Efficient Connectivity Analysis and Path Management in Massive Graphs

by

Yilun Huang

A THESIS SUBMITTED IN FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

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

Jun, 2023

CERTIFICATE OF ORIGINAL AUTHORSHIP

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

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

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

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

Signature:

Date: 12/06/2023

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my principal supervisor Prof. Ying Zhang, for his invaluable guidance, support, and encouragement throughout my research career. His expertise and profound insights have shaped my research endeavors, and I am immensely thankful for the time and effort he has dedicated to my academic development. Even during the challenging times of the pandemic, Prof. Zhang's mentorship provided me with the strength to overcome obstacles and persevere. His timely advice and kind encouragement have had a profound impact on my life choices and personal growth. I extend my heartfelt appreciation to Prof. Ying Zhang for his exceptional mentorship and invaluable contributions that have made a lasting impact on my academic and personal journey.

Secondly, I would like to express my sincere gratitude to my co-supervisor, A/Prof. Lu Qin. Throughout my research journey, he has consistently provided invaluable insights and guidance on my chosen research topics. A/Prof. Qin's remarkable work efficiency has instilled in me the belief that I can conduct thorough research through difficult and seemingly insurmountable obstacles during my PhD study. His unwavering confidence in tackling complex research problems has inspired me to constantly think critically and push myself beyond boundaries. I consider myself privileged to have had the opportunity to collaborate with A/Prof. Lu Qin, benefiting from his extensive knowledge and expertise

in the field.

Thirdly, I would like to acknowledge my co-supervisor Dr. Dong Wen for his insightful feedback and constructive criticism, which have helped me to refine my research and strengthen my arguments. His expertise in the field has been invaluable, and I feel grateful to have had the opportunity to learn from him.

Fourthly, I would like to thank Prof. Xuemin Lin, Prof. Wenjie Zhang, Dr. Longbin Lai, and Dr. Zhengping Qian for their invaluable support throughout the research presented in this thesis. I am deeply grateful to Prof. Lin and Prof. Zhang for offering an interesting but rigorous research environment, which has played a pivotal role in nurturing my skills and expanding my horizon. I also appreciate Dr. Lai and Dr. Qian's insightful advice on my research works, which have helped me to refine my ideas and methods.

I would also like to thank Dr. Xing Feng, Dr. Fan Zhang, Dr. Hanchen Wang, Dr. Wanqi Liu, Dr. Conggai Li, Dr. Mingjie Li, Dr. Dian Ouyang, Dr. Wentao Li, Dr. Bohua Yang, Dr. Junhua Zhang, Mr. Yuanhang Yu, Mr. Peilun Yang, Mr. Yuxuan Qiu, Mr. Rong Hu, Mr. Lantian Xu, Dr. Xubo Wang, Dr. Kongzhang Hao, Dr. You Peng, Dr. Xiaoshuang Chen, Dr. Zhengyi Yang, Dr. Kai Wang, Mr. Shunyang Li, and Mr. Xiande Xu for giving me a pleasant time when working with them.

Finally, I would also like to acknowledge my father Mr. Yuequan Huang, and my mother Ms. Miaozhen Chen, who provide me with selfless love and endless encouragement. I would like to extend a special thanks to my beloved kitty Power, whose soft purrs and gentle presence provided a welcome distraction during moments of stress. I also thank all the love and support from other relatives and friends.

Abstract

Graph analysis serves as a cornerstone in numerous real-world applications, spanning domains like social networks, knowledge graphs, fraud detection, and trajectory monitoring. In this thesis, we focus on three fundamental problems in graph analysis: reachability queries, path enumeration, and path compression, each nuanced by distinct constraints within expansive graphs. Our work advances a suite of innovative and efficient strategies to address these challenges.

The first problem considers the reachability between nodes in a temporal graph, encapsulating connectivity insights. Specifically, vertices in a temporal graph are connected by edges with time stamps, and there could be multiple edges connecting two nodes. Existing approaches assume a time-respecting path, which fails to capture relationships between entities in the same group or activity. To address this, we propose the span-reachability model, which identifies relationships between entities in a given time period. We adopt a two-hop cover approach and propose an index-based method to efficiently answer span-reachability queries.

In the second problem, we further explore the paths between given nodes to explore the connectivity in detail. We delve into the exploration of hop-constrained time interval s-t path enumeration in temporal graphs. To effectively tackle this challenge, we first propose a data structure named TIPST bundle to avoid repeated visits and maintain intermediate results in a compact way. It leads to a significant reduction in space complexity for each bundle. We then propose an algorithm named TDDL-DFS leveraging the distance labels. In the search process, these labels are dynamically updated to prune the fruitless branches, which is a is a polynomial delay algorithm.

The third problem deals with path compression in large graphs to handle the considerable scale of path sets as query results. The challenge comes with

the numerous number of regularly generated paths in networks. We propose the Overlap-Free Frequent Subpath (OFFS) compression method that allows retrieval of any individual path while compressing the overall size. We leverage a lookup table to match frequent common subpaths to supernodes and adopt a bottom-up framework to construct the lookup table in given iterations. Each path is shortened by replacing subpaths with corresponding supernodes in the table. Several optimizations are proposed to improve the compression ratio and speed.

We conduct extensive experiments on real-world datasets to demonstrate the effectiveness and efficiency of our proposed approaches. Our approaches outperform baseline methods significantly in terms of query answering time, compression ratio, and enumeration time.

PUBLICATIONS

- Dong Wen, **Yilun Huang**, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. "Efficiently answering span-reachability queries in large temporal graphs." *In the Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE).* (**Chapter 3**)
- **Yilun Huang**, Dong Wen, Peilun Yang, Lu Qin, Ying Zhang, and Wenjie Zhang. "Efficient Hop-Constrained Time Interval *s-t* Path Enumeration in Large Temporal Graphs" *under review.*(**Chapter 4**)
- **Yilun Huang**, Dong Wen, Longbin Lai, Zhengping Qian, Lu Qin and Ying Zhang. "Efficient and Effective Path Compression in Large Graphs" *In the Proceedings of the 2023 IEEE 39th International Conference on Data Engineering (ICDE).* (**Chapter 5**)

Contents

CERTIFICATE OF AUTHORSHIP/ORGINALITY ii

ACKNOWLEDGEMENTS iii

PUBLICATIONS iv

1 Introduction 1

2 Literature Review 17
- 2.1 Span Reachability in Temporal Graphs . . . 17
- 2.2 Time Interval Paths in Temporal Graphs . . . 18
- 2.3 Path Compression in Large Graphs . . . 21

3 Span Reachability in Temporal Graphs 24
- 3.1 Chapter Overview . . . 24
- 3.2 Preliminary . . . 25
- 3.3 Solution Overview . . . 27
  - 3.3.1 A Straightforward Online Approach . . . 27
  - 3.3.2 The Time Interval Labeling Index . . . 29
- 3.4 Index Construction . . . 31
  - 3.4.1 The Labeling Framework . . . 31
  - 3.4.2 Theoretical Analysis . . . 35
  - 3.4.3 Implementation . . . 36
- 3.5 Query Processing . . . 40
  - 3.5.1 Span-Reachability Query Processing . . . 40
  - 3.5.2 θ-Reachability . . . 44
- 3.6 Experiments . . . 47
  - 3.6.1 Span-Reachability Query Processing . . . 48
  - 3.6.2 Index Construction . . . 49
  - 3.6.3 θ-Reachability Query Processing . . . 54
- 3.7 Conclusion . . . 55

4 Time Interval Paths in Temporal Graphs **56**
4.1 Overview . . . 56
4.2 Preliminary . . . 57
4.2.1 Problem statement . . . 58
4.2.2 Straightforward Method . . . 59
4.3 Temporal Bundled Solution . . . 59
4.3.1 Motivation . . . 60
4.3.2 Bundled Time Interval DFS Approach . . . 61
4.3.3 Algorithm Description . . . 63
4.4 Dynamic Distance Label . . . 65
4.4.1 Motivation . . . 65
4.4.2 Distance Label . . . 66
4.4.3 Algorithm Description . . . 66
4.5 Analysis . . . 69
4.6 Experimental Study . . . 71
4.6.1 Experimental Setting . . . 71
4.6.2 Metrics . . . 73
4.6.3 Comparison with Baselines . . . 73
4.6.4 Effectiveness of Bundles . . . 75
4.6.5 Scalability . . . 77
4.7 Conclusion . . . 78

5 Path Compression in Large Graphs **79**
5.1 Overview . . . 79
5.2 Preliminary . . . 80
5.2.1 Problem Statement . . . 80
5.3 The Framekwork . . . 81
5.3.1 Overview . . . 81
5.3.2 Decompression and Compression . . . 81
5.4 Frequent Subpaths . . . 82
5.5 Supernode Table Compression . . . 87
5.5.1 Match Collision Issue . . . 87
5.5.2 Identifying Supernodes by Merge and Expansion . . . 89
5.5.3 Practical Implementation . . . 92
5.5.4 Possible Optimizations . . . 96
5.6 Complexity Analysis . . . 96
5.6.1 Time Complexity . . . 96
5.6.2 Space Complexity . . . 97
5.7 Experimental Study . . . 98
5.7.1 Experimental Setting . . . 98
5.7.2 Metrics . . . 100

5.7.3 Impacts of Parameters 101
5.7.4 Comparison with Baselines 104
5.7.5 Retrieval and Scalability 105
5.8 Conclusion 107

6 EPILOGUE **108**

List of Figures

1.1 A temporal graph $\mathcal{G}$ where each number represents the timestamp of the edge below 4
1.2 A simplified transaction in a purchase action 12
1.3 An example in Alibaba Cloud Service 13

3.1 The projected static graph of $\mathcal{G}$ in the time interval $[2, 4]$ 25
3.2 The data structure used to store $\mathcal{L}_{in}(v_4)$ and $\mathcal{L}_{out}(v_6)$ 42
3.3 Performance of span-reachability query processing 49
3.4 Index Size 50
3.5 Indexing Time 50
3.7 Scalability of index construction 52
3.6 Varying ϑ of TILL-Construct* 53
3.8 Performance of θ-reachability query processing 55

4.1 Corresponding projected graph and static graph 58
4.2 An example of time interval paths 59
4.3 An example of time interval paths to TIPST bundles 62
4.4 Comparison with baselines on running time and throughput . . . 74
4.5 Changes of compression ratio with hop-constraint k and interval size t 75
4.6 Impacts of hop-constraint k on throughput 77
4.7 Impacts of time interval size t on throughput 77

5.1 Compression with tables of frequent subpaths 87
5.2 Impacts of parameters i (a–d) and k (e–h) on compression speed and compression ratio 102
5.3 Comparison with baselines on compression ratio and compression speed 103
5.4 Decompression comparison and scalability test 105

Chapter 1

Introduction

The graph models are widely used to represent the relationships between entities, which play an essential role in both academia and industry. With the rapidly growing number of graphs being collected and maintained every day, graph analysis has emerged as a crucial research field in various applications such as social networks, database management systems, fraud detection, and trajectory monitoring[49, 68, 104]. However, we observe that the challenges come with the considerable data scale and complex side information of graphs[61, 48, 100, 67]. In this thesis, we undertake a comprehensive examination of the issue pertaining to the connectivity of two nodes. Reachability queries involve determining the presence of a path connecting two specified nodes within a given graph, thus affording a succinct evaluation. Addtionally, path enumeration provides furhter details beyond reachability, entailing the discovery of all feasible pathways between two nodes. The inherent complexity of this challenge is further exacerbated by the involvement of temporal data, thereby introducing an additional stratum of intricacy. Furthermore, the effective management of path sets assumes paramount significance, especially as the magnitude of data escalates due to the recurring nature of these queries. To tackle these challenges, we propose a

novel span-reachability model that identifies relationships between entities in a given time period and develop a static index-based method to efficiently answer span-reachability queries. For the path enumeration problem, we put forward a strategic bundling approach, a distance-based indexing scheme to achieve polynomial delayed searchs and yield compact results. To handle the ever-growing number of paths as query results, we propose the compression method based on Overlap-Free Frequent Subpath (OFFS), which uses a tailored dictionary to compress paths. This thesis provides effective and efficient solutions to connectivity queries, path compression, and constrained path enumerations in graphs. We provide the detailed background, motivations, and contributions of each problem as follows.

Span reachability in temporal graphs. Determining the existence of a path between two query vertices is a crucial problem in network analysis, known as computing the reachability between vertices. Existing works have extensively studied this problem, resulting in numerous algorithms that offer efficient solutions [6, 23, 26, 28, 85, 103, 105, 99, 53, 97, 91]. Reachability queries have wide-ranging applications in various domains, including road networks, social networks, collaboration networks, PPI (protein-protein interaction) networks, and XML and RDF databases. Our study specifically benefits several real-world applications:

- *Biology Analysis.* In PPI networks, identifying whether two proteins participate in the same biological process or molecular function is critical [54]. When monitoring protein activities over a specific period, two proteins that belong to the same biological organization may not have direct time-respecting paths but may be controlled by or interact with a common protein. Our model can identify relationships between these proteins.

- *Security Assessment & Recommendation.* In the context of assessing secu-

rity, it is important to determine whether a particular person is associated with a known terrorist [13]. When organizing a terrorist activity, there may be several phone calls among suspects in a short time period. Finding a time-respecting path from the known terrorist to others can be challenging, especially when not all people in the organization take orders from the terrorist. Our model can capture related suspects of a targeted terrorist. Similarly, in social networks, our model can detect whether two users are involved in a social group during significant events such as the FIFA World Cup and the Olympic Games.

- *Money Transaction Monitor.* In e-commerce platforms and bank systems, we often have a graph in which each vertex represents a user account and each edge with a timestamp represents a money transaction between two user accounts. Detecting whether there is a path between two user accounts is critical for monitoring money transactions or illegal financial activities such as money laundering and fake transactions. Normally, a series of money transactions should follow an increasing order of timestamps. However, skilled users may borrow untraceable money to complete a transfer and avoid monitoring. For example, an account in the transaction path may transfer money to the next account in advance and receive the money from the prior account later. The existing order-dependent reachability model cannot capture this activity, but our model can be used by setting a specified time interval.

In these real-world applications, graph edges are associated with temporal information, adding an extra layer of complexity to the problem. Therefore, it is of significance to further dig into the problem of reachability in temporal graphs.

The problem of vertex reachability in temporal graphs has been addressed in existing literature using the concept of time-respecting paths. An existing

method to model the temporal reachability is based on the concept of time-respecting paths [50, 57, 51]. Specifically, a vertex u reaches v if there exists a path connecting u and v such that the times on the path follow a non-decreasing order. For example, in the temporal graph $\mathcal{G}$ of Figure 1.1, v_6 reaches v_{10} since there exists a path $\{\langle v_6, v_2, 5\rangle, \langle v_2, v_1, 6\rangle, \langle v_1, v_{10}, 8\rangle\}$ connecting them and the times $5, 6, 8$ are in a non-decreasing order. Semertzidis et al. [86] also model the temporal reachability that two vertices u, v are reachable if there exists path connecting them and the times of all edges in the path are consistent, i.e., u, v are reachable in a snapshot of the temporal graph at a given time.

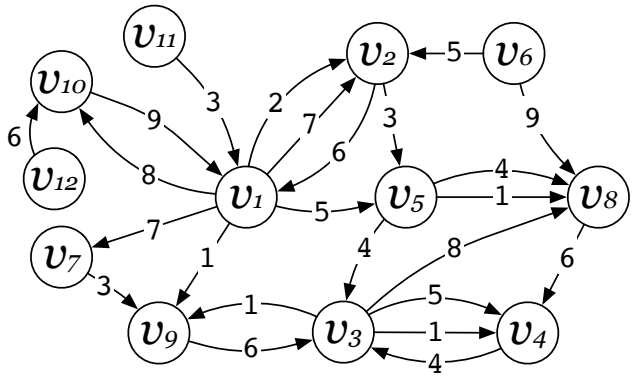


Figure 1.1: A temporal graph $\mathcal{G}$ where each number represents the timestamp of the edge below

Unfortunately, in many scenarios of temporal graph mining, we may only focus on the relationship between vertices in the projected graph of a small time interval without addressing any order limitation in the edge sequence. Here, the projected graph is the static graph containing all edges at times falling in the interval. For example, Gurukar el al. [46] compute the communication motifs in temporal graphs and show that two edges sharing a common vertex are related if the difference of their timestamps is very small. Authors in [96, 65] compute the community structures called Δ-clique and (θ, k)-persistent-core, respectively,

in temporal graphs. Their models require that the resulting subgraph satisfies some structural properties (e.g. vertex degree threshold) in the projected graph of a time interval. The aforementioned two reachability models are too strict and might fail to capture entity relationships in these scenarios.

Therefore, we introduce a new span-reachability model. Given a temporal graph and a time interval $\mathcal{I}$, a vertex u span-reaches v if u reaches v in the projected graph of $\mathcal{I}$. We investigate the efficient answering of the span-reachability query for an arbitrary pair of vertices and any possible time interval. Take $\mathcal{G}$ of Figure 1.1 for instance, we have v_1 span-reaches v_8 in the time interval $[3, 5]$, since there exists a path $\{\langle v_1, v_5, 5\rangle, \langle v_5, v_8, 4\rangle\}$ from v_1 to v_8 in the projected graph of $[3, 5]$. Besides, we also study a θ-reachability problem, which is a generalized version of span-reachability. Given a time interval $\mathcal{I}$ and a length threshold θ, two vertices are θ-reachable in $\mathcal{I}$ if they are span-reachable in a θ-length subinterval of $\mathcal{I}$. Taking the above case of monitoring money transactions, a more general task is to identify whether there exists a transaction chain between two accounts finished in a short period over a long monitoring period. Note that when the length of query interval equals to θ, θ-reachability is equivalent to span-reachability. The other special case is that when θ is 1, it is equivalent to the disjunctive historical reachability model studied in [86].

Time interval paths in temporal graphs.

Graphs play a crucial role in modeling relationships between entities across various domains, including social networks, road networks, web graphs, biological networks, and collaboration networks[49, 68, 104]. Among the essential topics in graph analytics[24, 34, 22], investigating the connectivity between two given vertices in a graph is of paramount importance. Specifically, a hop-constrained s-t path enumeration query[79] deals with a graph $\mathcal{G}$, a source vertex s, a target vertex t, and a hop constraint k, aiming to identify all simple paths from s to

t where the number of hops in each path does not exceed k. Please note that in this paper, we exclusively focus on simple paths since paths with loops (i.e., containing repeated vertices) are less intriguing and could significantly inflate the total number of s-t paths.

Remarkably, in real-world applications, graphs commonly include temporal information associated with their edges. However, most existing algorithms for hop-constrained s-t path enumeration focus on simple graphs and assume queries without additional constraints. This approach neglects the importance of analyzing the temporal dynamics within a network, which is becoming increasingly significant. While the problem of hop-constrained s-t simple path enumeration has received considerable attention and has been thoroughly studied, the temporal dynamics of networks are poorly captured by their static structures[61]. This limitation hinders more real-world applications, where graphs are often time-stamped, and path enumeration involves temporal constraints. Temporal graphs find a wide range of applications, particularly in knowledge graphs, social networks, transportation systems, and financial markets, where information flows over time, and relationships between entities constantly evolve. Unlike traditional graphs, temporal graphs offer a more detailed representation of system dynamics, empowering researchers to analyze not only connectivity but also the temporal characteristics of relationships between entities. Gaining insights into the changes in these relationships over time opens up new avenues for valuable insights and informed decision-making across diverse domains.

Motivation. In this paper, our focus lies in studying the problem of hop-constrained time interval s-t path enumeration. An existing method to model the temporal connectivity is based on the concept of time-respecting paths[50, 57, 51]. Specifically, given a source vertex s, a target vertex t, it aims to enumerate all paths from s to t in such a way that the timestamps along

the path adhere to a non-decreasing order. For instance, considering the same temporal graph $\mathcal{G}$ depicted in Figure 1.1, from v_6 to v_{10}, there exists a path $\{(v_6, v_2, 5), (v_2, v_1, 6), (v_1, v_{10}, 8)\}$ connecting them and the times $5, 6, 8$ are in a non-decreasing order. This model proves useful in representing how influence broadcasts over time.

Unfortunately, in various temporal graph mining scenarios, the main focus revolves around understanding vertex relationships within a projected graph over specific time intervals. In such cases, the order of the edge sequence becomes less significant and is often overlooked. The projected graph represents a snapshot that consolidates all edges occurring within the specified interval. For instance, Gurukar et al. [46] explore communication motifs in temporal graphs, emphasizing that two edges with a common vertex are related if their timestamp difference is minimal. Similarly, researchers in [96, 65] delve into community structures in temporal graphs. Their models require that the resulting subgraph adheres to specific structural properties, such as a vertex degree threshold, within the projected graph of a time interval. Moreover, real-life applications often introduce a hop constraint when enumerating s-t paths[79]. This constraint arises from the significant reduction in vertex relationship strength with an increasing number of hops. It is widely known that in practical situations, the number of paths can grow exponentially in correlation with the number of hops in real-life graphs. Therefore, imposing a hop constraint is a crucial means to manage complexity and ensure relevant results in practice.

Time interval paths. In this paper, we study the problem of hop-constrained time interval s-t path enumeration. Given a directed temporal graph, a source vertex s, a destination vertex t, a hop constraint k, and a temporal interval $[t_s, t_e]$, we aim to efficiently enumerate all time interval paths from s to t only using edges with timestamps in the given interval $[t_s, t_e]$ with number of hops

not larger than k.

Example 1. *In the temporal graph $\mathcal{G}$ of Figure 1.1, suppose the time interval is $[3, 5]$ and the hop constraint is k, there exists a time interval path $\{(v_1, v_5, 5), (v_5, v_8, 4)\}$ from v_1 to v_8 with all timestamps falling into $[3, 5]$.*

Applications. The proposed model offers an effective means to delve into the intricate relationships between entities, allowing us to concentrate on item interactions within specific time periods. Several real-world applications stand to benefit significantly from this study. For example:

- *Knowledge Graph Completion.* Knowledge graphs are key to various applications, including recommendation systems, search engines, and question-answering. Due to their inherent incompleteness, the task of knowledge graph completion, which involves predicting missing relations, holds paramount importance. In recent years, Path Ranking (PR) algorithms (e.g., [89, 90]) have garnered increasing attention. These algorithms enumerate paths between entities in a knowledge graph and employ these paths as features to train models for predicting missing facts [72]. An important consideration is that lengthy paths may not effectively capture the relationship between two entities, as the strength of their relation tends to decline significantly with the number of hops (i.e., interactions). Moreover, the emergence of temporal knowledge graphs[40] adds a new dimension of significance, where temporal constraints play a crucial role.

- *Biology Analysis.* Protein-Protein Interaction (PPI) networks are a type of biological network that represents physical interactions between proteins in a cell. One of the key tasks in PPI analysis is to determine whether two proteins participate in the same biological process or molecular function [54]. When monitoring protein activities within a specific time frame, it

is possible that two proteins belonging to the same biological organization might not have direct time-respecting paths. However, they could be influenced or interact with a common protein. Our model offers a valuable approach to enumerate and analyze the relationships between these proteins. This enables researchers to gain a comprehensive understanding of the underlying molecular mechanisms in biological processes.

- *E-Commerce Merchant Fraud Detection.* In e-commerce platforms and bank systems, we encounter graphs where each vertex represents a user account, and edges with timestamps denote money transactions between accounts. In monitoring financial activities, such as money laundering and fake transactions[10], it becomes crucial to detect the existence of a path between two user accounts. Typically, a series of money transactions should follow an increasing order of timestamps. However, skilled malicious users may exploit techniques like borrowing untraceable money to evade monitoring. For instance, an account in the transaction path might transfer money to the next account in advance and receive funds from the prior account at a later time. The existing order-dependent reachability model fails to capture such activities, but our model effectively addresses this by setting a specified time interval. By incorporating temporal constraints, we gain a more comprehensive view of money transaction patterns and bolster the detection of illicit financial practices.

Challenges. The primary challenge in solving this problem lies in the vast search space, even for a small k value, as the number of paths may exponentially increase with respect to k. Overcoming this complexity is crucial to finding practical solutions. Additionally, the temporal constraints add another layer of complexity, demanding efficient algorithms capable of processing substantial volumes of data swiftly. Handling the temporal properties of edges, including their

start and end times, poses an additional challenge in ensuring the accuracy of the results. Furthermore, maintaining the temporal results proves problematic, as their size grows linearly with the number of temporal stamps and exponentially with k. Therefore, the development of efficient and accurate algorithms for hop-constrained time interval s-t path enumeration and path maintenance in temporal graphs is of paramount importance and possesses significant potential to impact a wide array of real-world applications.

DFS-based Solution. To address this problem, a straightforward method is to directly process a hop-constrained time interval depth-first search from the source vertex and output the time interval path when meeting the terminate vertex. While this method works in finding the desired paths, it exhibits considerable time and space complexity, making it impractical for large graphs.

To achieve efficient query processing and enhance scalability, we propose an online solution called Temporal Dynamic Distance Labels Depth-First Search (TDDL-DFS), which leverages temporal bundles and dynamic distance labels. Specifically, our approach begins by compacting interval paths based on snapshots of temporal graphs, organizing the timestamps on edges, and introducing the notion of TIPST bundles to avoid redundant searches. This strategic approach effectively reduces unnecessary computations, leading to improved search efficiency. Additionally, given a query $q(s, t, t_e, t_e)$, for each vertex u in the temporal graph, we maintain a distance label $d'_s(u)$ from the source vertex s and a distance label $d_t(u)$ to the terminate vertex respectively. Given a query interval $[t_s, t_e]$, we traverse the search space starting from s, determining if reaching t is possible by considering both the distance label and the timestamps on the edges. We output the result and decrease the distance label if reaching t in less than k hops. Alternatively, if reaching t within k hops is not feasible, we increase the distance label to block that particular search branch. This combination of

temporal bundles and dynamic distance labels significantly enhances the performance and efficiency of our approach for Time Interval Path Enumeration queries in temporal graphs.

Contributions. Our principal contribution in this paper is summarized as follows.

- We present a novel temporal path model based on time intervals, which effectively illustrates the intersections between entities within specific periods of a temporal graph. Utilizing this model, we introduce a new data structure called **TIPST bundles** to prevent redundant visits and store intermediate results in a more efficient manner during the search process.

- We develop a novel algorithm called **TDDL-DFS**, which leverages the hierarchical structure of the temporal graph and employs a distance-based index to efficiently prune the search space. Our theoretical analysis demonstrates that TDDL-DFS is a polynomial delay algorithm with $O(km_s \log \theta)$ time per output, where m_s is the number of edges in the snapshot and k is the hop constraint. The overall time complexity is $O(km_s\delta \log \theta)$, and the space complexity is $O(k\delta)$, where δ represents the number of output bundles.

- We perform comprehensive experiments on real-world graphs with diverse characteristics. The experimental results demonstrate the scalability and efficiency of our proposed algorithm in handling hop-constrained time interval s-t path enumeration queries in temporal graphs. Specifically, we demonstrate that our TDDL-DFS algorithm outperforms the baseline significantly, with a speed improvement of up to three orders of magnitude. Furthermore, we observe that TIPST bundles play a pivotal role in reducing space costs substantially, with potential reductions of up to five orders

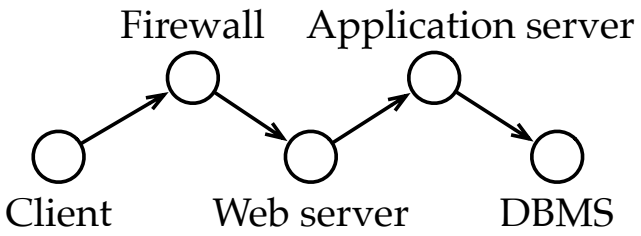


Figure 1.2: A simplified transaction in a purchase action

of magnitude.

Path compression in large graphs. The graph data model is widely used in various applications, including social analysis, e-commerce transactions, the World Wide Web, cybersecurity, and protein interactions, to capture complex relationships between entities. In a graph, paths refer to a sequence of vertices, where each adjacent pair of vertices is connected by an edge. Many graph-based analysis tasks generate paths through various queries, such as shortest paths on road networks, planning routes in public-transaction networks, routing records in telephone networks, or message transmissions in social networks. Inevitably, a set of paths needs to be recorded in these applications. Real-life graphs usually contain millions or billions of edges and vertices, and sophisticated analytics over such big graphs can produce an abundance of paths. However, many overlaps among paths can make the total path size significantly greater than the graph size, making it challenging to store paths effectively and reduce space usage. Therefore, it is essential to find effective methods for reducing the space cost of these paths.

