience, stability of a connection, seat reservability, and etc. It will be a possible direction to extend *TTL* to support the above mentioned sophisticated queries. However, it is challenging to devise an efficient index for such problems. For example, consider a query that asks for the cheapest route among all possible ones that arrive no later than a specified threshold arrival time t_θ . This problem will be even more involved than the constrained shortest path problem, since it not only includes the constraint on the arrival time, but also includes temporal-spatial constraints between different edges. To solve these problems, we will investigate how to provide approximate solutions under such scenarios, and make full use of the property of transportation networks to devise efficient indices.

Besides, in real world scenarios, we may want to find reliable routes such that we have enough time to do the transfer. In this case, we can pre-define a reliability threshold T as the time allowed for a public transportation transfer, e.g. from one bus to another. Afterwards, we find the EAP/LDP/SDPs that satisfy the reliability constraint. In this case, the queries will be more involved, and current *TTL* indexing approach cannot be easily applied to answer such queries. It will be interesting to consider how to devise efficient indices to answer such queries.

6.2.3 Shortest Path Problems on Time Dependent Road Networks

While public transportation networks only allow discrete time to transfer from one place to another, road networks allow travellers to change from one place to another at any time in an ideal environment. Most existing work assumes that the road networks are static. However, in reality, the travel time on road networks is usually dynamically changing. For example, in peak hours, some main roads will be persistently congested and incurs more travel time. Fortunately, most traffic conditions and the travel time are predictable, e.g., through signals collected from the embedded sensors on the roads, and the network hence can be modeled by a *time-dependent graph*. In time-dependent graphs, the length of an edge (u, v) varies with time t . Denote the length function by $\ell(t)$, previous work assumes that the length function is piecewise linear and can be split into segment functions. The shortest path between two nodes s and d can change over time. Most existing work studies how to obtain the shortest path from time to time. One interesting direction is to investigate how to adopt the optimization techniques in *COLA* to build an efficient index to answer the shortest path queries in *time-dependent* road networks.

On time-dependent graphs, Dijkstra’s algorithm can be augmented to compute shortest paths [27, 39], as long as the cost function is non-negative and FIFO. However, the worst case complexity to find all shortest paths from one vertex to another is $O(\log^2(n) \cdot n^{\log(n)})$ [42], where n is the number of vertices. On one hand, for existing state-of-the-art indexing techniques, e.g., *HHL*, they require to invoke the Dijkstra’s algorithm to build the index, or to proceed the queries. So the first step to build an efficient index for the time-dependent shortest path queries is to consider a more efficient Dijkstra-like traversal algorithm, so that the traversal algorithm consumes less time and space.

On the other hand, we will examine the optimization techniques used in *COLA* to exploit the feature of road networks. For example, on road networks, the graph partitioning technique can be applied to split the road network into edge-disjoint subgraphs, and then an overlay graph can be constructed among the boundary vertices, i.e., vertices appear in more than one subgraph. Then the indices can be built on the overlay graph, and time-dependent shortest path queries can be answered through the subgraphs and the indices. Finally, as the space consumption could be prohibitive if the indices need to store many piecewise linear segment functions for the shortest paths, we will consider approximate solutions instead of exact answers to reduce the index size, and help improve the query performance, which has shown to be very effective on the *COLA* index.

Appendix A

Proofs

Proof of Lemma 3.1. Denote $x = |\{s | (s, t) \in C_i(v)\} \setminus \{v\}|$ and denote $y = |\{t | (s, t) \in \mathcal{N}_i(v)\} \setminus \{v\}|$, then $I_i(v) = x + y$, at the meantime, we have:

$$\sqrt{|C_i(v)|} \leq \sqrt{x \cdot y + x + y} \leq x + y = I_i(v).$$

This finishes the proof of the LHS of Lemma 3.1. As G is a DAG, inequality $I_i(v) \leq n$ will always hold. On the other hand, $I_i(v) \leq |C_i(v)|$. Combining these, we obtain that:

$$I_i(v) = \sqrt{I_i(v)} \cdot \sqrt{I_i(v)} \leq \sqrt{n} \cdot \sqrt{|C_i(v)|}.$$

This finishes the proof of Lemma 3.1. □

Proof of Theorem 3.3. The main challenge for AC-Greedy algorithm is to calculate the number r of reachable pairs that v covers, the number c of label sets that v is inserted into, and the updates of r and c after ordering a vertex. To tackle the problem, AC-Greedy can be implemented by maintaining three tables *Reach*, *Cov* and *Pass*, where *Reach*(u, v) indicates whether $u \rightarrow v$ in G , *Cov*(u, v) indicates whether (u, v) is covered, and *Pass*(u, v) indicates how many reachable pairs (u, w) that satisfies $u \rightarrow v$ and $v \rightarrow w$. Also it needs to maintain two counters $r(v)$ and $c(v)$ for each $v \in V$, where $r(v)$ is the number of reachable pairs v can cover and $c(v)$ is the number of label sets v inserted to.

Note that the above score can be initialized with $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space cost. To update the scores, when a new vertex v^* is ordered, both $r(v)$ and $c(v)$ need to be updated for the remaining vertices. Let Q be the set of uncovered reachable pairs that v^* covers. For each pair $(u, w) \in Q$ and each vertex $v \in V$, we decrease $r(v)$ and *Pass*(u, v) by 1 if v covers (u, w) . If *Pass*(u, w) is zero, then we decrease $c(v)$ by 1.

