Efficient Substructure Mining in Large Networks

by

YUXUAN QIU

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)

February, 2023

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, Yuxuan Qiu, 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: Production Note: Signature removed prior to publication.

Date: 11/02/2023

ACKNOWLEDGEMENTS

First of all, I am extremely grateful to my supervisor, A/Prof. Lu Qin, for his unwavering guidance and support throughout my research career. He is a responsible and patient mentor who has effectively guided me into the realm of academia. As an exceptionally intelligent and proficient researcher, he consistently provides fresh insights and unique perspectives on my research topic in each of our interactions. In addition to being my supervisor, A/Prof. Qin is also a good friend who genuinely cares about my personal life. Whenever I face challenges or setbacks, he is always there to offer a helping hand and words of encouragement. Without his invaluable support, I would not have been able to complete my PhD thesis to the high standard that it is today.

Secondly, I would like to extend my gratitude to my co-supervisors, Prof. Ying Zhang and Dr. Dong Wen, for their invaluable assistance and guidance during my PhD studies. Prof. Zhang has been instrumental in providing me with valuable advice and support for my research projects. Her expertise and insights have been indispensable to my success in completing my PhD. Dr. Wen has also played an essential role in my research journey. He has consistently encouraged me to explore the core of research problems, allowing me to experience the joy of conducting research. His patience in guiding me through acquiring research skills, such as experiment designing and paper writing, has been greatly appreciated. Moreover, I have gained a lot from his problem-solving approach and rigorous research attitude.

Thirdly, I would like to thank Prof. Xuemin Lin, Prof. Wenjie Zhang, A/Prof. Lijun Chang, and Dr. Wentao Li for their invaluable support and guidance throughout my research work and career. They have provided me with practical advice and served as exemplary role models for me to become a good researcher.

Fourthly, I would like to express my gratitude to Prof. Rui Mao and Prof. Rong-Hua Li, my supervisors at Shenzhen University, for their role in inspiring my research interests and offering valuable advice during my studies.

I would also like to express my gratitude to a large number of individuals who have supported me and made my journey through PhD study all the more enjoyable. These individuals include Dr. Xin Cao, Dr. Xiaoyang Wang, Prof. Fan Zhang, A/Prof. Yixiang Fang, A/Prof. Dian Ouyang, Dr.Junhua Zhang, Mr. Rong Hu, Dr. Conggai Li, Dr. You Peng, Asst. Prof. Kai Wang, Dr. Hanchen Wang, Ms. Yiqi Wang, Dr. Zhengyi Yang, Dr. Mingjie Li, Dr. Bohua Yang, Dr. Xiaoshuang Chen, Dr. Longbin Lai, Dr. Xubo Wang, Dr. Yuren Mao, Dr. Yixing Yang, Dr. Yu Hao, Dr. Qingyuan Linghu, Dr. Michael Ruisi Yu, Dr. Chenji Huang, Mr. Yilun Huang, Mr. Yuanhang Yu, Mr. Kongzhang Hao, Mr. Lantian Xu, Mr. Deming Chu, Mr. Jianwei Wang, Mr. Shiding Zhang, Mr. Yizhang He, Mr. Qingqiang Sun, Mr. Shunyang Li, Mr. Gengda Zhao, Mr. Qingshuai Feng, Mr. Zhuo Ma, Ms. Yuting Zhang, Ms. Yanping Wu, Mr. Guangxin Su, Mr. Yufan Sheng, Mr. Kaiyu Chen, Dr. Mingshan Jia, Dr. Joakim Skarding, Dr. Yanbin Liu, Dr. Xiaolin Zhang, Dr. Yiliao Song, Mr. Wenhao Ma, Dr. Kai Yao, Dr. Guangxi Li, and Dr. Huaxi Huang. These individuals have all provided their support and made significant contributions to my journey, and I have enjoyed the time spent with each of them.

Last but not least, I would like to express my heartfelt gratitude to my family for their unwavering love and support throughout my academic journey. I am deeply grateful to my father, Mr. Zhenghong Qiu, and my stepmother, Ms. Ling Wang, for always being there for me and for their encouragement every step of the way. I would also like to take a moment to remember my late mother, Ms. Jiaying Xu, whose love and support were instrumental in shaping the person I am today. Her memory and her profound impact on my life will always be with me. I am also grateful to my girlfriend, Ms. Shuxin Zhong, for her constant support, understanding, and encouragement. Her unwavering belief in my abilities has been a great source of motivation throughout my studies. Lastly, I want to thank my grandparents, sisters, relatives, and friends for their support and encouragement in various ways. I am blessed to have them in my life.

ABSTRACT

Graph models have been widely applied to represent the relationships between objects or entities. In graph models, the objects or entities are represented by vertices, and their relationships are represented by edges. In graph analysis, a subset of vertices and edges with distinct patterns form a substructure in the graph. Cohesive subgraphs and the shortest paths are two typical types of substructures that are widely used in many real-world applications. Given the importance of these substructures, in this thesis, we study the problems of mining them on large graphs.

Firstly, we study the cohesive subgraph substructures. We propose a novel cohesive subgraph model named the statistically significant cliques to fill the gap where most cohesive subgraph models do not consider the statistical significance. We propose an efficient branch-and-bound method with carefully designed pruning techniques to compute the maximal significant cliques.

Secondly, we investigate the shortest path substructures. We specifically study the shortest path counting problem, which is an important closeness metric and also serves as the building block for betweenness centrality. We observe the limitations of the state-of-the-art method and the opportunities to improve. A more advanced index structure based on tree decomposition is designed for the shortest path count computation. We also provide efficient algorithms to construct such an index.

Thirdly, we investigated the dynamic updating of our proposed index in response to graph updates. To achieve this, we developed a basic updating method that identifies the affected area in the index and updates the labels without requiring a complete recomputation. We then proposed enhanced algorithms to expedite these updates.

We conduct extensive experiments, and the results validate the effectiveness and efficiency of our proposed methods.

PUBLICATIONS

- ***Yu-Xuan Qiu**, Dong Wen, Rong-Hua Li, Lu Qin, Michael Yu, Xuemin Lin. "Computing Significant Cliques in Large Labeled Networks." In IEEE Transactions on Big Data, vol 9(3):904-917, 2023(**Chapter 3**)*
- ***Yu-Xuan Qiu**, Dong Wen, Lu Qin, Wentao Li, Rong-Hua Li, Ying Zhang. "Efficient Shortest Path Counting on Large Road Networks." In Proceedings of the 48th International Conference on Very Large Databases. VLDB, vol 15(10):2098-2110, 2022. (**Chapter 4**)*
- ***Yu-Xuan Qiu**, Dong Wen, Lu Qin, Wentao Li, Rong-Hua Li, Ying Zhang. "Efficient Shortest Path Counting with Dynamic Index Maintenance on Large Road Networks". In submission. (**Chapter 5**)*

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Graph models have been used to capture the relationships among entities in a wide spectrum of applications, such as social networks [44, 75], biological networks [85, 80], collaboration networks [98, 97], and road networks [76, 77]. The surge of graph-based applications has shifted the focus of research toward addressing the difficulties of graph management and analysis.

In graph analysis, substructures play a crucial role. Substructures are smaller components of a larger graph with specific patterns or structures. The usefulness of these substructures extends to a range of applications, such as detecting social network patterns [20] or scrutinizing the architecture of a protein [104]. By studying these substructures, analysts can gain insights into the overall structure of a graph and potentially uncover important relationships or trends.

Cohesive subgraphs and shortest paths are two representative substructures in graph analysis. Cohesive subgraphs are subgraphs that are more strongly connected than the rest of the graph. The shortest paths, on the other hand, are paths between two vertices in a graph that have the smallest lengths. This thesis will study the substructure mining problems from these two aspects.

1.1 Significant Clique Computation

Discovering cohesive subgraphs has been recognized as a fundamental problem with numerous applications like detecting social communities [96, 35] and mining protein complexes [65]. This report aims to mine statistically significant cohesive subgraphs, which have never been considered in previous studies. Generally, the task identifies a set of densely connected subgraphs with certain properties beyond the standard distribution.

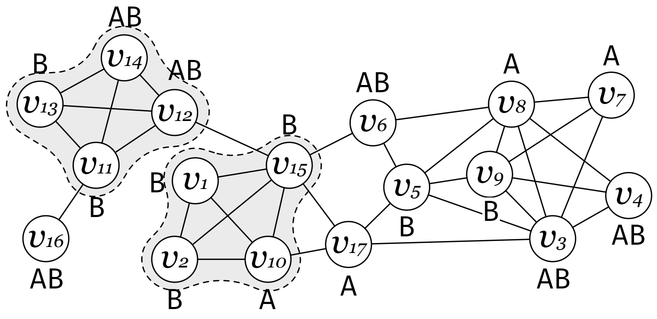


Figure 1.1: A labeled graph G and maximal (k, θ)-significant cliques in G given $k = 4, \theta = 6.0, p_A = 0.8$, and $p_B = 0.2$

Many efforts have been made on extracting significant substructures [81, 49, 87, 110] among the studies for mining subgraph patterns. Several works study the problem of frequent subgraph mining [87, 110], where a subgraph is considered to be significant if its frequency exceeds a predefined threshold. However, the subgraph frequency only considers the structural property, while vertices in a tremendous amount of real-world graphs are always associated with a set of labels or attributes. For example, in a protein-protein-interaction network, each vertex represents a protein, and the labels may represent protein functionalities. To capture the label statistics in significant subgraph mining, Arora et al. studied the problem of computing *statistically significant connected subgraphs*[9].

In statistics, the significance provides the evidence concerning the plausibility of the *null hypothesis*, which hypothesizes that the data distribution is only affected by random chance. If we have evidence to reject the null hypothesis, the corresponding result is considered statistically significant. In the statistically significant connected subgraph model, the null hypothesis is that the labels on each vertex are assumed to be assigned independently and randomly from a fixed probability distribution. The deviation between the actual labels and the expected labels measures the statistical significance and is computed via a function called *chi-square statistic* [88], which has also been used in many other works [9, 38, 107, 111]. The higher the chi-square, the higher the statistical significance [9, 87]. In this report, we also evaluate the significance by the chi-square statistic. Given a set of vertices U and their labels, the chi-square statistic is formally defined as follows.

$$f(U) = \sum_{i=1}^{l} \frac{(y_i - yp_i)^2}{yp_i},$$

where l is the number of distinct labels, y is the total number of all labels in U, y_i is the number of the i-th labels, and p_i is the expected frequency of the i-th label. A higher chi-square statistic means a higher deviation between the real and expected label distributions which may indicate some intrinsic properties. For example, the numbers of male and female staff are expected to be similar in a company. Given the real numbers of them, a high chi-square statistic may indicate some gender inequality in the company.