To illustrate the above concept, take the platform of Alibaba Cloud for instance, where a graph model is used to facilitate daily analysis and monitoring. In an e-commerce system as shown in Figure 1.2, whenever a buyer submits an order, a network message is sent to a series of middle-tier servers via the Internet, which are represented by vertices in the graph model. The communication be-

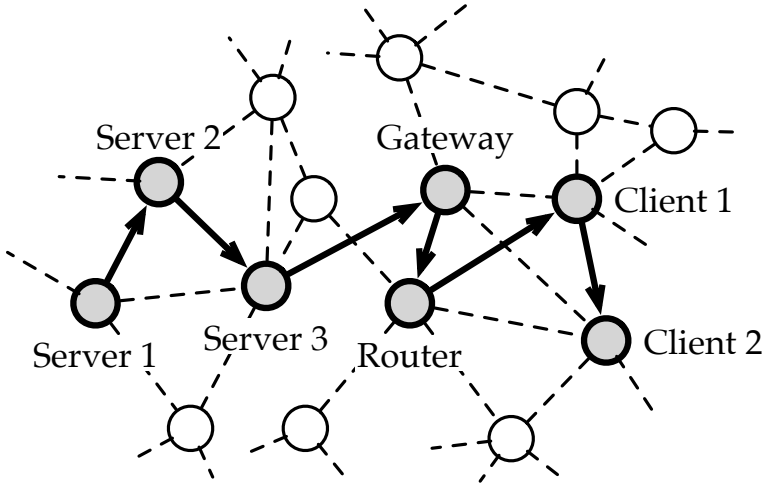


Figure 1.3: An example in Alibaba Cloud Service

tween the servers is denoted by an edge, and a purchase action can be described by a path in the graph. Each step in the path may involve multiple machines, and maintaining a transaction path for a user action helps locate affected users when an anomaly is detected in a specific machine.

A more general case is illustrated in Figure 1.3. Many devices are deployed on the platform Alibaba Cloud to support a wide range of services. Network messages between machines are recorded as IP pairs in system logs. An IP address may match a host server, a portable device, a client machine, or a gateway. The topological relationship between IPs in a certain task is denoted as IP hops (path). Two common instances of utilizing IP paths in daily system maintenance are identifying affected nodes and locating anomalies. In the former case, when there is an anomaly in a host server, it is necessary to identify all affected segments for altering routing plans. By retrieving all indexed IP paths containing the issue node, we can fetch all affected IP nodes accurately. In the latter case, when network issues are reported by customers, we need to investigate all intermediate IP nodes of network transactions for troubleshooting, which can be done by collecting all IP paths with given terminals.

Storing IP paths facilitates many services in Alibaba Cloud, while the number of paths can be numerous. For instance, a table to record IP hops usually accumulates more than 50 GB of data from nearly one million data transmissions in one day. There are massive data to be collected by more tables every day.

One may try to organize all paths as a whole and apply generic compression methods[30, 2, 1]. However, they are designed for general data, and their compression quality for path compression is poor compared with our tailored approach according to our experiments. In addition, generic compression methods do not support decompressing part of data, which is an essential need in aforementioned applications. One may also attempt to divide paths into several fine-granularity blocks and compress them individually. However, this approach leads to a significant drop in compression performance.

Graph summarization is another related work that aims to reduce a large graph to a smaller one without sacrificing the correctness of certain graph queries. The lossless graph summarization aims to reduce a big graph to a smaller one, without sacrificing the correctness of certain graph queries. In contrast, we aim to compress a set of paths so that each path can be retrieved individually. To apply the graph summarization techniques, an input graph is expected. An intuitive idea is to generate a graph using all edges in the set of paths. However, even never compressing/summarizing the graph, we cannot correctly identify every path without side information. Take two paths, $\{a, b, c\}$ and $\{d, b, e\}$, for instance. We unearth a star-like graph with them, where the vertex b is the center with four neighbors. In this case, even if starting from a to recover the path $\{a, b, c\}$, we cannot avoid spurious answers as there are four neighbors of b. Namely, we fail to recover paths from the graph correctly and the graph summarization techniques do not work in our case.

Therefore, we propose a compression method called Overlap-Free Frequent

Subpath (OFFS) to address the challenge of the considerable data scale of IP paths while allowing for retrievals of any individual path. Our approach involves building a lookup table to match a series of frequent common subpaths to supernodes. Each path is then shortened by replacing subpaths with corresponding supernodes in the table. We adopt a bottom-up framework to construct the lookup table in given iterations and propose several optimizations to improve the compression ratio and speed.

Contributions. To sum up, the principle contributions of this thesis composed of the aforementioned 3 works are summarized as follows:

- We introduce a span-reachability model and investigate a generalized version called θ-reachability. Based on that, we propose an index-based method using two-hop cover, optimizing index construction and θ-reachability query processing efficiency. Addtionally, our experiments on 17 real-world datasets validate the effectiveness of our optimizations and the efficiency of our solutions.

- We pioneer the study of this problem, enumerating hop-constrained s-t paths in a given interval. Our algorithm introduces bundling strategy to prevent redundant visits and store intermediate results in a more efficient manner. Then we develop a novel polynomial delayed algorithm called **TDDL-DFS**, which employs a distance-based index to efficiently prune the search space. Furthermore, comprehensive experiments on real-world graphs show the scalability and efficiency of our proposed algorithm.

- We implement a lightweight compression method, Overlap-Free Frequent Subpath (OFFS), for path sets. By leveraging a lookup table based on global information, we identify frequent common subpaths and compress them into supernodes, resulting in shorter paths and improved compres-

sion. To address match collision challenges, we devise an algorithm for constructing a supernode table that effectively handles overlapping subpaths. Extensive experiments on real datasets demonstrate the effectiveness and efficiency of our approach, achieving a high compression ratio in a shorter time compared to straightforward methods.

Chapter 2

Literature Review

In this chapter, we provide an overview of the related research in connectivity analysis and path management in large-scale graphs. We begin by reviewing reachability in temporal graphs, static graphs and dynamic graphs. Then we introduce existing works of path enumeration in simple graphs and complicated graphs, top-K shortest path enumeration, and reachability queries. Additionally, we review existing works related to path compression in simple graphs, including generic compression, lightweight compression, database management systems, trajectory simplification, string compression, and graph summarization.

2.1 Span Reachability in Temporal Graphs

Reachability in Temporal Graphs The time-respecting path is defined in [57] to model the reachability problem in temporal graphs. The similar concept is also studied using the terms journey [102, 37] or non-decreasing path [25]. Based on the time-respecting path, an index-based algorithm to efficiently answer the reachability problem in temporal graphs is studied in [101] and is improved in [110] for the distributed environment. The historical reachability problem is

studied in [86]. Given an interval $[t_1, t_2]$ and a pair of vertices u, v, the conjunctive historical reachability of u, v is true if for each possible $t \in [t_1, t_2]$, there exists a path connecting u, v and all timestamps in the path are t. The disjunctive historical reachability of u, v is true if there exists a timestamp $t \in [t_1, t_2]$ and a path connecting u, v in which all timestamps in the path are t [86]. Other mining problems in temporal graphs can be found in surveys [51, 21, 75].

Reachability in Static Graphs & Dynamic Graphs A large number of works have been done to design an index for answering the reachability query in static graphs [6, 23, 26, 28, 85, 103, 105, 99, 53, 97, 91]. Interested readers can find more details in surveys [108, 18]. Several works study the index maintenance in dynamic graphs [20, 85, 106, 111]. Estimating reachability based on random walks is studied in [87].

Note that even though the concept of the two-hop cover has been studied or used in several existing works [9, 4, 28, 98], our method is not a naive extension of existing techniques. Unlike the previous studies, our method is carefully tailored for temporal graphs. The proposed optimizations for index construction centers mainly on the relationships between different time intervals, such as containment and intersection.

2.2 Time Interval Paths in Temporal Graphs

Path Enumeration in Simple Graphs. Existing works on the problem of enumerating s-t paths[59, 78] aim at providing a succinct structure to represent paths. Rather than storing each path explicitly, these studies concentrate on efficient abstraction methods for path representation. However, their scalability is limited, typically only accommodating graphs with a few thousand vertices. Additionally, enumerating all s-t paths (or cycles) without the hop constraint is

a classical problem[56, 93, 16]. Recently, several pruning-based works [79, 44, 83] are proposed to answer the hop-constrained s-t queries $q(s, t, k)$ in simple graphs. These methodologies, which adopt a backtracking strategy based on a depth-first search framework, have all demonstrated performance of $O(km)$ per output. The state-of-the-art method, PathEnum[92], constructs a lightweight index to reduce the number of edges involved in the enumeration, thereby optimizing the running time by circumventing costly pruning operations. It is important to note that ignoring the temporal constraints and directly enumerating all paths before filtering the valid ones can result in significant time and space costs. This is because the complexity would be multiplied by $O(\theta^k)$. Such an approach becomes highly inefficient for large temporal graphs, making it impractical for real-world scenarios.

Path Enumeration in Complicated Graphs. In the context of distributed graphs, [48] proposes a new hybrid search paradigm that utilizes a divide-and-conquer approach to enumerate s-t paths. Additionally, [67, 66] address s-t path enumeration in labelled graphs and uncertain graphs, respectively, by imposing additional constraints in the query. However, in their cases, the side information is not explicitly included in the outputs, and the timestamps between vertices are ordered while labels are unordered, making it challenging to directly extend their methodologies to our scenario. Another recent work, 2SCENT[61], is proposed to enumerate all simple temporal cycles. Notably, this approach requires non-decreasing time order and does not involve hop constraints, distinguishing it from our problem setting. Overall, while these methodologies offer valuable insights into handling path enumeration in various graph settings, they do not directly address our specific scenario of hop-constrained time interval s-t path enumeration in temporal graphs.

Top-K Shortest Path Enumeration. Top-K shortest path enumeration

$q(s, t, K)$ is a valuable tool utilized in a wide range of applications, including route planning, network optimization, and logistics management. It facilitates efficient decision-making by providing multiple alternative paths ranked by their lengths. The query evaluation, conducted using Top-K shortest path algorithms [32, 39, 69, 8], aims to enumerate all paths shorter than a given threshold k. To achieve this, the parameter K is set to a sufficiently large value, allowing the process to terminate once the latest output path exceeds the length limit k. While these algorithms proficiently yield the results of the hop-constrained enumeration problem, it is pertinent to note that their result enumeration operates based on the ascending order of the result lengths, which consequently adds to the unnecessary computational overhead.

Reachability Queries. Reachability queries $q(s, t)$ in simple graphs aim to determine whether a path exists between two given vertices s and t. Existing methods[9, 6, 26, 103, 105, 99, 53, 91], such as pruned landmark labeling[9], involve constructing an index during an offline preprocessing step to efficiently handle subsequent queries. These methods evaluate queries using pre-computed information, and the index retains distance information to a set of vertices for each vertex in the graph, preserving global statistics. Additionally, there are plenty of works[57, 101, 110, 100] focusing on reachability queries in temporal graphs. Span reachability defined in [100] focuses on the reachability between s, t within a given time interval $[t_s, t_e]$ in a temporal graph. Their primary focus lies in achieving a balance between index construction costs and query efficiency. It is worth noting that the answer of reachability queries is a boolean value while the detailed paths are expected in our case.

2.3 Path Compression in Large Graphs

Generic Compression When it comes to compression scheme, a straightforward idea is generic compression, such as zlib[30], lz4[1], and zstd[2], which typically reduce file sizes using classic LZ77[112], LZ78[113], or their variants. However, these algorithms are typically used in default block mode, where they only consider duplication within the local block. Since the effective minimal size of a block, 1 KB, is much larger than a path, it is not efficient to use these algorithms for data retrieval in path compression.

Lightweight Compression In contrast to generic compression schemes, lightweight compression algorithms are less computationally expensive while achieving similar or even better compression rates in finer granularity. Five basic categories of lightweight techniques are identified in a recent survey [29]: frame-of-reference (FOR) [42, 114], delta coding (DELTA) [64, 84], dictionary compression (DICT) [114, 3], run-length encoding (RLE) [84, 3], and null suppression (NS) [84, 3]. These techniques focus on different data levels. FOR, DELTA, DICT, and RLE consider the logical data level, while NS addresses the physical level of bits or bytes. FOR represents each value as the difference to a given certain value, DELTA represents each value as the difference to its predecessors, and DICT replaces values as symbols in the dictionary. It is easier for NS to omit unnecessary zeros after representing smaller values with the aforementioned three schemes. Finally, RLE deals with continuous sequences of occurrences of the same value, i.e., runs, and each run is represented as its value and length.

Database Management System Modern systems usually use data compression engines to deal with the exponentially growing data volume. Several works [43, 36, 5] have aimed at performing read-only operations directly on compressed

data in database management systems. These works apply the idea of lazy decompression to avoid redundant compression and decompression operations. CompressDB by Zhang et al.[109] proposes an efficient technique to support write operations like updates, inserts, and deletes directly on compressed data. However, the finest granularity of compression and decompression of these methods is usually 1 KB to be compatible with existing systems, which is much larger than that of a path and is therefore unsuitable for path compression.

Trajectory Simplification With the development of location-based services, it is becoming increasingly costly to collect, store and transmit trajectory data. There have been works on line simplification to reduce the trajectory size while preserving necessary information. They can be classified into two classifications, batch solutions, like the classic Douglas-Peucker algorithm [31], Bellman algorithm [14] and Top-Down Time Ratio [74], and online solutions like Open Window[58] and Dead Reckoning [94]. It is worth noting that all of them exploit temporospatial metrics such as Synchronous Euclidean Distances [81], directions, speeds etc. to compress trajectories, which are not available in our case. In addition, lossy compression is usually acceptable for trajectories but does not work for our application.

String Compression String columns serve as necessary components in modern database systems and there are plenty of works on string compression. Classical string compression methods usually consider the special quality or operation of string, several works [11, 15] study the order-preserving compression of strings, [12] proposes a method specifically for compressing keys in B-trees and [35] explores string matching of LZ compressed string. Recent works focus on Single Instruction Multiple Data (SIMD), CPU-level parallelism, to improve the efficiency of compression. BRPFC [62] adopts strong dictionary compression based on Re-Pair Front Coding, and FSST [17] adopts a lightweight method based

on DICT. However, these methods usually come with hard-coded solutions or require specific constraints of order-persevering in compressed data like lexicographical order, which is inappropriate in our case.

Graph Summarization Graph summarization aims at reducing big graphs to smaller ones by contracting vertices or edges into more compacted supernodes (merged nodes) and superedges (edges between supernodes)[70]. Most work focuses on simple graphs without side information or labels. Based on the core methodology, they can be categorized into four classifications, grouping-based[60, 71], bit-compression based[77], simplification-based[88], and influence-based methods[73]. Grouping approaches are among the most popular techniques for summarization. The utility-driven graph summarization[60] allows users to provide different utilities to measure the data loss and outputs summaries within bound, which can be lossless. Recently, Fan et al.[33] propose a scheme to contract obsolete components, stars, cliques, and paths into supernodes, prioritize up-to-date data and store extra synopsis on disk. In contrast to compression, which aims at minimizing the data size as small as possible, those summarization methods explore structural patterns[70]. Several works[63, 82] aim at answering queries on graphs with no correction set. Namely, a correction set is the extra information of edge correction needed for recreating graphs from summarization. In another latest work, Yong et al.[107] propose an algorithm called LDME, which summarizes the input graph into a summary graph with correction sets at billion-scale. For complicated cases like Resource Description Framework (RDF) graphs, [41] formulates weak/strong node equivalences based on property relations and proposes algorithms with incremental versions to output typed summarizations considering RDF ontologies.

Chapter 3

Span Reachability in Temporal Graphs

3.1 Chapter Overview

This chapter presents a comprehensive account of the methodological details and evaluation results of our work on span reachability in temporal graphs, which is published in [100]. In Section 3.2, we provide formal definitions of the span reachability problem. Following this, Section 3.3 introduces a naive online baseline and explains the basic ideas of the index-based method. We then proceed to Section 3.4, which details how indexes are constructed, and Section 3.5, which illustrates how span reachability queries can be answered using indexes. Section 3.6 reports the experimental results, which demonstrate the effectiveness and efficiency of our proposed methods. Finally, Section 3.7 concludes the chapter.

3.2 Preliminary

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a directed temporal graph, where $\mathcal{V}$ and $\mathcal{E}$ denote the set of vertices and the set of temporal edges respectively. Each temporal edge $e \in \mathcal{E}$ is a triplet $\langle u, v, t \rangle$, where u, v are the vertices in $\mathcal{V}$ and t is the connection time from u to v. Without loss of generality, we assume t is an integer since the timestamp in real-world applications is normally an integer. We use $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ to denote the number of vertices and the number of temporal edges respectively. Given a vertex $u \in \mathcal{V}$, the out-neighbor set of u is defined as $N_{out}(u) = \{\langle v, t \rangle | (u, v, t) \in \mathcal{E}\}$, and the in-neighbor set is defined similarly. The out-degree (resp. in-degree) of u is denoted as $degr_{out}(u) = |N_{out}(u)|$ (resp. $degr_{in}(u) = |N_{in}(u)|$). Given a time interval $[t_s, t_e]$, the projected graph of $\mathcal{G}$ in $[t_s, t_e]$, denoted by $\mathcal{G}_{[t_s,t_e]}$, where $V(\mathcal{G}_{[t_s,t_e]}) = \mathcal{V}$ and $E(\mathcal{G}_{[t_s,t_e]}) = \{(u, v) | (u, v, t) \in \mathcal{E}, t \in [t_s, t_e]\}$. The length or width of an interval $[t_s, t_e]$ is the number of timestamps in the interval, i.e., $t_e - t_s + 1$. Given the temporal graph $\mathcal{G}$ in Figure 1.1, its projected graph in the interval $[2, 4]$ is given in Figure 3.1.

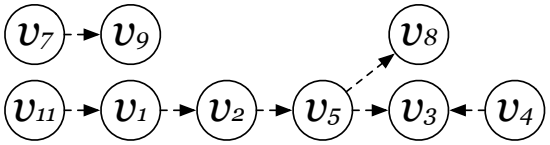


Figure 3.1: The projected static graph of $\mathcal{G}$ in the time interval $[2, 4]$

Based on the concept of the projected graph, we define the span-reachability as follows.

Definition 1. (Span-Reachability) *Given a temporal graph $\mathcal{G}$, two vertices u, v, and a time interval $[t_s, t_e]$, u span-reaches v in $[t_s, t_e]$, denoted as $u \rightsquigarrow_{[t_s,t_e]} v$, if u reaches v in the projected graph $\mathcal{G}_{[t_s,t_e]}$.*

Given the temporal graph $\mathcal{G}$ in Figure 1.1, we have $v_1 \rightsquigarrow_{[2,4]} v_3$ since v_1 reaches v_3 in the projected graph of $[2, 4]$ in Figure 3.1. We define the first problem studied in this paper based on Definition 1 as follows.

Problem 1. *Given a temporal graph $\mathcal{G}$, an arbitrary pair of vertices u, v, and a time interval $\mathcal{I}$, we aim to efficiently answer whether u span-reaches v in the interval $\mathcal{I}$.*

In addition to identifying the span-reachability, we further define a generalized reachability model in a temporal graph $\mathcal{G}$ as follows.

Definition 2. (θ-REACHABILITY) *Given a temporal graph $\mathcal{G}$, two vertices u, v, a parameter θ, and a time interval $[t_s, t_e]$ s.t. $t_e - t_s + 1 \geq \theta$, u θ-reaches v if there exists an interval $[t'_s, t'_e] \subseteq [t_s, t_e]$ such that $t'_e - t'_s + 1 = \theta$ and u reaches v in $\mathcal{G}_{[t'_s,t'_e]}$.*

Example 2. *Given the temporal graph $\mathcal{G}$ in Figure 1.1, let $\theta = 3$. We have v_1 3-reaches v_{12} in the interval $[1, 5]$ since there exists an interval $[3, 5] \subseteq [1, 5]$ such that the length of $[3, 5]$ is 3 and v_1 reaches v_{12} in the projected graph $\mathcal{G}_{[3,5]}$.*

Relationship of Two Reachability Models. Given an arbitrary pair of vertices u, v, a threshold θ and a time interval $\mathcal{I}$, we also study the issue of computing θ-reachability from u to v in $\mathcal{I}$, denoted by Problem 2. Definition 1 is a special case of Definition 2 when θ is equal to the length of the input interval. We also see a growing strictness from Definition 1 to Definition 2, which is shown in the following lemma.

Lemma 1. *Given an arbitrary pair of vertices u, v and an interval $\mathcal{I}$, u span-reaches v in $\mathcal{I}$ if u θ-reaches v in $\mathcal{I}$.*

For ease of presentation, we assume the input temporal graph is a directed graph, and our proposed techniques can easily handle undirected graphs. We

omit the proofs of several lemmas and theorems when they are straightforward due to space limitation.

3.3 Solution Overview

We give an overview of our solution in this section. We start by presenting a straightforward online algorithm for our research problems and then introduce several basic ideas of our index-based method.

3.3.1 A Straightforward Online Approach

Given a time interval $[t_s, t_e]$, the span-reachability of two vertices u and v in $[t_s, t_e]$ can be answered by a modified bidirectional breath-first search. Specifically, we begin by alternatively picking one of u and v in each round, and exploring the unvisited vertices that are reachable from u or can reach v. We have u reaches v once the search scopes of two vertices intersect. The detailed pseudocode of this approach is given in Algorithm 1. Note that we assume $u \neq v$ in all proposed algorithms to answer the reachability queries in this paper. Alternatively, we directly return *true* without the algorithm invocation.

In line 1, R_u and R_v are used to collect all vertices that u can reach and all vertices that can reach v, respectively. In line 5, $Q_u \cup Q_v = \emptyset$ means there does not exist any unexplored vertex for both u and v. The variable *toggle* initialized in line 4 represents the processed vertex in the last iteration, and we process u in lines 7–15 if $toggle = v$. We explore the out-neighbors of all vertices in the queue in lines 9–15. In line 11, we only access edges whose time falls into the input interval. We return *true* if a common vertex of R_u and R_v is found in line 12, or push the new found vertex into the queue in line 14. The algorithm essentially performs a bidirectional BFS in the projected graph $\mathcal{G}_{[t_1,t_2]}$. The time

complexity of Algorithm 1 is given as follows.

Algorithm 1: Online-Reach()

```
   Input: a temporal graph G, two vertices u and v, and an interval [t1, t2]
   Output: the span-reachability of u and v in [t1, t2]
 1 Ru ← {u}, Rv ← {v};
 2 Qu ← a queue containing u;
 3 Qv ← a queue containing v;
 4 toggle ← v;
 5 while Qu ∪ Qv ≠ ∅ do
 6     if toggle = v ∧ Qu ≠ ∅ then
 7         toggle ← u;
 8         l ← |Qu|;
 9         for 1 ≤ i ≤ l do
10             w ← Qu.pop();
11             foreach ⟨w′, t⟩ ∈ Nout(w) : t ∈ [t1, t2] do
12                 if w′ ∈ Rv then return true;
13                 if w′ ∉ Ru then
14                     Qu.push(w′);
15                     Ru ← Ru ∪ {w′};
16     else
17         repeat lines 7–15 to search the vertices that reach v by toggling
           between u and v, and replacing the subscript out with in
18 return false;
```

Lemma 2. *The running time of Algorithm 1 is bounded by* $O(m+n)$.

Proof. Observe that every vertex is marked as either visited or not visited in Algorithm Algorithm 1, indicating that it will be traversed at most once. For each dequeued vertex w from the queue Q_u, the computational cost incurred by iterating through all its neighbors amounts to $degr_{out}(w)+1$ (or $degr_{in}(w)+1$ when in the toggled mode). Consequently, the overall computational complexity is determined as: $O(\sum_{w\in V}(degr_{out}(w)+degr_{in}(w)+2)) = O(2m+2n) = O(m+n)$. □

Likewise, Problem 2 can be answered by invoking Algorithm 1 as a subroutine. We can sequentially check each possible θ-length subinterval in the given query interval $[t_1, t_2]$ and return *true* immediately if u reaches v in any one of them. In the worst case, the time complexity of this algorithm is bounded by $O((t_2 - t_1 - \theta) \cdot (n + m))$.

Even though the bidirectional search method can successfully answer span-reachability queries and θ-reachability queries, the algorithms suffer from a poor scalability since the whole graph may be visited during query processing. One might ponder an alternative strategy, involving the construction of a projected graph before executing an online reachability algorithm. As far as we are concerned, the state-of-the-art work of refined online search approach for reachability queries is [95]. Nevertheless, it is imperative to acknowledge the substantial overhead incurred by this approach. Enumerating all nodes and edges alone entails a computational complexity of $O(m + n)$. Furthermore, a prerequisite for preprocessing the input graph into a directed acyclic graph (DAG) significantly diminishes the online nature of the process. Consequently, the prospect of optimizing performance with respect to time complexity remains elusive. To improve query efficiency, we propose an index-based method in the following section.

3.3.2 The Time Interval Labeling Index

We introduce our index structure called Time Interval Labeling (TILL-Index) in this section. TILL-Index adopts the idea of two-hop cover (or two-hop labeling) [9, 4]. In a nutshell, for each vertex u, we maintain an in-label set $\mathcal{L}_{in}(u)$ and an out-label set $\mathcal{L}_{out}(u)$. Each item in $\mathcal{L}_{in}(u)$ is a triplet $\langle w, t_s, t_e \rangle$ which means that w reaches u in the projected graph $\mathcal{G}_{[t_s,t_e]}$. Each item in $\mathcal{L}_{out}(u)$ is a triplet $\langle w, t_s, t_e \rangle$ which means that u reaches w in $\mathcal{G}_{[t_s,t_e]}$. A triplet is called a w-triplet if the first item of the triplet is w. We call $\langle u, v, t_s, t_e \rangle$ a reachability tuple

Table 3.1: A Time Interval Labeling of $\mathcal{G}$

$\mathcal{L}_{in}(v_2)$	$\langle v_1, 2, 2\rangle$	$\langle v_1, 7, 7\rangle$	$\mathcal{L}_{out}(v_2)$	$\langle v_1, 6, 6\rangle$	$\mathcal{L}_{in}(v_3)$
$\langle v_1, 2, 4\rangle$	$\langle v_1, 4, 5\rangle$	$\langle v_2, 3, 4\rangle$	$\mathcal{L}_{in}(v_4)$	$\langle v_1, 1, 4\rangle$	$\langle v_1, 4, 5\rangle$
$\langle v_2, 3, 5\rangle$	$\langle v_2, 1, 4\rangle$	$\langle v_3, 1, 1\rangle$	$\langle v_3, 5, 5\rangle$	$\langle v_3, 6, 8\rangle$	$\mathcal{L}_{out}(v_4)$
$\langle v_3, 4, 4\rangle$	$\mathcal{L}_{in}(v_5)$	$\langle v_1, 2, 3\rangle$	$\langle v_1, 5, 5\rangle$	$\langle v_2, 3, 3\rangle$	$\mathcal{L}_{out}(v_5)$
$\langle v_3, 4, 4\rangle$	$\mathcal{L}_{out}(v_6)$	$\langle v_1, 5, 6\rangle$	$\langle v_2, 5, 5\rangle$	$\langle v_4, 6, 9\rangle$	$\mathcal{L}_{in}(v_7)$
$\langle v_1, 7, 7\rangle$	$\mathcal{L}_{out}(v_7)$	$\langle v_3, 3, 6\rangle$	$\mathcal{L}_{in}(v_8)$	$\langle v_1, 1, 3\rangle$	$\langle v_1, 2, 4\rangle$
$\langle v_1, 4, 5\rangle$	$\langle v_2, 1, 3\rangle$	$\langle v_2, 3, 4\rangle$	$\langle v_3, 8, 8\rangle$	$\langle v_5, 1, 1\rangle$	$\langle v_5, 4, 4\rangle$
$\langle v_6, 9, 9\rangle$	$\mathcal{L}_{out}(v_8)$	$\langle v_3, 4, 6\rangle$	$\langle v_4, 6, 6\rangle$	$\mathcal{L}_{in}(v_9)$	$\langle v_1, 1, 1\rangle$
$\langle v_1, 3, 7\rangle$	$\langle v_2, 1, 4\rangle$	$\langle v_3, 1, 1\rangle$	$\langle v_7, 3, 3\rangle$	$\mathcal{L}_{out}(v_9)$	$\langle v_3, 6, 6\rangle$
$\mathcal{L}_{in}(v_{10})$	$\langle v_1, 8, 8\rangle$	$\mathcal{L}_{out}(v_{10})$	$\langle v_1, 9, 9\rangle$	$\mathcal{L}_{out}(v_{11})$	$\langle v_1, 3, 3\rangle$
$\mathcal{L}_{out}(v_{12})$	$\langle v_1, 6, 9\rangle$	$\langle v_{10}, 6, 6\rangle$			

if $u \rightsquigarrow_{[t_s,t_e]} v$, and we say a vertex w covers a reachability tuple $\langle u, v, t_s, t_e\rangle$ if $u \rightsquigarrow_{[t_s,t_e]} w$ and $w \rightsquigarrow_{[t_s,t_e]} v$. For ease of presentation, we focus mainly on Problem 1 now. Problem 2 can also be solved based on the TILL-Index, and Section 3.5 will discuss its solution in detail by extending the techniques in answering Problem 1. Given two vertices u and v, u span-reaches v in an interval $[t_1, t_2]$ if any one of the following equations holds:

1. $\exists\langle v, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$: $[t_s, t_e] \subseteq [t_1, t_2]$;
2. $\exists\langle u, t_s, t_e\rangle \in \mathcal{L}_{in}(v)$: $[t_s, t_e] \subseteq [t_1, t_2]$;
3. $\exists\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u), \langle w', t'_s, t'_e\rangle \in \mathcal{L}_{in}(v)$: $w = w' \wedge [t_s, t_e] \subseteq [t_1, t_2] \wedge [t'_s, t'_e] \subseteq [t_1, t_2]$.