As v covers (u, w) if and only if both $u \rightarrow v$ and $v \rightarrow w$ holds, we can check whether v covers (u, w) in constant time by looking at *Reach*(u, v) and *Reach*(v, w). It also requires only constant time to update $c(v)$. Thus, once a reachable pair is covered, it needs $\mathcal{O}(n)$ time to update $r(v)$ and $c(v)$ for each $v \in V$. Since we will cover $\mathcal{O}(n^2)$ reachable pairs in total, the algorithm runs in $\mathcal{O}(n^3)$ time.

Proof of Lemma 3.2. The correctness of our lemma is guaranteed by the fact that the score function is always non-decreasing. In particular, for each vertex v , the estimated score function by using k-min sketch is:

$$f(v) = (\hat{S}_{in}(v) + \hat{S}_{out}(v)) / (\hat{S}_{in}(v) * \hat{S}_{out}(v) + \hat{S}_{in}(v) + \hat{S}_{out}(v)).$$

where $\hat{S}_{in}(v)$ and $\hat{S}_{out}(v)$ are defined in Algorithm 3. When a vertex u is ordered and removed from the graph, then $\hat{S}_{in}(v)$ and $\hat{S}_{out}(v)$ values of the remaining vertices may change. We first discuss the monotonicity of $\hat{S}_{in}(v)$ and $\hat{S}_{out}(v)$. Consider the i -th copy of O_{min} value for a remaining vertex v after deleting a vertex u , if v cannot reach u , the $O_{min}(v)$ value will not be affected by the deletion of vertex u . When v can reach u , if $O_{min}^i(u) > O_{min}^i(v)$, let the vertex with X^i key that is equals to O_{min}^i be w , then there exists a path from v to w does not pass through u and the $O_{min}^i(v)$ value will not be affected by the deletion of vertex u . Otherwise as v can reach w , all paths from v to w passes through u , which means $O_{min}^i(u)$ should be $X^i < O_{min}^i(u)$, leading to a contradiction. If $O_{min}^i(u) = O_{min}^i(v)$, after the deletion of vertex u , the new value of $O_{min}^i(v)$ will be no more than $O_{min}^i(v)$. So $O_{min}^i(v)$ will be non-decreasing in all cases. As i here chosen can be any copy of the sketch, the value of $\sum_{i=1}^k O_{min}^i(v)$ for any remaining vertex v will also be non-decreasing, indicating that $\hat{S}_{out}(v)$ is always non-increasing. Similarly, we can prove that $\hat{S}_{in}(v)$ for a remaining vertex v is always non-increasing.

Finally, consider $f(v)$ for any remaining vertex v , it can be verified that when $\hat{S}_{in}(v)$ (resp. $\hat{S}_{out}(v)$) decreases, $f(v)$ increases. As after the updates, both $\hat{S}_{in}(v)$ and $\hat{S}_{out}(v)$ will be non-increasing, $f(v)$ will be non-decreasing.

During the lazy update, if an updated score of a vertex v is minimum in the heap, even if there are some vertices with obsoleted score in the heap, vertex v will still have the minimum score after all the updates. The reason is due to the non-decreasing property of the score value as we have proved. This establishes the correctness of the score values with lazy update approach.

Proof of Lemma 4.1. We prove the lemma by contradiction: Assume that there does not exist a label $\langle w, t_d, t_a, b, p \rangle$ in $L_{out}(u)$ or a label $\langle w', t'_d, t'_a, b', p' \rangle$ in $L_{in}(v)$ such that either of them satisfies any one of the conditions in the lemma. In other words, there are two cases, stated as follows.

Case 1: There does not exist a canonical path from u to v .

Case 2: There does not exist a node w^* such that both the paths from u to w^* and from w^* to v contain a canonical path respectively.

Given that P is the result of the path query q from u to v , we have that P is not dominated by the other paths that answer q by Definitions 4.2, 4.3, and 4.4. Besides,

any sub-path of P is also not dominated by the other paths. If not, we can replace the sub-path with the dominated one.

Let w^o be the node with highest rank among nodes in P . If w^o equals u or v , then P is a canonical path (by Definition 4.5), which contradicts *Case 1*. On the other hand, if w^o does not equal u or v , let P_1 be the sub-path from u to w^o , and P_2 be the sub-path from w^o to v . As such, we have P_1 and P_2 as canonical paths (by Definition 4.5), which contradicts *Case 2*. Thus, the lemma is proved. $\square$

Proof of Lemma 4.2. Let $l = \langle u, t_d, t_a, b, p \rangle$ be an in-label in $L_{in}(v)$, and $l' = \langle u', t'_d, t'_a, b', p' \rangle$ be an out-label in $L_{out}(v)$. The lemma says that if we remove l or l' , there exists a path query q whose result P does not satisfy the constraints in Lemma 4.1.