Clique is a fundamental and commonly used model for cohesive subgraph detection [71]. An induced subgraph S is a clique if there exists an edge between every pair of vertices in S. The clique model has drawn a great amount of research attention, such as enumerating maximal cliques [26, 29, 5], computing the maximum clique [67, 25], and mining clique-based subgraphs, e.g., signed cliques [64, 53, 74]. Several other cohesive subgraph models are briefly introduced in Section 2.1.

Our Model. Based on the concepts of chi-square statistic and clique, we propose a novel significant cohesive subgraph model, which is called (k, θ)-significant cliques, in vertex-labeled graphs. Given a size constraint k, a significance threshold θ, and a probability distribution P as the input, a (k, θ)-clique S satisfies the following three properties: (i) S is a clique in which every pair of vertices is connected; (ii) the chi-square statistic of S is at least equal to θ; and (iii) the number of vertices in S is no smaller than k. The first two properties support us to find significant cohesive subgraphs in the graph, and the third property helps us avoid some small graphlets like edges and triangles. We study the problem of enumerating all maximal (k, θ)-significant cliques in a graph. Given an integer $k = 4$, a real value $\theta = 6$ and a label probability distribution $\{p_{\mathsf{A}} = 0.8, p_{\mathsf{B}} = 0.2\}$, Fig. 1.1 shows an example of all maximal (k, θ)-significant cliques.

Applications. The problem of enumerating all maximal (k, θ)-significant cliques has many applications, including but not limited to discovering influential research groups in collaboration networks [57], detecting topic-centric communities in social networks [34], and revealing important functional organizations in PPI networks [66].

Research group discovery in co-author networks. In a co-author network (e.g., DBLP), two researchers are connected by an edge if they have a common publication. A researcher may have several labels or attributes, and each label represents a conference or a journal where the researcher has a paper published.

Setting a relatively low expected frequency for some high-ranking conferences or journals in a research domain leads to a higher chi-square statistic for groups with more such publications. Our model can be applied to identify outstanding research groups with many high-quality publications. We have conducted a case study on DBLP to discover such research groups. The details can be found in Section 4.5.

Topic-centric community detection in social networks. In social networks, each user may have several labels representing the followed topics, like soccer and basketball. The model can help mine topic-centric communities that have a strong association with some specific topics (or features) far beyond others.

For example, in sports marketing, it is crucial to locate the avid sports fans. Assume that we would like to mine a set of pure "soccer" communities for recommendations and advertisements. A straightforward method is to collect all the vertices following "soccer" and compute cliques in the induced subgraph. However, this method does not consider other labels, and the resulting communities may also be highly interested in other sports. By setting a suitable parameter, our model can find a set of communities whose members care about "soccer" far beyond other sports.

Organization mining in biological networks. In PPI networks, each protein is assigned several labels by its functionalities. By setting specific parameters, our model can be used to identify a set of biological organizations (closely connected proteins) with some particular statistics far beyond normal. The derived structures may play crucial roles in certain biological processes.

Note that there have been several keyword-based community models in labeled networks. However, they cannot easily cover our research problem and techniques. First, existing works either cannot guarantee strong structural cohesion [72, 83] or focus on other cohesion models like k-core and k-truss [32]. To the best of our knowledge, we are the first to study significant clique mining in labeled graphs given the prevalence of the fundamental clique model. Our model guarantees both the strongest structural cohesion and flexible keyword significance. Second, several works only accept an input graph and cannot support personalized keyword (distribution) queries [32, 15, 73, 72, 84]. For example, Chu et al. [32] find cohesive subgraphs where the common pattern is frequent in all vertices. The common pattern is generated by the algorithm. Third, several keyword-based community detection models accept a set of keywords as the input and only consider the existence of keywords[42, 54, 117]. Such models compute cohesive subgraphs where each vertex covers as many given keywords as possible. In these models, the importance of all input keywords is always the same. By contrast, the statistical significance model provides an input of keyword distribution. Thus, we can flexibly assign different strengths to input keywords according to specific scenarios.

Challenges. It is challenging to compute all maximal (k, θ)-significant cliques. First, the problem is NP-hard, which is proved in Section 3.2.2 by showing that the maximal clique enumeration is a special case of our problem. Second, the significance constraint in the model is not anti-monotonic. In other words, even though we find a clique S with a chi-square statistic less than θ, a sub-clique of S may still have a chi-square statistic larger than θ. As a result, we still need to check every possible sub-clique of S further. Therefore, the technical challenges include how to correctly output maximal (k, θ)-significant cliques without any duplication and how to prune the search space effectively.

Our Solution. Based on the concept of k-core [94] and graph coloring [99] in existing studies, we propose a basic branch-and-bound algorithm. However, the integer k is normally small to only filter out some small motifs, which diminishes the pruning effectiveness of basic structural rules. To improve the practical efficiency, we observe an upper bound for the chi-square statistic of a given vertex set (Theorem 2). Based on the upper bound, we combine the concepts of k-core and graph coloring and derive a vertex reduction strategy with stronger structural pruning effectiveness and a statistical pruning rule. We further extend the ideas from vertices to edges and propose an edge pruning strategy.

Contributions. We summarize the main contributions in this work as follows.

- *An elegant significant cohesive subgraph model.* We propose a novel subgraph model, called (k, θ)-significant clique in labeled graphs. We prove that the problem of computing all maximal (k, θ)-significant cliques is NP-hard.

- *An algorithm for significant clique enumeration.* We propose a novel branch-and-bound algorithm to enumerate maximal (k, θ)-significant cliques without any duplication.

- *Several strategies to prune the search space.* We propose two effective pruning strategies from the perspectives of both vertex reduction and edge reduction, which take $O(m \cdot \log deg_{max})$ and $O(m^{1.5})$ times, respectively. m is the number of edges, and deg_{max} is the maximum degree.

- *Extensive performance studies.* We conduct extensive experiments on seven real-world datasets to evaluate the efficiency of our proposed algorithms. We also conduct a case study to show the effectiveness of our model.

The details of this work are presented in Chapter 3.

1.2 Shortest Path Counting

Given the strong expressive power of the graph model, road maps are often abstracted as graphs, aka road networks, in many real-world location-based services and analytical tasks. In these applications, each road is represented by an edge, and each intersection of roads is represented by a graph vertex. The real distance of each road is modeled as a weight value for each edge in the graph.

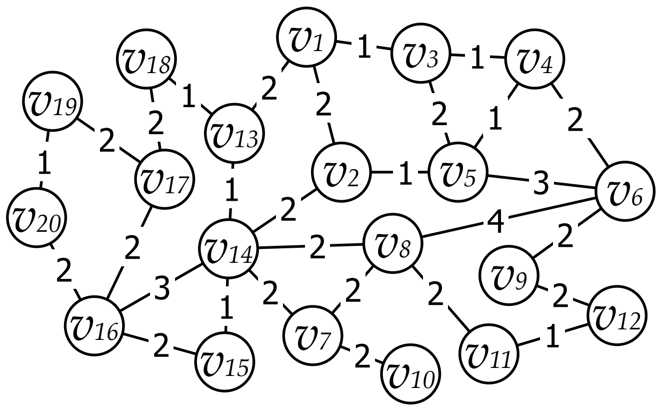


Figure 1.2: A road network $G(V, E)$.

In analyzing road networks, the concept of the shortest path is important and lays the foundation of many complex location-based queries, like the shortest distance [76], kNN [77] and betweenness centrality [10]. The distance or length of a path is the sum of weights of all edges in the path. A path p is the shortest path if there does not exist a path with the same terminal vertices and a distance value smaller than p. The shortest distance between two vertices is the distance of their shortest path. It is a standard metric to evaluate how close (or similar) the two vertices are.

A great deal of research effort has been contributed to efficiently querying the shortest distance between query vertices in graphs [76, 46, 4, 12, 108, 118]. However, a vertex may reach multiple other vertices with the same shortest distance, which weakens the effectiveness of the shortest distance as a closeness metric. A recent work [115] breaks the tie by formulating the shortest path counting problem, which aims to compute the number of shortest paths between two query vertices in a graph. In this thesis, we study the shortest path counting problem on road networks, where the distance is rounded to a specific precision, e.g., meters.

In real road network applications, more shortest paths indicate more traffic options and more flexibility for route planning from the start vertex to the destination. For instance, top-k nearest neighbors search aims at finding k objects close to the query vertex from a candidate set. It is a key operator in taxi-hailing (e.g., Uber), restaurant (e.g., Tripadvisor), and hotel recommendation (e.g., Booking) services. A candidate object can be more desirable than others with the same or similar distance if many shortest paths lead to the object since we have more backup routing plans and a higher probability of avoiding traffic jams. For example, in a movie ticket application, there are two cinemas with the similar shortest distance to the source location. We may prefer the one with more shortest paths considering the traffic options.

In addition to serving as a closeness metric, the shortest path count has been used as a building block of betweenness centrality computation [82, 86]. On road networks, the betweenness centrality is widely used as a static predictor of congestion and load, which helps predict the traffic flow [60]. Given a vertex u, the betweenness centrality of u, denoted by $C_B(u)$, is the fraction of shortest paths passing u, i.e., $C_B(u) = \sum_{s \neq u \neq t \in V} \frac{\mathsf{spc}_u(s,t)}{\mathsf{spc}(s,t)}$, where $\mathsf{spc}(s,t)$ is the number of shortest paths between s and t, and $\mathsf{spc}_u(s,t)$ is the number of paths passing u in $\mathsf{spc}(s,t)$. [86] observes that $\mathsf{spc}_u(s,t) = \mathsf{spc}(s,u) \cdot \mathsf{spc}(u,t)$ if $\mathsf{sd}(s,u) + \mathsf{sd}(u,t) = \mathsf{sd}(s,t)$, where $\mathsf{sd}(s,t)$ is the shortest distance between s and t. Based on this property, several works precompute the shortest distances and the shortest path counts for a set of vertex pairs to approximately compute the betweenness centrality [90, 86, 21, 11]. The techniques in this thesis can significantly improve the efficiency of counting shortest paths and boost the efficiency of betweenness centrality computation in practice. Apart from road networks, the proposed method can also be applied to other infrastructure networks, like power grid networks and public transportation networks, which have a small tree height [69, 19].