Based on the above equations, a TILL-Index is a minimal index that can be used to answer correctly all possible span-reachability queries in $\mathcal{G}$. Here, by minimal, we mean that removing any item in the index cannot correctly determine all possible span-reachability in the graph. We will give detailed proof of correctness and minimality in Section 3.4.2. An example of a TILL-Index of the temporal graph $\mathcal{G}$ in Figure 1.1 is given in Table 3.1.

Example 3. *Assume that we aim to answer the span-reachability from* v_6 *to* v_3 *in the time interval* $[4, 8]$*. We first locate the out-label set of* v_6 *in Table 3.1, which are* $\mathcal{L}_{out}(v_6) = \{\langle v_1, 5, 6\rangle, \langle v_2, 5, 5\rangle, \langle v_4, 6, 9\rangle\}$*. The in-label set of* v_3 *are* $\mathcal{L}_{in}(v_3) = \{\langle v_1, 2, 4\rangle, \langle v_1, 4, 5\rangle, \langle v_2, 3, 4\rangle\}$*. We can see that there is a common vertex* v_1 *such that both* $\langle v_1, 5, 6\rangle \in \mathcal{L}_{out}(v_6)$ *and* $\langle v_1, 4, 5\rangle \in \mathcal{L}_{in}(v_3)$ *fall in the query interval* $[4, 8]$*. Therefore, the answer of this query is true.*

Even though the idea of two hop cover is simple, it is non-trivial to efficiently compute a small TILL-Index and answer the reachability queries based on the index. We give the details about index construction and query processing in Section 3.4 and Section 3.5, respectively.

3.4 Index Construction

3.4.1 The Labeling Framework

We begin by presenting several basic concepts before introducing the details of the index construction.

Definition 3. (Dominance and Skyline Reachability Tuple) *Given two vertices* u *and* v*, a reachability tuple* $\langle u, v, t'_s, t'_e\rangle$ *dominates* $\langle u, v, t_s, t_e\rangle$ *if* $[t'_s, t'_e] \subset [t_s, t_e]$*. A reachability tuple* $\langle u, v, t_s, t_e\rangle$ *is a skyline (or non-dominated) reachability tuple (SRT) if it is not dominated by other tuples.*

Given a vertex u, we also use the term *skyline* in Definition 3 for the triplets in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{in}(u)$) since a triplet $\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$ represents a reachability tuple $\langle u, w, t_s, t_e\rangle$. In constructing TILL-Index, we only need to compute labels that can cover all SRTs since a vertex covering an SRT also covers all its dominating tuples. Therefore, our research task in the index construction is to cover all SRTs in the graph with the total index size as small as possible.

The Minimum Two-Hop Cover. [28] studies the two-hop cover for the shortest distance and reachability queries in general graphs. They proved that computing the minimum two-hop cover is NP-hard and can be transformed to a minimum cost set cover problem [27]. They use a greedy algorithm to compute a two-hop cover and achieve an $O(\log n)$ approximation factor. The proposed algorithm is inefficient since a procedure of densest subgraph computation is invoked every time they select a vertex to cover several reachability (or shortest distance) vertex pairs.

Hierarchical Two-Hop Cover. The aforementioned theoretical results also hold in our scenario, and we omit the detailed proof. Due to the difficulty of the optimal cover computation, we adopt a hierarchical labeling approach [9, 4] which follows a strict total order on the vertices in $\mathcal{G}$, and we will prove the minimality of our TILL-Index under the total order constraint. We use $\mathcal{O}$ to denote the vertex order. We say the rank of a vertex u is higher than that of a vertex v if $\mathcal{O}(u) < \mathcal{O}(v)$. By the total order, we mean to sequentially process each vertex in $\mathcal{O}$. Once we process a vertex w, we add w and corresponding intervals to the labels of u and v for all uncovered reachability tuples containing u, v covered by w. Intuitively, a vertex playing an important role in $\mathcal{G}$ should be put at the front of the order. Next, we adopt the ordering method in [53]. Given each vertex u, we use the formula $(degr_{in}(u)+1) \times (degr_{out}(u)+1)$ as the importance of u. We sort the vertices in a decreasing order of their importance and break the tie by selecting a vertex with smaller ID. Given the total vertex order, we immediately have the following lemmas for our TILL-Index.

Lemma 3. *Given an arbitrary vertex u, for every triplet $\langle w, *, * \rangle$ in $\mathcal{L}_{out}(u) \cup \mathcal{L}_{in}(u)$, $\mathcal{O}(w) < \mathcal{O}(u)$.*

Lemma 4. *Given an SRT $\langle u, v, t_s, t_e \rangle$ in $\mathcal{G}$, let w be the first vertex (the highest rank) in $\mathcal{O}$ that can cover $\langle u, v, t_s, t_e \rangle$. $w \neq u \neq v$. There exists a triplet*

$\langle w, t'_s, t'_e \rangle \in \mathcal{L}_{out}(u)$ such that $[t'_s, t'_e] \subseteq [t_s, t_e]$ and a triplet $\langle w, t''_s, t''_e \rangle \in \mathcal{L}_{in}(v)$ such that $[t''_s, t''_e] \subseteq [t_s, t_e]$.

Without loss of generality, we maintain only skyline triplets in labels of TILL-Index since a dominated triplet can be always replaced by a corresponding skyline triplet without influencing calculation's accuracy. We define an important concept in computing TILL-Index as follows.

Definition 4. (Canonical Reachability Tuple) *A reachability tuple $\langle u, v, t_s, t_e \rangle$ is a canonical reachability tuple (CRT) if (i) $\langle u, v, t_s, t_e \rangle$ is a skyline reachability tuple, and (ii) there does not exist a vertex w such that $u \rightsquigarrow_{[t_s,t_e]} w$, $w \rightsquigarrow_{[t_s,t_e]} v$, $\mathcal{O}(w) < \mathcal{O}(u)$, and $\mathcal{O}(w) < \mathcal{O}(v)$.*

Given a vertex order $\mathcal{O}$ and a vertex u, we say a tuple is an SRT (resp. CRT) of u if the tuple is an SRT (resp. CRT) containing u and the rank of u is higher in the tuple. We have following lemmas based on Definition 4.

Lemma 5. *Given an arbitrary vertex u and any (skyline) triplet $\langle w, t_s, t_e \rangle$ in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{in}(u)$), $\langle u, w, t_s, t_e \rangle$ (resp. $\langle w, u, t_s, t_e \rangle$) is a CRT.*

Lemma 6. *For each CRT $\langle u, v, t_s, t_e \rangle$ in $\mathcal{G}$, there is a triplet $\langle u, t_s, t_e \rangle$ in $\mathcal{L}_{in}(v)$ if $\mathcal{O}(u) < \mathcal{O}(v)$. If this is not the case, there is a triplet $\langle v, t_s, t_e \rangle$ in $\mathcal{L}_{out}(u)$.*

Example 4. *The labels in Table 3.1 are computed following the total alphabetical order of the vertices in $\mathcal{G}$ of Figure 1.1. For the in-labels of v_8, we can find that the rank of all vertices v_1, v_2, v_3, v_4, v_5 and v_6 appearing in $\mathcal{L}_{in}(v_8)$ have ranks higher than v_8. For an arbitrary triplet $\langle v_2, 3, 4 \rangle$ in $\mathcal{L}_{in}(v_8)$, there does not exist any vertex with higher rank than v_8, and v_2 that can cover the reachability tuple $\langle v_2, v_8, 3, 4 \rangle$.*

Based on Lemma 5 and Lemma 6, there is a one-to-one correspondence between CRTs and triplets in TILL-Index. It now follows that we can construct

TILL-Index by computing all CRTs. A framework to construct TILL-Index is presented in Algorithm 2.

Algorithm 2: A Framework of Index Construction

```
1 for 1 ≤ i ≤ n do
2     u_i ← the i-th vertex in the order O;
3     compute all SRTs of u_i;
4     compute all CRTs by refining the computed SRTs;
5     add corresponding triplet of each CRT to in-labels or out-labels of
        other vertices;
```

In the framework, we process each vertex sequentially in the vertex order. In line 3, the SRTs of u_i can be computed in two phases. One computes all vertices and corresponding time intervals that are reachable from u, while the other computes those that can reach u. Taking the first one as an example, a basic implementation uses a queue to maintain the discovered reachable triplets of u_i. To be specific, the queue is initialized as a special triplet containing u_i. We iteratively pop a triplet $\langle v, t_s, t_e \rangle$, which means u can reach v in $[t_s, t_e]$. For each out-neighbor $\langle v', t \rangle$ of v, we expand $\langle v, t_s, t_e \rangle$ to $\langle v', \min(t_s, t), \max(t_e, t) \rangle$, which means u_i reaches v' in the interval $[\min(t_s, t), \max(t_e, t)]$. We mark this new triplet $\langle v', \min(t_s, t), \max(t_e, t) \rangle$ as discovered and push it into the queue if it is not dominated by other discovered triplet, and remove all its dominating discovered triplets. In line 3, for every SRT computed in line 2, we check whether there exists a vertex with a higher rank that can cover the SRT based on Definition 4. This can be done by performing a query processing procedure based on the labels computed by higher-rank vertices. The details of query processing will be given in the Section 3.5. If yes, we omit such SRT, and derive all CRTs when all SRTs are checked.

3.4.2 Theoretical Analysis

We prove the correctness and the minimality of TILL-Index computed by Algorithm 2.

Theorem 1. (CORRECTNESS) *The span-reachability query of any pair of vertices can be correctly answered (any one of three conditions presented in Section 3.3.2 holds) based on the index computed by Algorithm 2.*

Proof. The theorem can be easily derived according to Definition 4, Lemma 5 and Lemma 6. □

Theorem 2. (MINIMALITY) *For any vertex u and any triplet $\langle w, t_s, t_e\rangle$ in $\mathcal{L}_{in}(u)$ or $\mathcal{L}_{out}(u)$ of the index computed by Algorithm 2, there exists a pair of vertices u', v' and a corresponding interval $[t'_s, t'_e]$ such that the span-reachability of u' and v' in $[t'_s, t'_e]$ cannot be correctly answered after removing $\langle w, t_s, t_e\rangle$.*

Proof. Given a triplet $\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$, we prove that after removing $\langle w, t_s, t_e\rangle$, the span-reachability from u to w in $[t_s, t_e]$ cannot be correctly answered. If this query can be correctly answered, then at least one of the following two condition hold: (i) there exists a triplet $\langle u, t'_s, t'_e\rangle$ in $\mathcal{L}_{in}(w)$ such that $[t'_s, t'_e] \subseteq [t_s, t_e]$; (ii) there exists a triplet $\langle v, t'_s, t'_e\rangle \in \mathcal{L}_{out}(u)$ and a triplet $\langle v, t''_s, t''_e\rangle \in \mathcal{L}_{in}(w)$ such that $[t'_s, t'_e] \subseteq [t_s, t_e]$ and $[t''_s, t''_e] \subseteq [t_s, t_e]$.

Given that $\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$, we have $\mathcal{O}(w) > \mathcal{O}(u)$ according to Lemma 3, and a triplet containing u cannot appear in $\mathcal{L}_{in}(w)$ and $\mathcal{L}_{out}(w)$. Therefore, condition i cannot hold. Condition ii holds if v covers the reachability tuple $\langle u, w, t_s, t_e\rangle$ and the rank of v is higher than those of u and w. This contradicts Lemma 5 that $\langle u, w, t_s, t_e\rangle$ is a CRT. This completes the proof of the theorem. □

3.4.3 Implementation

The basic implementation incurs high computational cost. We discuss several techniques to efficiently compute SRTs and CRTs as follows.

Efficient SRT Computation

We propose a priority queue based method to efficiently compute all SRTs of a given vertex. A key idea of this method is given in the following lemma.

Lemma 7. *Given a vertex u and a set of known SRTs S containing u, a reachability tuple $\langle u, v, t_s, t_e \rangle$ is an SRT if (i) $\langle u, v, t_s, t_e \rangle$ is not dominated by any other SRT in S, and (ii) the length of $[t_s, t_e]$ is the smallest among those of all tuples that are not in S.*

Example 5. *We consider the temporal graph $\mathcal{G}$ in Figure 1.1. Assume that we aim to compute SRTs of v_5. For ease of presentation, we only consider the SRTs starting from v_5. Initially, $S = \emptyset$ and we have several reachability tuples with the smallest interval length. They are $\langle v_5, v_3, 4, 4 \rangle$, $\langle v_5, v_8, 1, 1 \rangle$, and $\langle v_5, v_8, 4, 4 \rangle$, and all of them are SRTs. Now we have $S = \{\langle v_5, v_3, 4, 4 \rangle, \langle v_5, v_8, 1, 1 \rangle, \langle v_5, v_8, 4, 4 \rangle\}$. $\langle v_5, v_8, 4, 8 \rangle$ is not an SRT since it is dominated by $\langle v_5, v_8, 4, 4 \rangle$ in S, and $\langle v_5, v_{12}, 4, 5 \rangle$ is an SRT since its interval length is smallest among all possible reachability tuples except the SRTs in S.*

Based on Lemma 7, to compute all non-dominated reachability triplets (a target and the corresponding time interval) from a vertex u, we preserve all discovered reachability triplets in a priority queue, and always pop the triplets with the smallest time interval length in the priority queue. According to Lemma 7, a popped triplet $\langle v, t_s, t_e \rangle$ must be an SRT if it is not dominated by any previously found SRT. We compute the new interval of each neighbor of v that can

be reached from $\langle v, t_s, t_e \rangle$ and push the corresponding new triplet into the priority queue if necessary. Following this, we compute all SRTs when the priority queue is empty. A detailed pseudocode of our final algorithm will be given in the following section.

Efficient CRT Computation

We reduce the CRT checks by making use of the transitive property of the dominance relationship. The following lemma provides an early termination condition in the search of SRT computation.

Lemma 8. *Given a reachability tuple $\langle u, v, t_s, t_e \rangle$ and a vertex w, for any reachability tuple $\langle u, v', t'_s, t'_e \rangle$, we have w covers $\langle u, v', t'_s, t'_e \rangle$ if (i) w covers $\langle u, v, t_s, t_e \rangle$, (ii) $[t_s, t_e] \subseteq [t'_s, t'_e]$, and (iii) v span-reaches v' in $[t'_s, t'_e]$.*

Given the i-th vertex u_i in $\mathcal{O}$, assume that we have detected a vertex v that u_i can reach in an interval $[t_s, t_e]$, and the corresponding tuple $\langle u_i, v, t_s, t_e \rangle$ has been covered. Based on Lemma 8, we immediately terminate any further exploration of v since all other vertices that are reachable from $\langle v, t_s, t_e \rangle$ must have been covered too. By adopting this pruning technique, we not only avoid a large number of CRT checks but also reduce the search scope in SRT computation. We give the pesudocode of the final algorithm for the index construction by combing two optimization techniques in Algorithm 3.

In Algorithm 3, we use a parameter ϑ to achieve a trade-off between the index size and the index coverage practically. ϑ represents the largest interval length of span-reachability query that TILL-Index can support. In most applications, users may be only interested in the span-reachability queries in a small-length interval. We will show the index size and its construction time under different ϑ selections in Section 3.6.

Lines 4–16 of Algorithm 3 compute all reachable verices and corresponding intervals from u_i. As discussed in Section 3.4.3, we always pop a triplet $\langle v, t_s, t_e \rangle$ with the smallest value of $t_e - t_s$ in line 8. Based on Lemma 8, we check if the reachability tuple $\langle u_i, v, t_s, t_e \rangle$ has been covered in line 10. Here, $u_i \rightsquigarrow^{\mathcal{L}}_{[t_s,t_e]} v$ means the answer of the span-reachability query from u_i to v in $[t_s, t_e]$ is *true* according to the current TILL-Index $\mathcal{L}$ ($\mathcal{L}$ includes the in-label $\mathcal{L}_{in}$ and out-label $\mathcal{L}_{out}$ of every vertex). Note that $\mathcal{L}$ dynamically increases during the execution process of the algorithm. We omit this tuple and stop further exploration of it if it is covered by the previously computed index (line 10). Lemma 7 and Lemma 8 guarantee that $\langle u_i, v, t_s, t_e \rangle$ must be an CRT, and we safely add u_i with corresponding interval to the in-labels of v in line 11. Lines 12–16 explore the out-neighbors of v. We omit the neighbor with higher rank in line 13 since their reachability tuples have been covered in previous iterations. We compute the updated reachability interval for each neighbor v' in line 14. We push the triplet into the priority queue in line 16 if the interval gap is not larger than the threshold ϑ.

Example 6. *We give a running example of Algorithm 3. The default value of the parameter ϑ is $+\infty$. Given a graph $\mathcal{G}$ in Figure 1.1 and an alphabetical order, assume that we have processed the first 4 vertices. We have $i = 5$ in line 3 and $u_i = v_5$ in line 4. The priority queue is initialized with one special element $\langle v_5, +\infty, -\infty \rangle$. We pop $\langle v_5, +\infty, -\infty \rangle$ in line 8 and scan out-neighbors of v_5 including $\langle v_3, 4 \rangle$, $\langle v_8, 1 \rangle$, and $\langle v_8, 4 \rangle$. We omit the out-neighbor $\langle v_3, 4 \rangle$ since $\mathcal{O}(v_3) > \mathcal{O}(v_5)$ in line 13, and push $\langle v_8, 1, 1 \rangle$ and $\langle v_8, 4, 4 \rangle$ into $\mathcal{Q}$. Assume the next popped triplet in line 8 is $\langle v_8, 1, 1 \rangle$. v_8 has only one out-neighbor $\langle v_4, 6 \rangle$ and we have $t'_s = 1, t'_e = 6$ in line 14. We push $\langle v_4, 1, 6 \rangle$ into $\mathcal{Q}$. In the next round, we pop $\langle v_8, 4, 4 \rangle$ and push $\langle v_4, 4, 6 \rangle$ into $\mathcal{Q}$. Now, $\mathcal{Q}$ contains two triplets, $\langle v_4, 4, 6 \rangle$ and $\langle v_4, 1, 6 \rangle$. We do not push any new triplet into $\mathcal{Q}$ in the following*

Algorithm 3: TILL-Construct*()

Input: a temporal graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, a vertex order $\mathcal{O}$ and a parameter ϑ
Output: the TILL-Index of $\mathcal{G}$

```
1  foreach u ∈ V do
2      L_in(u), L_out(u) ← ∅;
3  for 1 ≤ i < n do
4      u_i ← the i-th vertex in O;
5      Q ← an empty priority queue;
6      Q.push(⟨u_i, +∞, −∞⟩);
7      while Q is not empty do
8          ⟨v, t_s, t_e⟩ ← Q.pop();
9          if u_i ≠ v then
10             if u_i ⇝^L_[t_s,t_e] v then continue;
11             else L_in(v) ← L_in(v) ∪ {⟨u_i, t_s, t_e⟩};
12         foreach ⟨v', t⟩ ∈ N_out(v) do
13             if O(v') ≤ O(u) then continue;
14             t'_s ← min(t_s, t), t'_e ← max(t_e, t);
15             if t'_e − t'_s + 1 > ϑ then continue;
16             else Q.push(⟨v', t'_s, t'_e⟩);
17     repeat lines 2–16 to construct L_out of each vertex by toggling
         between the subscripts in and out;
```

rounds since both $\langle v_4, 4, 6\rangle$ *and* $\langle v_4, 1, 6\rangle$ *are covered by* v_3*, and the condition in line 10 holds. Till now, we have computed all CRTs of* v_5 *which start from* v_5*.*

Let $C^{out}_{\leq\vartheta}(u)$ (resp. $C^{in}_{\leq\vartheta}(u)$) be the set of all CRTs containing u whose interval length is not larger than ϑ and the first (resp. second) item is u. Let $c_{\leq\vartheta} = \max_{u\in\mathcal{V}}(\max(|C^{out}_{\leq\vartheta}(u)|, |C^{in}_{\leq\vartheta}(u)|)$ and d be the largest out-degree or in-degree of the vertices in the graph, i.e., $d = \max_{u\in\mathcal{V}} \max(degr_{out}(u), degr_{in}(u))$. The time complexity of Algorithm 3 is summarized as follows.

Theorem 3. *The running time of Algorithm 3 is bounded by* $O(ndc_{\leq\vartheta}(\log dc_{\leq\vartheta} + c_{\leq\vartheta}))$.

Proof. We first focus on one iteration of line 3. Based on Lemma 5 and Lemma 6,

line 11 is performed $O(c_{\leq\vartheta})$ times. We scan the out-neighbors of v' if line 11 holds. Therefore, lines 13-16 are performed $O(dc_{\leq\vartheta})$ times, and the total number of items appended to the priority queue is bounded by $O(dc_{\leq\vartheta})$. In line 10, we check whether $\langle u_i, v, t_s, t_e\rangle$ is covered by prior verices. This can be done by sequentially scanning the existing out-label of u_i and in-label of v and returning true if there is a common vertex in the interval $[t_s, t_e]$. The running time can be bounded by $O(|C^{out}_{\leq\vartheta}(u_i)| + |C^{in}_{\leq\vartheta}(v)|)$ or $O(c_{\leq\vartheta})$. In line 7 and 16, it requires $O(\log dc_{\leq\vartheta})$ to push a new item or get the top item in the priority queue. By combing the results, we have the total time complexity $O(ndc_{\leq\vartheta}(\log dc_{\leq\vartheta} + c_{\leq\vartheta}))$. □

Undirected Graphs. In undirected graphs, we only need to maintain one label set for each vertex. Therefore, we omit line 17 of Algorithm 3 when constructing the index of an undirected graph.

3.5 Query Processing

We study the query processing strategies based on the TILL-Index computed by Algorithm 3. We discuss the algorithm to answer the span-reachability followed by a full discourse of the algorithms for the θ-reachability query.

3.5.1 Span-Reachability Query Processing

Our first step is to present several basic pruning strategies to check span-reachability. Given a vertex u, let $t_{min}(N_{out}(u))$ (resp. $t_{max}(N_{out}(u))$) be the smallest (resp. largest) timestamp in out-neighbors of u. $t_{min}(N_{in}(u))$ and $t_{max}(N_{in}(u))$ are defined similarly. We have the following lemmas.

Lemma 9. *A vertex u span-reaches a vertex v in $[t_1, t_2]$ only if there exist a neighbor $\langle w, t\rangle \in N_{out}(u)$ and $\langle w', t'\rangle \in N_{in}(v)$ such that $t \in [t_1, t_2]$ and $t' \in$*

$[t_1, t_2]$.

Lemma 10. *A vertex u span-reaches a vertex v in $[t_1, t_2]$ only if $t_2 \geq \max(t_{min}(N_{out}(u)), t_{min}(N_{in}(v)))$ and $t_1 \leq \min(t_{max}(N_{out}(u)), t_{max}(N_{in}(v)))$.*

We can check the conditions in above two lemmas simply by scanning the neighbors of each query vertex. If the conditions do not hold, we immediately return *false* and do not invoke any query processing procedure.

Given a pair of query vertices u, v and an interval $[t_s, t_e]$, a straightforward method to answer the span-reachability of u and v is to scan $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(v)$. Let $\mathcal{L}_{out}(u)_{[t_s,t_e]}$ (resp. $\mathcal{L}_{in}(u)_{[t_s,t_e]}$) be the set of all triplets in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{out}(v)$) falling in the interval $[t_s, t_e]$. We answer *true* if there exists a common vertex in $\mathcal{L}_{out}(u)_{[t_s,t_e]} \cup \{u\}$ and $\mathcal{L}_{in}(v)_{[t_s,t_e]} \cup \{v\}$. Otherwise, we return *false*. This can be done by using a hash table to preserve the vertices.

To improve the query efficiency, we group the triplets in the out-label or in-label of each vertex by their target vertices (the first item in the triplet). Let $\mathcal{V}(\mathcal{L}_{out}(u))$ be the set of vertices in the reachability triplet of $\mathcal{L}_{out}(u)$, i.e., $\mathcal{V}(\mathcal{L}_{out}(u)) = \{v \in \mathcal{V} | \langle v, t_s, t_e \rangle \in \mathcal{L}_{out}(u)\}$. Given a vertex w in $\mathcal{V}(\mathcal{L}_{out}(u))$, we use $\mathcal{L}_{out}(u)_w$ to denote the intervals that u can reach w in $\mathcal{L}_{out}(u)$, i.e., $\mathcal{L}_{out}(u)_w = \{[t_s, t_e] | \langle w, t_s, t_e \rangle \in \mathcal{L}_{out}(u)\}$. We check the span-reachability in two phases. In the first one, we check if there exists a common vertex in $u \cup \mathcal{V}(\mathcal{L}_{out}(u))$ and $v \cup \mathcal{V}(\mathcal{L}_{out}(v))$. This can be done in a merge sort like strategy by arranging the vertices in the label of each vertex by their ranks. Once finding a common vertex w, we further check if there exist intervals falling in the query interval in $\mathcal{L}_{out}(u)_w$ and $\mathcal{L}_{in}(v)_w$, respectively. If yes, we immediately return *true*. Otherwise, we resume the search and look for the next common vertex. Recall that in Algorithm 3, the triplets appended to the out-label or in-label of each vertex follow the order of the vertex rank. Therefore, the group operation can be done naturally in the index construction without incurring extra cost.

To check whether there exists an interval falling in the query interval, we sort the intervals of each vertex in chronological order. So, given two intervals $[t_s, t_e]$ and $[t'_s, t'_e]$, $[t_s, t_e]$ is prior to $[t'_s, t'_e]$ if (i) $t_s < t'_s$, or (ii) $t_s = t'_s \wedge t_e < t'_e$. Therefore, given a query interval $[t_1, t_2]$ and an arbitrary interval $[t_s, t_e]$, if an interval $[t^*_s, t^*_e] \subseteq [t_1, t_2]$ exists, $[t^*_s, t^*_e]$ must appear after $[t_s, t_e]$ if $t_s < t_1$ or appear before $[t_s, t_e]$ if $t_e > t_2$. This sorting task can be done at the end of Algorithm 3 after all labels are completely computed. The time usage for sorting can be bounded by $O(nc_{\leq\vartheta} \log c_{\leq\vartheta})$, and this would not increase the total time complexity in Theorem 3.