Consider l . Given that $\langle u, t_d, t_a, b, p \rangle \in L_{in}(v)$, we have $o(u) < o(v)$ and l is pertinent to a canonical path P from u to v . Let q be a path query from u to v with starting timestamp t_d and ending timestamp t_a , and P be the result of q . By contradiction, assume that after removing l from $L_{in}(v)$, P still satisfies Lemma 4.1. Since $o(u) < o(v)$, by Definition 4.7, v cannot appear in any label of $L_{out}(u)$. Then, there must exist a node w , such that $\langle w, t_d, t_a^*, \cdot, \cdot \rangle \in L_{out}(u)$, $\langle w, t_d^*, t_a, \cdot, \cdot \rangle \in L_{in}(v)$, and $t_a^* \leq t_d^*$. That is, the path P from u to v passes w , and w has a higher rank than both u and v , which contradicts the Rank Constraint in Definition 4.5. The case for l' can be proved in a similar manner. $\square$

Proof of Lemma 4.3. Consider a path query q from u to v with a starting timestamp t and an ending timestamp t' . Let P_{ans} be the result of q , $l = \langle w, t_d, t_a, \cdot, \cdot \rangle$ be an out-label of u , and $l' = \langle w', t'_d, t'_a, \cdot, \cdot \rangle$ be an in-label of v . If l is pertinent to P_{ans} or the sub-path of P_{ans} , we have $t \leq t_d$ and $t_a \leq t'$, which is checked by Line 4 of Algorithm 6. Similarly, the timestamps of l' are checked by Line 10 of Algorithm 6. By Lemma 4.1, there are three cases:

Case 1: If $w = v$, we have P_{ans} , which is pertinent to l .

Case 2: If $w' = u$, we have P_{ans} , which is pertinent to l' .

Case 3: If $w \neq v$, $w' \neq u$, $t_a \leq t'_d$ and $w = w'$, we have l and l' each of which is pertinent to a sub-path of P_{ans} .

Obviously, *Case 1* and *Case 2* can be correctly handled by Lines 6-8 and Lines 12-14 respectively. Now, we focus on the discussion of *Case 3*. By contradiction, assume that Lines 15-20 cannot find a sketch corresponding to P_{ans} . Since labels in $L_{out}(u)$ and $L_{in}(v)$ are sorted, we have $t_a > t'_d$ for all pairs of l and l' where $w = w'$. This contradicts *Case 3*, which completes the proof. $\square$

Proof of Lemma 4.4. Without loss of generality, let $o(u) < o(v)$, i.e., the rank of u is higher than that of v . Let P_1 denote the sub-path of P from u to p , and P_2 the sub-path

of P from p to u . By Definition 4.6, the rank of p is higher than all nodes in P , except u and v . As such, both P_1 and P_2 satisfy the Rank Constraint in Definition 4.5.

Assume on the contrary that either P_1 or P_2 is not a canonical path. Consider that P_1 is not a canonical path. Since P_1 satisfies the Rank Constraint, there must exist another path P^o from u to p dominating P_1 . This contradicts that P is a canonical path. The case for P_2 can be similarly established. $\square$

Proof of Lemma 4.5. By Lemma 4.1 and Lemma 4.3, given a starting timestamp t and an ending timestamp t' , we can find the sketches pertinent to all paths from u to v , each of which is an SDP regarding its departure time $t_d \geq t$ and arrival time $t_a \leq t'$. Denote these paths by $SDP(u, v, t, t')$. As such, when $t' = +\infty$, we can obtain the sketches pertinent to paths in $SDP(u, v, t, +\infty)$. Hence, we are able to find the EAP whose arrival time is the minimum in $SDP(u, v, t, +\infty)$. The similar analysis can be applied to the extension of LDP queries. $\square$

Proof of Lemmas 4.6 and 4.7. Consider Lemma 4.6. Given the non-dominated path P starting from u at time $t \in T_d$ and ending at v at time t' , assume that P is not an EAP. In other words, there exists another path P' such that P' also starts from u at time t but ends at v at time t^* , which is earlier than t' , i.e., $t^* < t'$. However, this violates that P is a non-dominated path, which completes the proof of Lemma 4.6. The proof of Lemma 4.7 can be established in the similar manner. $\square$

Proof of Lemma 4.8. We first show that each label in $\mathcal{L}$ constructed by Algorithm 8 is pertinent to a canonical path of G . Consider a label $l = \langle u, t_d, t_a, \cdot, \cdot \rangle$ in the in-label set $L_{in}(v)$ of v , which is pertinent to a path P from u to v with departure time t_d and arrival time t_a . As all nodes whose ranks are higher than u are removed before the computation on u , we have that P fulfills Rank Constraint in Definition 4.5. Assume that P does not satisfy Dominance Constraint, i.e., there exists another canonical path P' from u to v with departure time $t'_d > t_d$ and arrival time $t'_a < t_a$. Since we compute the outgoing edges of u in descending order of their departure timestamps, P' is computed before P . As such, P should not be added to $L_{in}(v)$ due to that $[t'_d, t'_a] \subset [t_d, t_a]$, which leads to a contradiction. In sum, P is a canonical path.