The State-Of-The-Art Solution. The state-of-the-art algorithm for shortest path counting is proposed in [115]. In [115], the authors devise a labeling-based index by assigning a total order for all vertices. Specifically, for each vertex u, they precompute the shortest distances and the shortest path counts to some vertices with higher ranks than u in the order. To query the number of shortest paths between two vertices u and v, they find every common vertex in the label sets of these two vertices. Each common vertex p acts as a bridge to connect two shortest sub-paths. The sum of two sub-paths' distances is their distance, and the product of two sub-paths' counts is the number of the shortest paths between u and v in terms of p in the index. They sum the counts for all common vertices whose corresponding distances are the shortest between u and v.

Challenges. Their indexing scheme works well as a general method. However, there still exist several challenges and room for improvement. First, based on a total vertex order, a low-ranking vertex may have a large number of labels in the index. More labels imply more index space usage and more comparisons in the query processing. Second, they order the labels for each vertex and perform a merge-sort-like strategy to find common vertices in query processing. Given that the label size for each vertex is not well-bounded, the query strategy needs to access all labels, thus incurring much time overhead. Third, to compute the order-based labels, [115] searches every vertex in the induced subgraph of all vertices with lower ranks. The search space can be the whole graph, which makes the index construction inefficient in large graphs.

Our Approach. In this thesis, we propose a new labeling-based index structure that is carefully defined for road networks and other sparse graphs. We adopt the concept of tree decomposition [18] and propose a tree-based labeling structure, given that real-world road networks normally have a low average degree and small treewidth [76, 77]. Specifically, we organize all vertices in the graph into a tree structure such that there is a one-to-one correspondence between the vertices and the tree nodes. By our indexing scheme, we only store a label for each ancestor of a vertex in the tree, which bounds the label size of each vertex well. In query processing, we derive several useful properties which enable us to only consider the common ancestors of two query vertices in the tree. As a result, we only check a small number of labels which significantly improves the query efficiency. Our index is also a labeling-based structure and satisfies the concept of exact shortest path covering defined in [115]. Their hub-pushing-based index construction paradigm can be naturally adapted to construct our tree-based index. In order to enhance indexing efficiency, we propose a new index construction framework and avoid the costly graph search in [115]. A crucial phase in the index construction process involves calculating the shortest distance and path count from a vertex u to one of its ancestors v. We propose several rules to reduce all the descendants of u in the tree while preserving the correctness of all shortest paths in the small reduced graph. We compute the result from u to v by utilizing the values derived in previous rounds and avoid searching the graph by only scanning the neighbors of u in the reduced graph.

Contributions. We summarize our main contributions as follows.

- *A novel tree-based index algorithm.* We design a novel index structure called TL-Index. Let n be the number of vertices, h be the treeheight, and w be the treewidth. Our index size is bounded by $O(nh)$, and the query time is bounded by $O(h)$. By contrast, the index size and the query processing time of the state-of-the-art solution are bounded by $O(nw \log n)$ and $O(w \log n)$, respectively [115]. As shown in our experiments (Section 4.5), h is much smaller than $w \log n$ in real-world graphs. For instance of the New York City map, we have $h = 505$ and $w \log n = 2412$. Therefore, our solution achieves higher query efficiency than the state-of-the-art method with smaller space usage.

- *A new index construction paradigm.* We propose a new paradigm to construct the index and two optimizations to improve efficiency. Compared to the index construction framework proposed in [115], we improve the time complexity of index construction from $O(nh^2 + nh \log n)$ to $O(nhw + n \log n)$, because w is typically several times smaller than h in practice.

- *Extensive experiments and evaluations.* We conduct extensive experiments on 14 real-world networks, including the USA map with 24 million vertices and 58 million edges. The state-of-the-art method cannot finish indexing within 24 hours on the USA map, while our proposed method only takes one hour. On other large real-world maps, our method achieves 20 times faster indexing and seven times faster querying than the state-of-the-art method. The results validate the effectiveness of our index structure and the efficiency of the indexing algorithm.

The details of this work are presented in Chapter 4.

1.3 TL-Index Maintenance

In real-world applications, the edge weight on road networks can change. For example, assuming that the weight on each edge indicates the average traveling time on each road segment, it may vary between rush hour and off-peak. As the index size is typically very large, it would be inefficient to recompute the entire index from scratch each time we have a weight change on the road network. Therefore, we aim to find an efficient index maintenance method for updating the index when the weight of some edges on the road network changes.

The problem of updating tree-decomposition-based index for shortest distance queries has been studied in some existing works [113, 116]. However, their methods cannot be directly applied to update our index for several reasons.

- First, their methods only consider the updates of the shortest distances in the index. In our index, apart from the shortest distances, there are also labels for the shortest path counts. It is possible that the shortest distance between two nodes in the tree remains unchanged, but the shortest path count may vary when the index updates. Therefore, it is necessary to consider more elements within the index that may be impacted by the updates.
- Second, while the shortest distance values may always exhibit the minimality property, meaning that the updated value will always be the minimum after multiple updates, the same property does not hold for the shortest path count values. If the same edge is updated twice, the value may become incorrect. Hence, we must design the updating process cautiously to prevent repeat updates.
- Third, the shortest distance labels in their index differ from ours. In their index, the shortest distance between two nodes is the global shortest distance, whereas, in our index, we use the local shortest distance as described in Section 4.4.4. This means it is unnecessary to compute the global shortest distances during the update process.

As a result, this work presents a novel up-and-down updating paradigm to identify nodes that may be affected by the updates. Additionally, we developed a sophisticated label update strategy to maintain our index in the presence of an increase and decrease in the weight of the road network, respectively. Furthermore, we explore the impact of the local distance labels in our index. Following the setting of existing works [113, 116], we only consider the weight change as the vertices and edges can be reasonably assumed to be stable since road construction and destruction are rare in practice.

The details of this work are presented in Chapter 5.

1.4 Roadmap

The rest of this thesis is organized as follows. Chapter 2 discusses related works. Chapter 3 introduces the significant clique model on large labeled graphs. Chapter 4 presents the shortest-path count queries on large road networks. Chapter 5 proposes a maintenance framework for the TL-Index proposed in Chapter 4. Chapter 6 concludes the whole thesis.

Chapter 2

LITERATURE REVIEW

Due to the wide applications of graph substructures like cohesive subgraphs and shortest paths, efficient computation has drawn a lot of research work. In this chapter, we first survey the literature on cohesive subgraph mining and then the research on shortest path queries.

2.1 Techniques for Cohesive Subgraph Mining

Significant Sub-structures on Graphs. Many real-world applications rely on exploiting statistically significant sub-structures on graphs, which include significant paths [102], trees [50], and subgraphs [87, 110]. Zhang et al. propose a sampling method based on modularity to detect significant communities on graphs [114]. He et al. utilize the p-value bound to develop a local search algorithm to find significant subgraphs[52]. A brief survey on significant sub-structures can be found in [28]. However, most previous works are tailored to unlabeled graphs that ignore the label information on vertices. Arora et al. propose a statistically significant connected subgraph model, which depicts the label figure with chi-square statistics [9]. This model may not be applicable to our problem, as the connection between the nodes inside a connected subgraph may be very loose.

Community Modeling. The community models over graphs have been extensively explored. Communities in a graph are often modeled by a group of densely connected nodes. In the literature, various community models and algorithms have been proposed, which include clique [30, 24, 25], k-core [94, 106], k-truss [105], k-clan [70], k-plex [16], and so on. In recent years, more models for community detection that consider graph label information have also been developed. Notable examples include the k-core-based attributed community model [42], the truss-based attributed community model [54], the keyword-centric attributed community searching model [117]. However, all the above models never consider the statistical significance of the discovered community, which cannot be directly applied to our problem.

Maximal Clique Enumeration. The clique model has a wide range of applications. The enumeration of all maximal cliques in a graph has long been a popular problem in graph data mining. Many existing works have been put forward to study the problem. Most of them are based on a backtracking diagram [24, 100, 41]. Tomita et al. [100] propose a pivoting technique which is proved to be worst-case optimal. [41] further improves the time complexity of maximal clique enumeration on sparse graphs. Jin et al. [58] propose an approach combining a hybrid data structure and a new pivot selection rule to accelerate the enumeration. [30] and [109] propose I/O efficient and distributed algorithms, respectively. Chang et al. [26] proposes an algorithm to progressively compute maximal cliques in polynomial delay. More recently, enumerating cliques with additional information has been researched. For example, [112] investigates the problem in the context of a spatial database. Li et al. [64] propose a signed clique model on signed networks.

2.2 Techniques for Shortest Path Queries

Shortest Distance Query in Networks. Querying the shortest distance is one of the most critical problems in graph data analysis as it has many real-world applications like driving directions or network routing. The Dijkstra algorithm [40] is one of the most renowned algorithms for this problem. However, when the network is large, such online algorithms may not be efficient in solving the shortest distance queries. Thus, existing research works mainly focus on pre-computing an effective index to accelerate query processing. For example, Goldberg et al. proposed an A* search method accelerated by precomputed shortest distances [48]. Gavoille et al. studied the labeling methods for undirected graphs [45]. There is a class of algorithms that exploits the graph hierarchies to accelerate the query. Sanders et al. designed the Highway Hierarchies, which imitates the natural hierarchies of road network [93]. When answering the query, it utilizes the highway to reduce the search space. Geisberger et al. proposed another hierarchy-based algorithm named Contraction Hierarchies [46] which relies on a pre-assigned total order. It removes the less important vertices along the pre-assigned order and generates shortcuts between the remaining vertices. Another important class of methods for shortest distance query is hub-labeling-based algorithms [33]. The hub labeling is also named 2-hop labeling, which assigns a 2-hop label for each vertex in the graph. To answer the shortest distance queries, it simply joins the 2-hop labels to compute the answers. Abraham et al. studied efficient hub-labeling algorithms for road networks [1, 2]. Ouyang et al. leveraged the advantages of both hub labeling and hierarchy to devise a Hierarchical 2-Hop (H2H) labeling scheme for road networks [76]. This approach organizes the hub-labels into a tree structure and utilizes the tree decomposition to facilitate hop-link searches. The H2H-Index assigns a label for each vertex and at the same time preserves a hierarchy among all vertices. When query the index for the shortest distance between two given vertices, H2H examines their labels for the vertices recorded in their common ancestor node to get the result. The time complexity of this approach is $O(w)$, where w represents the treewidth, which is equivalent to the number of vertices recorded in the ancestor node. Nonetheless, despite the similar tree structure used by H2H, their method isn't directly applicable to our scenario since it is specifically engineered for querying the shortest distance and falls short when computing shortest path count values. Unlike shortest distance computations, the shortest path count displays a more intricate property, which presents a challenge in designing refined structures to circumvent repetition or omission. Chen et al. [27] proposed the P2H method which improves the H2H labeling scheme by reducing the label size. Akiba et al. presented the pruned highway labeling [3] and the pruned landmark labeling [4] for road networks and scale-free networks, respectively.