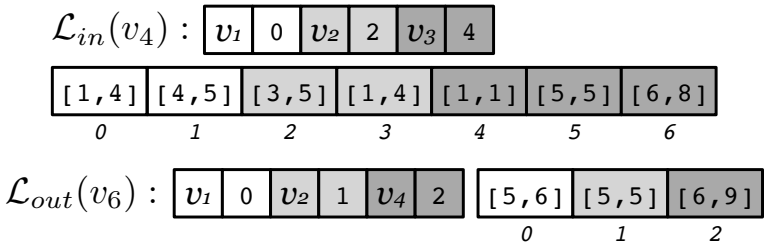


Figure 3.2: The data structure used to store $\mathcal{L}_{in}(v_4)$ and $\mathcal{L}_{out}(v_6)$

Example 7. *Figure 3.2 shows the data structure used to store the labels of each vertex. We take $\mathcal{L}_{in}(v_4)$ and $\mathcal{L}_{out}(v_6)$ as examples. All triplets in these two label sets can be found in Table 3.1. Two arrays are used to store the triplets in the label of each vertex. One interval array stores the intervals for each vertex in the label, and the other vertex array stores all vertices in the label and the start position of their intervals in the interval array. For $\mathcal{L}_{in}(v_4)$ in Figure 3.2, the intervals of v_1, v_2, and v_3 are marked by white, light gray and dark gray, respectively. The intervals of v_2 in $\mathcal{L}_{in}(v_4)$ in the interval array start from the position of v_2 (i.e., 2), and end at the position of the next vertex v_3 in the vertex array (i.e., 4).*

A complete pseudocode of the span-reachability query processing is presented

Algorithm 4: Span-Reach()

Input: TILL-Index of $\mathcal{G}$, two vertices u and v, and an interval $[t_1, t_2]$
Output: the span-reachability of u and v in $[t_1, t_2]$

```
1  i, i' ← 1;
2  while i ≤ |V(L_out(u))| ∧ i' ≤ |V(L_in(v))| do
3      w ← the i-th vertex in V(L_out(u));
4      w' ← the i'-th vertex in V(L_in(v));
5      if w = v ∧ ∃[t_s, t_e] ∈ L_out(u)_w : [t_s, t_e] ⊆ [t_1, t_2] then return true;
6      else if w' = u ∧ ∃[t'_s, t'_e] ∈ L_in(v)_w' : [t'_s, t'_e] ⊆ [t_1, t_2] then return
         true;
7      else if O(w) < O(w') then i ← i + 1;
8      else if O(w) > O(w') then i' ← i' + 1;
9      else if ∃[t_s, t_e] ∈ L_out(u)_w : [t_s, t_e] ⊆ [t_1, t_2] ∧
         ∃[t'_s, t'_e] ∈ L_in(v)_w' : [t'_s, t'_e] ⊆ [t_1, t_2] then
10         return true;
11     else i ← i + 1, i' ← i' + 1;
12 return false;
```

in Algorithm 4, and is self-explanatory. In lines 5, 6, and 9, we use the binary search method described above to find a subinterval of $[t_1, t_2]$.

Example 8. *Assume that we aim to answer the span-reachability from v_6 to v_4 in the interval $[3, 5]$. We scan the vertex array of $\mathcal{L}_{out}(v_6)$ and $\mathcal{L}_{in}(v_4)$ to look for a common vertex. We first find a common vertex v_1. However, there does not exist a subinterval of $[3, 5]$ of v_1 in the interval array of $\mathcal{L}_{out}(v_6)$. We continue to search the next common vertex and find v_2. We find there exist a subinterval $[5, 5]$ of v_2 in $\mathcal{L}_{out}(v_6)$ and a subinterval $[3, 5]$ of v_2 in $\mathcal{L}_{in}(v_4)$. Therefore, we return true for this query.*

Theorem 4. *Given a pair of vertices u and v, the running time of Algorithm 4 is bounded by $O(|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|)$.*

3.5.2 θ-Reachability

Based on the idea for the span-reachability query processing, we study the θ-reachability query in this subsection. Given two vertices u, v, a threshold θ and an interval $[t_1, t_2]$, a straightforward idea to answer the θ reachability query is to invoke Algorithm 4 for every possible interval (from $[t_1, t_1+\theta-1]$ to $[t_2-\theta+1, t_2]$). The time complexity of this method is $O((t_2 - t_1 - \theta) \cdot (|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|))$. We improve the time complexity to $O(|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|)$ by taking a sliding window based approach. Before discussing the details of the algorithm, we show that u θ-reaches v in $[t_1, t_2]$ if one of the following equations holds:

1. $\exists \langle v, t_s, t_e \rangle \in \mathcal{L}_{out}(u)$: $[t_s, t_e] \subseteq [t_1, t_2] \wedge t_e - t_s + 1 \leq \theta$;
2. $\exists \langle u, t_s, t_e \rangle \in \mathcal{L}_{in}(v)$: $[t_s, t_e] \subseteq [t_1, t_2] \wedge t_e - t_s + 1 \leq \theta$;
3. $\exists \langle w, t_s, t_e \rangle \in \mathcal{L}_{out}(u), \langle w', t'_s, t'_e \rangle \in \mathcal{L}_{in}(v)$: $w = w' \wedge [t_s, t_e] \subseteq [t_1, t_2] \wedge [t'_s, t'_e] \subseteq [t_1, t_2] \wedge \max(t_e, t'_e) - \min(t_s, t'_s) + 1 \leq \theta$.

Based on the conditions above, we can follow the same framework of Algorithm 4. We add the limitation $t_e - t_s + 1 \leq \theta$ in line 5 and line 6 of Algorithm 4 respectively to check the first two conditions. To check the third condition of finding a common vertex w in $\mathcal{V}(\mathcal{L}_{out}(u))$ and $\mathcal{V}(\mathcal{L}_{in}(v))$, we first filter out all intervals in $\mathcal{L}_{out}(u)_w$ and $\mathcal{L}_{in}(v)_w$ not found in $[t_1, t_2]$. With the concept of sliding window, the window is always θ. Recall that the intervals in each label are sorted in chronological order. The initial start time of the window is the smallest start time of the remaining intervals in the labels. If both the first intervals of two labels fall in the sliding window, we return *true*. Alternatively, we filter out the interval with the smallest start time and move the sliding window forward to the next smallest start time of the intervals. This step is repeated until no interval remains.

The pseudocode to answer the θ-reachability query is given in Algorithm 5. Lines 5 and 6 correspond to the θ-reachability conditions 1 and 2 respectively. Lines 9–20 correspond to condition 3. In lines 10 and 11, we use a binary search to locate the first interval falling in $[t_1, t_2]$. The condition of line 15 holds if all intervals of $\mathcal{L}_{out}(u)_w$ (or $\mathcal{L}_{in}(v)_w$) in $[t_1, t_2]$ are scanned, and we break the loop. Line 17 holds if we find a pair of intervals falling in the same sliding window. In lines 19 and 21, we move the sliding window with a new start time of $\min(t_s, t'_s)$.

Theorem 5. *Given a pair of vertices u and v, the running time of Algorithm 5 is bounded by $O(|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|)$.*

Example 9. *Given a query interval $[1, 8]$ and $\theta = 3$, assume that we aim to answer 3-reachability from v_6 to v_4. The out-label and in-label of v_6 and v_4 are given in Figure 3.2, respectively. In line 9 of Algorithm 5, we find a common vertex v_1 in $\mathcal{V}(\mathcal{L}_{out}(v_6))$ and $\mathcal{V}(\mathcal{L}_{in}(v_4))$. We have $[t_s, t_e] = [5, 6]$ in line 13 and $[t'_s, t'_e] = [1, 4]$ in line 14. The conditions in lines 15, 17,and 19 do not hold. As a result, line 21 is executed. In the next iteration, we have $[t'_s, t'_e] = [4, 5]$ and $[t_s, t_e]$ is kept constant. The condition in line 17 holds, and true is returned.*

Algorithm 5: ES-Reach*()

Input: TILL-Index of $\mathcal{G}$, a parameter θ, two vertices u and v, and an interval $[t_1, t_2]$
Output: the θ-reachability of u and v in $[t_1, t_2]$

```
1  i, i' ← 1;
2  while i ≤ |V(L_out(u))| ∧ i' ≤ |V(L_in(v))| do
3      w ← the i-th vertex in V(L_out(u));
4      w' ← the i'-th vertex in V(L_in(v));
5      if w = v ∧ ∃[t_s, t_e] ∈ L_out(u)_w : [t_s, t_e] ⊆ [t_1, t_2], t_e − t_s ≤ θ then return true;
6      else if w' = u ∧ ∃⟨w', t'_s, t'_e⟩ ∈ L_in(v) : [t'_s, t'_e] ⊆ [t_1, t_2], t'_e − t'_s ≤ θ then return
         true;
7      else if O(w) < O(w') then i ← i + 1;
8      else if O(w) > O(w') then i' ← i' + 1;
9      else if ∃[t_s, t_e] ∈ L_out(u)_w : [t_s, t_e] ⊆ [t_1, t_2] ∧
         ∃[t'_s, t'_e] ∈ L_in(v)_w' : [t'_s, t'_e] ⊆ [t_1, t_2] then
10         k ← the position of the first interval [t_s, t_e] ∈ L_out(u)_w s.t. [t_s, t_e] ⊆ [t_1, t_2];
11         k' ← the position of the first interval [t'_s, t'_e] ∈ L_in(v)_w' s.t. [t'_s, t'_e] ⊆ [t_1, t_2];
12         while k ≤ |L_out(u)_w| ∧ k' ≤ |L_in(v)_w'| do
13             [t_s, t_e] the k-th interval in L_out(u)_w;
14             [t'_s, t'_e] the k'-th interval in L_in(v)_w';
15             if [t_s, t_e] ⊈ [t_1, t_2] ∨ [t'_s, t'_e] ⊈ [t_1, t_2] then
16                 break;
17             else if max(t_e, t'_e) − min(t_s, t'_s) ≤ θ then
18                 return true;
19             else if t_e − t_s > θ ∨ t_s < t'_s then
20                 k ← k + 1;
21             else k' ← k' + 1;
22         i ← i + 1, i' ← i' + 1;
23     else i ← i + 1, i' ← i' + 1;
24 return false;
```

3.6 Experiments

Table 3.2: Network Statistics

Dataset	$\mathcal{M}$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\vartheta_{\mathcal{G}}$
CollegeMsg	D	1,899	59,835	16,736,181
Chess	D	7,301	65,053	99
Slashdot	D	51,083	140,778	1,157,361,660
MathOverflow	D	24,818	506,500	203,068,736
Facebook_f	U	63,731	817,035	1,232,231,923
Epinions	D	131,828	841,372	944
Facebook_wp	D	46,952	876,993	134,873,285
AskUbuntu	D	159,316	964,437	225,834,442
Enron	D	87,273	1,148,072	1,401,187,797
SuperUser	D	194,085	1,443,339	239,614,928
Digg	D	279,630	1,731,653	1,247,032,805
Wiki	U	118,100	2,917,785	239,001,193
Prosper	D	89,269	3,394,979	2,142
Arxiv	U	28,093	4,596,803	3,649
Youtube	U	3,223,589	9,375,374	225
DBLP	U	1,314,050	18,986,618	76
Flickr	D	2,302,925	33,140,017	197

We conducted extensive experiments to evaluate the performance of our proposed algorithms, summarized as follows:

- Online-Reach: Algorithm 1.
- Span-Reach: Algorithm 4.
- ES-Reach: a naive method to answer θ-reachability by invoking several runs of Span-Reach(). More details can be found in Section 3.5.2.
- ES-Reach*: Algorithm 5.
- TILL-Construct: A basic implementation of Algorithm 2. We use a queue to compute all SRTs and get CRTs by checking whether every SRT can be

covered by existing labels. More details can be found in Section 3.4.1.

- TILL-Construct*: Algorithm 3.

All algorithms were implemented in C++ and compiled using a g++ compiler at a -O3 optimization level. All the experiments were conducted on a Linux Server with an Intel Xeon 2.7GHz CPU and 180GB RAM.

Datasets. We conducted experiments on seventeen publicly-available real-world graphs. The detailed statistics of these datasets are summarized in Table 3.2. $\mathcal{M}$ demonstrates the types of datasets, where D represents the directed graph and U represents the undirected graph. $\vartheta_{\mathcal{G}}$ demonstrates the number of atomic units between the smallest timestamp and the largest timestamp. All networks and corresponding detailed descriptions can be found in SNAP[1] and KONECT[2].

The rest of this section is organized as follows. Section 3.6.1 provides the performance of answering span-reachability queries. Section 3.6.2 evaluates the index construction algorithms, and Section 3.6.3 reports the performance of answering θ-reachability queries.

3.6.1 Span-Reachability Query Processing

We evaluate the performance of span-reachability query processing. To generate input queries, we randomly pick 100 vertex pairs in each graph $\mathcal{G}$. For each vertex pair, we randomly generate subintervals of $[1, \vartheta_{\mathcal{G}}]$ and only keep intervals if the conditions in Lemma 9 and Lemma 10 are satisfied. We repeat this step until 10 intervals are found. This strategy works because the query algorithm is only invoked if the conditions in Lemma 9 and Lemma 10 hold. As a result, we fully prepare 1000 span-reachability queries. We report the running time of Span-Reach with Online-Reach as a comparison in Figure 3.3.

[1] `http://snap.stanford.edu/data/index.html`
[2] `http://konect.cc`

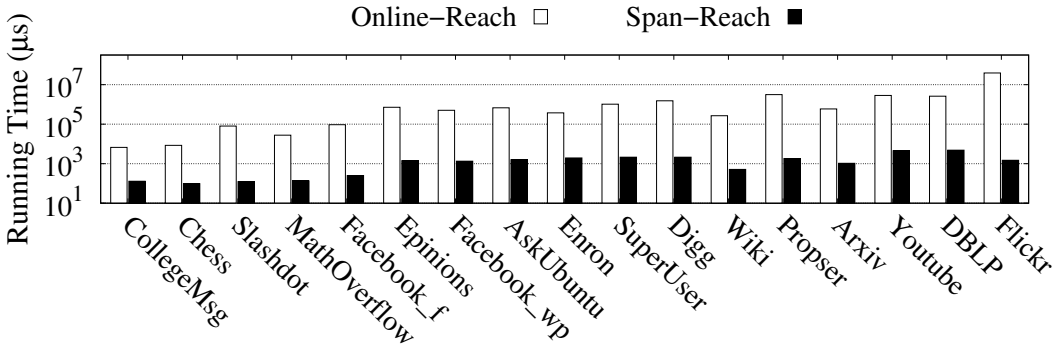


Figure 3.3: Performance of span-reachability query processing

We can see that the running time of Span-Reach is at least two orders of magnitude smaller than that of Online-Reach in all datasets in the experiment. For example, in the largest dataset Flickr, Online-Reach takes over 300 seconds while our Span-Reach algorithm takes only about 1.4 ms (1s= 10^3ms= $10^6\mu$s).

3.6.2 Index Construction

This section is devoted to evaluating the performance of index construction algorithms.

Index Size

We report the index size of all datasets in Figure 3.4, and also add the size of datasets as a comparison. We can find that in several large datasets, the index size is smaller than the graph size. For example, in Flickr, the dataset takes about 400 MB while the index takes only about 350 MB.

Indexing Time

The running time of TILL-Construct* for all datasets is reported with TILL-Construct as a comparison in Figure 3.5

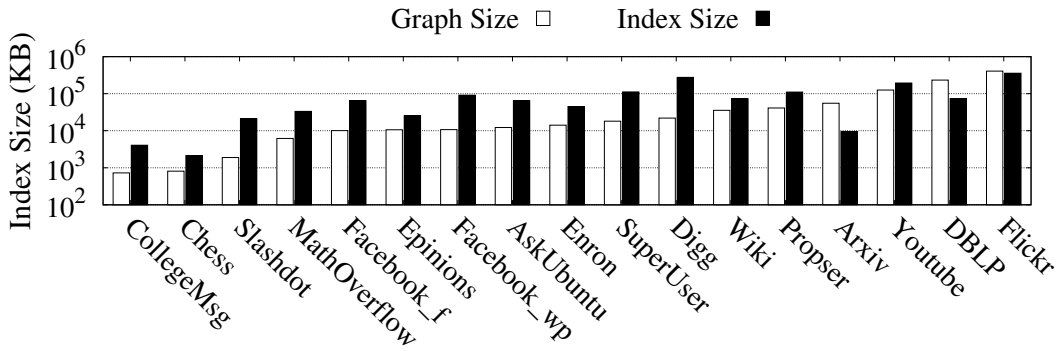


Figure 3.4: Index Size

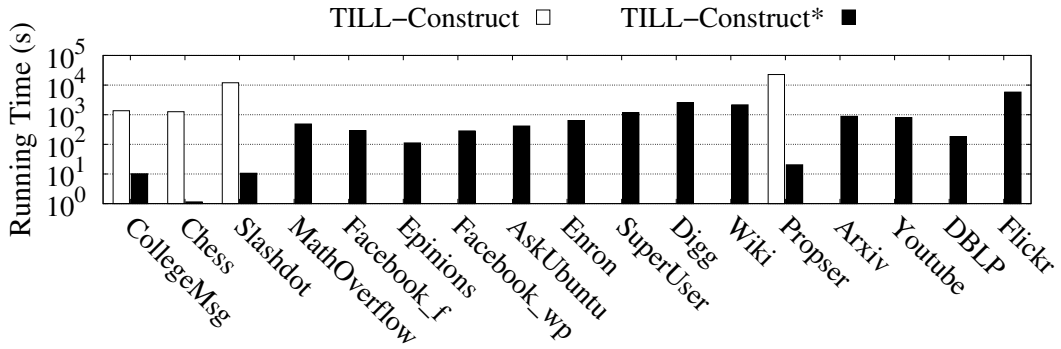


Figure 3.5: Indexing Time

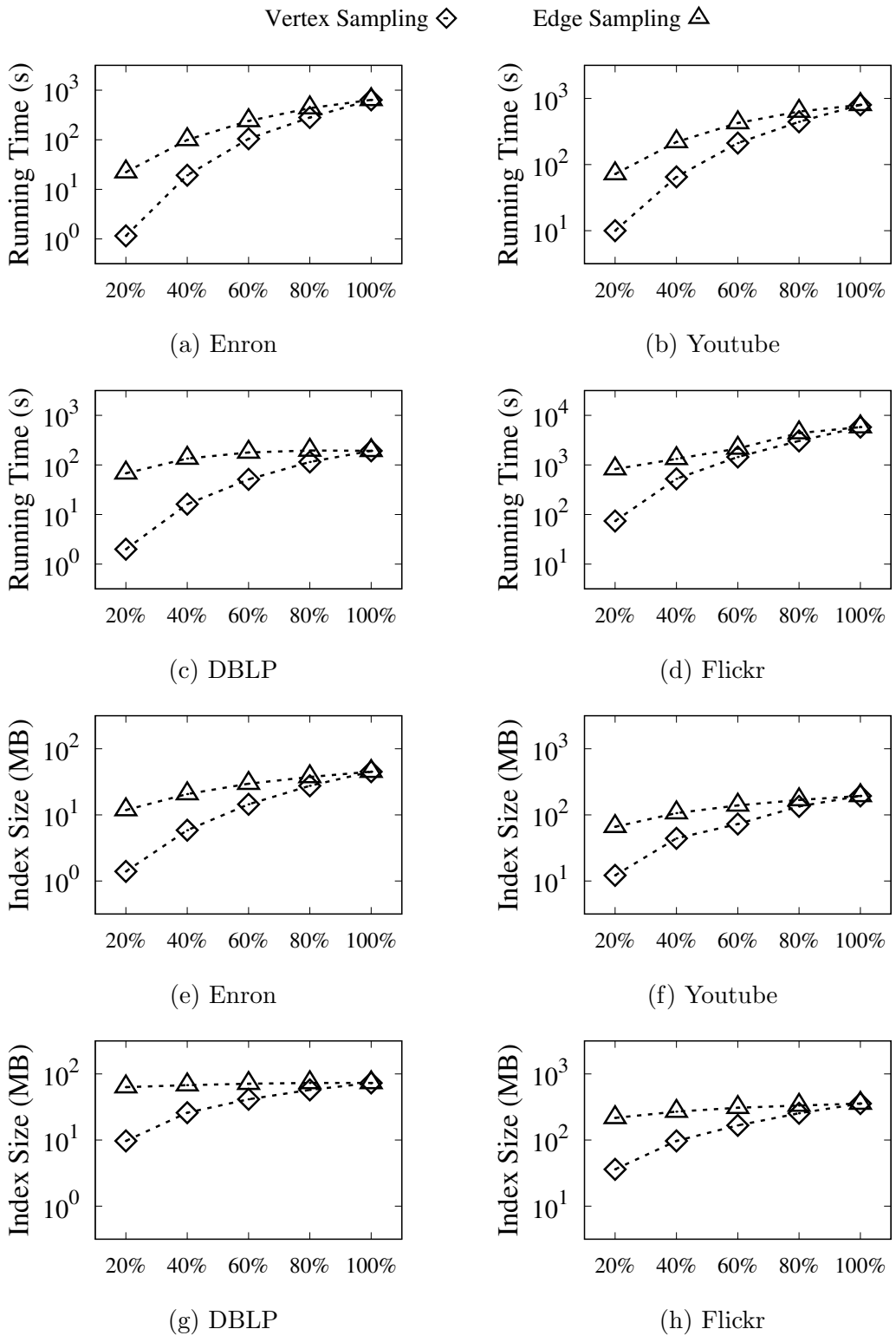
Note that the running time of TILL-Construct on several datasets are not given as the algorithm cannot finish in six hours. It is clear that in comparing all reported times of TILL-Construct, TILL-Construct* is at least two orders of magnitude faster. In the largest dataset Flickr, TILL-Construct* takes about 1.5 hours to compute TILL-Index. TILL-Construct* takes about 1 second on Chess, which is the shortest on all reported times. By contrast, the running time of TILL-Construct on Chess is about 20 minutes.

Varying ϑ

The running times and index sizes of TILL-Construct*() are presented in Figure 3.6 by varying the input parameter ϑ from 20% to 100% of $\vartheta_{\mathcal{G}}$ for each dataset $\mathcal{G}$. Note that $\vartheta = \vartheta_{\mathcal{G}}$ is equivalent to the default setting ϑ as $+\infty$. Due to limited space here, Figure 3.6 shows only the results of four datasets — Enron, Youtube, DBLP and Flickr. The results for other datasets display similar trends.

We can see from figures (a)–(d) that the increasing speed of running time becomes small when both the vertex and edge sampling ratio increases. For example, the running time of TILL-Construct* on Flickr is about 14 minutes when the edge sampling ratio is 20%. It reaches to 22 minutes, 35 minutes and 73 minutes when the edge sampling ratio is 40%, 60%, and 80% respectively. Finally, on the ratio of 100%, the time reaches about 90 minutes. The increasing trends for the index size in figures (e)–(h) are similar and even more gentle.

Figure 3.6 (a)–(d) reports the running times. We can see that the increases on both Enron and DBLP are not obvious (does not exceed 20 seconds) from 20% to 100%. The lines are almost linear in Youtube and Flickr, which start from about 500 seconds and 25 minutes, ending at about 750 seconds and 1.5 hours, respectively. Figure 3.6 (e)–(h) reports the index size. The change on all reported datasets is very small. The group of figures shows that the index size and indexing time are confined even though we do not set any interval length limitation ($\vartheta = +\infty$) in TILL-Construct*.



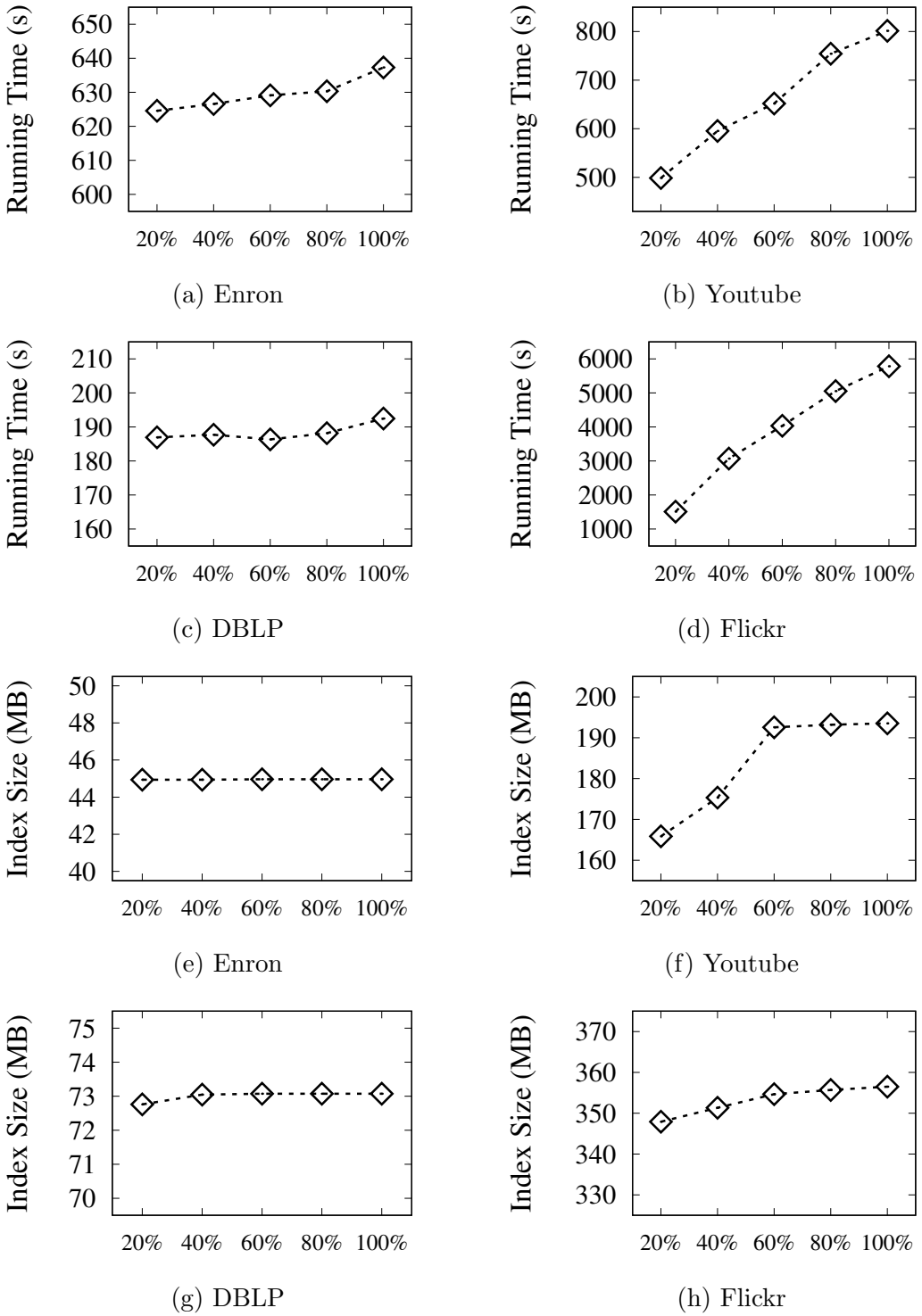
(a) Enron (b) Youtube

(c) DBLP (d) Flickr

(e) Enron (f) Youtube

(g) DBLP (h) Flickr

Figure 3.7: Scalability of index construction



(a) Enron (b) Youtube

(c) DBLP (d) Flickr

(e) Enron (f) Youtube

(g) DBLP (h) Flickr

Figure 3.6: Varying ϑ of TILL-Construct*

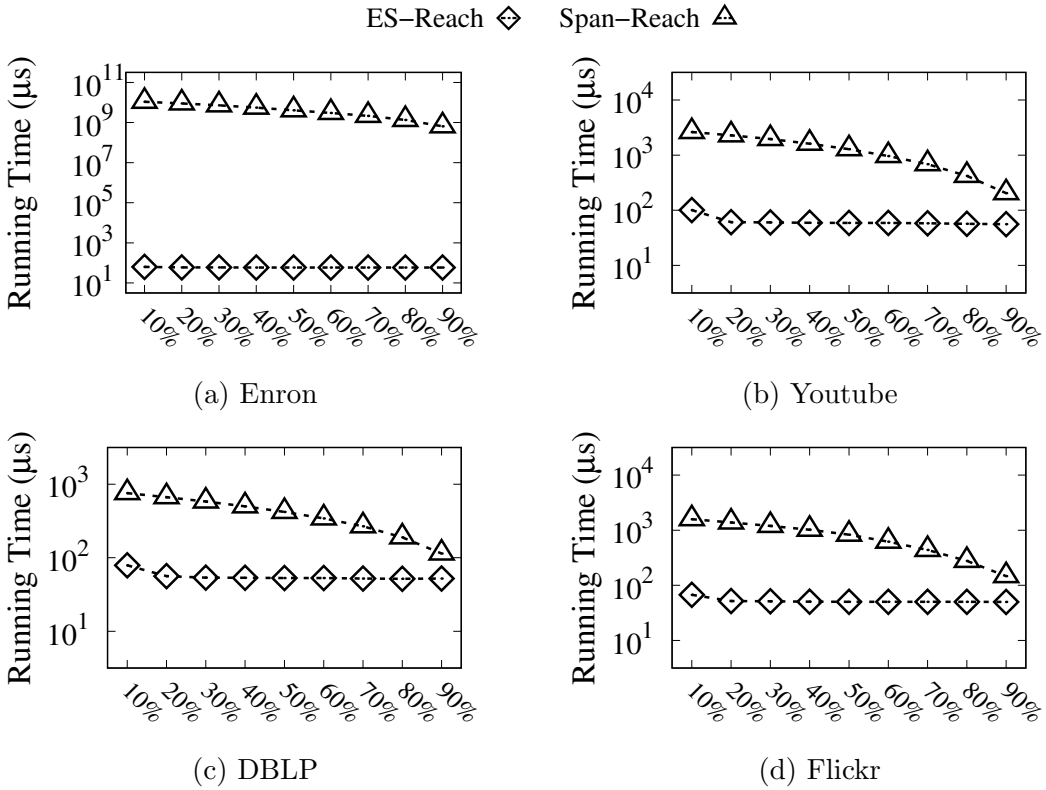
Scalability

This experiment tests the scalability of our index construction algorithm. Again, with limited space, we only report the results for four real-world graph datasets as representatives — Enron, Youtube, DBLP and Flickr. The results on other datasets show similar trends. For each dataset, we vary the graph size and graph density by randomly sampling vertices and edges from 20% to 100%. When sampling vertices, we derive the induced subgraph of the sampled vertices, and when sampling edges, we select the incident vertices of the edges as the vertex set.