Now we prove that each canonical path of G is pertinent to a label in $\mathcal{L}$. Assume by contradiction that there exists a canonical path P^* from u to v such that P^* is not pertinent to any label in $\mathcal{L}$. By Lemmas 4.6 and 4.7, given a starting timestamp t , we can find all EAPs from u by the modified version of Dijkstra's algorithm; and given an ending timestamp t' , we can find all LDPs to v by the reverse version of Dijkstra's algorithm. As such, P^* must be found by the modified version or the reverse version of Dijkstra's algorithm. This indicates that there must already exist a label l' in $\mathcal{L}$ such that l' is pertinent to another path P' and P' dominates P^* . This contradicts to the assumption

that P^* is a canonical path, as a consequence, *IndexBuild* correctly produces a *TTL* index.

To analyze the complexity, we consider the time cost in each iteration. In each iteration, a vertex is chosen and a temporal Dijkstra is started from this vertex. Observe that each edge will be traversed at most once, and during the traversal, a vertex is popped at most once from the heap. By maintaining a Fibonacci Heap, the amortized cost hence is $O(n \cdot \log n)$. The label insertion process as shown in Lines 29-33 requires only $O(1)$ time for each insertion by forcing the vertices sorted in the out-label (resp. in-label) by the vertex level order and then in decreasing order of arrival timestamps (resp. in increasing order of departure timestamps). So in each iteration, the complexity is $O(m + n \cdot \log n)$. There are n iterations, and the time-complexity can be bounded by $O(n \cdot m + n^2 \cdot \log n)$, which completes the proof. $\square$

Proof of Lemma 4.9. By the definition of $I(u_i)$, we have the following two inequalities:

$$|I(u_i)| \leq \sum_{v \in V} |T_a(v)| + \sum_{v \in V} |T_d(v)| = \alpha, \quad (\text{A.1})$$

$$|I(u_i)| \leq |R_i(u_i)|. \quad (\text{A.2})$$

Based on the above inequalities, we have

$$|I(u_i)| = \sqrt{|I(u_i)|} \cdot \sqrt{|I(u_i)|} \leq \sqrt{\alpha} \cdot \sqrt{|R_i(u_i)|}. \quad (\text{A.3})$$

Proof of Theorem 4.1. The approximation guarantee and complexity can be proved similarly using the same technique in the proof of Theorem 3.2, and we omit it for brevity.

Proof of Corollary 4.1. Consider a label $l = \langle \cdot, \cdot, \cdot, \cdot, p \rangle$. If $p = \text{null}$, l is pertinent to an edge in G . Otherwise, by Definition 4.7, l is pertinent to a canonical path consisting of at least two edges. If $p \neq \text{null}$, we recursively find the edges that form the sub-paths P_1 and P_2 pivoted by p , and then concatenate the sub-paths. By Lemma 4.4, P_1 and P_2 are canonical paths, which proves the correctness of Algorithm 7. $\square$

Proof of Observations 4.1 and 4.2. To prove why Observation 4.1 holds, assume to the contrary that P is dominated by another path P' from u to v . In that case, P' either (i) departs from u later than t_{-} , or (ii) departs from u at t_{-} but arrives at v earlier than P . This, however, contradicts the facts that t_{-} is the largest timestamp in T_d , and P is an EAP. The rationale of Observation 4.2 is that, if P is dominated by another path P^* , then P^* must depart from u after t_{-}^* (i.e., P^* has departure time t_{-}); otherwise, the existence of P^* violates the assumption that P is an EAP. In other words, if P is not dominated by any EAP from u at t_{-} , then P would not be dominated by any other paths, either.

Proof of Lemma 5.1. Let v_1° be the first boundary vertex that P_{opt} goes through, and v_2° be the last boundary vertex that P_{opt} goes through. We consider three sub-paths P_1 , P_2 , and P_3 , where P_1 is the path from s to v_1° , P_2 is the path from v_1° to v_2° , and P_3 is the path from v_2° to t . Clearly, P_1 is within G_s , otherwise, it must go to another subgraph, which indicates that there must exist a boundary vertex v° such that P_1 goes through. This contradicts to the assumption that v_1° is the first boundary vertex in P_{opt} . Similarly, P_3 is within G_t . Meanwhile, based on the property of overlay graph, there exists a path P° in G° such that P° can α -dominate P_2 , i.e., $c(P^\circ) \leq c(P_2)$, $\ell(P^\circ) \leq \alpha \cdot \ell(P_2)$. Let P' be the path by concatenating P_1 , P° , and P_3 , then $c(P') \leq c(P_{opt}) \leq \theta$, and $\ell(P') \leq \alpha \ell(P_{opt})$. This finishes the proof. $\square$

Proof of Lemma 5.2. From Algorithm 11, it is easy to verify that *LabelJoin* algorithm since it never goes back to an entry previously scanned. So we mainly focus on the correctness proof here. We say a path P is covered by *COLA* if it can be reconstructed by an out-label ρ_o and an in-label ρ_i .

Let P be the path with cost no more than θ and has a minimum length concatenated from $B_{out}(s)$ and $B_{in}(t)$. Then it is an α -CSP query. The reason is that, if there exists an α -CSP query, then *COLA* index contains an out-label ρ_o and an in-label ρ_i to cover it. Let the path P' be the α -CSP path covered by *COLA*. Then as P is minimum among all paths covered by *COLA*, $\ell(P') \leq \ell(P)$. If P is an α -CSP query, then P' will also be an α -CSP query.