In real-world networks, the weights of edges may fluctuate over time, thereby affecting the resulting shortest distance between vertices. Consequently, several algorithms have been introduced to update indexes for dynamic networks. For instance, Geisberger et al. proposed a vertex-centric method for maintaining Contraction Hierarchies (CH) based index [47]. This method identifies the affected vertices and then re-contracts the shortcut index for updated networks. In a similar vein, Delling et al. suggested a method for maintaining overlay graphs for the CRP algorithm [37]. Ouyang et al., on the other hand, proposed a shortcut-centric algorithm to update the CH [78]. For the H2H index, Zhang et al. presented a method known as DTDH for updating the shortest distance labels [113]. Despite its effectiveness in updating tree-decomposition-based index for shortest distance queries, as discussed in Section 1.3, this method cannot be directly applied to our index for shortest path counting. This limitation arises because DTDHL only contemplates the shortest distance, which is simpler to maintain during value changes compared to shortest path count values. Moreover, the information our index stores differs from theirs, making the application of the DTDHL method to our index even more infeasible. On top of DTDHL, Zhang and Yu have developed an improved method for updating the H2H index [116]. However, just like its predecessor, this method encounters significant challenges when applied to our specific scenario.

Network Substructure Counting. Counting the occurrence of certain substructures is also a fundamental problem in graph data analysis. In the literature, many research works have been done on counting small subgraphs like motifs [23, 68] or graphlets [22]. Jain et al. proposed an elegant clique counting algorithm based on classic pivoting techniques [56]. Shi et al. developed a parallel clique counting algorithm [95]. A comparison between different k-clique counting or listing algorithms can be found in [63].

Triangle counting has gained popularity in graph substructure counting due to its fundamental significance in diverse fields, including social network analysis, computational biology, and recommendation systems. The simplest method to count triangles is a brute-force approach: enumerating all vertex triplets and verifying if these three vertices constitute a triangle. The complexity of this method is $O(n^3)$. However, for real-world graphs, a more efficient node-iterative approach can be employed by examining the neighboring vertex pairs for each vertex in the graph, reducing the time complexity to $O(n \cdot d_{max}^2)$.

In the literature, many methods boasting better runtime performance have been proposed. Itai and Rodeh developed one of the earliest triangle enumeration methods with a running time of $O(m^{\frac{3}{2}})$ [55]. Alon, Yuster, and Zwick devised the AYZ method to count triangles by combining the node-iterator and matrix multiplication [7]. Their method achieves a complexity of $O(m^{\frac{2\gamma}{\gamma+1}})$. Due to its complex nature, researchers have sought ways to accelerate triangle counting. One such approach is approximation, with Tsourakakis et al. proposing one of the earliest methods for approximating triangle counting named DOULION[101].Recently, GPU-based acceleration has garnered increased interest in academia. For instance, Pandey et al. proposed a vertex-centric hashing-based method called TRUST that achieves over one trillion Traversed Edges Per Second (TEPS) rate for triangle counting on GPUs [79]. A comprehensive survey on triangle counting can be found in [6].

There are also many works study the counting of paths or cycles in the literature. Flum et al. proved that counting the cycles and paths of length k in both directed and undirected graphs, parameterized by k, is #W[1]-complete [43]. Valiant proved that the $s-t$ simple path counting problem is #P-complete [103]. Because of its #P-complete complexity, Roberts gave an estimating algorithm to estimate the number of simple paths [91]. Given the specific query vertices s and t, Bezakova et al. provided a shortest paths counting query method for planar graphs [17]. Zhang et al. devised a hub-labeling-based method for shortest path counting on large graphs [115]. There are also studies on other specific graphs. Ren et al. studied the problem of shortest path counting in probabilistic biological networks[89]. He et al. proposed a data structure for categorical path counting queries, which asks the number of distinct categories on a path in a given tree between two query nodes [51].

Chapter 3

COMPUTING SIGNIFICANT CLIQUES IN LARGE LABELED NETWORKS

3.1 Chapter Overview

In this chapter, we study the significant cliques computation in large labeled networks. This chapter is organized as follows. Section 3.2 introduces background knowledge and defines the problem. Section 3.3 proposes a non-trivial baseline algorithm. Section 3.4 gives several pruning strategies. Section 3.5 presents the final algorithm. Section 3.6 reports the performance studies. Section 3.7 concludes this chapter.

3.2 Preliminaries

We first introduce the problem definition of maximal statistical significant clique enumeration in Section 3.2.1. The problem has two main challenges, i.e., NP-hard time complexity and non-monotonicity, which is illustrated in Section 3.2.2.

3.2.1 Problem Definition

We consider an undirected labeled graph $G(V, E, \mathcal{L})$. V is the set of vertices. $E \subseteq (V \times V)$ is the set of edges. $\mathcal{L}$ assigns one or more labels to each vertex from a label set L, i.e. $\mathcal{L} : V \to \bigcup_{v \in V, L_v \subseteq L} L_v$. We use n and m to represent $|V|$ and $|E|$, respectively. Given a vertex u, the neighbor set of u is denoted by $N(u)$, i.e., $N(u) = \{v \in V | (u, v) \in E\}$. The degree of u is denoted by $deg(u)$, i.e., $deg(u) = |N(u)|$. A subgraph $S(V_S, E_S)$ is called an induced subgraph of G if $V_S \subseteq V$ and $E_S = \{(u, v) \in E | u \in V_S, v \in V_S\}$. A subgraph S of G is a clique if every two vertices in S are connected, i.e., $\forall u, v \in V_S, (u, v) \in E_S$. The clique S is called a k-clique if there are k vertices in S, i.e., $|V_S| = k$.

Given a set of vertices $U \subseteq V$, assume that l is the number of distinct labels in U, i.e., $l = |\bigcup_{u \in U} \mathcal{L}(u)|$. We have an observed frequency vector $Y = \{y_1, y_2, ..., y_l\}$, where $y = \sum_{i=1}^{l} y_i = \sum_{u \in U} |\mathcal{L}(u)|$. Given a fixed label probability distribution $P = \{p_1, p_2, ..., p_l\}$, the *chi-square* statistic [88] (also called statistical significance [9]) of U is defined as follows.

$$f(U) = \sum_{i=1}^{l} \frac{(y_i - yp_i)^2}{yp_i} = \sum_{i=1}^{l} \frac{y_i^2}{yp_i} - y \tag{3.1}$$

Example 1. *Given the graph G in Fig. 1.1, we consider the induced subgraph of $\{v_5, v_6, v_8\}$. We have two* A *labels and two* B *labels. We have $l = 2$. The observed frequency vector is $Y = \{y_\mathsf{A} = 2, y_\mathsf{B} = 2\}$, and $y = 4$. Assume that the probability distribution of the labels is $P = \{p_\mathsf{A} = 0.8, p_\mathsf{B} = 0.2\}$. The chi-square of $\{v_5, v_6, v_8\}$ is $\frac{4}{4\times0.8} + \frac{4}{4\times0.2} - 4 = 2.25$.*

The chi-square statistic of a subgraph represents the deviation of the observed label frequency from the expected frequency distribution, which is a widely used metric to quantify the statistical significance [88, 38, 107, 111]. Arora et al.[9] show that the subgraph with a large chi-square statistic is considered to be highly significant. Based on Eq. 3.1, we define a new subgraph model, called (k, θ)-*significant clique*, as follows.

Definition 1. (SIGNIFICANT CLIQUE) *Given a graph G, a probability distribution $P = \{p_1, p_2, ..., p_l\}$, an integer k and a real value θ, a (k, θ)-significant clique is an induced subgraph C that satisfies the following constraints:*

- *Clique constraint: C is a clique;*
- *Chi-square constraint: $f(V_C) \geq \theta$;*
- *Size constraint: $|V_C| \geq k$.*

In Definition 1, the clique constraint ensures that the subgraph is densely connected and can be a cohesive pattern or a social community in real-world graphs. The chi-square constraint ensures that the subgraph is highly significant in the given graph. The size constraint filters out small resulting motifs in the (k, θ)-significant cliques. The probability distribution P enables flexibility to adjust the importance of the labels.

Definition 2. (MAXIMAL SIGNIFICANT CLIQUE) *A subgraph C of G is a maximal (k, θ)-significant clique if (i) C is a (k, θ)-significant clique, and (ii) there is no (k, θ)-significant clique C' in G which contains clique C.*

A (k, θ)-significant clique may contain several subgraphs which are still (k, θ)-significant cliques. The maximality of the model reduces the redundancy in resulting subgraphs.

Example 2. *Fig. 1.1 shows an example of the maximal significant cliques. Given $k = 4, \theta = 6.0, p_{\mathsf{A}} = 0.8$ and $p_{\mathsf{B}} = 0.2$, all maximal $(4, 6)$-significant cliques in G*

are marked by gray. Note that if $k = 3$*, we have* $f(v_{11}, v_{12}, v_{13}) = 7.563$*. As a result, the induced subgraph of* $\{v_{11}, v_{12}, v_{13}\}$ *is a* $(3, 6)$*-significant clique but not maximal.*

We use (k, θ)-clique to represent the maximal (k, θ)-significant clique for short when the context is clear. Based on Definition 2, we define the research problem as follows.

Problem Statement. Given a labeled graph G, a probability distribution P, an integer k and a real value θ, we aim to enumerate all maximal (k, θ)-significant cliques in G.

Table 3.1: Frequent notations used in Chapter 3.

Notation	Meaning
$G = (V, E, \mathcal{L})$	undirected labeled graph
V	the set of vertices
$E \subseteq (V \times V)$	the set of edges
$\mathcal{L} : V \to \bigcup_{v \in V, L_v \subseteq L} L_v$	label assigning function
$N(u)$	the neighbor set of u
$deg(u)$	the degree of vertex u
$deg_c(u)$	colorful degree of a vertex u
$f(U)$	chi-square significance of vertex set U
$f(u)$	chi-square significance of a vertex u
$f_n(u)$	neighborhood significance of a vertex $f_n(u)$
$f_{cn}(u)$	colorful neighborhood significance of a vertex u
$f_n(u, v)$	support significance of an edge (u, v)

3.2.2 Hardness and Challenges.

NP-hard Time Complexity

We prove the hardness of our problem by considering a closely related problem — maximal clique enumeration, which has been widely studied in the literature [30, 24, 100, 41]. All maximal cliques are the results of a special case of our problem.