3.6.3 θ-Reachability Query Processing

The performance of answering θ-reachability is evaluated next. To prepare the input queries, we adopt the same strategy described in Section 3.6.1 and randomly pick 100 vertex pairs and 10 intervals for each vertex pair. For each interval, we set θ as a fraction of its length, and adjust the fraction from 10% to 90%. The running time of ES-Reach* on four representative datasets is given in Figure 3.8, with ES-Reach as a comparison.

We can see from Figure 3.8 that ES-Reach* is faster than ES-Reach on all parameter settings. Their times trend towards equal when θ increases, since two algorithms are equivalent when θ is the length of the query interval. For the performance of ES-Reach*, it is clear that all lines present roughly downward trends.



(a) Enron (b) Youtube

(c) DBLP (d) Flickr

Figure 3.8: Performance of θ-reachability query processing

3.7 Conclusion

In this chapter, we define a span-reachability model to capture entity relationships in a specific period of temporal graphs. We propose an index-based method based on the concept of two-hop cover to answer the span-reachability query for any pair of vertices and time intervals. Several optimizations are given to improve the efficiency of index construction. We also study the problem of θ-reachability, which is a generalized version of span-reachability. We conduct extensive experiments on 17 real-world datasets to show the efficiency of our proposed algorithms.

Chapter 4

Time Interval Paths in Temporal Graphs

4.1 Overview

In this chapter, we introduce the proposed TDDL-DFS's methodological details along with its evaluation results.

In Section 4.2, we provide formal definitions of time interval paths and projected graphs of directed temporal graphs. Based on these ideals, we formally define the problem of hop-constrained time interval s-t (TIPST) path enumeration in the temporal network and introduces a straightforward method based on the Depth First Search (DFS). Following this, Section 4.3 puts forward the idea of path bundles. Section 4.4 studies a DFS-based approach leveraging dynamic distance labels. Section 4.5 proves the correctness and analyses the theoretical performance. Section 4.6 reports the experimental results. Section 4.7 concludes the chapter.

4.2 Preliminary

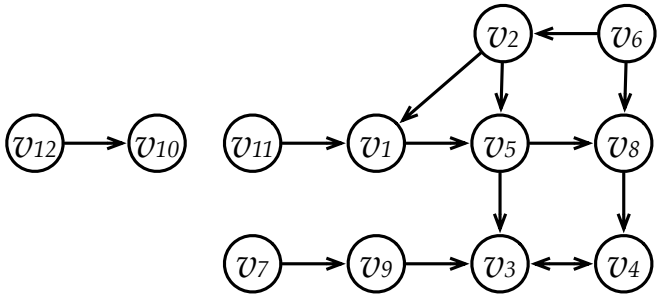
Given a directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents the set of vertices and $\mathcal{E}$ represents the set of temporal edges. In this graph, $e(u, v, t) \in \mathcal{E}$ represents a directed temporal edge from u to v at time t. Let n and m denote the number of vertices and edges, respectively, such that $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$. Additionally, we use θ to represent the number of time stamps, where $\theta = |\mathcal{T}|$, and $\mathcal{T}$ is the set of time stamps. We use *omega* to denote an inclusive time interval, where $\omega = [t_s, t_e]$ with t_s and t_e as starting and ending time.

Definition 5 (Time interval path)**.** *A time interval path p between two nodes $v_s, v_t \in \mathcal{V}$ within the inclusive interval ω is a sequence of temporal edges $(v_s, v_1, t_1), (v_1, v_2, t_2), ..., (v_{k-1}, v_t, t_k)$, such that for all $1 \leq i \leq k, t_i \in \omega$ and all edges in p belong to $\mathcal{E}$. We denote the length of path p as $|p|$, which corresponds to the number of hops. Another more compact notation for a time interval path is $v_s \xrightarrow{t_1} v_1 \xrightarrow{t_2} v_2 ... \xrightarrow{t_k} v_t$. Given a hop constraint k, a path p is hop constrained if $|p| \leq k$.*

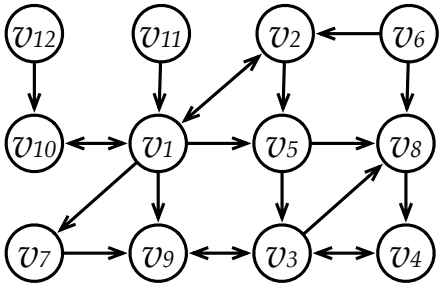
A time interval path is considered simple if each vertex in p is distinct. When referring to time interval paths, we will focus on simple time interval paths.

Definition 6 (Projected Graph)**.** *Given a time interval ω, the projected graph of $\mathcal{G}$ in ω is denoted as $\mathcal{G}_{t_s,t_e}$. The set of vertices $V(\mathcal{G}_{t_s,t_e}) = V$ and the set of edges $E(\mathcal{G}_{t_s,t_e}) = (u, v)|e(u, v, t) \in \mathcal{E}, t \in \omega$.*

In particular, we refer to the corresponding projected graph of $[0, +\infty]$ as a static graph, denoted as $\mathcal{G}_s$, where $V(\mathcal{G}_s) = \mathcal{V}, E(\mathcal{G}_s) = (u, v)|e(u, v, t) \in \mathcal{E}$. Additionally, we use $m_s = |E(\mathcal{G}_s)|$ to denote the number of edges in the static graph. Note that any projected graph or static graph is simple, and it can be disconnected. For the temporal graph in Figure 1.1, its projected graph in the



(a) Projected graph in $[3, 6]$



(b) Static graph

Figure 4.1: Corresponding projected graph and static graph

interval $[3, 6]$ is shown in Figure 4.1a, and the corresponding static graph is illustrated in Figure 4.1b.

4.2.1 Problem statement

In this work, we study the problem of hop-constrained time interval *s-t* (TIPST) path enumeration in the temporal network, which is defined as follows.

Definition 7 (hop-constrained time interval *s-t* path enumeration)**.** *Given a temporal network $\mathcal{G}(\mathcal{V}, \mathcal{E})$, a hop-constrained time interval s-t path query $q(v_s, v_t, k, \omega)$, we aim to enumerate all simple time interval paths $P(q)$ from v_s to v_t within ω such that $\forall p \in P, |p| \leq k$. To make it clear, we name the problem as TIPST here and throughout the chapter.*

Suppose the query is $q(v_1, v_3, 5, [3, 6])$ for the temporal graph in Figure 1.1, the paths satisfying the query are illustrated in Figure 4.2, note that different

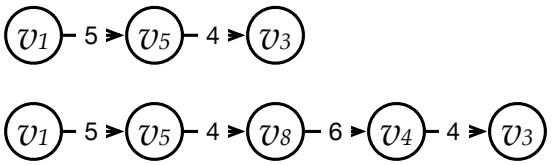


Figure 4.2: An example of time interval paths

from simple paths, they are stamped with time intervals in $[3, 6]$.

4.2.2 Straightforward Method

Hop-Constrained Time Interval Depth-First-Search. One straightforward approach to solve the hop-constrained temporal interval path enumeration problem is to perform a Depth-First Search (DFS) starting from the source node s with a maximum of k hops. To maintain the current state during the search, we use two stacks: one for node IDs ($\mathcal{S}_i$) and another for timestamps ($\mathcal{S}_t$). The size of the stack is denoted by $|\mathcal{S}|$, and $p(\mathcal{S}_i, \mathcal{S}_t)$ represents the corresponding temporal path with a length equal to $|p(\mathcal{S}_i, \mathcal{S}_t)|$, which can be either $|\mathcal{S}_i| - 1$ or $|\mathcal{S}_t|$. Importantly, we ensure not to revisit any node already present in $\mathcal{S}_i$ to ensure the output path remains simple (Line 5). Additionally, we verify whether the timestamp t on each edge satisfies the query constraint (Line 5), ensuring that t is no earlier than the starting time t_s and no later than the ending time t_e. The time complexity of the TI-DFS algorithm is $O((n \cdot \theta)^k)$, where n is the number of vertices and θ is the number of timestamps, and the space complexity is $O((n \cdot \theta)^k)$. Please note that the factor of $n \cdot \theta$ arises from considering the presence of n potential vertices and θ distinct timestamps at each search depth.

4.3 Temporal Bundled Solution

In this section, we present a bundled solution tailored for the TIPST query.

Algorithm 6: TI-DFS($u, t_c, \mathcal{S}_i, \mathcal{S}_t$)

Input: A vertex u, a timestamp t_c, a stack of IDs of vertices $\mathcal{S}_i$ and a stack of timestamps $\mathcal{S}_t$

```
1 S_i.push(u), S_t.push(t_c);
2 if u == t then
3     output p(S_i, S_t) where P_t ← P_t ∪ p(S_i, S_t);
4 else if |p(S_i, S_t)| < k then
5     for each out-going edge {v, t_uv} of u where v ∉ S_i, t_uv ∈ [t_s, t_e] do
6         TI-DFS(v, t_uv, S_i, S_t);
7 u is unstacked from S_i and t_c is unstacked from S_t;
```

4.3.1 Motivation

As mentioned earlier, TI-DFS is a straightforward adaptation from simple graphs to temporal graphs for handling TIPST queries. However, this approach can lead to redundant searches in our specific case. The problem arises from the failure to fully consider the projected graph with a given interval of the temporal graphs. Consequently, unnecessary computations are performed, impacting the overall efficiency of the algorithm.

Example 10. *Consider the process of Algorithm 6 for the given temporal graph shown in Figure 1.1 and the query* $q(v_2, v_4, 3, [1, 6])$*. Initially, we have two possible choices for the outgoing edges from* v_2*:* $e(v_2, v_1, 6)$ *and* $e(v_2, v_5, 3)$*. However, selecting* $e(v_2, v_1, 6)$ *would lead to a search depth that exceeds the hop constraint of 3, resulting in no valid output. Therefore, we proceed with the edge* $e(v_2, v_5, 3)$*, which starts from* v_2 *within the time interval* $[1, 6]$*. In the next hop, from* v_5*, there are three optional edges:* $e(v_5, v_3, 4)$*,* $e(v_5, v_8, 1)$*, and* $e(v_5, v_8, 4)$*. All of these edges fall within the time interval* $[1, 6]$*. Depending on the chosen edge, the rest of the path will be different. If we select the branch with* $e(v_5, v_3, 4)$*, the subsequent path could either be* $\{e(v_3, v_4, 1)\}$ *or* $\{e(v_3, v_4, 5)\}$*. On the other hand, if we choose* $e(v_5, v_8, 1)$ *or* $e(v_5, v_8, 4)$*, the rest of the path will be* $\{e(v_8, v_4, 6)\}$*.*

In the example provided, after removing the edge $e(v_5, v_8, 1)$ from $\mathcal{S}_i$ and $\mathcal{S}_t$, the subsequent search branch starts from $e(v_5, v_8, 4)$. Remarkably, we observe that despite having different timestamps, $e(v_5, v_8, 1)$ and $e(v_5, v_8, 4)$ share the same vertex ID. Consequently, the algorithm ends up repeatedly searching the same branch from the same node v_8, resulting in redundant computations. We note that revisiting the same branch is inevitable since the states in $\mathcal{S}_t$ do not affect the subsequent checks in line 5 of Algorithm 6. Even worse, it leads to considerable space costs. For instance, assuming an average of 10 timestamps between each pair of adjacent vertices and a hop constraint of 4 (a moderate case in real life), the runtime memory cost would be increased up to 10^4 times, most of which is redundancy and unnecessary. To overcome this issue and improve efficiency, we propose a novel data structure that leverages the projected graph of the temporal graph with a given interval to compactly organize the involved timestamps. This approach effectively reduces redundant searches and substantially improves space efficiency, leading to a more optimized and practical solution for path enumeration in temporal graphs.

4.3.2 Bundled Time Interval DFS Approach

To optimise the number of repeated visits in DFS, we proposed the idea of bundle for TIPST path enumeration.

Definition 8 (TIPST bundle)**.** *A TIPST bundle B in a temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ satisfying $q(v_s, v_t, k, \omega_q)$ consists of a sequence of vertices $v_0 = v_s, \ldots, v_l = v_t$ and sets of time intervals $\{\omega_1, ..., \omega_l\}$ such that $1 \leq l \leq k$ and $\forall 1 \leq i \leq l, \omega_i \subset \omega_q$. Additionally, for any edge $(v_i, v_{i+1}, t) \in \mathcal{E}$, we have $t \in \omega_i$ if $t \in \omega_q$. We denote the TIPST bundle B as $v_0 \xrightarrow{\omega_1} v_1 \xrightarrow{\omega_2} v_2 ... \xrightarrow{\omega_l} v_l$.*

The set of time interval paths represented by B, denoted $\mathcal{P}(B)$ is defined as:

$\mathcal{P}(B) = \{v_0 \xrightarrow{t_1} v_1 ... \xrightarrow{t_k} v_k | \forall i : t_i \in \omega_i \& (v_{i-1}, v_i, t_i) \in \mathcal{E}\}$

A TIPST bundle is called minimal if $\forall i = 1...k, \omega^- \subset \omega_i$ it holds that

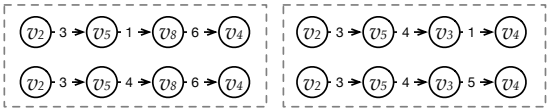
$\mathcal{P}(v_0 \xrightarrow{\omega_1} ...v_i \xrightarrow{\omega_i \setminus \omega^-} ... \xrightarrow{\omega_k} v_k) \subsetneq \mathcal{P}(v_0 \xrightarrow{\omega_1} ...v_i \xrightarrow{\omega_i} ... \xrightarrow{\omega_k} v_k)$

Lemma 11. *Let B be a TIPST bundle, there exists a unique minimal TIPST bundle B' such that $\mathcal{P}(B) = \mathcal{P}(B')$.*

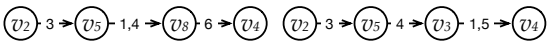
Remarkably, we define the length of a TIPST bundle B as $|B|$, which corresponds to the number of hops. For any path $p \in P(B)$ within the bundle, we have $|p| = |B|$.

Lemma 12. *Let B be a TIPST bundle, if B is constrained by a maximum of k hops, then for any path $p \in P(B)$, we have $|p| \leq k$.*

This lemma ensures that TIPST can accurately represent all valid interval paths for a given query $q(s, t, k, \omega)$. The bundle B containing all paths with a maximum of k hops from s to t within the specified time interval ω correctly captures all the feasible paths, allowing for efficient and accurate enumeration.



(a) Time interval paths



(b) TIPST bundles

Figure 4.3: An example of time interval paths to TIPST bundles

Figure 4.3 showcases the results for the query $q(v_2, v_4, 3, [1, 6])$ in Figure 1.1. The paths that belong to the same bundle are effectively grouped with dotted boxes in Figure 4.3a, while the corresponding bundles are visually represented in Figure 4.3b. Besides, the length of bundles equals the length of contained paths, which is 4 in the figure. It is important to note that paths can be bundled only

if they share the same sequences of vertex IDs. This bundle-based optimization significantly reduces the search space by eliminating redundant computations, thus enhancing the overall efficiency of the algorithm for time interval path enumeration in temporal graphs.

4.3.3 Algorithm Description

Algorithm 7: TI*-DFS($u, itv_c, \mathcal{S}_i, \mathcal{S}_{itv}$)

Input: A vertex u, a time interval itv_c, a stack of IDs of vertices $\mathcal{S}_i$ and a stack of time intervals $\mathcal{S}_{itv}$

```
1 S_i.push(u), S_itv.push(itv_c);
2 if u == t then
3     output B(S_i, S_itv) where B_t ← B_t ∪ B(S_i, S_itv);
4 else if |B(S_i, S_itv)| < k then
5     for each out-going neighbor v of u such that v ∉ S_i do
6         find the valid interval itv_(u,v) from u to v with binary search;
7         if itv_uv is not empty then
8             TI*-DFS(v, itv_(u,v), S_i, S_itv);
9 u is unstacked from S_i and itv_c is unstacked from S_itv;
```

In practical implementation, we efficiently track the valid sets of timestamps using time intervals, which are usually organized in non-decreasing order, representing the chronological order in most applications. To represent the corresponding TIPST bundle $B(\mathcal{S}_i, \mathcal{S}_{itv})$, we utilize a stack of vertex IDs $\mathcal{S}_i$ and another stack of intervals $\mathcal{S}_{itv}$ in the TI*-DFS algorithm (Algorithm 7). For each pair of adjacent vertices (u, v), the time set $\omega_{(u,v)}$ can be represented as $[t_{\min}, t_{\max}]$, where $t_{\min}$ and $t_{\max}$ are the lower and upper bounds of the timestamps between u and v that satisfy the query interval $[t_s, t_e]$. Since the timestamps between vertices are naturally organized in non-decreasing order as ground truth in most real-life scenarios, we can efficiently obtain $itv_{(u,v)}$ using binary search

in line 6 and enumerate the timestamps within that interval when necessary. If the interval is valid, we proceed with the search in line 8. We output results if we find the terminal vertex (lines 2-3) or backtrack (line 9) during the traversal. This efficient representation and retrieval of time intervals using stacks effectively optimize the performance for time interval path enumeration in temporal graphs.

Example 11. *Consider the same query* $q(v_2, v_4, 3, [1, 6])$ *as in Example 10, where we aim to find the path between vertices* v_2 *and* v_4 *with a hop constraint of 3 within the time interval* $[1, 6]$*. The search begins at* v_2*, and after identifying* v_1 *as a fruitless branch, we proceed to explore* v_5*. We push* v_5 *into the stack* $\mathcal{S}_i$ *and record the interval* $[0, 1)$ *in the stack* $\mathcal{S}_{itv}$*. This interval indicates that we can retrieve valid timestamps between* v_2 *and* v_5 *from the* 0*-th position (inclusive) to the* 1*-th position (exclusive) in the set* 3*. In the next hop, when we push* v_3 *into the stack* $\mathcal{S}_i$ *and update the interval to* $[0, 1)$*, we successfully reach the target vertex* v_4*. The updated stacks are* $\mathcal{S}_i = v_2, v_5, v_3, v_4$ *and* $\mathcal{S}_{itv} = [0, 1), [0, 1), [0, 2)$*. It is noteworthy that there is no need to store timestamps explicitly with the graph in the memory as ground truth, which helps optimize space usage. Likewise, for the next output in the other branch, the stacks become* $\mathcal{S}_i = \{v_2, v_5, v_8, v_4\}$ *and* $\mathcal{S}_{itv} = \{[0, 1), [0, 2), [0, 1)\}$ *respectively. This effective bundling and pruning strategy in Algorithm 7 enable us to avoid repeated searches and output more compact results, significantly enhancing the efficiency of the path enumeration process. Better still, it also optimizes memory usage, making our algorithm a more practical and effective solution.*

Lemma 13. *The time complexity of TI*-DFS is* $O((n \cdot \log \theta)^k)$ *and the space complexity is* $O(n^k)$*.*

Note that $\log \theta$ in the time complexity of the algorithm is due to the two binary searches performed for each outgoing neighbor. Additionally, the value

of θ no longer affects the space complexity, as we only need to store an interval (two integers) between any pairs of adjacent vertices.

In our evaluation of real-world datasets, we observed a wide range of values for θ, ranging from 100 to 1,000,000. This significant variability indicates considerable costs associated with the baseline approach. However, the introduction of the TIPST bundle idea not only accelerates the algorithm but also results in a reasonable runtime space cost, making it a more efficient and practical solution.

4.4 Dynamic Distance Label

In this section, we present a polynomial delay algorithm, namely TDDL-DFS.

4.4.1 Motivation

TI*-DFS utilizes the temporal constraint, but it falls short in exploring the hop constraint, resulting in reduced efficiency. As a remedy, we introduce a pioneering approach called Temporal Dynamic Distance Labelling DFS (TDDL-DFS) to address these limitations. Our method aims to learn from previous shortcomings by employing a strategy of blocking and increasing distance labels for nodes in dead-end branches. Conversely, we unblock and decrease labels for nodes whenever a solution is obtained, allowing for improved exploration and search efficiency.

As demonstrated in Example 11, although v_1 is a valid choice within the interval $[1, 6]$, it yields no output in this branch. Furthermore, it results in excessive and unnecessary computations due to the considerable number of out-neighbors of v_1. To resolve this issue, a deeper analysis of the distances between nodes becomes imperative.

4.4.2 Distance Label

One might question the potential benefits of using static labels to record the distances between vertices in order to prune unnecessary branches. On the one hand, the time complexity of this method, $O((n \cdot \log \theta)^k)$, remains non-polynomial in theory. On the other, we have observed that static labels cannot accurately reflect the current states of stacks and outputs in practice. Specifically, the actual distance from a vertex u to the target t should be increased if the intermediate vertices between u and t have already been visited. One possible approach to address this is to aggressively update the distance labels while updating the stacks, as seen in T-DFS [55]. However, this method incurs prohibitive overhead due to the BFS operation required for label updates. Instead, we adopt a passive approach that focuses on the results of each search branch to update the labels. The core idea behind our proposed method is to increase the labels whenever encountering a dead end and decrease them whenever nodes in between produce valid outputs.

4.4.3 Algorithm Description

The pseudo-code for our proposed Temporal Dynamic Distance Labelling DFS (TDDL-DFS) approach is illustrated in Algorithm 8. The algorithm begins with the initial call to TDDL-DFS($s, t_c = \emptyset, \mathcal{S}_i = \emptyset, \mathcal{S}_{itv} = \emptyset$) for the given query $q(s, t, k, \omega)$.

In the algorithm, $d_v(u)$ represents the distance from vertex u to vertex v, while $d'_v(u)$ represents the distance from vertex v to vertex u. During initialization, we perform a BFS from s to update $d'_s(u)$ and a reverse BFS from t to update and $d_t(u)$ respectively. Specifically, if there is no path between them or $d'_s(u) + d_t(u) > k$, we set both $d'_s(u)$ and $d_t(u)$ to $k + 1$.

Algorithm 8: TDDL-DFS($u, itv_c, \mathcal{S}_i, \mathcal{S}_{itv}$)

Input: A vertex u, a time interval itv_c, a stack of IDs of vertices $\mathcal{S}_i$ and a stack of time intervals $\mathcal{S}_{itv}$

```
1  S_i.push(u), S_itv.push(itv_c);
2  D ← k + 1;
3  block u;
4  if u == t then
5      output B(S_i, S_itv) where B_t ← B_t ∪ B(S_i, S_itv);
6      u is unstacked from S_i and itv_c is unstacked from S_itv;
7      return 0;
8  else if |B(S_i, S_itv)| < k then
9      foreach out-going neighbor v of u such that
         v ∉ S_i, |B(S_i, S_itv)| + d_t(v) ≤ k do
10         find the valid interval itv_(u,v) from u to v with binary search;
11         if itv_(u,v) is not empty then
12             d ←TDDL-DFS(v, itv_(u,v), S_i, S_itv);
13             if d < k + 1 then D ← min(D, d + 1) ;
14 if D == k + 1 then
15     d_t(u) ← k + 1 − |B(S_i, S_itv)|;
16 else UpdateL(S_i, u, D, k) ;
17 u is unstacked from S_i and itv_c is unstacked from S_itv;
18 return D;

1  Procedure UpdateL(S_i, u, l, k)
2      if d_t(u) > l or u is blocked and d_t(u) == l then
3          d_t(u) ← l;
4          unblock u;
5          foreach incoming neighbor v of u do
6              if v ∉ S_i and d'_s(v) + d_t(u) + 1 <= k then
7                  UpdateL(S_i, v, d_t(u) + 1, k);
```

For each visited node u, we initially assume that it cannot reach t and set the temporal record of distance D to $k+1$ in Line 2. If we find the target, i.e., $u == t$, we add the current TIPST bundle to the result in Line 4. If there are remaining hops, we continue the search through the out-neighbors of u in Lines 8-13. For each neighbor $v \notin \mathcal{S}_i$, we check if its recorded distance to the target $d_t(v)$ is available for the remaining hops, as well as the existence of valid temporal intervals in Lines 9-11.

If there is no valid output after exploring all possible search branches, that is D remain unchanged, we need to increase the distance label $d_t(u)$ and set the value to $k+1-|B(\mathcal{S}_i, \mathcal{S}_{itv})|$ in Line 15. This indicates that t cannot be reached with $k-|B(\mathcal{S}_i, \mathcal{S}_{itv})|$ hops left and we should never visit it when the stack has the same size as $|B(\mathcal{S}_i, \mathcal{S}_{itv})|$. Note that the updated value $k+1-|B(\mathcal{S}_i, \mathcal{S}_{itv})|$ will always be larger than the label value of $d_t(u)$ before in this case. It is because if $d_t(u) \geq k+1-|B(\mathcal{S}_i, \mathcal{S}_{itv})|$ before u is visited, u cannot pass the check of hop constraint in Line 9 and will never be visited. Otherwise, if D is updated, it means that u is included in any valid output considering the current stack. In this case, we need to broadcast to unblock and decrease the labels of potential vertices in Line 16, as shown in UpdateL. It is important to note that d represents the number of hops from v to t, and D is set to the minimum value of $d+1$ in Line 10 to reflect the existence of shortest valid outputs. Additionally, UpdateL only continues broadcasting when $d_t(u) > l$ or when u is blocked and $d_t(u) == l$. It skips any neighbor v if it has been in $\mathcal{S}_i$ or $d'_s(v) + d_t(u) + 1 > k$, which does not have the potential to reach t within k hops.

Example 12. *Suppose the temporal graph is as shown in Figure 1.1, and the query is* $q(v1, v_9, 8, [0,6])$. *Assume the current state is* $\mathcal{S}_i = \{v_1, v_2, v_5, v_3\}$ *and* $\mathcal{S}_{itv} = \{[0,1), [0,1), [0,1)\}$, *there are three possible search branches starting from* v_4, v_8, v_9 *respectively. Starting with* v_4, *it passes the hop-constraint check since*

$|\mathcal{S}_i| + d_t(v_4) < 8$. However, there is no valid output due to the collision with v_3 in the stack. Consequently, $d_t(v_4)$ is increased to 4 when v_4 is popped. Moving on to the next candidate, v_9 is the target and produces a valid TIPST bundle for $\mathcal{S}_i = \{v_1, v_2, v_5, v_3, v_9\}$ and $\mathcal{S}_{itv} = \{[0,1), [0,1), [0,1), [0,1)\}$. On the other hand, the last option, v_8, cannot pass the interval check and will not be visited. When v_3 is popped, it triggers UpdateL since $d_t(v_3) == 1$ and v_3 is blocked. The update is then broadcasted to v_4, correctly setting $d_t(v_4)$ to 2. During the subsequent check of v_4 when $\mathcal{S}_i = \{v_1, v_2, v_5, v_8\}$ and $\mathcal{S}_{itv} = \{[0,1), [0,1), [0,2)\}$, v_4 successfully passes the hop-constraint check. As a result, we correctly obtain the next output when $\mathcal{S}_i = \{v_1, v_2, v_5, v_8, v_4, v_3, v_9\}$ and $\mathcal{S}_{itv} = \{[0,1), [0,1), [0,2), [0,1), [0,1)\}$.

4.5 Analysis

We begin by focusing on the correctness of distance labels for each vertex. Given the current TIPST bundle $B(\mathcal{S}_i, \mathcal{S}_{itv})$, and we suppose there exists a path p from u to t for the given query. For for every vertex $v \in p$, excluding u and not in $\mathcal{S}_i$, we define $d_t(u)$ as correct if and only $d_t(u) \leq |p|$. It is worth noting that $d_t(u)$ is always considered correct when $u \in \mathcal{S}_i$.

Lemma 14. *Given a temporal graph $\mathcal{G}$, a TIPST query q and a vertex $v \in \mathcal{V}$, if there exists a TIPST bundle D from s to t with $|p(B)| \leq k$, and suppose the distance labels of every vertex is corrected maintained, then for any $v \in p$ at the i-th position where $0 \leq i < |p|$, we have $d'_s(v) \leq i$ and $d_t(v) \leq k - i$.*

Proof. If $0 \leq i < |p|$, then $v = p[i]$ must be reachable from s and reachable to t within i and $|p| - i$ hops respectively given the current distance label. Moreover, we have $|p| \leq k$ based on the definition of TIPST query, thus, $d_t(v) \leq |p| - i \leq k - i$, which suggests it should be reachable to t within $k - i$ hops. □

We proceed to prove the correctness of updating dyamic distance labels in Algorithm 8.