On the other hand, we prove *LabelJoin* can correctly derive P' . In particular, during the scan of the labelling, if the labels concern different vertices, then, by concatenating these two labels, we will not be able to derive a path, so it will not miss any α -CSP. On the other hand, as $B_{out}(s)$ is sorted in increasing order of cost, and $B_{in}(t)$ is sorted in decreasing order of cost, if an out-label and an in-label concerning a vertex w , has a total cost larger than θ , then going to the next label in $B_{in}(v)$ will not miss any path since an previous entry in $B_{in}(v)$ either has a larger cost, which indicates that the total cost of the labels will also exceed θ and will not be a valid answer. Meanwhile, if a previous label includes another node, it will not be a valid path either.

On the other hand, if the total cost of the two labels does not exceed θ , then it compares with the existing path P^* , if it is larger than P^* , then it will not be the minimum path, and can be discarded. Then $B_{out}(s)$ goes to the next label entry. This operation, we prove that will also not miss P' . In particular, as label entries are stored in ascending of the cost, which indicates the paths are also stored in descending order of the lengths. Then, a previous label entry in $B_{out}(s)$ will have a larger length, considering the total length with the current label entry in $B_{in}(t)$. So in either case, *LabelJoin* will not miss P' , indicates that it will find the minimum-path covered by $B_{out}(s)$ and $B_{in}(t)$. This finishes the proof. $\square$

Proof of Theorem 5.1. For convenience, we abuse notation and use $\tau(\rho)$ to denote the surrogate value of an entry ρ in $L(v)$. Assume that P' contains k edges $e_1, e_2, \dots, e_k$, such that e_i ($i \in [1, k]$) points to a node v_i . We recursively construct k paths $P_1, P_2, \dots, P_k$, and define a surrogate value $\tau(P_i)$ for each path P_i as follows. First, we define P_1 as a path that contains only e_1 . Observe that $L(v_1)$ must contain an entry ρ_1 that represents P_1 (i.e., $\rho_1 = \langle v_1, c(P_1), \ell(P_1), *, s \rangle$), since such an entry would be inserted into $L(v_i)$ right after α -Dijk pops the first entry $\langle s, 0, 0, 0, null \rangle$ from the min-heap H . We set $\tau(P_1) = \tau(\rho_1)$.

Assume that P_i ($i \in [1, k-1]$) is constructed, and L_i contains an entry ρ_i that represents P_i . Then, right after ρ_i is popped from H and inserted into L_i , α -Dijk would insert an entry into H for each outgoing edge e of v_i . Since e_{i+1} is an outgoing edge of v_i , one of the entries inserted should be $\rho' = \langle v_{i+1}, c(P_i) + c(e_{i+1}), \ell(P_i) + \ell(e_{i+1}), \tau(P_i) + \ell(e_{i+1}), v_i \rangle$. Then, when ρ' is popped from H , it is either pruned by an existing entry in $L(v_{i+1})$ or inserted into $L(v_{i+1})$. In either case, there must exist an entry ρ_{i+1} with a cost at most $c(P_i) + c(e_{i+1})$ and a surrogate value $\tau(\rho_{i+1}) \leq \tau(P_i) + \ell(e_{i+1})$. We set $\tau(P_{i+1}) = \tau(\rho_{i+1})$.

With the above constructions, the following equations hold for any $i \in [1, k-1]$:

$$\begin{aligned} c(P_{i+1}) &\leq c(P_i) + c(e_{i+1}); \\ \tau(P_{i+1}) &\leq \tau(P_i) + \ell(e_{i+1}); \\ \ell(P_{i+1}) &\leq \alpha \cdot \tau(P_{i+1}) \leq \alpha \cdot \ell(P_{i+1}). \end{aligned}$$

Consequently, we have

$$\begin{aligned} c(P_k) &\leq c(P_{k-1}) + c(e_k) \leq c(P_{k-2}) + c(e_{k-1}) + c(e_k) \\ &\leq \dots \leq \sum_{i=1}^k c(e_i) = c(P'); \end{aligned} \tag{A.4}$$

$$\begin{aligned} \tau(P_k) &\leq \tau(P_{k-1}) + \ell(e_k) \leq \tau(P_{k-2}) + \ell(e_{k-1}) + \ell(e_k) \\ &\leq \dots \leq \sum_{i=1}^k \ell(e_i) = \ell(P'); \\ \ell(P_k) &\leq \alpha \cdot \tau(P_k) \leq \alpha \cdot \ell(P'). \end{aligned} \tag{A.5}$$

Equations (A.4) and (A.5) indicate that P' is α -dominated by a path P_k represented by an entry in $L(v_k)$. This completes the proof. $\square$

Proof of Lemma 5.3. Let v be the destination node and G_i be the input graph for Algorithm 13, then base on Theorem 5.1 and how labels are derived, it is guaranteed that each label is constructed from a path. Let this path be P , and the τ value is set to be no smaller than $\ell(P)$ based on Algorithm 13. In addition, for any path P' , as the label stores paths that cannot α -dominate each other from the smallest cost to the

largest cost, if a path P' has cost c , let ρ be the label entry for path P that has cost smaller than P' , and the next label has cost larger than P' ,

then if P has length such that $\tau(P) \leq \ell(P')$, then the requirement is satisfied. Meanwhile, if $\tau(P) > \ell(P')$, since P' has a cost smaller than the label entry next to the one concerning P , then it should be added into $\mathcal{L}(v)$ based on the Algorithm 13 Lines 16-18. So in either cases, P' satisfy that there exists a label ρ such that $c(\rho) \leq c(P')$ and $\tau(\rho) \leq \ell(P')$, which finishes the proof. $\square$