Specifically, given $k = 0$ and $\theta = 0$, the chi-square statistic of an arbitrary vertex set is always no less than θ, and the problem of enumerating $(0, 0)$-significant cliques is equivalent to the problem of maximal clique enumeration. Given that the maximal clique enumeration problem is NP-hard, our problem is also NP-hard.

Non-Monotonicity

Given a set S, an anti-monotonic constraint means that if S satisfies (or does not satisfy) the constraint, any subset of S also satisfies (or does not satisfy) the constraint. For example, the clique constraint in Definition 1 is anti-monotonic since any subgraph of a clique is also a clique. The size constraint in Definition 1 is anti-monotonic since if a graph S has fewer than k vertices, any subgraph of S also has fewer than k vertices. However, the chi-square constraint in Definition 1 is not anti-monotonic. In other words, given a graph S with $f(V_S) \geq \theta$ and an arbitrary subgraph S' of S, we cannot derive $f(V_{S'}) \geq \theta$ and vise versa.

Example 3. *Given the graph G in Fig. 1.1, assume that $k = 4, \theta = 6.0, p_{\mathsf{A}} = 0.8$ and $p_{\mathsf{B}} = 0.2$. Considering the induced triangle of $\{v_{12}, v_{13}, v_{14}\}$, we have the chi-square value $f(v_{12}, v_{13}, v_{14}) = 5$, which is less than the expected threshold θ. However, we cannot remove the vertices since an induced supergraph of $\{v_{11}, v_{12}, v_{13}, v_{14}\}$ has a chi-square value 8.167. On the other hand, we consider the vertex set $\{v_1, v_2, v_{10}, v_{15},\ v_{17}\}$, whose chi-square is 5 and less than θ. However, we still cannot remove the vertices since a subset $\{v_1, v_2, v_{10}, v_{15}\}$ has a chi-square value 7.563, which is larger than θ.*

Without the anti-monotonicity, we cannot immediately borrow the idea of existing algorithms for maximal clique enumeration. Specifically, once finding a maximal clique C, even $f(V_C) < \theta$, it is possible that a sub-clique C' of C satisfies $f(V_{C'}) \geq \theta$. Consequently, we cannot filter out C and need to further

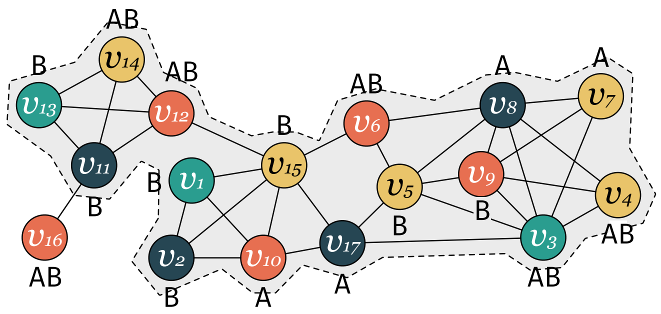


Figure 3.1: A coloring and the 3-core of the graph G

check every possible sub-clique of C with no fewer than k vertices. The method works but produces numerous intermediate results since a (k, θ)-significant clique may be involved in several maximal cliques. In addition, the number of cliques can be extremely large (up to $3^{n/3}$ in the worst case [100]), which makes the naive solution costly in big graphs. Therefore, the main challenges are how to avoid outputting the duplicated results and how to prune the search space effectively.

3.3 A Branch-and-Bound Algorithm

3.3.1 Basic Structural Graph Reduction

To handle the challenges discussed in Section 3.2.2, we give a non-trivial baseline algorithm in this section. We start by introducing several basic pruning rules, which can be easily derived from existing clique studies, like k-clique enumeration [36] and the maximum clique computation [25].

Core based Pruning. The first structural pruning rule is based on k-core, which is formally defined as follows.

Definition 3. (k-Core) *Given a graph G and an integer k, a k-core in G is a maximal connected subgraph in which the degree of every vertex is at least k [94].*

Lemma 1. *Given a graph G and a vertex u, u is contained in a k-clique only if it is contained in a $(k-1)$-core [92].*

Based on Lemma 1, all vertices not belonging to the $(k-1)$-core can be safely removed before (k,θ)-clique computation. Given an integer k, we can compute the $(k-1)$-core by iteratively removing all vertices with degree less than $k-1$. The running time is bounded by $O(m)$ [13]. An example of the 3-core in the graph G of Fig. 1.1 is marked by gray in Fig. 3.1.

Graph Coloring-based Pruning. The second rule for structural pruning utilizes graph coloring. In a graph $G(V,E)$, a coloring is an arrangement of a *color number* to each vertex u, denoted by $\mathsf{color}(u)$, such that adjacent vertices have distinct colors; i.e., $\forall(u,v) \in E, \mathsf{color}(u) \neq \mathsf{color}(v)$. Given a colored graph G and a subgraph S of G, we use $\mathsf{colors}(S)$ or $\mathsf{colors}(V_S)$ to denote all distinct color numbers in S, i.e., $\mathsf{colors}(S) = \{\mathcal{C} | \exists u \in V_S, \mathsf{color}(u) = \mathcal{C}\}$.

Lemma 2. *A graph G contains a k-clique only if there are at least k distinct colors in G, i.e., $|\mathsf{colors}(G)| \geq k$ [99].*

Based on Lemma 2, we avoid enumerating (k,θ)-significant cliques of a subgraph S if the number of distinct colors in V_S is less than k, i.e., $|\mathsf{colors}(S)| < k$. The pruning effectiveness closely depends on the coloring result. The fewer distinct color numbers, the more subgraphs can be pruned. However, it is NP-hard to color a graph with the minimum distinct color numbers [59]. Several heuristic methods have been proposed for coloring graphs in practice, and a widely used one is the greedy method following the graph degeneracy order [106]. Specifically, a vertex permutation $\{v_1, v_2, ..., v_n\}$ is a *degeneracy order* if every vertex v_i has the smallest degree in the induced subgraph of $\{v_i, v_{i+1}, ..., v_n\}$. Computing the degeneracy order takes $O(m)$ time. The coloring algorithm processes vertices in the reverse degeneracy order and greedily assigns each vertex the smallest color number that is not the same as that of any colored neighbor. The degeneracy-order-based coloring can be conducted in $O(m)$ time. A graph coloring for the graph G in Fig. 1.1 is provided in Fig. 3.1.

3.3.2 Computing Maximal Significant Cliques

The Key Idea. We propose a branch-and-bound algorithm, called SigClique, to enumerate all (k, θ)-significant cliques. Without loss of generality, we assume that the input graph is connected. Given a set of vertices R initialized as V, we aim to compute all (k, θ)-significant cliques in the induced subgraph of R. We first identify whether $G[R]$ is a (k, θ)-significant clique. If $G[R]$ is not a valid (k, θ)-significant clique, SigClique randomly picks a vertex u and divides the search space into two subspaces: (i) the subspace of all (k, θ)-cliques containing u, and (ii) the subspace of all (k, θ)-cliques excluding u. Then SigClique recursively performs the same strategy for each subspace. In each recursion, we use I to denote the set of vertices that must be included in the (k, θ)-cliques in R. I is initialized as $\emptyset$. Consequently, R consists of I and a set of candidate vertices, which can be potentially included in the (k, θ)-clique. In each recursion, we pick a vertex from the set $R \setminus I$. We immediately terminate the search if $R = I$ since R is not a (k, θ)-clique and no subspace can be further explored.

If R is a valid (k, θ)-significant clique, we check the maximality of R by adding all possible common neighbors of R. No matter whether R is maximal or not, we terminate the current search space since all following (k, θ)-significant cliques are subsets of R and can never be maximal.

The Algorithm. The pseudocode of SigClique is provided in Algorithm 1. In addition to R and I, we input k and θ to the procedure Enum. In Line 2 of Enum, we reduce the graph by computing the $(k-1)$-core based on Lemma 1. Since all vertices in I must be contained in the (k, θ)-clique, we terminate the

Algorithm 1: SigClique($G(V, E), \theta, k$)

```
1  color $G$ based on the degeneracy order;
2  Enum($V, \emptyset, \theta, k$);
1  Procedure Enum($R, I, \theta, k$) :
2      $R \leftarrow$ all vertices in $(k-1)$-core of $G[R]$;
3      if $R \cap I \neq I$ then return;
4      if $|$colors$(R)| < k$ then return;
5      if $R$ is a $(k, \theta)$-significant clique then
6          if IsMax($R, \bigcap_{v \in R} N(v), \theta$) then output $R$;
           // early termination
           return;
7      if $R = I$ then return;
8      pick a vertex $u$ from $R \setminus I$;
9      Enum($I \cup N_R(u) \cup \{u\}, I \cup \{u\}, \theta, k$);
10     Enum($R \setminus \{u\}, I, \theta, k$);
1  Procedure IsMax($R, C, \theta$) :
2      if $C = \emptyset$ then return $true$;
3      pick a vertex $u$ from $C$;
4      if $f(R \cup \{u\}) \geq \theta$ then return $false$;
5      if !IsMax($R \cup \{u\}, C \cap N(u), \theta$) then
6          return $false$;
7      if !IsMax($R, C \setminus \{u\}, \theta$) then return $false$;
8      return $true$;
```

current search space if a vertex in I is removed in the $(k-1)$-core computation in Line 3. Based on Lemma 2, we count the number of distinct colors in R and terminate the search if R cannot be a k-clique. Line 5 identifies whether R is a (k, θ)-significant clique by checking the degree of each vertex and the chi-square statistic of R. Line 6 checks the maximality of R by invoking IsMax. Line 9 searches the subspace including u, where $N_R(u)$ represents the neighbors of u in R. Line 10 searches the subspace excluding u.

In the procedure IsMax, R is the set of vertices to be checked, and C is all common neighbors of vertices in R. In Line 2, $C = \emptyset$ means no candidate vertex can be added to R, and R must be maximal. The maximality search is also divided into two subspaces. Line 5 identifies whether $R \cup \{u\}$ is maximal. Line 7 checks whether R is maximal when excluding u from the candidate set.

Theorem 1. *Algorithm 1 correctly computes all maximal (k, θ)-significant cliques in the graph G.*

Proof. We first prove the correctness. Based on Lemma 1, line 2 correctly prunes the vertices that cannot be contained in a k-clique. Lines 3-4 judge whether R contains k-cliques based on Lemma 2. Line 5 judges if R is a (θ, k)-significant clique and line 5 ensures the maximality. Thus, the results are correct.