Theorem 6. *For any vertex u, the distance label $d_t(u)$ is correctly updated in Algorithm 8.*

Proof. If the vertex u is visited and pushed into $\mathcal{S}_i$, $d_t(u)$ is always correct since it will never be updated until it is popped. When updating the value of $d_t(u)$ after exploring all possible search branches, there are two possible cases: increasing or decreasing $d_t(u)$. In the case where there is no valid output as in Line 14, $d_t(u)$ is increased since $d_t(u)+|B(\mathcal{S}_i, \mathcal{S}_{itv})| < k+1$ for u to pass the hop-constraint check. Otherwise, $d_t(u)$ is set to D and we broadcast the changes through UpdateL in Line 16. Note that $D \leftarrow min(D, d+1)$ suggests D is correctly updated with the minimal distance to the target t considering the current $\mathcal{S}_i$. Thus, $d_t(u)$ is correctly updated when u is popped from $\mathcal{S}_i$.

Another case is when u is not present in $\mathcal{S}_i$ and is updated through UpdateL. In this case, it must satisfy one of two conditions: either $d_t(u) > l$ or u is blocked with $d_t(u) == l$. If $d_t(u) > l$, it indicates that u has been visited before, and the label is increased due to fruitless explorations. $d_t(u)$ is decreased and unblocked in this case since there exists a path from u to t according to Line 6 in UpdateL. Likewise, we correctly unblock u when u is blocked and $d_t(u) == l$. Thus, $d_t(u)$ is correctly updated through UpdateL. □

According to this theorem, the distance labels are correctly maintained. Given the correctness of the updates of labels, the correctness of Algorithm 8 immediately follows since it is based on dynamic distance labels.

Theorem 7. *Algorithm 8 is a polynomial delay algorithm with $O(km_s \log\theta)$ time per output. The time complexity of it is $O(km_s\delta \log\theta)$, where δ is the number of TIPST bundles. The space complexity of it is $O(k\delta)$.*

Proof. Suppose a vertex u is unstacked twice while there is no new output in Algorithm 8. Let $|S_1|$ and $|S_2|$ denote the stack size after u is pushed into $\mathcal{S}_i$ at the first and the second time, respectively. After u is unstacked for the first time, $d'_t(u)$ is increased to $k - |S_1| + 2$. Since no new output is generated, the UpdateL function will not be triggered, and therefore, the distance label remains unchanged when u is pushed into the stack for the second time.

When u is visited for the second time, it implies that $|S_2| + d'_t(u) \leq k$, and we can deduce that $|S_2| < |S_1|$. This is because the label will be increased by at least one when encountering a dead end and being unstacked. Since $d'_t(u) < k$, it indicates that a vertex cannot be visited and pushed into a stack more than k times unless there is any output. Additionally, an edge in the static graph will be visited whenever u is pushed into the stack. Considering the cost of unblocking in the UpdateL function, an edge in the static graph will be visited at most $k+1$ times before a new output is found or the call terminates.

It is important to note that each time an edge is visited, we check the existence of a valid interval, which incurs a cost of $\log \theta$. Therefore, TDDL-DFS is a polynomial delay algorithm with $O(km_s \log \theta)$ time complexity per output. The overall time complexity of the algorithm is $O(km_s\delta \log \theta)$.

The space complexity of Algorithm 8 is $O(k\delta)$ since we need to spare $O(k)$ space for each valid output.

□

4.6 Experimental Study

4.6.1 Experimental Setting

We conducted experiments on 8 publicly-available real-world graphs. The detailed statistics of these datasets are summarized in Table 4.1. θ is the number

of atomic units between the smallest timestamp and the largest timestamp. m_s is the number of edges in the accordingly static graph. All networks and corresponding detailed descriptions can be found in SNAP[1] and KONECT[2].

Table 4.1: Real-world networks

Dataset	n	m	m_s	θ
CollegeMsg	1,899	59,835	20,296	58,911
Chess	7,301	65,053	60,046	100
Slashdot	51,083	140,778	131,175	90,345
MathOverflow	24,818	506,550	239,978	505,784
Epinions	131,828	841,372	841,372	936
Facebook	46,952	876,993	274,086	867,939
AskUbuntu	159,316	964,437	596,933	960,866
Prosper	89,269	3,394,979	3,330,225	1,259

We conducted extensive experiments to evaluate the performance of our proposed algorithms:

- TI-DFS: Algorithm 6 considering the hop constraint and time interval.
- TI*-DFS: Algorithm 7, a bundled DFS considering the hop constraint and time interval.
- TDDL-DFS: Algorithm 8, the proposed method with dynamic distance labels.

All algorithms were implemented in C++ and compiled using a g++ compiler with the -O3 optimization level. All the experiments were conducted on a Linux Server with an Intel Xeon 2.7GHz CPU and 180GB RAM.

The rest of this section is organized as follows. Section 4.6.3 compares the performance of baselines answering TIPST queries. Section 4.6.3 illustrates the

[1] `http://snap.stanford.edu/data/index.html`
[2] `http://konect.cc/`

effectiveness of bundles in compression. Section 4.6.5 demonstrates the scalability of the proposed method with varying hop constraints k and time intervals t.

4.6.2 Metrics

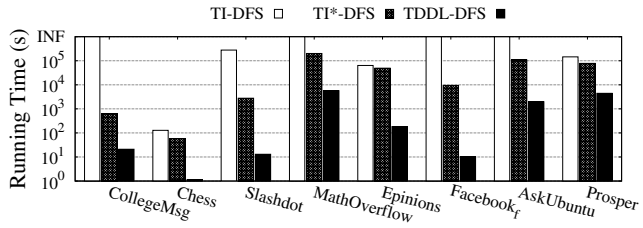
We next describe our metrics of query performance in experiments.

Compression ratio. Suppose the cost to store TIPST bundles equals the number of output $|\mathcal{B}|$, and the corresponding number of time interval paths is $|\mathcal{P}(\mathcal{B})|$, the compression ratio, CR of the set of TIPST bundles $\mathcal{B}$ is $CR = \frac{|\mathcal{P}(\mathcal{B})|}{|\mathcal{B}|}$.

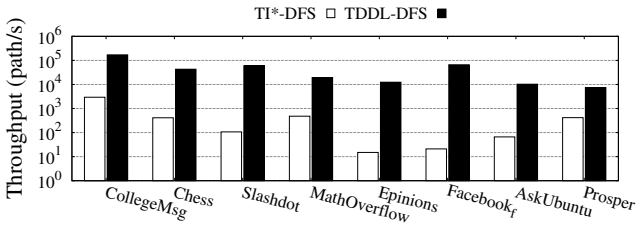
Throughput. Suppose the time cost to get $|\mathcal{B}|$ TIPST bundles for query $q(s, t, k, itv)$ is t, the throughput to measure the speed of q is defined to be $\frac{|\mathcal{B}|}{t}$.

4.6.3 Comparison with Baselines

In this subsection, we conduct an evaluation of the time efficiency of the algorithms by analyzing their query response time and throughput. We perform a random generation of 30 query pairs (s, t) for each time interval, where the source vertex s can reach the target vertex t within k hops in different windows of size $\theta/3$. We begin by comparing the average running time of the three algorithms across all eight datasets. For Prosper, Slashdot, and Facebook, we set $k = 8$ to achieve similar performance with other graphs, while for the remaining datasets, we set $k = 6$. In cases where an algorithm fails to complete within 72 hours, we assign its runtime as Inf. The results, illustrated in Figure 4.4a, demonstrate that the proposed TDDL-DFS algorithm outperforms the other two algorithms significantly. TDDL-DFS achieves query response times that are 1-3 orders of



(a) Running time



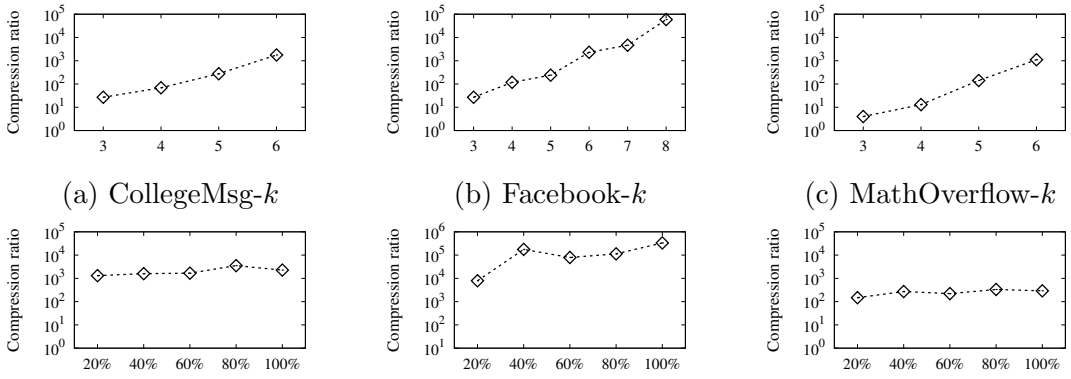
(b) Results per second

Figure 4.4: Comparison with baselines on running time and throughput

magnitude faster than TI*-DFS, while TI-DFS often exceeds the time limit. The inferior performance of TI-DFS can be attributed to its failure to consider the projected graph shared by paths and its repetitive search for paths within the same bundle. On the other hand, TI*-DFS performs better than TI-DFS by avoiding redundant computations with the assistance of bundles. However, it still struggles to eliminate useless search branches, which is not satisfactory in practical scenarios. Additionally, we observe that for datasets with coarse timestamps, such as Chess, TI*-DFS exhibits similar performance to TI-DFS. This is because the average number of paths in a bundle is approximately 1 when θ is small.

Next, we compare the running speed of TI*-DFS and TDDL-DFS using throughput as the metric of efficiency, measured in terms of results per second. As shown in Figure 4.4b, the proposed TDDL-DFS achieves throughput that is 1-3 orders of magnitude faster than TI*-DFS. Moreover, the average throughput of TDDL-DFS is around 50,000 results per second, making it highly competitive

for real-time applications. These findings demonstrate the outstanding performance and efficiency of TDDL-DFS, making it a promising solution for time interval path enumeration in temporal graphs.



(a) CollegeMsg-k (b) Facebook-k (c) MathOverflow-k

(d) CollegeMsg-t (e) Facebook-t (f) MathOverflow-t

Figure 4.5: Changes of compression ratio with hop-constraint k and interval size t

4.6.4 Effectiveness of Bundles

Table 4.2: Compression ratio of bundles

Dataset	θ	$\lvert\mathcal{P}(B)\rvert$	$\lvert\mathcal{B}\rvert$	CR
CollegeMsg	58,911	1,282,528,263	724,714	1,769.70
Chess	100	15,744	9,332	1.69
Slashdot	90,345	431,173	132,205	3.26
MathOverflow	505,784	35,658,711,811	31,774,449	1,122.24
Epinions	936	1,635,738	1,635,738	1.00
Facebook	867,939	9,188,873,087	155,874	58,950.65
AskUbuntu	960,866	88,739,520	1,895,453	46.82
Prosper	1,259	27,004	16,391	1.65

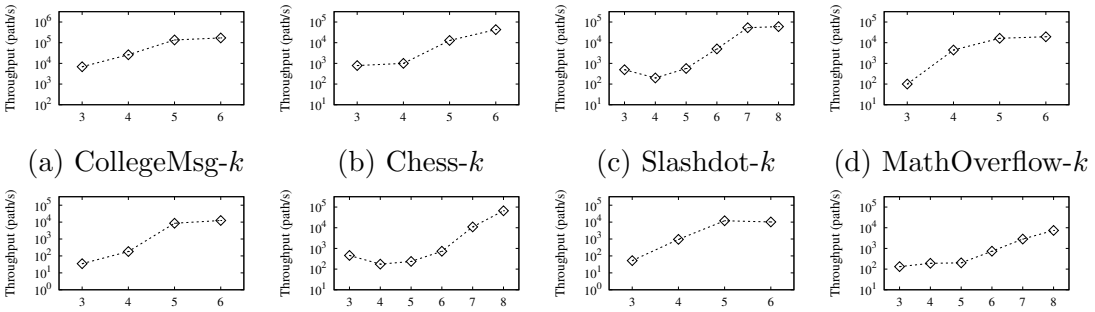
In this section, we demonstrate the effectiveness of TIPST bundles in improving compression ratios CR. The results, as shown in Table 4.2, underscore the significant impact of TIPST bundles on datasets such as CollegeMsg, MathOverflow, Facebook, and AskUbuntu. The findings highlight the substantial reduction in runtime memory costs and search space redundancy achieved through the bundled approach. Notably, certain datasets exhibit CR values ranging between

1 and 10, possibly attributed to coarsely distributed timestamps. This evidence validates the efficiency of TIPST bundles in optimizing storage and enhancing the overall performance of the algorithm for time interval path enumeration in temporal graphs.

To further visualize the relationship between the Compression Ratio CR and different hop-constraints k for queries, we present Figure 4.5a, Figure 4.5c, and Figure 4.5b depicting the changes in CR for three representative datasets: CollegeMsg, MathOverflow, and Facebook. The default interval size is set to 1/3 of θ. The plots clearly demonstrate that the changes in CR increase exponentially with higher hop-constraints k. This observation underscores the necessity of implementing TIP bundles to optimize runtime memory utilization and expedite query response times, making it a crucial strategy for improving the efficiency of our approach.

Additionally, to examine the relationship between CR and different interval sizes, we present Figure 4.5d, Figure 4.5f, and Figure 4.5e. The default hop-constraint remains consistent with Section 4.6.3. The outcomes of these plots reveal that CR remains stable despite changes in the interval size. These findings collectively support the adoption of TIPST bundles as an effective strategy for reducing memory usage and improving query performance, further validating the efficiency and effectiveness of our approach.

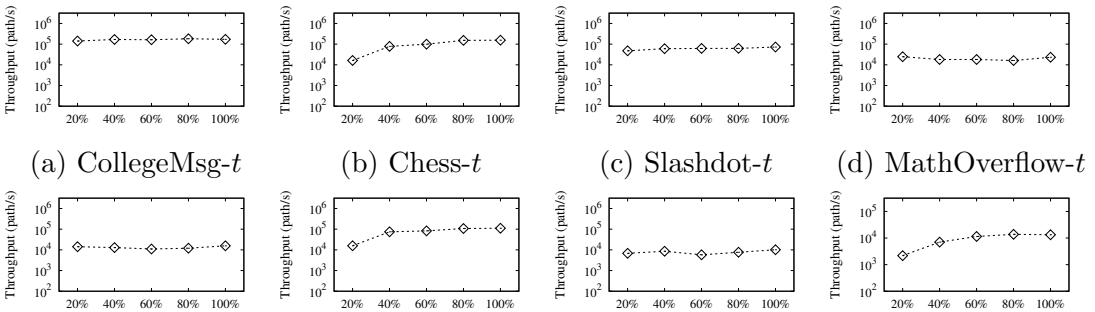
Scalability with hop-constraint k. The assessment of the running speed of TDDL-DFS on 8 graphs, with varying hop-constraint k, is presented in Figure 4.6. The results demonstrate that throughput exhibits an exponential increase as k increases, highlighting the scalability of the algorithm.



(a) CollegeMsg-k (b) Chess-k (c) Slashdot-k (d) MathOverflow-k

(e) Epinions-k (f) Facebook-k (g) AskUbuntu-k (h) Prosper-k

Figure 4.6: Impacts of hop-constraint k on throughput



(a) CollegeMsg-t (b) Chess-t (c) Slashdot-t (d) MathOverflow-t

(e) Epinions-t (f) Facebook-t (g) AskUbuntu-t (h) Prosper-t

Figure 4.7: Impacts of time interval size t on throughput

4.6.5 Scalability

Scalability with interval size. The evaluation of the running speed of TDDL-DFS is depicted in Figure 4.7, where different interval sizes are considered relative to the maximum time interval of the datasets. Our observations indicate that the throughput remains stable or gently increases with the growth of the query interval. This promising trend can be attributed to the augmented potential for leveraging bundles as the interval size grows, ultimately contributing to the overall performance improvement.

4.7 Conclusion

In this paper, we defined a time interval path model to capture entity relationships in a specific period of temporal graphs. Building upon this model, we investigated the problem of hop-constrained time interval s-t path enumeration and proposed a novel and efficient DFS-based method called TDDL-DFS. Leveraging bundled structures and dynamic distances, TDDL-DFS significantly optimizes the search process by avoiding repeated and fruitless searches. Theoretical analysis demonstrated that TDDL-DFS is a polynomial delay algorithm with $O(km_s\theta)$ time per output, and the space complexity for each output is $O(k)$. In our practical evaluation, we conducted extensive experiments on 8 real-world datasets to demonstrate the efficiency and effectiveness of our proposed algorithm.

Chapter 5

Path Compression in Large Graphs

5.1 Overview

In this chapter, we introduce the proposed OFFS's methodological details along with its evaluation results, which is published in ICDE 2023[52]. In Section 5.2, we provide formal definitions of paths and (de)compression. Following this, Section 5.3 introduces the framework of (de)compression of dictionary-based methods for path compression. We then proceed to Section 5.4, which put forwards a baseline for table construction based on gross frequencies. Section 5.5 details the proposed bottom-up framework of table construction with merge and expansion and Section 5.6 provides the complexity analysis and runtime cost. Section 5.7 reports the experimental results, which demonstrate the effectiveness and efficiency of our proposed methods. Finally, Section 5.8 concludes the chapter.

5.2 Preliminary

In this section, we introduce definitions of graphs and simple paths. In addition, we present the definitions of four representative distance measures between trajectories.

5.2.1 Problem Statement

Given a directed graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges, $e(u, v)$ denotes a directed edge from u to v. A path P is a sequence of vertices $\{v_1, ..., v_l\}$ such that $\forall 1 < i \leq l, e(v_{i-1}, v_i) \in E$. A path P is simple if all vertices in P are distinct. When the context is clear in the rest, we use the term "path" to represent "simple path" for short. We use $|P|$ to denote the number of vertices in P. Given two integer x and y with $x < y \leq |P|$, we use $P[x : y]$ to denote the subpath from x-th vertex to $(y - 1)$-th vertex, and x is starting from 0. Likely, we use $P[x]$ to denote x-th vertex in P. For example, given a path $P = \{1, 2, 3, 5, 8, 13\}$, we have $P[1 : 4] = \{2, 3, 5\}$ and $P[4] = \{8\}$.

Lossless compression and decompression. In this topic, we study the problem of lossless compression of a set of simple paths with access to any individual path. Specifically, given a set of paths P, our aim is to find a compression scheme $f : P \Rightarrow (P', R)$, where P' is a set of contracted paths and R is the compression rule. Meanwhile, there exists a decompression scheme $f^T : (P', R) \Rightarrow P$ that restores P from P' with R losslessly. Given any subset $Q' \subseteq P'$, $f^T : (Q', R) \Rightarrow Q$ returns its corresponding uncontracted paths $Q \subseteq P$ without decompressing the other compressed paths in $P' - Q'$.

5.3 The Framekwork

In this section, we give an overview of our proposed framework and then introduce the details of decompression and compression.

5.3.1 Overview

To improve the compression quality for our problem, we adopt the framework of dictionary compression DICT, which contracts each input path into a shorter path with the assistance of a lookup table. The paths in our problem come from the same graph, and common subpaths appear repeatedly in different paths. Our basic idea is to merge those common subpaths into supernodes. We record the corresponding subpath for each supernode in the lookup table.

5.3.2 Decompression and Compression

We first introduce the decompression scheme as shown in Algorithm 9. Given a supernode table $\mathcal{ST}$ and an arbitrary compressed path, we scan the compressed path. When meeting a supernode, we look up the supernode table and replace the supernode with the corresponding subpath. In line 4, $\mathcal{ST}[P_c[pos]]$ derives the subpath for the supernode $P_c[pos]$, and $\oplus$ is a concatenation operation. If $P_c[pos]$ is not a supernode, we keep the original vertex and continue to scan. To deal with an overwhelming number of path decompression queries, we can process paths in parallel to improve efficiency.

Lemma 15. *Given a compressed path P_c, decompressing P_c takes $O(|P|)$ time, where P is the decompressed path.*

Assume that we already have a supernode (lookup) table. The compression scheme is presented in Algorithm 10. We use $\mathcal{ST}^{-1}$ to represent the inverted

lookup table, which returns a supernode given a subpath. In lines 3–9, we adopt a greedy strategy to match subpaths to supernodes. We use a parameter δ to denote the longest length among all compressed subpaths. Starting from the first vertex $pos = 0$, we find the longest subpath in P such that a matching supernode exists. If a matched subpath exists (line 5), we replace it with the corresponding supernode in line 6. Otherwise, we skip the vertex at the current position (line 8) and start matching from the next vertex in the next iteration.

Lemma 16. *Given a path P, compressing P takes $O(|P| \cdot \delta^2)$ time, where δ is the longest subpath length in the lookup table.*

Note that the square of δ comes from δ times possible hashes of up to δ elements, but never mind since δ is small in practice. In our evaluation and real deployment, δ is set as 8.

Based on that, we will give straightforward ideas on how to build $\mathcal{ST}$ and $\mathcal{ST}^{-1}$ in the following subsections.

Algorithm 9: Decompress($P_c, \mathcal{ST}$)

Input: A compressed path P_c and a supernode table $\mathcal{ST}$
Output: An orginal path P

```
1 P ← ∅;
2 pos ← 0;
3 for 0 ≤ pos < |P_c| do
4     if P_c[pos] ∈ ST then P ← P ⊕ ST[P_c[pos]] ;
5     else P ← P ⊕ P_c[pos] ;
6 return P;
```

5.4 Frequent Subpaths

One naive solution for table construction comes from brute-force enumeration. The cost of testing the compression performance of each table is $O(|\mathbb{P}| \cdot \delta^2)$ as

Algorithm 10: Compress($P, \mathcal{ST}^{-1}$)

Input: A path P and an inverted supernode table $\mathcal{ST}^{-1}$
Output: A compressed path P_c

```
1  P_c ← ∅;
2  pos ← 0;
3  while pos < |P| do
4      for min(δ, |P| − pos) ≥ l > 1 do
5          if P[pos : pos + l] ∈ ST^{-1} then
6              P_c ← P_c ⊕ ST^{-1}[P[pos : pos + l]];
7              break;
8      if l = 1 then P_c ← P_c ⊕ P[pos : pos + 1] ;
9      pos ← pos + l;
10 return P_c;
```

Lemma 16 shows. More specifically, given the maximum supernode size δ, the number of candidates is bounded by $\delta \cdot |\mathbb{P}|$. The possible number of tables composed of c supernodes is $\binom{\delta \cdot |\mathbb{P}|}{c}$. It is bounded by $O(min((\delta \cdot |\mathbb{P}|)^c, (\delta \cdot |\mathbb{P}|)^{\delta \cdot |\mathbb{P}| - c}))$ and is beyond polynomial time. For instance, if we set δ as 8, and construct a table of the top 1,000 out from 12,500 paths, a toy case compared to real-life datasets, the estimated cost 100000^{1000} has too many digits to count and becomes unacceptable for applications.

Instead, we are exploring in another way via finding frequent patterns, which is one of the classic topics in data mining. There are several methods on that, such as Apriori[7] and FP-Tree[47]. As far as we know, the state-of-the-art solution for computing frequent subpaths is Apriori for Frequent Subpaths (AFS)[45]. AFS adopts a bottom-up framework as in Algorithm 11 to get the result sets L_i of length i through iterations. It starts from short frequent subpaths of length one in line 1. Next, it joins current frequent subpaths with outgoing edges of the last element in the subpath (suppose there is a graph as ground truth), and checks whether the extended subpath excluding the first element is in the last set

L_{i-1} in line 4. After that, it counts their gains (i.e., the product of frequency and length) in data paths and keeps candidates with gains larger than a threshold k in line 5. Then it increases the count of iterations i and checks whether i has reached the target l, if so returns results and ends the process, otherwise goes to the next iteration.

Algorithm 11: AFS$(\mathbb{P}, l, k)$

```
   Input: A set of paths ℙ, the maximum length l, and a threshold k
   Output: A set of frequent subpaths L
   // initialize the set with nodes
 1 L_1 ← {v|v ∈ ℙ};
 2 i ← 2;
 3 while i ≤ l do
 4 |   C_i ← JoinWithCheck(L_{i−1}, i − 1);
 5 |   L_i ← CountGain(C_i, ℙ, k, i);
 6 |   i ← i + 1;
 7 return L ← L_1, ..., L_l;

 1 Procedure JoinWithCheck(L, i)
 2 |   C ← ∅;
 3 |   foreach P = {v_0, v_1, ..., v_{i−1}} ∈ L do
 4 |   |   foreach u as neighbor of v_{i−1} do
 5 |   |   |   P' ← {v_0, v_1, ..., v_{i−1}, u};
 6 |   |   |   if P'[1 : i + 1] ∈ L then
 7 |   |   |   |   C ← C ∪ {P'};
 8 return C;
```

However, (1) the time complexity of this method is unacceptable. Suppose that we gain $|L|$ subpaths of length i in i-th iteration and there are n nodes in the graph, then the cost to extend in each iteration is $O(i \cdot n \cdot |L|)$ due to checks in L_{i-1}. Assume the number of subpaths returned in line 12 is λ, it will cost $O(l^2 \cdot n \cdot \lambda)$ in total. Even worse, (2) unlike Apriori for sets, the generated longer subpaths in C_i are not guaranteed valid on data paths, as the constraint

Algorithm 11: AFS$(\mathbb{P}, l, k)$ (continued)

```
9  Procedure CountGain(C, ℙ, k, i)
10     L ← ∅;
11     foreach P ∈ ℙ do
12         j ← 0;
13         while j + i ≤ |P| do
14             if P[j : j + i] ∈ C then
15                 increase P[j : j + i].count;
16             j ← j + 1;
17     foreach subpath ∈ C do
18         if subpath.count ≥ k then
19             L ← L ∪ {subpath};
20 return L;
```

of order and adjacency of paths is stricter than that of sets. Therefore, it is essential to prune the useless subpaths by identifying them on datasets once more. It requires another $O(l^2 \cdot |\mathbb{P}|)$ where $|\mathbb{P}|$ is the node number in $\mathbb{P}$. (3) Another serious issue is that the generated subpaths tend to be overlapped with each other and be covered by the longer subpaths. Namely, they cannot be matched effectively due to match collisions. According to our observation, (1) and (2) make it take too much time to generate a tiny table, whose compression quality is far from satisfactory. Instead, we come up with two more practical one-pass baselines named RSS and GFS. RSS stands for randomly sampling all candidates, which does not consider any measure. GFS stands for gross frequent subpaths, which picks top candidates in the order of a gross measure.

The process of one-pass baselines carries out the following steps as Algorithm 12 shows, where $\mathcal{H}$ is a hash table to maintain all vertex sequences and corresponding weights. The weight of a vertex sequence *seq* in $\mathcal{H}$ is initialized by 1 once found. We increase the weight of *seq* in $\mathcal{H}$ every time it is matched elsewhere. At the start, it traverses paths and counts frequencies of all subpaths

Algorithm 12: TConstruct($\mathbb{P}$)

Input: A set of paths $\mathbb{P}$
Output: A lookup table $\mathcal{ST}$, and its inverted table $\mathcal{ST}^{-1}$
// collect and count $O(|P|^2)$ subpaths
1 **foreach** $P \in \mathbb{P}$ **do**
2 **foreach** $0 \leq i < |P|$ **do**
3 **foreach** $i < j \leq |P|$ **do** $\mathcal{H}$.add($P[i:j]$);
4 **if** $|\mathcal{H}| > \lambda$ **then** pick λ items in $\mathcal{H}$ via its rule;
5 create a table $\mathcal{ST}$ for decompression and an inverted table $\mathcal{ST}^{-1}$ for compression from $\mathcal{H}$;

1 **Procedure** $\mathcal{H}.add(seq)$
2 **if** $seq \in \mathcal{H}$ **then** $\mathcal{H}[seq] \leftarrow \mathcal{H}[seq] + 1$;
3 **else** $\mathcal{H}[seq] \leftarrow 1$;

(lines 1–2). Then it picks the top λ with its rule if the number of candidates in $\mathcal{H}$ is more than the given threshold λ (line 4). Consequently, it builds a lookup table based on the remaining candidates(line 5). It is easy to realize that AFS serves as lines 1—4 in picking top candidates, however, the cost makes it impossible to build a large enough table in practice. Instead, we exploit RSS and GFS. RSS is a naive solution that randomly samples c out of candidates without considering any measure. GFS is named after the measure *gross weighted frequency* (for subpaths), the product of frequency and size. It works as the equivalent of AFS if we set the maximum size of candidates as l in lines 2—3. Better still, GFS solves both problems (1)(2) of AFS via collecting subpaths and counting their frequencies directly in data paths. Specifically, it takes $O(l^2 \cdot |\mathbb{P}|)$ to collect and count all subpaths and $O(l \cdot |\mathbb{P}| \cdot log(\lambda))$ to pick the top λ. Although GFS is suitable for pattern mining like AFS, we will show in the following that it is not an ideal measure for compression. Sometimes it is even worse than the most naive solution RSS, due to the problem (3). We will introduce a more efficient strategy focusing on the match collision issue in Section 5.5.