Proof of Corollary 5.1. This corollary can be directly derived by Lemma 5.3 by setting the origin as v_1 , and destination as v_2 . In particular, Lemma 5.3 satisfy that for any path P from v_1 to v_2 , there exists a label in $L(v_2)$ can α -dominate P . Meanwhile, based on how the edges are constructed (Algorithm 14 Line 14), it is guaranteed that for any path P from v_1 to v_2 , there exists a stored entry that can α -dominate P . As such entries are mapped to an edge e° , and these entries are pertinent to a path from v_1 to v_2 , the correctness is guaranteed. This finishes the proof. $\square$

Proof of Lemma 5.4 We prove that after the i -th iteration, for all paths P whose rank (see 5.3.3) $r(P) \leq i$, it will be α -dominated by a path P that can be concatenated by an in-label and an out-label.

In the first iteration, let u_1 be the node with $r(u_1) = 1$. Since initially B_{out} and B_{in} are empty, Line 5 condition 2 will not be triggered. The correctness is guaranteed by Theorem 5.1.

Consider the i -th iteration, an α -Dijk will be started from vertex u_i with $r(u_i) = i$. During the traversal, if a path P from u_i to v is pruned by α -Dijk, the subsequent correctness can be guaranteed by 5.1. However, if a path is pruned by label checking, i.e., Line 5 condition 2, then there exists an out-label in $B_{out}(u_i)$ and an in-label in $B_{in}(v)$ such that their concatenated path P' satisfy that $c(P') \leq c(P)$ and $\ell(P') \leq \tau(P)$. In this case, consider any path P^+ that contains P as a sub-path. We prove that for any such paths P^+ , there will exists some label that can α -dominate P^+ .

Denote the path corresponding to the labels that prune P as P_α , clearly $r(P_\alpha) < i$. For each path P^+ , we divide them into two sub-paths with $P_1 = P$, and P_2 is the remaining sub-path. Let P_α^+ be the path by concatenating P_α and P_2 . Then it is guaranteed that, $c(P_\alpha^+) \leq c(P^+)$, and $\ell(P_\alpha^+) \leq \ell(P^+)$. Based on the induction assumption, path P_α^+ satisfies that there exist a label in B_{out} and a label in B_{in} , such that their corresponding path P_x satisfy that $c(P_x) \leq c(P_\alpha^+)$, and $\ell(P_x) \leq \alpha \cdot \ell(P_\alpha^+) \leq \alpha \cdot \ell(P^+)$. In other words, for any path P^+ derived by the above method, there will exist a path pertinent to an out-label and an in-label that can α -dominate P^+ . So pruning such paths will be safe, and will not affect the correctness.

Consider any path P whose rank is i , based on the above construction, for the sub-path P_1 from u_i to the ending node of P , there exists a label ρ that will be stored such

that $c(\rho) \leq c(P_1)$ and $\ell(\rho) \leq \alpha \cdot \ell(P_1)$. As a reversed version of α -Dijk is applied. Similarly, there exists a label ρ' such that, for the sub-path P_2 from the source of P to u_i , such that $c(\rho') \leq c(P_2)$ and $\ell(\rho') \leq \alpha \cdot \ell(P_2)$. Adding them together, we can conclude that for any path P whose rank is i , it will be α -dominated by a path P concatenated from an in-label and an out-label.

This indicates that after the i -iteration, all paths P whose rank below i satisfy Lemma 5.4. After n iterations, the label then guarantees for any path, it satisfies Lemma 5.4, which finishes the proof. $\square$

Publication List

Paper published:

- **Sibo Wang**, Wenqing Lin, Yi Yang, Xiaokui Xiao, Shuigeng Zhou. Efficient Route Planning on Public Transportation Networks: A Labelling Approach. Proceedings of the ACM SIGMOD International Conference on Management of Data (**SIGMOD**), pages 967-982, 2015.
- **Sibo Wang**, Xiaokui Xiao, and Chun-Hee Lee. Crowd-Based Deduplication: An Adaptive Approach. Proceedings of the ACM SIGMOD International Conference on Management of Data (**SIGMOD**), pages 1263-1277, 2015.
- Andy Diwen Zhu, Wenqing Lin, **Sibo Wang**, and Xiaokui Xiao. Reachability Queries on Large Dynamic Graphs: A Total Order Approach. Proceedings of the ACM SIGMOD International Conference on Management of Data (**SIGMOD**), pages 1323-1334, 2014.
- Andy Diwen Zhu, Xiaokui Xiao, **Sibo Wang** and Wenqing Lin. Efficient Single-Source Shortest Path and Distance Queries on Large Graphs. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (**SIGKDD**), pages 998-1006, 2013.

Paper under submission:

- **Sibo Wang**, Xiaokui Xiao, Yin Yang, Wenqing Lin. Effective Indexing for Approximate Constrained Shortest Path Queries on Large Road Networks.
- Youze Tang, **Sibo Wang**, Xiaokui Xiao, Yin Yang, Zengxiang Li. HubPPR: Effective Indexing for Approximate Personalized PageRank.