Next, we prove the completeness. Line 8 picks each vertex, and lines 9-10 search both the spaces with or without the vertex. Thus, it searches all the possible space, and the theorem is proved. □

3.4 Statistical Graph Reduction

Even though Algorithm 1 successfully computes (k, θ)-cliques without any redundancy, the pruning effectiveness is still limited, especially in large graphs and given a small size constraint. In this section, we study several pruning strategies regarding the chi-square statistic. Section 3.4.1 formulates a new cohesive subgraph model called (k, θ)-significant core. Section 3.4.2 embeds the concept of graph coloring to the (k, θ)-significant core, which improves the effectiveness of both structural pruning and statistical pruning. Section 3.4.3 extends the idea of (k, θ)-significant core to prune edges.

3.4.1 Pruning via Significant Core

To support the statistical pruning over the graph, we first give a key theorem as follows.

Theorem 2. *Given a set of labeled vertices V, a probability distribution P, two arbitrary subsets V_1 and V_2 with $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$, we have $f(V_1) + f(V_2) \geq f(V)$.*

Proof. We prove the theorem by showing $f = f(V_1) + f(V_2) - f(V) \geq 0$. We expand f as follows based on Eq. 3.1.

$$f = \sum_{i=1}^{l} \frac{y_{1i}^2}{y_1 p_i} - y_1 + \sum_{i=1}^{l} \frac{y_{2i}^2}{y_2 p_i} - y_2 - \sum_{i=1}^{l} \frac{y_i^2}{y p_i} + y$$

Given that $y_1 + y_2 = y$ and $y_{1i} + y_{2i} = y_i$, we transform the formula as follows.

$$\begin{aligned} f &= \sum_{i=1}^{l} \frac{1}{y_1 y_2 y p_i} \cdot (y_{1i}^2 y_2^2 + y_{2i}^2 y_1^2 - 2 y_{1i} y_{2i} y_1 y_2) \\ &= \sum_{i=1}^{l} \frac{(y_{1i} y_2 - y_{2i} y_1)^2}{y_1 y_2 y p_i} \geq 0 \end{aligned}$$

We have $f \geq 0$, and the proof is completed. □

Based on Theorem 2, we formulate a new cohesive subgraph model, called (k, θ)-significant core, to prune unpromising vertices in the problem of (k, θ)-clique enumeration. Related definitions are given as follows.

Definition 4. (NEIGHBORHOOD SIGNIFICANCE) *Given a vertex u, the neighborhood significance of u, denoted by $f_n(u)$, is the sum of the significance of u and all its neighbors, i.e., $f_n(u) = f(u) + \sum_{v \in N(u)} f(v)$.*

Definition 5. (SIGNIFICANT CORE) *Given a graph G, an integer k and a positive real value θ, (k, θ)-Significant Core (SC for short) is a maximal connected subgraph of G in which every vertex u satisfies (i) $deg(u) \geq k$, and (ii) $f_n(u) \geq \theta$.*

Based on Definition 5, we can prune vertices supported by the following lemma.

Lemma 3. *A maximal (k, θ)-significant clique must be contained in a $(k-1, \theta)$-significant core.*

Proof. We prove Lemma 3 by contradiction. Assume that we have a (k, θ)-significant clique C which is not contained in any $(k-1, \theta)$-significant core. Because C is a (k, θ)-significant clique, for each $u \in V_C$, we have $f_n(u) \geq f(C) \geq \theta$, which satisfies Definition 5 (ii). As C is not contained in any $(k-1, \theta)$-significant core, there must exist a vertex $u \in V_C$ whose degree is less than $(k-1)$, i.e. $\exists u \in V_C$, $deg(u) < (k-1)$. Also, as we know that C is a (k, θ)-significant clique, then for each $u \in V_C$, we have $deg(u) \geq (k-1)$. The contradiction exists. □

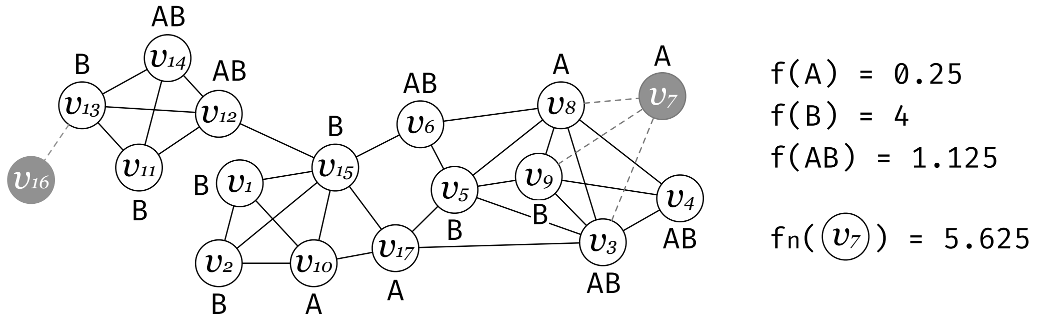


Figure 3.2: Pruning G via $(3, 6)$-significant core

Example 4. *Fig. 3.2 gives an example of the pruning result via the $(3, 6)$-significant core. The chi-square statistics of several required label sets are given on the right of Fig. 3.2. After computing the $(3, 6)$-significant core, the vertices v_7 and v_{16} are removed. Specifically, v_{16} is removed since $deg(v_{16}) < 3$. For the vertex v_7, we have $f_n(v_7) = f(v_7) + f(v_3) + f(v_8) + f(v_9) = 5.625 < 6$. All the remaining vertices have degrees no less than 3 and neighborhood significance no less than 6. For example, the neighborhood significance of the vertex v_4 is 6.5.*

Given a graph G, we can compute all (k, θ)-significant cores by a method similar to k-core computation. We recursively remove a vertex with degree less than k or neighborhood significance less than θ. If a neighbor v of u is removed, we update $f_n(u)$ to $f_n(u) - f(v)$. The time complexity is analyzed as follows.

Theorem 3. *Computing all (k, θ)-cores takes $O(m)$ time.*

Proof. For each vertex u, it takes $O(deg(u))$ time to initialize $f_n(u)$. We remove the vertex u if $deg(u) < k$ or $f_n(u) < \theta$. Then, we update $deg(v)$ and $f_n(v)$ for each neighbor v of u, which takes constant time. Therefore, the overall time complexity is $O(m)$. □

3.4.2 Pruning via Colorful Significant Core

In this subsection, we further improve the pruning effectiveness by embedding the concept of graph coloring in (k, θ)-significant core. Compared with the significant core model, we strictly prune more vertices in terms of both graph structure and label statistics. From the structural perspective, we can combine the degree-based bound (Lemma 1) and the coloring-based bound (Lemma 2) as follows.

Definition 6. (Colorful Degree) *The colorful degree of a vertex u, denoted by $deg_c(u)$, is the number of distinct colors in $N(u)$, i.e., $deg_c(u) = |\mathsf{colors}(N(u))|$.*

Lemma 4. *For any (k, θ)-significant clique C, we have $deg_c(u) \geq k-1$ for every $u \in V_C$.*

Proof. We prove Lemma 4 by contradiction. We assume that there exists a vertex $u \in V_C$ whose degree is less than $k - 1$. Then, we have $|\mathsf{colors}(N(u))| < k - 1$. According to Lemma 2, $N(u)$ cannot contain a $(k - 1)$-clique, thus, $\{u\} \cup N(u)$ cannot contain a k-clique. However, as C is a (k, θ)-significant clique, for every vertex $u \in V_C$, we have $\{u\} \cup N(u)$ contains a k-clique. That is a contradiction. $\square$

Example 5. *We give an example in the colored graph G of Fig. 3.1. Considering the vertex v_6, the degree of v_6 is 3. However, the colorful degree of v_6 is only 2, since there are only two distinct colors in the neighborhood of v_6. According to Lemma 4, v_6 cannot be in any $(4, \theta)$-significant clique.*

From the statistical perspective, we combine the concepts of neighborhood significance (Definition 4) and the coloring-based bound (Lemma 2) as follows.

Definition 7. (Colorful Neighborhood Significance) *The colorful neighborhood significance of a vertex u, denoted by $f_{cn}(u)$, is the sum of significance of u and the maximum vertex significance for each color in $N(u)$, i.e.,*

$$f_{cn}(u) = f(u) + \sum_{\mathcal{C} \in \mathsf{colors}(N(u))} \max_{v \in N(u) | \mathsf{color}(v) = \mathcal{C}} f(v).$$

Lemma 5. *Given a (k, θ)-significant clique C, we have $f_{cn}(u) \geq \theta$ for every $u \in V_C$.*

Proof. Given a (k, θ)-significant clique C, we prove $f_{cn}(u) \geq \theta$ for every $u \in V_C$. Because C is a (k, θ)-significant clique, for each $u \in V_C$, we have $f(u) + \sum_{v \in V_C, v \neq u} f(v) \geq f(C) \geq \theta$. Given an arbitrary vertex $u \in V_C$, every vertex $v \in C$ $(v \neq u)$ has a distinct $\mathsf{color}(v)$. For each such a vertex v, we assume there is a vertex $v' \in N(u)$ that satisfies $\mathsf{color}(v) = \mathsf{color}(v')$ and v' has the maximum vertex significance over all the vertices in $N(u)$ who have the same $\mathsf{color}(v)$ (v' can be the same vertex as v). Obviously, we have $\sum_{v'} f(v') \geq \sum_{v} f(v)$. In terms of the number of colors, we also have that $|\mathsf{colors}(N(u))| \geq |\mathsf{colors}(V_C \backslash \{u\})|$. Thus, we can get $f(u) + \sum_{\mathcal{C} \in \mathsf{colors}(N(u))} \max_{v \in N(u) | \mathsf{color}(v) = \mathcal{C}} f(v) \geq f(u) + \sum_{v'} f(v') \geq f(u) + \sum_{v} f(v) \geq \theta$, i.e., $f_{cn}(u) \geq \theta$. $\square$

Example 6. *We give an example to explain Definition 7. Considering the vertex v_{17} in Fig. 3.1, we have three distinct colors in $N(v_{17})$. For the yellow color, we have two vertices v_5 and v_{15} in $N(v_{17})$. The labels of them are the same, and the largest statistic for the yellow color is $f(\mathsf{B}) = 4$. Each other color in $N(v_{17})$ has only one vertex. As a result, we have $f_{cn}(v_{17}) = f(v_{17}) + f(v_5) + f(v_{10}) + f(v_3) = 5.625$. By contrast, the neighborhood significance of v_{17} is $f_n(v_{17}) = 9.625$.*

Based on Definition 6 and Definition 7, we give an extended version of (k, θ)-significant core.