Table 5.1: Supernode tables with different measures

$\mathcal{ST}$		$\mathcal{ST}^*$	
subpaths	supernodes	subpaths	supernodes
$v_2, v_3, v_5, v_8, v_{12}$	u_0	$v_2, v_3, v_5, v_8, v_{12}$	u_0^*
v_2, v_3, v_5, v_8	u_1	v_{13}, v_{21}	u_1^*
v_3, v_5, v_8, v_{12}	u_2	v_{17}, v_9	u_2^*
v_2, v_3, v_5	u_3	v_2, v_2	u_3^*
v_3, v_5, v_8	u_4		

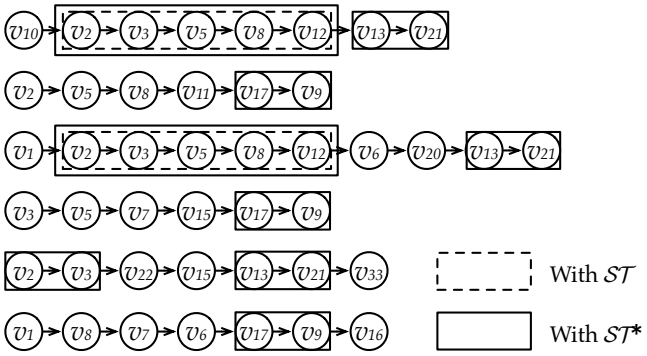


Figure 5.1: Compression with tables of frequent subpaths

5.5 Supernode Table Compression

5.5.1 Match Collision Issue

We clarify why the gross weighted frequency is far from a good choice for compression. As aforementioned, a straightforward and naive solution is to consider both frequency and size, as GFS does, to pick the most common subpaths. It is reasonable for frequent pattern mining, however, it is not an ideal measure for compression. Here is an instance to show its drawback due to match collision issues.

Example 13. *Assume there is a path set as in Figure 5.1, the maximum size of supernodes is 5, and the capacity of the lookup table is 5. Note that if two*

candidates have the same weighted frequency, we prefer the longer one unless it has a frequency of 1. Following the scheme of collecting and selecting in Algorithm 12, the final supernode table $\mathcal{ST}$ *is composed of* $u_0 = \{v_2, v_3, v_5, v_8, v_{12}\}$, $u_1 = \{v_2, v_3, v_5, v_8\}$, $u_2 = \{v_3, v_5, v_8, v_{12}\}$, $u_3 = \{v_2, v_3, v_5\}$, $u_4 = \{v_3, v_5, v_8\}$. *And the result of performing Algorithm 10 with that is as dotted boxes in Figure 5.1 shows, where only two paths get compressed. Suppose the issue is considered, and somehow the table* $\mathcal{ST}^*$ *is composed of* $u_0^* = \{v_2, v_3, v_5, v_8, v_{12}\}$, $u_1^* = \{v_{13}, v_{21}\}$, $u_2^* = \{v_{17}, v_9\}$, $u_3^* = \{v_2, v_3\}$ *under the same given constraints. Then the result after compression will be like solid boxes in Figure 5.1 with each path contracted, which has a better compression quality as well as a smaller table size.*

The problem with GFS is that only one-fifth of the table is helpful for compression performance, while the rest contribute nothing since they have been covered as shown in Table 5.1. In other words, the gross weighted frequency is inappropriate given a limited capacity λ. The result includes too many overlapped subpaths that it fails to give good compression performance. In contrast, candidates in a good lookup table are supposed to be complementary, like u_0^*, u_1^*, and u_2^* in $\mathcal{ST}^*$.

That suggests it unreasonable to generate and add every possible frequent subpath greedily in Algorithm 12. The naive solution has serious computational redundancy and misses potential complementary candidates due to the match collision issue. More specifically, the overlapping problem lies in the process of extending and counting supernodes. For instance, suppose $\{v_2, v_3, v_5, v_8, v_{12}\}$ is matched in Figure 5.1, the frequencies of others appearing within that matched interval will never count under the compression framework in Algorithm 10. The practical frequencies of them during matching are definitely zero in this case. Following this idea, we put forward our measure to solve matched collision

issues, *practical weighted frequency*, the product of practical frequency and size. The difference between gross and practical frequencies is that only the valid ones following the compression scheme count instead of counting occurrences at any position. Based on that, we will give a detailed description of how to construct a representative lookup table.

5.5.2 Identifying Supernodes by Merge and Expansion

We introduce our method to compute the supernode table. The key is selecting a set of vertex sequences that frequently appear in many paths.

Our idea is to start by considering some frequent pairs of vertices (i.e., edges) as supernodes. Then in each iteration, we extend the current candidate vertex sequences to a set of new longer sequences. The number of iterations is controlled by a parameter τ.

The formal pseudocode for table construction is described in Algorithm 13. Lines 1–2 add all edges in the path set to the hash table $\mathcal{H}$ as the initial candidate patterns (supernodes). Next, for each path P in each iteration, we start from the first node and identify the target sequence with the assistance of LongestPrefix, that matches a corresponding supernode in the candidate set $\mathcal{H}$ (line 5). Every time we identify a matched sequence (line 9), we increase the weight of the supernode in $\mathcal{H}$ (line 4 of Algorithm 14). At the end of each iteration, we generate new supernodes based on the current supernode (lines 10–15) and skip the sequence (line 16) to identify new patterns.

Merge and Expansion. We generate new candidate supernodes in each iteration using two strategies, merge and expansion, to leverage the latest matches as much as possible. Specifically, the merge strategy concatenates a pair of continuous supernodes found in a path (lines 10–13). Note that the length of a compressed sequence is bounded by δ. We cut the later part if the size of the

Algorithm 13: TConstruct*($\mathbb{P}$)

Input: A set of paths $\mathbb{P}$
Output: A supernode table $\mathcal{ST}$ and an inverted supernode table $\mathcal{ST}^{-1}$

```
   // initialize H
1  foreach P ∈ ℙ do
2      foreach e ∈ P do H.add(e);
3  for 1 ≤ i ≤ τ do
4      foreach P ∈ ℙ do
5          match ← LongestPrefix(0, P, i, H);
6          pos ← |match|;
7          while pos < |P| do
8              pre_match ← match;
9              match ← LongestPrefix(pos, P, H);
               // Merge
10             if |pre_match| + |match| ≤ δ then
11                 H.add(pre_match ⊕ match);
12             else
13                 H.add(pre_match ⊕ match[0 : δ − |pre_match|]);
               // Expansion
14             if |match| > 1 ∧ |pre_match| < δ then
15                 H.add(pre_match ⊕ P[pos]);
16             pos ← pos + |match|;
17     if |H| > λ then keep top-λ items in H;
18 create a supernode table ST for decompression and an inverted table
       ST^{-1} for compression from H;
```

Algorithm 14: LongestPrefix($pos, P, i, \mathcal{H}$)

Input: A position indicator pos, a path P, an iteration indicator i, the supernode table $\mathcal{H}$

Output: A prefix of P

```
1 l ← min(2^{i+1}, δ, |P| − pos);
2 while l > 1 and P[pos : pos + l] ∉ H do
3     l ← l − 1;
4 if l > 1 then H.add(P[pos : pos + l]);
5 return P[pos : pos + l];
```

Table 5.2: Candidates and weights during table updates

Stage	Initialization		1st iteration		2nd iteration		Finalization	
	candidates	weights	candidates	weights	candidates	weights	subpaths	supernode ids
Top-5	v_2, v_{10}	1	v_3, v_5, v_8, v_{12}	2	$v_2, v_3, v_5, v_8, v_{12}$	2	$v_2, v_3, v_5, v_8, v_{12}$	u_0^*
	v_2, v_3	1	v_3, v_5, v_8	2	v_{13}, v_{21}	3	v_{13}, v_{21}	u_1^*
	...	...	v_3, v_5	3	v_{17}, v_9	3	v_{17}, v_9	u_2^*
	v_{17}, v_9	1	v_{13}, v_{21}	3	v_2, v_3	2	v_2, v_3	u_3^*
	v_9, v_{16}	1	v_{17}, v_9	3	v_3, v_5, v_8, v_{12}	1		

merged sequence exceeds δ (lines 12–13). As for the expansion strategy, a complement to merging, it adds the following vertex to the sequence (lines 14–15). We only keep at most λ items in $\mathcal{H}$ in each iteration. Here we use the same input and constraints in Example 13 to show how to construct a supernode table from scratch.

Example 14. *Suppose the input is as Figure 5.1, and we update the lookup table Table 5.2 in two iterations, along with initialization and finalization. Primarily, we initialize the table with all 26 edges with frequency one, where the weight suggests existence. In the first iteration, the maximum size of matched supernodes is two whenever calling* LongestPrefix. *If there are two adjacent successful matches, take* $\{v_3, v_5\}$ *and* $\{v_8, v_{12}\}$ *for instance, we not only increase their weights by one, but also count for the results of merge and expansion,* $\{v_3, v_5, v_8, v_{12}\}$ *and* $\{v_3, v_5, v_8\}$. *The top 5 candidates after the first iteration are shown in Table 5.2. Then, the maximum size of supernodes increases to four, which is also the value*

of l in Algorithm 14 for the next iteration. Likewise, we keep matching, counting and picking the top 5 at the second iteration. After that, the candidate table is updated to include more representative sequences like $\{v_2, v_3, v_5, v_8, v_{12}\}$ and exclude overlapped ones like $\{v_3, v_5, v_8\}$. During finalization, we drop the useless ones with weight one, like $\{v_3, v_5, v_8, v_{12}\}$, and generate the same $\mathcal{ST}^*$ as in Table 5.1.

5.5.3 Practical Implementation

We give a detailed description of how to identify the longest matched supernodes in Algorithm 15.

As aforementioned, the upper bound for identifying the longest matched supernode at any position is $O(\delta^2)$, because we only store the exact key in the hash table and the cost of each hash is linear to the size of sequences. The drawback is that there will be much redundancy in hashing common prefixes when we start identifying the prefix from the longest length.

Example 15. *Assume the inputs i and pos of Algorithm 14 make $l \geq 8$ in line 1, P is $\{v_8, v_5, v_0, v_9, v_1, v_3, v_4, v_2\}$, and the return value is $\{v_8\}$. That is, we meet the worst case with no valid matched supernode. During identification, we compute and combine the hashes from v_8 to v_2 at the beginning. However, there is no matched key in $\mathcal{H}$, so we pop the back v_2. We repeat hashing and popping back until only v_8 remains and return it. The total cost for that is $(8+2)(8-2+1)/2 = 35$, where the common prefix are involved repeatedly.*

It is not difficult to realize the problem lies in the storing data structure, namely, the hash table does not support prefix key matching. However, we cannot afford to store all possible prefixes either, as it requires $O(\delta^2)$ space for each supernode. It is a reasonable tradeoff between space and time costs to

leverage a multi-level hash scheme. Specifically, all supernodes with sizes no larger than α are kept in a one-level hash map $\mathcal{H}_1$. The others are maintained in a two-level hash map $\mathcal{H}_2$ with the primary key $P[pos : pos+\alpha]$ and the secondary key $P[pos + \alpha : pos + l]$.

Algorithm 15: LongestPrefix*$(pos, P, i, \mathcal{H})$

Input: A position indicator pos, a path P, an iteration indicator i, the supernode table $\mathcal{H}$
Output: A prefix of P

```
1  l ← min(2^{i+1}, δ, |P| − pos);
2  if l ≤ α then
3      while l > 1 and P[pos : pos + l] ∉ H_1 do
4          l ← l − 1;
5      if l > 1 then
6          H_1.add(P[pos : pos + l]);
7  else if P[pos : pos + α] ∈ H_2 then
8      while l > α and P[pos + α : pos + l] ∉ H_2[P[pos : pos + α]] do
9          l ← l − 1;
10     if l > δ then
11         H_2[P[pos : pos + α]].add(P[pos + α : pos + l]);
12     else
13         H_1.add(P[pos : pos + α]);
14 else
15     return LongestPrefix*(pos, P[0 : pos + α], i, H);
16 return P[pos : pos + l];
```

Based on that, we implement our scheme for identifying the longest matched supernodes in Algorithm 15. Line 1 starts with initializing the possible maximum size l for matching. If it is too small to have a secondary key for matching (line 2), the process is reduced to what Algorithm 14 does in lines 3—6, otherwise, we look it up in $\mathcal{H}_2$. If there is a match of the primary key $P[pos : pos + \alpha]$ in $\mathcal{H}_2$, we just move on matching the remaining suffix as the secondary key $P[pos + \alpha : pos + l]$ (lines 8—9) and increase the weight for successful matches

of suffixes (lines 10—11) or prefixes (lines 12—13), otherwise, it falls back to $\mathcal{H}_1$ with its prefix $P[pos : pos + \alpha]$ (line 15). Besides, a small trick to optimize the matching in lines 7—13 is that there will always be a valid key of $P[pos : pos+\alpha]$ in $\mathcal{H}_1$ if it exists in $\mathcal{H}_2$. It guarantees that we never fall back to lines 3—6 whenever line 7 is satisfied. Line 16 returns the longest matched subpath.

Lemma 17. *Given a path P, compressing or identifying longest matched supernodes in P takes $O(max(|P| \cdot \alpha^2, |P| \cdot (\delta - \alpha)^2))$ time, where α is the maximum length of the primary hash key and $\delta - \alpha$ is the maximum length of the secondary hash key.*

As we can get in Lemma 17, the refined upper bound is less than $O(|P| \cdot \delta^2)$ and the optimal value of α is supposed to be approximately $\delta/2$.

Example 16. *Assume the input is the same as in Example 15, and the return value is still the worst case $\{v_8\}$, and α is 5. Since $8 > 5$, there remain two possible cases, (1) the prefix $\{v_8, v_5, v_0, v_9, v_1\}$ is not matched in two-level hash table $\mathcal{H}_2$, the total cost in this case is $(5 + 2)(5 - 2 + 1)/2 = 14$ in line 3—6, otherwise, (2) it is matched as primary key in $\mathcal{H}_2$, then the upper bound for hashing is $5 + (3 + 1) \cdot 3/2 = 11$. And the maximum value 14 is less than 35 in Example 15.*

In practice, it works pretty fine for simple paths. And we will cover other path types in the following.

Other Path Types. Our proposed algorithms can also process other path types. We provide two instances as follows, including undirected paths and weighed paths.

For paths in undirected graphs, it is straightforward to apply our method for directed graphs by selecting an arbitrary terminal as the start. If both undirected

and directed paths are included, we additionally use a boolean value for each path to indicate whether it is directed.

For weighted paths, the challenge comes with weights of edges. Suppose the paths are represented by $u_0 \xrightarrow{w_0} u_1 ... u_{l-1} \xrightarrow{w_{l-1}} u_l$, where each $u_{i-1} \xrightarrow{w_{i-1}} u_i, 1 \leq i \leq l$ represents an edge weighted w_{i-1} from u_{i-1} to u_i. We represent each weighted edge $u_{i-1} \xrightarrow{w_{i-1}} u_i$ by a new vertex ID if the edge with the same terminals and the same weight has never been visited before. Otherwise, we use the existing ID. In this way, we represent a weighed path as an unweighed path with new IDs, and we can compress them using our algorithm. To decompress a path, we additionally replace each vertex in the path with the corresponding weighed edge.

Dynamic Paths. We discuss the case that paths dynamically update. To insert a new path P, we could compress P using the existing supernode table. To remove an existing path P, we do nothing but only removing the compressed path of P. Changing a path can be viewed as a path removal operation followed by a path insertion operation. When a large number of paths update, the compression quality may be affected since the supernode table is constructed by frequent subpaths in old paths. In this case, we reconstruct the supernode table based on the updated paths.

For paths coming in a stream, we follow the approach of building tables and then compressing paths. Once the volume of accumulated paths reaches a certain threshold, we build the supernode table using paths randomly sampled from them and compress all paths from then on with the constructed table. As more paths come in and the supernode table becomes outdated, we periodically reconstruct the supernode table based on the updated paths.

Circles. Note that circles may exist in paths in certain scenarios. Our proposed algorithms process each path as an integer array and scan elements in the array

sequentially. We just aim to identify frequent subpaths where repeating vertices is possible. Therefore, our algorithms still work even if there are circles in the paths.

5.5.4 Possible Optimizations

As a component of the real-life industrial system of Alibaba Cloud, it is essential to consider all possible edge cases for comprehensiveness and robustness sake. The cost of the current version might grow dramatically when the average length of paths extends to hundreds or even thousands. There are two possible optimizations focusing on that. They have not been tested yet due to the absence of datasets with longer average lengths at the moment. (1) We can implement a hybrid framework combining top-down with bottom-up to identify shorter supernodes and longer ones parallelly. (2) We can adopt a new data structure to serve as Prefix-Tree or Trie[38] for vertex sequences. This structure tailored for prefixes will identify matches faster with less run-time memory cost. It will improve the upper bound for each prefix match $O(\delta^2)$ to $O(\delta)$.

5.6 Complexity Analysis

5.6.1 Time Complexity

The time cost for prefix matching during table construction (Algorithm 13) could be divided into initialization (lines 1—2), iteration (lines 3—17), and finalization (line 18). As for the main part during iteration, the cost to identify matched supernodes (line 9) for a path set of total $|\mathbb{P}|$ nodes is $O(|\mathbb{P}| \cdot \delta^2)$ bounded by constant times of hashes. Given the bound of the capacity of the candidate set λ, the cost to update the table (line 17) is $O(|\mathbb{P}| \cdot \ log(\lambda))$. It is derived

from keeping top-λ elements in a min-heap of size λ, with at most $|\mathbb{P}|$ updates. Likewise, the overhead for the initialization with edges (lines 1—2) and the finalization to return the lookup table (line 18) is $O(|\mathbb{P}| + \lambda \cdot \delta)$. Better still, we could make it more efficient by sampling. In our observation, it is still effective to sample less than one percent of all paths during table construction. In sum, the total cost of i iterations with sample rate s, i.e., one in every s paths, is $O((i \cdot (|\mathbb{P}| \cdot \delta^2 + |\mathbb{P}| \cdot log(\lambda)) + |\mathbb{P}| + \lambda \cdot \delta)/s)$. Note that we set λ linear to $|\mathbb{P}|$ with a fixed factor β in practice. Therefore, that could be further simplified as $O(\gamma \cdot |\mathbb{P}|)$, where the factor $\gamma = (i \cdot (\delta^2 + log(|\mathbb{P}| \cdot \beta)) + \beta \cdot \delta)/s$. As we can see, the total time cost heavily relies on i, δ, β and s. In practice, δ is set as a constant, and β is large enough to trigger filters only in the last few iterations. There remain two significant parameters i and s, to which we will pay close attention in the following experiments.

As for the compression phase in Algorithm 10, what runs likewise as table construction. The difference is that the table becomes static, where matching does not lead to merging or expanding. Since it is one-pass travel in the whole dataset, the upper bound is $O(|\mathbb{P}| \cdot \delta^2)$ with no doubt. Better still, it is safe for us to run compression with the static table in parallel. More specifically, we are able to implement pleasing parallelism on a finer granularity as small as a path in $O(|\mathbb{P}| \cdot \delta^2/p)$ on a p-core machine. Likely, the decompression phase in Algorithm 9 is also easy to parallelize with the bound $O(|\mathbb{P}|/p)$, which makes it a lightweight and fast method.

5.6.2 Space Complexity

The overhead during compression is under strict control, as stated in the section before. The runtime memory during table construction in Algorithm 13 only requires a tiny sample $O(|\mathbb{P}|/s)$. The space for the heap of candidates is bounded

by $O(|\mathbb{P}| \cdot \beta \cdot \delta)$ in each iteration. Therefore, the total space cost is $O(|\mathbb{P}| \cdot \nu)$, where ν is bounded by $O(1/s + \beta \cdot \delta)$. In our observation, the supernode table is still effective when the overhead factor ν is less than 0.03.

As for phases of compression and decompression of Algorithm 10 and Algorithm 9, the required space can be as small as several I/O blocks plus dictionary size $O(|\mathbb{P}| \cdot \beta \cdot \delta)$ thanks to our finer granularity. The upper bound of output size is linear to $|\mathbb{P}|$ with a small overhead factor $\beta \cdot \delta$. In other words, the worst ratio of input size to output size is $(1/(1 + \beta \cdot \delta))$. Assume all supernodes are of length δ, and each subpath is exactly replaced by a supernode, then the ideal ratio for the best case is δ.

Table 5.3: Real-world paths

Dataset	path number	node number	id number	maximum length	average length
Alibaba Cloud	171,024,135	2,941,010,457	426,248	30	17.20
Rome	3,426,475	229,972,163	39,078	503	67.12
Porto	36,898,213	1,207,828,790	137,288	1,355	32.73
San Francisco	5,857,208	102,038,897	6,026	103	17.42

5.7 Experimental Study

Using real-life data, we conducted three sets of experiments to evaluate the (1) impacts of parameters for compression, (2) comparison between OFFS and baselines in compression ratio and compression speed, and (3) retrieval and scalability.

5.7.1 Experimental Setting

Datasets. We use four real-life datasets to evaluate our algorithm. The first one is a private dataset from Alibaba Cloud that monitors IP hops from servers to clients in its network on June 18th 2021. We also use three additional public

datasets of taxi trajectories[19, 76, 80] which cover trajectories in Rome, Porto and San Francisco respectively. The details about their path numbers, node numbers, id numbers, maximum lengths and average lengths are shown in Table 5.3. It can be observed that the average length of paths is less than 100 in all datasets, while the maximum length can be several hundred times larger.

Please note that we make use of the following preprocessing to get datasets ready for further experiments.

New id. It is easy to assign new ids to IPs in the Alibaba Cloud network, i.e., integers from zero to indicate different vertices. Trajectories are recorded as sequences of pairs of $\{longitude, latitude\}$ in time order. Due to vehicle movements and GPS errors, it does not make sense to denote distinct pairs by new ids. Namely, it is abnormal for taxi drivers in the same city to never drive on the same road. Therefore, we need to increase spatial granularity by dividing the space into grids based on various time intervals. In this way, we merge nodes in the same grid into one.

Simple path. There can be duplicated nodes in the same path as a consequence of several reasons. (a) Noise. Whenever we encounter a sequence of adjacent duplicate vertices, we keep only the first one and drop the rest. (b) Cycle. We solve the loop issue by cutting before the first recurring node and generating two shorter paths. In addition, we prune the trivial data by discarding paths of size no more than 2. Having solved (a)(b), we can rest assured that the output paths always stay simple.

Group set. The generated path sets are grouped according to given rules for future mining in the dataset of Alibaba Cloud. Likewise, we organize paths of taxi drivers into sets. For instance, we classify them according to their starting and ending vertices, or passing vertices of interest. Note that although paths are distinct in a given set, they can recur among sets.

Implementation details. We then provide the necessary details about the baseline implementations.

The algorithms were implemented in C++ and compiled by clang++ 14.0.6 with -O3. We conducted all experiments on a Unix machine with eight processors of 3.20 GHz and 16 GB memory. It is essential to note that, in compliance with the data confidentiality principles of Alibaba Group, we were obliged to conduct experiments on our work computer, thus necessitating a distinct experimental environment from the prior two scenarios.

All competitors involved work under the framework of building lookup dictionaries and compressing data path by path. The sample rate for table construction is set to 128 during comparison. The dictionary of Dlz4 is constructed using zdict with enough samples, as suggested in the official documentation. Specifically, we pick one in every 128 as sample, and divide them into blocks of 1 KB for training a dictionary. We attempted to allocate blocks path by path or provide more samples but it did not make a difference in our observation. As for RSS and GFS, the table capacity c is the same as for OFFS. In our observation, they still take too much time for the candidate collection given a limited supernode size. Therefore, we set the threshold $5 \cdot c$ to speed them up during table construction. For OFFS, we set the maximum length of primary keys α to 5, the maximum length of subpaths in the table δ to 8, and the factor between the table capacity and the input size β to 500. Last but not least, all of them are implemented with OpenMP default parallelism during compression and decompression for fairness.

5.7.2 Metrics

We next describe our metrics of compression performance in experiments.

Compression ratio. Suppose the cost to store contracted paths and rule is $|P'| + |R|$ with the raw size of paths $|P|$, the compression ratio, CR of the

corresponding compression scheme $f : P \Rightarrow (P', R)$ is $CR = \frac{|P|}{|P'| + |R|}$.

Compression speed. Given the time of compression and decompression T_c, T_d, the compression speed is defined as $CS = \frac{|P|}{T_c}$, likewise, the decompression speed is defined as $DS = \frac{|P|}{T_d}$. For any partial decompression $f^T : (Q', R) \Rightarrow Q, Q \subseteq P$ with decompression time T_{pd}, the partial decompression speed is defined as $PDS = \frac{|Q|}{T_{pd}}$.

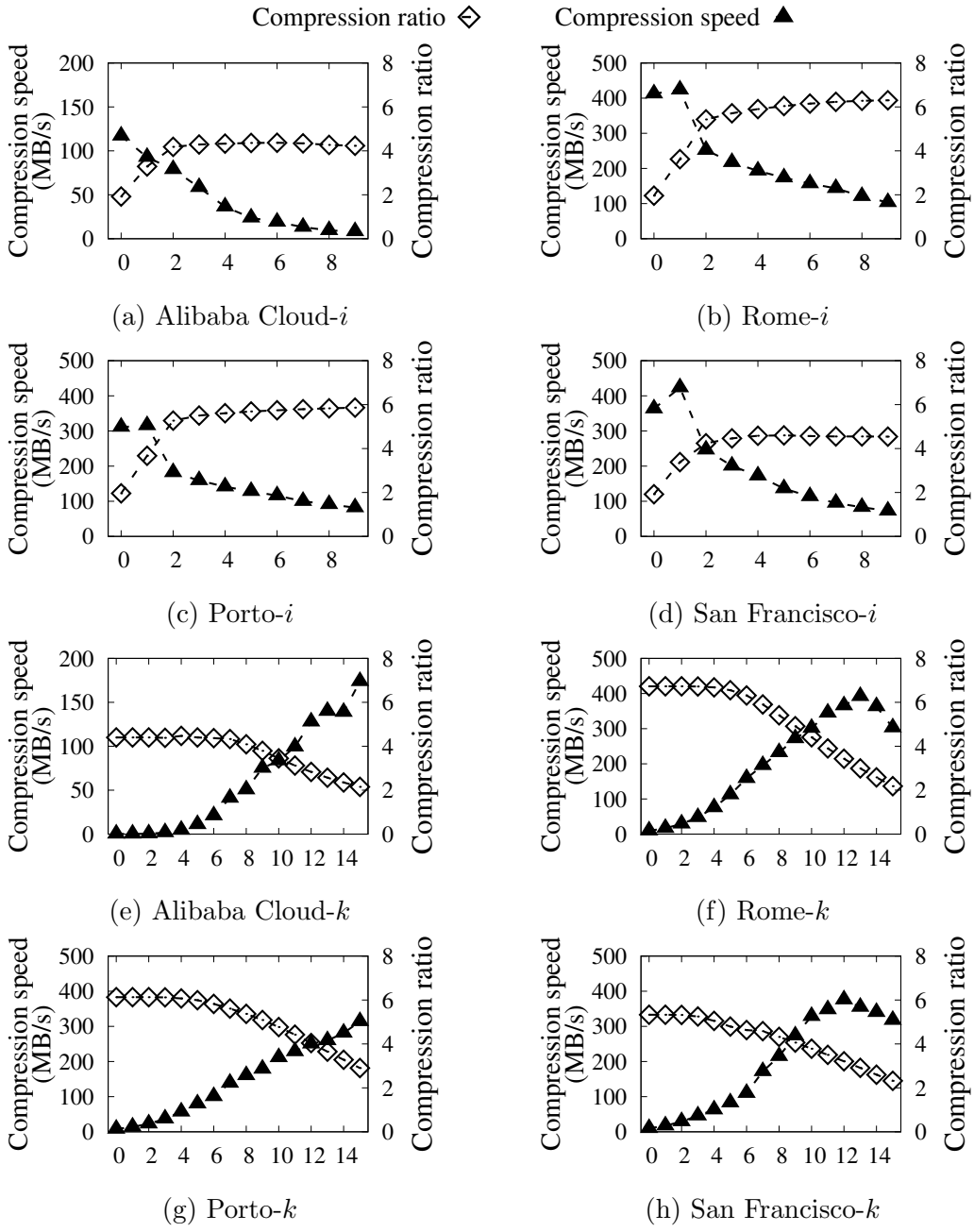
5.7.3 Impacts of Parameters

We first study the impacts of two significant parameters during table construction, the number of iterations i and the exponent k of the base 2 as the sample rate. The results are shown in Figure 5.2. We provide a detailed analysis of their impacts in terms of CR and CS in the following.