For the above publications, we present an initial study on the impact of node ordering for *Hierarchical Hub Labelling* in our paper titled "Reachability Queries on Large Dynamic Graphs: A Total Order Approach". We then further present a more detailed study in Chapter 3, and will extend it to a journal version. The paper titled "Efficient Route

Planning on Public Transportation Networks: A Labelling Approach” focuses on the shortest path queries on timetable graphs as elaborated in Chapter 4. Finally, the paper titled ”Effective Indexing for Approximate Constrained Shortest Path Queries on Large Road Networks” works on the constrained shortest path queries on road networks as shown in Chapter 5.

References

- [1] <https://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds>.
- [2] 9th DIMACS Implementation Challenge. <http://www.dis.uniroma1.it/challenge9/download.shtml>.
- [3] Apple iOS Maps. <http://www.apple.com/ios/maps>.
- [4] Bing Maps. www.bing.com/maps.
- [5] Google Maps. www.google.com/maps.
- [6] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35, 2012.
- [7] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.
- [8] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360, 2013.
- [9] S. Aljubayrin, J. Qi, C. S. Jensen, R. Zhang, Z. He, and Z. Wen. The safest path via safe zones. In *ICDE*, pages 531–542, 2015.
- [10] M. A. Babenko, A. V. Goldberg, H. Kaplan, R. Savchenko, and M. Weller. On the complexity of hub labeling (extended abstract). In *MFCS*, pages 62–74, 2015.
- [11] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger. Fast routing in very large public transportation networks using transfer patterns. In *ESA*, pages 290–301, 2010.
- [12] 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. *CoRR*, abs/1504.05140, 2015.

REFERENCES

- [13] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *ALENEX*, volume 9, 2009.
- [14] R. Bauer, D. Delling, and D. Wagner. Experimental study of speed up techniques for timetable information systems. *Networks*, 57(1):38–52, 2011.
- [15] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379–394, 1989.
- [16] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.
- [17] A. Bernstein. Near linear time $(1 + \epsilon)$ -approximation for restricted shortest paths in undirected graphs. In *SODA*, pages 189–201, 2012.
- [18] F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen. Distance oracles in edge-labeled graphs. In *EDBT*, pages 547–558, 2014.
- [19] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
- [20] T. Breugem. Evaluating FPTASes for the multi-objective shortest path problem. Bachelor’s thesis, Erasmus University Rotterdam, Jul 2014.
- [21] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, pages 493–504, 2005.
- [22] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, pages 893–902, 2008.
- [23] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*, pages 193–204, 2013.
- [24] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, pages 193–204, 2008.
- [25] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

- [26] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.
- [27] K. L. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of mathematical analysis and applications*, 14(3):493–498, 1966.
- [28] B. C. Dean. *Continuous-time dynamic shortest path algorithms*. PhD thesis, Massachusetts Institute of Technology, 1999.
- [29] D. Delling. Time-dependent sharc-routing. *Algorithmica*, 60(1):60–94, 2011.
- [30] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *ESA*, pages 321–333. Springer, 2014.
- [31] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. F. Werneck. Graph partitioning with natural cuts. In *IPDPS*, pages 1135–1146, 2011.
- [32] D. Delling, B. Katz, and T. Pajor. Parallel computation of best connections in public transportation networks. *Journal of Experimental Algorithmics (JEA)*, 17:4–4, 2012.
- [33] D. Delling and G. Nannicini. Bidirectional core-based routing in dynamic time-dependent road networks. In *Algorithms and Computation*, pages 812–823, 2008.
- [34] D. Delling, T. Pajor, and R. F. F. Werneck. Round-based public transit routing. In *ALENEX*, pages 130–140, 2012.
- [35] D. Delling and D. Wagner. Pareto paths with sharc. In *SEA*, pages 125–136, 2009.
- [36] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. Intriguingly simple and fast transit routing. In *SEA*, pages 43–54, 2013.
- [37] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [38] Y. Disser, M. Müller-Hannemann, and M. Schnee. Multi-criteria shortest paths in time-dependent train networks. In *SEA*, pages 347–361, 2008.
- [39] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations research*, 17(3):395–412, 1969.

REFERENCES

- [40] F. Ergün, R. K. Sinha, and L. Zhang. An improved FPTAS for restricted shortest path. *Inf. Process. Lett.*, 83(5):287–291, 2002.
- [41] L. R. Ford and D. R. Fulkerson. *Flows in networks*. Princeton University Press Princeton, N.J, 1962.
- [42] L. Foschini, J. Hershberger, and S. Suri. On the complexity of time-dependent shortest paths. *Algorithmica*, 68(4):1075–1097, 2014.
- [43] S. Funke, A. Nusser, and S. Storandt. On k-path covers and their applications. *PVLDB*, 7(10):893–902, 2014.
- [44] S. Funke and S. Storandt. Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In *SOCS*, 2013.
- [45] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.
- [46] R. Geisberger. Contraction of timetable networks with realistic transfers. In *SEA*, pages 71–82, 2010.
- [47] R. Geisberger, M. Kobitzsch, and P. Sanders. Route planning with flexible objective functions. In *ALLENEX*, volume 10, pages 124–137, 2010.
- [48] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.
- [49] A. Goel, K. G. Ramakrishnan, D. Kataria, and D. Logothetis. Efficient computation of delay-sensitive routes from one source to all destinations. In *INFOCOM*, pages 854–858, 2001.
- [50] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, pages 156–165, 2005.
- [51] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In *ALLENEX*, pages 129–143, 2006.
- [52] G. Y. Handler and I. Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293–309, 1980.