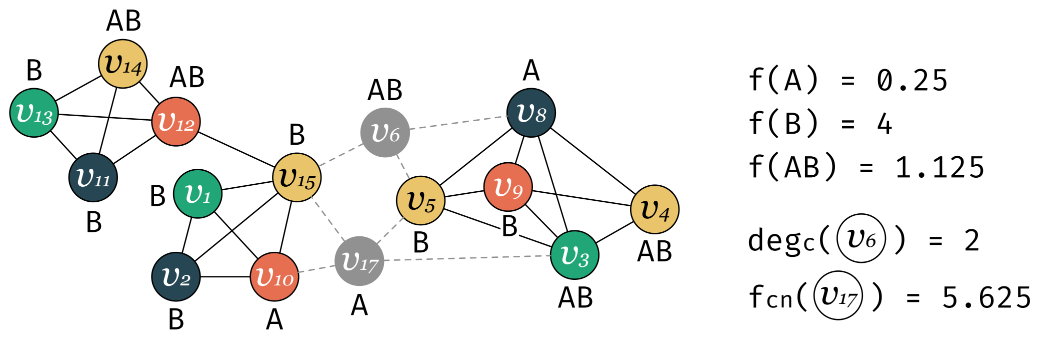


Figure 3.3: Pruning G via colorful $(3, 6)$-significant core

Definition 8. (Colorful Significant Core) *Given a colored graph G, an integer k and a positive real value θ, a Colorful (k, θ)-Significant Core (CSC for short) is a maximal connected subgraph of G in which every vertex u satisfies (i) $deg_c(u) \geq k$, and (ii) $f_{cn}(u) \geq \theta$.*

Example 7. *We give an example of the $(3, 6)$-CSC in Fig. 3.3. Two vertices v_6 and v_{17} are removed from the graph. Based on Lemma 4 and Lemma 5, the two vertices can never be in any $(4, 6)$-clique.*

The following theorem shows that the pruning effectiveness of (k, θ)-CSC is guaranteed to be stronger than that of (k, θ)-SC.

Theorem 4. *A colorful (k,θ)-significant core must be contained in a (k,θ)-significant core.*

Proof. Given a colorful (k,θ)-significant core C, for each vertex $u \in V_C$, we have $deg_c(u) \geq k$ and $f_{cn}(u) \geq \theta$. Obviously, $deg(u) \geq deg_c(u) \geq k$, and $f(u) \geq f_{cn}(u) \geq \theta$, thus, C is contained in a (k,θ)-significant core. □

Algorithm 2: CSC(G,θ,k)

```
// $f_n(u) = f(u) + \sum_{v \in N(u)} f(v)$, $f_{cn}(u)$ is colorful $f_n(u)$, $deg_c(u)$
   is colorful $deg(u)$
1  $Q \leftarrow$ initialized an empty queue;
2  foreach $u \in V$ do
3      compute $f_{cn}(u)$, and $deg_c(u)$;
4      if invalid$(u)$ then $Q.push(u)$;
5  while $Q \neq \emptyset$ do
6      $u \leftarrow Q.pop()$;
7      foreach $v \in N(u)$ do
8          if invalid$(v)$ then continue;
9          update $deg_c(v)$;
10         update $f_{cn}(v)$;
11         if invalid$(v)$ then $Q.push(v)$;
12     remove $u$ and all connected edges;
1  Procedure invalid$(u)$ :
2      if $deg_c(u) < k$ then return $true$;
3      if $f_{cn}(u) < \theta$ then return $true$;
4      return $false$;
```

Colorful SC Computation. Algorithm 2 presents the pseudocode for colorful (k,θ)-significant core computation. The strategy is similar to computing k-cores. Line 3 initializes the colorful degree and the colorful neighborhood significance for each vertex u. Line 4 adds u to the queue if it cannot be in the (k,θ)-CSC. The procedure invalid() is used to identify the validity of a vertex according to Lemma 4 and Lemma 5. After popping an invalid vertex u in Line 6, we update neighbors of u if necessary. Let the color of u be $\mathcal{C}$. In Line 9, we decrease $deg_c(u)$ by one if no vertex has the color $\mathcal{C}$ in $N(v)$ after removing u. In Line 10, let $N_{\mathcal{C}}(v)$ be the set of all neighbors of v with the color $\mathcal{C}$. If u has the largest chi-square statistic in $N_{\mathcal{C}}(v)$, let s be the second largest chi-square statistic in $N_{\mathcal{C}}(v)$. We set $s = 0$ if there exists only u in $N_{\mathcal{C}}(v)$. We update $f_{cn}(v)$ to $f_{cn}(v) - f(u) + s$. We do not change $f_{cn}(v)$ if u is not the vertex with the largest chi-square statistic in $N_{\mathcal{C}}(v)$.

Implementation and Complexity Analysis. The key step in Algorithm 2 is to efficiently maintain the colorful degree (Line 9) and colorful neighborhood significance (Line 10) of each vertex. For the colorful degree of a vertex u, we use a hash table to store the number of vertices for each color in $\mathsf{colors}(N(u))$. Initializing the hash table for u takes $O(deg(u))$ time, and locating the value for a specific color number takes constant average and amortized time.

Next, we discuss the implementation to update the colorful neighborhood significance. For each color $\mathcal{C} \in \mathsf{colors}(N(v))$, we sort all vertices with the color $\mathcal{C}$ in a non-increasing order of their chi-square statistics, which takes $O(deg(v) \log deg(v))$ time. Given an invalid neighbor u, we also use a hash table to locate u in the sorted list and mark the position of u as empty. If u is not the first vertex in the corresponding color, $f_{cn}(v)$ does not need to be updated. If u is the first vertex, we iteratively search the following positions with the same color until finding a nonempty vertex w. We update $f_{cn}(v)$ to $f_{cn}(v) - f(u) + f(w)$, where $f(w) = 0$ if w does not exist. Note that it may take several movements to find w. However, the total number of operations for each vertex v in Line 10 of Algorithm 2 is bounded by $O(deg(v))$ since we always move forward to locate the second largest chi-square statistic.

Example 8. *In Fig. 3.4, we give an example of the data structure to maintain $f_{cn}(v_{15})$ in the graph of Fig. 3.1. v_{15} has six neighbors with three distinct colors.*

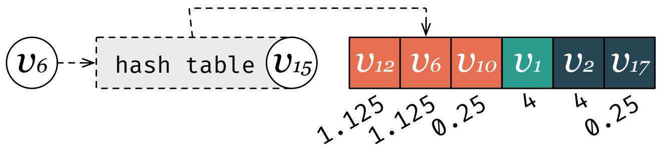


Figure 3.4: The data structure for v_{15} to maintain $f_{cn}(v_{15})$

$f_{cn}(v_{15})$ is initialized as the sum of $f(v_{15})$ and the chi-square statistics of all the first vertices of distinct colors. Assume that the vertex v_6 is removed, and we need to update $f_{cn}(v_{15})$. We first use the hash function to locate the position of v_6. By checking the previous vertex, v_6 is not the first vertex in the table with the same color. Therefore, we set the position of v_6 as empty and do not change $f_{cn}(v_{15})$.

The following theorem presents the theoretical analysis of Algorithm 2.

Theorem 5. *The time complexity of Algorithm 2 is $O(m \cdot \log deg_{max})$, where deg_{max} is the maximum degree in the graph. The space complexity of Algorithm 2 is $O(m)$.*

Proof. We first prove the time complexity. Line 1 takes $O(1)$ time. We need $O(deg(u))$ time to compute $f_{cn}(u)$ and $deg_c(u)$ in line 3. Thus, Lines 2–4 take $O(m)$ time. With the data structure above, lines 9–10 cost $O(deg(v))$ time for each vertex v, while the initialization costs $O(deg(v) \log deg(v))$ time. Thus, the time complexity of lines 5–12 is $O(m \cdot \log deg(v)$ which is also the time complexity of Algorithm 2.

As for each vertex u, we only store $O(deg(u))$ information for the data structure, the space complexity is $O(m)$. □

3.4.3 Pruning via Significant Truss

Recall that Section 3.4.2 extends (k, θ)-SC and strengthens the vertex pruning rule. In this subsection, we will strengthen the (k, θ)-SC from the perspective of edge reduction. In other words, we consider whether two connected vertices can be in the same (k, θ)-clique or not. The lemma for the structural edge reduction is given as follows, which further applies the k-core concept to the ego-network of each vertex.

Lemma 6. *Given a k-clique C and an arbitrary vertex $u \in V_C$, for every vertex $v \in N_C(u)$, the vertex v is contained in a $(k-2)$-core of $G[N(u)]$ [67].*

Theorem 6. *Two vertices u and v cannot be contained in the same (k, θ)-clique if v is not in a $(k-2)$-core of the neighborhood subgraph $G[N(u)]$.*

Proof. We can easily prove Theorem 6 by contradiction. Suppose that vertices u and v are contained in the same (k, θ)-clique C, i.e., $u, v \in C$, according to Lemma 6, the vertex v is contained in a $(k-2)$-core of $G[N(u)]$. However, as v is not in a $(k-2)$-core of the neighborhood subgraph $G[N(u)]$, the contradiction exists, thus the vertices u and v cannot be contained in the same (k, θ)-clique. □

Based on Theorem 6, if the vertex v is not in a $(k-2)$-core of $G[N(u)]$, we remove the edge (u, v), which guarantees that u and v cannot be enumerated in the same clique. Given the $O(m)$ time to compute the k-core, a straightforward method to recursively remove all unpromising edges in Theorem 6 takes $O(m \cdot h_{max})$ time, where h_{max} is the largest number of edges in neighborhood subgraphs. To remedy the cost, we give the following lemma.

Lemma 7. *Given a vertex u and a vertex v in $N(u)$, the degree of v in $G[N(u)]$ is equivalent to the number of triangles that contain the edge (u, v).*

Proof. Given the vertex $u \in G$ and $v \in N(u)$, the degree of v in $G[N(u)]$ is $deg_{G[N(u)]}(v) = |\{v'|(v', v) \in G[N(u)]\}|$. As both $v, v' \in N(u)$, for each v', the edges (u, v'), (v, v'), and (u, v) constitute a triangle. Thus, the number of such triangles made by (u, v) and v' equals the degree of v in $G[N(u)]$. □

Based on Lemma 7, removing all the edges not satisfying the condition in Theorem 6 is equivalent to computing the k-truss in graph, which is formally defined as follows.

Definition 9. (k-TRUSS) *Given a graph G and an integer k, a k-truss is a maximal connected subgraph in which every edge is contained in at least $k-2$ triangles [105].*

Let S be the maximal subgraph such that for each pair of connected vertices u and v in V_S, v is in $(k-2)$-core of $G[N(u)]$. Then, S is a k-truss of G.