(1) As i changes from 0 to 3, CR increases rapidly. Note that the maximum size of current candidates reaches δ at the third iteration. Thereafter, it grows gently as i changes from 3 to 9. On average, it increases by 3 as the maximum size of supernodes in the table grows, and by 0.6 as more iterations are involved to refine the candidates.

(2) As k increases from 0 to 7, CR slowly decreases. Meanwhile, the sample size goes from the full set to less than one percent. It then drops sharply as k changes from 7 to 15. On average, the drop from 0 to 7 is 0.7, while it becomes almost half from 7 to 15.

(3) CS becomes half as i changes from 0 to 4. Then it drops again by half as i increases from 4 to 9. Not surprisingly, the differences are significant when the main update in the table is to extend supernodes in the first three iterations. Thereafter, as the candidates in the table change less, the differences in CS become smaller.



(a) Alibaba Cloud-i (b) Rome-i

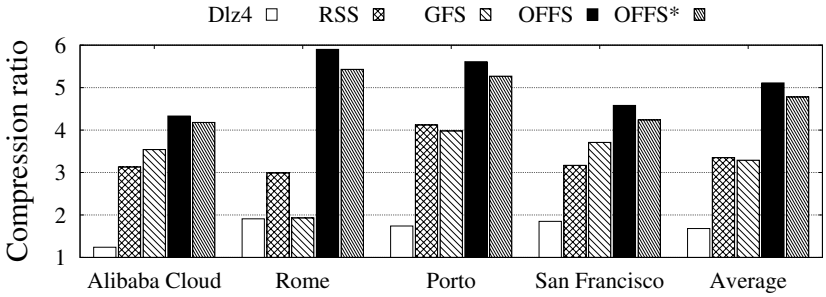
(c) Porto-i (d) San Francisco-i

(e) Alibaba Cloud-k (f) Rome-k

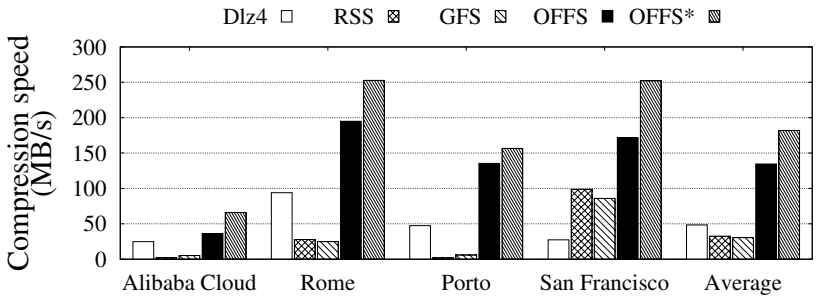
(g) Porto-k (h) San Francisco-k

Figure 5.2: Impacts of parameters i (a–d) and k (e–h) on compression speed and compression ratio

(4) CS increases sharply by 20 times as k changes from 0 to 7. It then slowly doubles with k increasing from 7 to 15. When k is small, the table construction



(a) Compression ratio



(b) Compression speed

Figure 5.3: Comparison with baselines on compression ratio and compression speed

phase accounts for most of the time cost, so CS changes considerably. While k becomes larger, the compression phase accounts for most of the time cost. Meanwhile, it might suffer from more useless matches during compression, which affects CS.

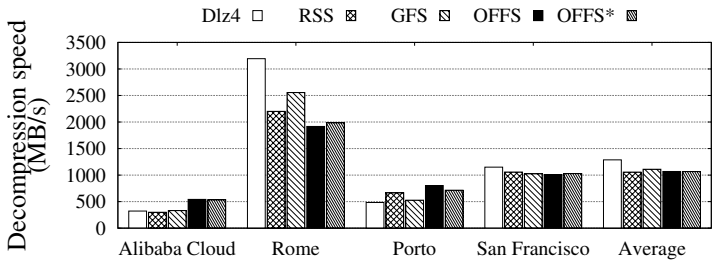
(5) Regarding the trade-off between CS and CR, we pick two sets of (i, k), the default mode $(4, 7)$ and the the fast mode $(2, 7)$. The default mode continues to refine candidates after the maximum size of candidates reaches δ, while the fast mode finishes the table construction after that. We denote them as OFFS and OFFS* for the following comparison.

5.7.4 Comparison with Baselines

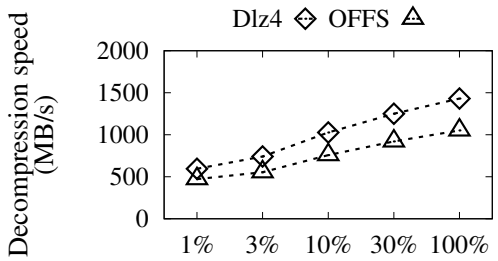
We then compare the proposed OFFS with two sets of baselines. Dlz4 serves as representative of generic compression methods, while RSS and GFS work as naive DICTs. Experiments are conducted on all datasets focused on CS and CR. The results shown in Figure 5.3 tell us the following.

(1) CR of OFFS is 5.11 on average as shown in Figure 5.3a, which is more than 3 times that of Dlz4 and 1.5 times those of RSS and GFS. Note that all dictionaries are trained from sufficient samples, which means the larger samples will not make a difference to CR. The result shows that OFFS works much better than Dlz4 and enhances a lot compared with GFS. It comes from the optimization of practical weighted frequency. We are not surprised to find that the average CR of GFS is worse than that of RSS due to match collisions. As for the quick mode, OFFS* only loses 0.33 compared to OFFS. These observations in Figure 5.3a suggest that OFFS improves a lot from Dlz4, providing better and stabler CR compared to the naive DICT solutions.

(2) CS of OFFS is 135 MB/s on average. It is 3 times faster than Dlz4 and 4 times faster than naive DICTs, while OFFS* further improves the speed of OFFS by 1.5 times. Figure 5.3b shows that naive DICTs are slower than Dlz4. Specifically, the gap could be an order of magnitude in large datasets like Porto and Alibaba Cloud. This suggests inefficiency and redundancy during the candidate collection process of naive DICTs in our case. As we notice in the Alibaba Cloud dataset, the speed drops when the space cost exceeds the available memory and causes a memory swap with I/O. Despite our attempts to avoid I/O, it cannot be neglected when the input size is large enough. It is preferable to adopt a more advanced stream mode that simultaneously handles reading and processing for that. However, here we leave it as a pressure test of



(a) Decompression speed



(b) Scalability of decompression

(c) Scalability of dictionary

Figure 5.4: Decompression comparison and scalability test

an extreme case of limited memory space. The result shows that OFFS is still more robust compared to baselines, and it is faster in CS and much better in CR when I/O is inevitable.

5.7.5 Retrieval and Scalability

We next evaluate the speed of data retrieval with decompression and the scalability of table construction. Note that all decompression processes start from the results in memory to avoid I/O impacts. For either full or partial decompression,

the output is the same set or subset as the input before compression, which is lossless. To test the scalability of table construction, we randomly pick paths to simulate the real-world case where we build a table based on first arriving samples.

(1) We first compare the DS of whole datasets between OFFS and baselines. Note that the DS of Dlz4 is almost ideal without reallocating memory because the exact sizes of input paths are recorded. It is to avoid memory issues on datasets like Porto whose maximum length is hundreds of times of average length. The results in Figure 5.4a show that OFFS is competitive against Dlz4 with DS around 1000 MB/s. It is not surprising to find that all DICT-based methods have approximately the same DS since they follow the same decompression strategy bounded by $O(|\mathbb{P}|/p)$.

(2) We next compare PDS between Dlz4 and OFFS with scalability from 1 to 100 percent of whole datasets. As shown in Figure 5.4b, DS of OFFS is 0.75 on average of that of Dlz4, and the average PDS is around 500 MB/s when the sample rate is 1 percent. It suggests that OFFS can handle partial decompression for data retrieval efficiently.

(3) We then demonstrate the scalability of OFFS based on various sample rates for table construction. The test sample rates change from 20 to 100 percent. As shown in Figure 5.4c, CR changes from 4.4 to 5.1 with the sample rate changing from 20% to 100%. In other words, the relative loss of CR is less than 15% when building on a sample as tiny as 20%. Better still, the CR of OFFS is more than 2.5 times that of Dlz4 with 20% sampling in our observation. It shows that OFFS is capable to build a more representative table than that of Dlz4 under scalability tests.

5.8 Conclusion

In this chapter, we present Overlap-Free Frequent Subpath, a strategy inspired by real-life cases from Alibaba Cloud to effectively reduce the overall size of path sets with easy retrievals to compressed paths. Specifically, we take advantage of a bottom-up framework of merge and expansion during the table construction stage to count practical weighted frequencies and refine the lookup table with iterations. Moreover, the granularity as finer as a path makes it friendly for parallelism in both compression and decompression stages. We clarify how to solve the challenge of match collisions of overlapped subpaths during table construction, provide analyses of time and space complexity, and compare with baselines in evaluation to highlight the performance of OFFS. Experiments show that OFFS dramatically improves compression ratio and compression speed, has competitive decompression speed, and demonstrates the scalability of table construction based on tiny samples.

Chapter 6

EPILOGUE

In this chapter, we summarize the works presented in this thesis and describe possible future research directions. In this thesis, we focus on efficient connectivity analysis and path management in massive graphs. The main contributions of this thesis can be concluded as follows:

- **Span Reachability in Temporal Graphs.** We define a span-reachability model to capture entity relationships in a specific period of temporal graphs. We propose an index-based method leveraging the concept of two-hop cover to answer the span-reachability query for any pair of vertices and time intervals. Notably, the proposed index construction method is guaranteed optimal both in terms of space and time complexity. Additionally, we address the problem of θ-reachability, a generalized version of span-reachability, and present an optimized solution with improved time complexity.

- **Time Interval Paths in Temporal Graphs.** We address the problem of enumerating time interval paths in temporal graphs, known as TIPST path enumeration, by introducing an efficient algorithm called TDDL-DFS.

Our algorithm incorporates novel temporal bundled dynamic labeling techniques, optimizing both time and space complexity to handle a considerable number of outputs during runtime processing. The theoretical analyses further establish the correctness and polynomial delay per output of our algorithm, providing robust assurances regarding its reliability and efficiency.

- **Path Compression in Large Graphs.** We present Overlap-Free Frequent Subpath (OFFS), a lossless strategy inspired by real-life cases to effectively reduce the overall size of path sets with easy retrievals to compressed paths. The proposed method is based on a bottom-up framework of merge and expansion during the table construction stage to refine the lookup table with iterations. We address the challenge of match collisions of overlapped subpaths during table construction to increase the compression ratio and provide analyses of time and space complexity.

Future Work and Research Opportunities. There are still some open problems and opportunities that need further research.

- **Grouped Span Reachability in Temporal Graphs.** Instead of considering a pair of nodes, grouped reachability focuses on the connectivity of a pair of groups of nodes. To address this problem, we could propose a novel grouped-based index for span reachability.

- **Parallel and Distributed Computing for Path Enumeration in Temporal Graphs.** Future work includes designing and implementing parallel and distributed algorithms for path enumeration in temporal graphs to optimize performance, scalability, and fault tolerance by leveraging the power of multiple processing units and distributed resources.

- **Incremental Framework for Path Compression.** Dynamic graphs with updated edges and nodes are common in path processing. A potential avenue for future work is to design an incremental framework to construct the supernode table when the paths are updated dynamically. With each new path, we can update the supernode table incrementally instead of recomputing it from scratch.

Bibliography

[1] "Lz4," https://lz4.github.io/lz4.

[2] "Zstd," http://facebook.github.io/zstd.

[3] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*, 2006, pp. 671–682.

[4] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical hub labelings for shortest paths," in *ESA*, 2012, pp. 24–35.

[5] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: enabling queries on compressed data," in *NSDI*, 2015, pp. 337–350.

[6] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *SIGMOD*, vol. 18, no. 2, 1989, pp. 253–262.

[7] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *PVLDB*, vol. 1215, 1994, pp. 487–499.

[8] S. Ahmadi, G. Tack, D. D. Harabor, and P. Kilby, "A fast exact algorithm for the resource constrained shortest path problem," in *AAAI*, vol. 35, no. 14, 2021, pp. 12 217–12 224.

[9] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *SIGMOD*, 2013, pp. 349–360.

[10] I. Alarab and S. Prakoonwit, "Graph-based lstm for anti-money laundering: Experimenting temporal graph convolutional network with bitcoin data," *Neural Process. Lett.*, vol. 55, no. 1, pp. 689–707, 2023.

[11] G. Antoshenkov, "Dictionary-based order-preserving string compression," *VLDBJ*, vol. 6, no. 1, pp. 26–39, 1997.

[12] G. Antoshenkov, D. Lomet, and J. Murray, "Order preserving string compression," in *ICDE*, 1996, pp. 655–663.

[13] K. Anyanwu and A. Sheth, "ρ-queries: enabling querying for semantic associations on the semantic web," in *WWW*, 2003, pp. 690–699.

[14] R. Bellman, "On the approximation of curves by line segments using dynamic programming," *CACM*, vol. 4, no. 6, p. 284, 1961.

[15] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *SIGMOD*, 2009, pp. 283–296.

[16] E. Birmelé, R. Ferreira, R. Grossi, A. Marino, N. Pisanti, R. Rizzi, and G. Sacomoto, "Optimal listing of cycles and st-paths in undirected graphs," in *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms*. SIAM, 2013, pp. 1884–1896.

[17] P. Boncz, T. Neumann, and V. Leis, "Fsst: fast random access string compression," *PVLDB*, vol. 13, no. 12, pp. 2649–2661, 2020.

[18] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets, "Querying graphs," *Synthesis Lectures on Data Management*, vol. 10, no. 3, pp. 1–184, 2018.

[19] L. Bracciale, M. Bonola, P. Loreti, G. Bianchi, R. Amici, and A. Rabuffi, "CRAWDAD dataset roma/taxi (v. 2014-07-17)," Downloaded from https://crawdad.org/roma/taxi/20140717, Jul. 2014.

[20] R. Bramandia, B. Choi, and W. K. Ng, "On incremental maintenance of 2-hop labeling of graphs," in *WWW*, 2008, pp. 845–854.

[21] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying graphs and dynamic networks," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 27, no. 5, pp. 387–408, 2012.

[22] L. Chang, X. Lin, L. Qin, J. X. Yu, and J. Pei, "Efficiently computing top-k shortest path join," in *EDBT*, 2015.

[23] Y. Chen and Y. Chen, "An efficient algorithm for answering graph reachability queries," in *ICDE*, 2008, pp. 893–902.

[24] Y. Chen, Y. Fang, R. Cheng, Y. Li, X. Chen, and J. Zhang, "Exploring communities in large profiled graphs," *TKDE*, vol. 31, no. 8, pp. 1624–1629, 2018.

[25] E. Cheng, J. W. Grossman, and M. J. Lipman, "Time-stamped graphs and their associated influence digraphs," *Discrete Applied Mathematics*, vol. 128, no. 2-3, pp. 317–335, 2003.

[26] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu, "Tf-label: a topological-folding labeling scheme for reachability querying in a large graph," in *SIGMOD*, 2013, pp. 193–204.

[27] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

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

[29] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner, "Lightweight data compression algorithms: an experimental survey (experiments and analyses)," in *EDBT*, 2017, pp. 72–83.

[30] P. Deutsch and J. L. Gailly, "Zlib compressed data format specification version 3.3," Tech. Rep. 2070-1721, 1996.

[31] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica*, vol. 10, no. 2, pp. 112–122, 1973.

[32] D. Eppstein, "Finding the k shortest paths," *SIAM Journal on computing*, vol. 28, no. 2, pp. 652–673, 1998.

[33] W. Fan, Y. Li, M. Liu, and C. Lu, "Making graphs compact by lossless contraction," in *SIGMOD*, 2021, pp. 472–484.

[34] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu, "Effective and efficient attributed community search," *VLDBJ*, vol. 26, pp. 803–828, 2017.

[35] M. Farach and M. Thorup, "String matching in lempel-ziv compressed strings," in *STOC*, 1995, pp. 703–712.

[36] P. Ferragina and G. Manzini, "Indexing compressed text," *JACM*, vol. 52, no. 4, pp. 552–581, 2005.

[37] A. Ferreira, "On models and algorithms for dynamic communication networks: The case for evolving graphs," in *In Proc. ALGOTEL*, 2002.

[38] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[39] J. Gao, H. Qiu, X. Jiang, T. Wang, and D. Yang, "Fast top-k simple shortest paths discovery in graphs," in *Proc. ACM Int. Conf. Inf. Knowl. Manag.*, 2010, pp. 509–518.

[40] A. Garcia-Duran, S. Dumančić, and M. Niepert, "Learning sequence encoders for temporal knowledge graph completion," in *EMNLP*, 2018, pp. 4816–4821.

[41] F. Goasdoué, P. Guzewicz, and I. Manolescu, "Rdf graph summarization for first-sight structure discovery," *VLDBJ*, vol. 29, no. 5, pp. 1191–1218, 2020.

[42] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes," in *ICDE*, 1998, pp. 370–379.

[43] R. Grossi, A. Gupta, and J. S. Vitter, "When indexing equals compression: experiments with compressing suffix arrays and applications." in *SODA*, vol. 4, 2004, pp. 636–645.

[44] R. Grossi, A. Marino, and L. Versari, "Efficient algorithms for listing k disjoint st-paths in graphs," in *LATIN 2018: Theoretical Informatics: 13th Latin American Symposium, Buenos Aires, Argentina, April 16-19, 2018, Proceedings 13*, 2018, pp. 544–557.

[45] S. Guha, "Efficiently mining frequent subpaths," in *AusDM*, 2009, pp. 11–15.

[46] S. Gurukar, S. Ranu, and B. Ravindran, "Commit: A scalable approach to mining communication motifs from dynamic networks," in *SIGMOD*, 2015, pp. 475–489.

[47] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM sigmod record*, vol. 29, no. 2, pp. 1–12, 2000.

[48] K. Hao, L. Yuan, and W. Zhang, "Distributed hop-constrained st simple path enumeration at billion scale," *PVLDB*, vol. 15, no. 2, pp. 169–182, 2021.

[49] Y. Hao, Y. Zhang, and J. Cao, "A novel qos model and computation framework in web service selection," *World Wide Web*, vol. 15, pp. 663–684, 2012.

[50] P. Holme, C. R. Edling, and F. Liljeros, "Structure and time evolution of an internet dating community," *Social Networks*, vol. 26, no. 2, pp. 155–174, 2004.

[51] P. Holme and J. Saramäki, "Temporal networks," *Physics reports*, vol. 519, no. 3, pp. 97–125, 2012.

[52] Y. Huang, D. Wen, L. Lai, Z. Qian, L. Qin, and Y. Zhang, "Efficient and effective path compression in large graphs," in *ICDE*, 2023, pp. 3093–3105.

[53] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," in *SIGMOD*, 2009, pp. 813–826.

[54] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *SIGMOD*, 2008, pp. 595–608.

[55] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, "On generating all maximal independent sets," *Inf. Process. Lett.*, vol. 27, no. 3, pp. 119–123, 1988.

[56] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.

[57] D. Kempe, J. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," *Journal of Computer and System Sciences*, vol. 64, no. 4, pp. 820–842, 2002.

[58] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "An online algorithm for segmenting time series," in *ICDM*, 2001, pp. 289–296.

[59] D. E. Knuth, *The art of computer programming, volume 4A: combinatorial algorithms, part 1.* Pearson Education India, 2011.

[60] K. A. Kumar and P. Efstathopoulos, "Utility-driven graph summarization," *PVLDB*, vol. 12, no. 4, pp. 335–347, 2018.

[61] R. Kumar and T. Calders, "2scent: An efficient algorithm to enumerate all simple temporal cycles," *PVLDB*, vol. 11, no. 11, pp. 1441–1453, 2018.

[62] R. Lasch, I. Oukid, R. Dementiev, N. May, S. S. Demirsoy, and K.-U. Sattler, "Faster & strong: string dictionary compression using sampling and fast vectorized decompression," *VLDBJ*, vol. 29, no. 6, pp. 1263–1285, 2020.

[63] K. LeFevre and E. Terzi, "Grass: Graph structure summarization," in *SDM*, 2010, pp. 454–465.

[64] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Softw., Pract. Exper.*, vol. 45, no. 1, pp. 1–29, 2015.

[65] R. H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai, "Persistent community search in temporal networks," in *ICDE*, 2018, pp. 797–808.

[66] X. Li, K. Hao, Z. Yang, X. Cao, and W. Zhang, "Hop-constrained s-t simple path enumeration in large uncertain graphs," in *ADC*, 2022, pp. 115–127.

[67] X. Li, K. Hao, Z. Yang, X. Cao, W. Zhang, L. Yuan, and X. Lin, "Hop-constrained s-t simple path enumeration in billion-scale labelled graphs," in *WISE*, 2022, pp. 49–64.

[68] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation: An index-based approach," in *The World Wide Web Conference*, 2019, pp. 1130–1141.

[69] H. Liu, C. Jin, B. Yang, and A. Zhou, "Finding top-k shortest paths with diversity," *TKDE*, vol. 30, no. 3, pp. 488–502, 2017.

[70] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *CSUR*, vol. 51, no. 3, pp. 1–34, 2018.

[71] A. Maccioni and D. J. Abadi, "Scalable pattern matching over compressed graphs via dedensification," in *SIGKDD*, 2016, pp. 1755–1764.

[72] S. Mazumder and B. Liu, "Context-aware path ranking for knowledge base completion," *arXiv preprint arXiv:1712.07745*, 2017.

[73] Y. Mehmood, N. Barbieri, F. Bonchi, and A. Ukkonen, "Csi: Community-level social influence analysis," in *ECML PKDD*, 2013, pp. 48–63.

[74] N. Meratnia *et al.*, "Spatiotemporal compression techniques for moving point objects," in *EDBT*, 2004, pp. 765–782.

[75] O. Michail, "An introduction to temporal graphs: An algorithmic perspective," *Internet Mathematics*, vol. 12, no. 4, pp. 239–280, 2016.

[76] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas, "Predicting taxi–passenger demand using streaming data," *TITS*, vol. 14, no. 3, pp. 1393–1402, 2013.

[77] S. Navlakha, R. Rastogi, and N. Shrivastava, "Graph summarization with bounded error," in *SIGMOD*, 2008, pp. 419–432.

[78] M. Nishino, N. Yasuda, S.-i. Minato, and M. Nagata, "Compiling graph substructures into sentential decision diagrams," in *AAAI*, vol. 31, no. 1, 2017.

[79] Y. Peng, Y. Zhang, X. Lin, W. Zhang, L. Qin, and J. Zhou, "Hop-constrained st simple path enumeration: Towards bridging theory and practice." *PVLDB*, vol. 13, no. 4, pp. 463–476, 2019.

[80] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser, "CRAWDAD dataset epfl/mobility (v. 2009-02-24)," Downloaded from https://crawdad.org/epfl/mobility/20090224, Feb. 2009.

[81] M. Potamias, K. Patroumpas, and T. Sellis, "Sampling trajectory streams with spatiotemporal criteria," in *SSDBM*, 2006, pp. 275–284.

[82] M. Riondato, D. García-Soriano, and F. Bonchi, "Graph summarization with quality guarantees," *ICDM*, vol. 31, no. 2, pp. 314–349, 2017.

[83] R. Rizzi, G. Sacomoto, and M.-F. Sagot, "Efficiently listing bounded length st-paths," in *IWOCA*, 2015, pp. 318–329.

[84] M. A. Roth and S. J. Van Horn, "Database compression," *SIGMOD Rec.*, vol. 22, no. 3, pp. 31–39, 1993.

[85] R. Schenkel, A. Theobald, and G. Weikum, "Efficient creation and incremental maintenance of the hopi index for complex xml document collections," in *ICDE*, 2005, pp. 360–371.

[86] K. Semertzidis, E. Pitoura, and K. Lillis, "Timereach: Historical reachability queries on evolving graphs." in *EDBT*, 2015, pp. 121–132.

[87] N. Sengupta, A. Bagchi, M. Ramanath, and S. Bedathur, "Arrow: Approximating reachability using random walks over web-scale graphs," in *ICDE*, 2019, pp. 470–481.

[88] Z. Shen, K. L. Ma, and T. Eliassi-Rad, "Visual analysis of large heterogeneous social networks by semantic and structural abstraction," *IEEE Trans. Vis. Comput. Graph.*, vol. 12, no. 6, pp. 1427–1439, 2006.

[89] P. Shiralkar, A. Flammini, F. Menczer, and G. L. Ciampaglia, "Finding streams in knowledge graphs to support fact checking," in *ICDM*. IEEE, 2017, pp. 859–864.

[90] D. Sorokin and I. Gurevych, "Context-aware representations for knowledge base relation extraction," in *EMNLP*, 2017, pp. 1784–1789.

[91] J. Su, Q. Zhu, H. Wei, and J. X. Yu, "Reachability querying: can it be even faster?" *TKDE*, vol. 29, no. 3, pp. 683–697, 2016.

[92] S. Sun, Y. Chen, B. He, and B. Hooi, "Pathenum: towards real-time hop-constrained st path enumeration," in *SIGMOD*, 2021, pp. 1758–1770.

[93] R. Tarjan, "Enumeration of the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 2, no. 3, pp. 211–216, 1973.

[94] G. Trajcevski, H. Cao, P. Scheuermanny, O. Wolfsonz, and D. Vaccaro, "On-line data reduction and the quality of history in moving objects databases," in *MobiDE*, 2006, pp. 19–26.

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

[96] T. Viard, M. Latapy, and C. Magnien, "Computing maximal cliques in link streams," *Theor. Comput. Sci.*, vol. 609, pp. 245–252, 2016.

[97] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *ICDE*, 2006, pp. 75–75.

[98] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *SIGMOD*, 2015, pp. 967–982.

[99] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *PVLDB*, vol. 7, no. 12, pp. 1191–1202, 2014.

[100] D. Wen, Y. Huang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "Efficiently answering span-reachability queries in large temporal graphs," in *ICDE*, 2020, pp. 1153–1164.

[101] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *ICDE*, 2016, pp. 145–156.

[102] B. B. Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *Int. J. Found. Comput. Sci.*, vol. 14, no. 02, pp. 267–285, 2003.

[103] 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*, 2013, pp. 1601–1606.

[104] W. Yao, J. He, G. Huang, J. Cao, and Y. Zhang, "A graph-based model for context-aware recommendation using implicit feedback data," *World Wide Web*, vol. 18, pp. 1351–1371, 2015.

[105] H. Yildirim, V. Chaoji, and M. J. Zaki, "Grail: a scalable index for reachability queries in very large graphs," *VLDBJ*, vol. 21, no. 4, pp. 509–534, 2012.

[106] ——, "Dagger: A scalable index for reachability queries in large dynamic graphs," *arXiv preprint arXiv:1301.0977*, 2013.

[107] Q. Yong, M. Hajiabadi, V. Srinivasan, and A. Thomo, "Efficient graph summarization using weighted lsh at billion-scale," in *SIGMOD*, 2021, pp. 2357–2365.

[108] J. X. Yu and J. Cheng, "Graph reachability queries: A survey," in *Managing and Mining Graph Data*, 2010, pp. 181–215.

[109] F. Zhang, W. Wan, C. Zhang, J. Zhai, Y. Chai, H. Li, and X. Du, "Compressdb: enabling efficient compressed data direct processing for various databases," in *SIGMOD*, 2022, pp. 1655–1669.

[110] T. Zhang, Y. Gao, L. Chen, W. Guo, S. Pu, B. Zheng, and C. S. Jensen, "Efficient distributed reachability querying of massive temporal graphs," *VLDBJ*, pp. 1–26, 2019.

[111] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: a total order approach," in *SIGMOD*, 2014, pp. 1323–1334.

[112] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[113] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[114] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *ICDE*, 2006, pp. 59–59.