REFERENCES

- [53] P. Hansen. Bicriterion path problems. In *Multiple criteria decision making theory and application*, pages 109–127, 1980.
- [54] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations research*, 17(1):36–42, 1992.
- [55] 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.
- [56] H. B. H. III, M. V. Marathe, V. Radhakrishnan, and R. E. Stearns. The complexity of planar counting problems. *SIAM J. Comput.*, 27(4):1142–1167, 1998.
- [57] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.
- [58] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *PVLDB*, pages 768–779, 2004.
- [59] R. Jin, N. Ruan, S. Dey, and J. X. Yu. Scarab: scaling reachability computation on large graphs. In *SIGMOD*, pages 169–180, 2012.
- [60] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *PVLDB*, 6(14):1978–1989, 2013.
- [61] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.
- [62] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, pages 595–608, 2008.
- [63] N. Jing, Y. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Trans. Knowl. Data Eng.*, 10(3):409–432, 1998.
- [64] H. C. Joksche. The shortest route problem with constraints. *Journal of Mathematical analysis and applications*, 14(2):191–197, 1966.
- [65] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Trans. Knowl. Data Eng.*, 14(5):1029–1046, 2002.

REFERENCES

- [66] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE*, page 10, 2006.
- [67] F. A. Kuipers, A. Orda, D. Raz, and P. V. Mieghem. A comparison of exact and *epsilon*-approximation algorithms for constrained routing. In *NETWORKING*, pages 197–208, 2006.
- [68] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. *Geoinformation*, 22:219–230, 2004.
- [69] A. Likhyan and S. J. Bedathur. Label constrained shortest path estimation. In *CIKM*, pages 1177–1180, 2013.
- [70] D. H. Lorenz and D. Raz. A simple efficient approximation scheme for the restricted shortest path problem. *Oper. Res. Lett.*, 28(5):213–219, 2001.
- [71] K. Mehlhorn and M. Ziegelmann. Resource constrained shortest paths. In *ESA*, pages 326–337, 2000.
- [72] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up dijkstra’s algorithm. In *WEA*, pages 189–202, 2005.
- [73] J. Monnot. A simple approximation algorithm for WIS based on the approximability in k . *European Journal of Operational Research*, 171(1):346–348, 2006.
- [74] M. Müller-Hannemann and M. Schnee. Paying less for train connections with MOTIS. In *ATMOS*, 2005.
- [75] M. Müller-Hannemann and M. Schnee. Efficient timetable information in the presence of delays. In *Robust and Online Large-Scale Optimization*, pages 249–272. Springer, 2009.
- [76] M. Müller-Hannemann, F. Schulz, D. Wagner, and C. Zaroliagis. Timetable information: Models and algorithms. In *Algorithmic Methods for Railway Optimization*, pages 67–90, 2007.
- [77] K. Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1):154–166, 1995.

- [78] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional a^* search for time-dependent fast paths. In *SEA*, pages 334–346, 2008.
- [79] G. Nannicini, D. Delling, D. Schultes, and L. Liberti. Bidirectional a^* search on time-dependent road networks. *Networks*, 59(2):240–251, 2012.
- [80] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithms*, 12:2–4, 2008.
- [81] M. N. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *PVLDB*, 4(2):69–80, 2010.
- [82] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.
- [83] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, pages 568–579, 2005.
- [84] P. Sanders and D. Schultes. Engineering highway hierarchies. In *ESA*, pages 804–816, 2006.
- [85] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*, pages 1009–1020, 2013.
- [86] A. J. V. Skriver and K. A. Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers & OR*, 27(6):507–524, 2000.
- [87] S. Storandt. Route planning for bicycles-exact constrained shortest paths made practical via contraction hierarchy. In *ICAPS*, volume 4, page 46, 2012.
- [88] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
- [89] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.
- [90] G. Tsaggouris and C. D. Zaroliagis. Multiobjective optimization: Improved FPTAS for shortest paths and non-linear objectives with applications. *Theory Comput. Syst.*, 45(1):162–186, 2009.

REFERENCES

- [91] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD Conference*, pages 913–924, 2011.
- [92] V. V. Vazirani. *Approximation Algorithms*. Springer, 2002.
- [93] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.
- [94] A. Warburton. Approximation of pareto optima in multiple-objective, shortest-path problems. *Operations research*, 35(1):70–79, 1987.
- [95] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: An independent permutation labeling approach. *PVLDB*, 7(12), 2014.
- [96] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *PVLDB*, 7(9):721–732, 2014.
- [97] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *PVLDB*, 5(5):406–417, 2012.
- [98] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pages 1601–1606, 2013.
- [99] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.
- [100] Y. Yoshida. Almost linear-time algorithms for adaptive betweenness centrality using hypergraph sketches. In *SIGKDD*, pages 1416–1425, 2014.
- [101] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*, pages 1323–1334, 2014.