Example 9. *We consider the graph in Fig. 3.5. Given the vertex v_{12} and $k = 4$, the degree of v_{15} in the neighborhood subgraph of v_{12} is 0. In other words, there is no triangle containing the edge (v_{12}, v_{15}). Therefore, v_{12} and v_{15} cannot be in the same $(4, \theta)$-clique according to Theorem 6.*

Similar to the concept of neighborhood significance, we define the support significance for the statistical edge reduction.

Definition 10. (SUPPORT SIGNIFICANCE) *The support significance of an edge (u, v), denoted by $f_n(u, v)$, is the sum of $f(u, v)$ and the chi-square statistics of all common neighbors of u and v, i.e., $f_n(u, v) = f(u, v) + \sum_{w \in N(u) \cap N(v)} f(w)$.*

Lemma 8. *Given a (k, θ)-significant clique C, we have $f_n(u, v) \geq \theta$ for every edge $(u, v) \in E_C$.*

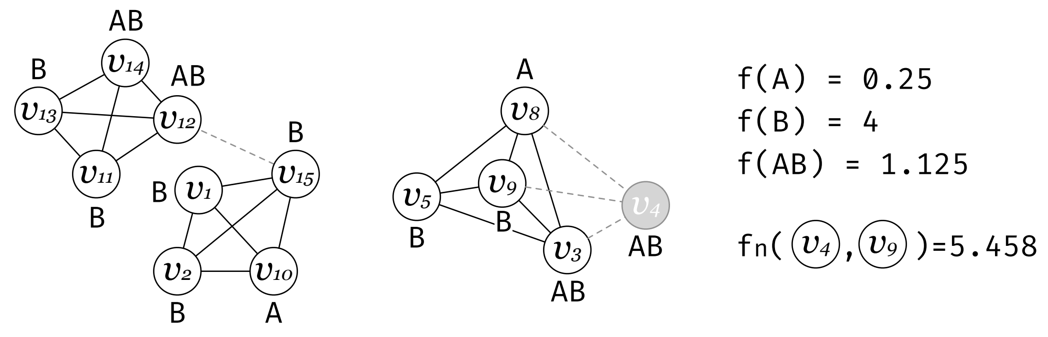


Figure 3.5: Pruning G via $(4, 6)$-significant truss

Proof. We prove Lemma 8 by contradiction. Given a (k, θ)-significant clique C, suppose we have an edge $(u, v) \in E_C$ and $f_n(u, v) < \theta$. As $u, v \in V_C$, all the other vertices in V_C are common neighbors of both u and v, i.e., $C\backslash\{u, v\} \subseteq N(u) \cap N(v)$. Hence, we have $f(u, v)+\sum_{w\in C\backslash\{u,v\}} f(w) \leq f(u, v)+\sum_{w\in N(u)\cap N(v)} f(w) = f_n(u, v)$. As $f(C) \leq \sum_{w\in C\backslash\{u,v\}} f(w)$ and $f_n(u, v) < \theta$, we have $f(C) < \theta$. This contradicts with the (k, θ)-significant clique C. □

Example 10. *We consider the edge* (v_4, v_9) *in the graph of Fig. 3.5. There are two common neighbors —* v_3 *and* v_8*. We have* $f_n(v_4, v_9) = f(v_4, v_9) + 1.125 + 0.25 = 5.458$*. Based on Lemma 8, the edge* (v_4, v_9) *cannot be in any* (k, θ)*-significant clique if* $\theta > 5.458$*.*

The support of an edge (u, v), denoted by $sup(u, v)$, is the number of triangles that contains (u, v). Based on Definition 10 and Definition 9, we define a new statistical cohesive subgraph model as follows.

Definition 11. (Significant Truss) *Given a graph G, an integer k and a positive real value θ, (k, θ)-Significant Truss (ST for short) is a maximal connected subgraph of G in which every edge (u, v) satisfies (i) $sup(u, v) \geq k - 2$, and (ii) $f_n(u, v) \geq \theta$.*

Example 11. *An example of the pruning result by applying* $(4, 6)$*-significant truss is given in Fig. 3.5. The edge* (v_{12}, v_{15}) *and all the edges connected to* v_4 *are removed.*

The following theorem shows that (k, θ)-*ST* is guaranteed to have stronger pruning effectiveness than (k, θ)-*SC*.

Theorem 7. *A* (k, θ)*-significant truss must be contained in a* $(k-1, \theta)$*-significant core.*

Proof. Given a (k, θ)-significant truss C and an arbitrary vertex $u \in C$, for each neighbor $v \in N(u) \cap C$, we have $f_n(u, v) \geq \theta$ (Lemma 8). Thus, $f_n(u) \geq \theta$. Also, we have $sup(u, v) \geq k - 2$ by which we can get $deg(u) \geq k - 1$. So, C is contained in a $(k - 1, \theta)$-significant core. □

ST Computation. We give the pseudocode for computing (k, θ)-significant truss in Algorithm 3. The idea is similar to that of truss decomposition [105]. The complexity of Algorithm 3 is summarized below.

Theorem 8. *The time complexity and space complexity of Algorithm 3 are* $O(\alpha m)$ *and* $O(m)$*, respectively.*

Proof. Lines 1–5 initialize the support and the support significance of each edge. The time complexity of enumerating all triangles is $O(\sum_{(u,v) \in E} \min(deg(u), deg(v)))$, i.e. $O(\alpha \cdot m)$, where $\alpha (\alpha < m^{0.5})$ is the graph arboricity and equals the minimum number of forests to cover all edges in the graph [31]. Lines 6–8 takes $O(m)$ time. We update necessary edges after (u, v) is removed in Line 10. We use a hash set to maintain all neighbors of each vertex. As a result, the time complexity for Lines 9–20 is $O(\alpha \cdot m)$. The total time complexity is $O(\alpha \cdot m)$. Note that we never store any triangle during the algorithm, thus the space complexity is $O(m)$. □

Algorithm 3: $\mathsf{ST}(G, \theta, k)$

```
1  $sup(u,v) \leftarrow 0, f_n(u,v) \leftarrow f(u,v)$;
2  foreach enumerated $\triangle_{u,v,w} \in G$ do
3      $sup(u,v) \leftarrow sup(u,v) + 1$;
4      $f_n(u,v) \leftarrow f_n(u,v) + f(w)$;
5      repeat two lines above for $(u,w)$ and $(v,w)$;
6  $Q \leftarrow$ initialized an empty queue;
7  foreach $(u,v) \in E_G : \mathsf{invalid}(u,v,\theta,k-2)$ do
8      $Q.push((u,v))$;
9  while $Q \neq \emptyset$ do
10     $(u,v) \leftarrow Q.pop()$;
11     if $deg(u) > deg(v)$ then swap $u$ and $v$;
12     foreach $w \in N(u)$ do
13         if $w \in N(v)$ then
14             if $\mathsf{invalid}(u,w,\theta,k)$ then continue;
15             $sup(u,w) \leftarrow sup(u,w) - 1$;
16             $f_n(u,w) \leftarrow f_n(u,w) - f(v)$;
17             if $\mathsf{invalid}(u,w,\theta,k)$ then
18                 $Q.push((u,w))$;
19             repeat five lines above for $(v,w)$;
20     remove $(u,v)$ from $E_G$;
21 remove all isolated vertices from $V_G$;
1  Procedure $\mathsf{invalid}(u,v,\theta,k)$ :
2      if $sup(u,v) < k-2$ then return $true$;
3      if $f_n(u,v) < \theta$ then return $true$;
4      return $false$;
```

Remark: Similar to Section 3.4.2, we can further embed the color-based bound in the concept of (k,θ)-significant truss and further improve the pruning effectiveness. However, maintaining such colorful supports and statistics for each edge incurs large space usage and processing time. Therefore, we only compute (k,θ)-significant truss in our final algorithm.

3.5 The Final Algorithm

Algorithm 4: RCSC(G, θ, k)

```
// $f_n(u) = f(u) + \sum_{v \in N(u)} f(v)$, $f_{cn}(u)$ is colorful $f_n(u)$, $deg_c(u)$
    is colorful $deg(u)$
1  $Q \leftarrow$ initialized an empty queue;
2  foreach $u \in V$ do
3      compute $f_n(u)$, $f_{cn}(u)$, and $deg_c(u)$;
4      if invalid$(u)$ then $Q.push(u)$;
5  while $Q \neq \emptyset$ do
6      $u \leftarrow Q.pop()$;
7      foreach $v \in N(u)$ do
8          if invalid$(v)$ then continue;
9          update $deg_c(v)$;
10         update $f_n(v)$;
11         if invalid$(v)$ then $Q.push(v)$;
12     remove $u$ and all connected edges;
1  Procedure invalid$(u)$ :
2      if $deg_c(u) < k$ then return $true$;
3      if $f_n(u) < \theta$ then return $true$;
4      if $f_{cn}(u) < \theta$ then return $true$;
5      return $false$;
```

Our final algorithm to enumerate maximal (k, θ)-significant cliques is given in Algorithm 5. We reduce the input graph G by computing the $(k-1, \theta)$-CSC and the (k, θ)-ST in Line 2 and Line 3, respectively. The order of CSC and ST does not affect the result. However, as CSC requires less computation than ST, we apply CSC first. Different from SigClique, the recursive enumeration procedure Enum* in SigClique* computes relaxed colorful $(k-1, \theta)$-significant cores ($RCSC$) in Line 3 instead of $(k-1)$-cores (Line 2) and color numbers (Line 4) of Enum.

Relaxed CSC. Given a graph G, an integer k and a real value θ, let S and S' be the vertices in (k, θ)-SC and (k, θ)-CSC, respectively. An induced subgraph of S'' is called a relaxed colorful (k, θ)-significant core if $S' \subseteq S'' \subseteq S$. The motivation of computing $RCSC$s is to hold the same $O(m)$ time complexity as k-core computation but bring stronger pruning effectiveness.

Recall that in Algorithm 2 to compute (k, θ)-CSCs, the dominating cost is to sort the neighbors of each vertex, which is used to efficiently update the colorful neighborhood significance (Fig. 3.4). To reduce the time complexity from $O(m \cdot \log deg_{max})$ to $O(m)$, we do not update the colorful neighborhood significance f_{cn} for each vertex in the iteration but just remove all vertices u with $f_{cn}(u) < k$ in the first round. Specifically, we modify Algorithm 2 to compute $RCSC$s in the following three parts. First, in Line 3, we additionally compute $f_n(u)$. Second, we replace Line 10 with "update $f_n(u)$". Third, we additionally check whether $f_n(u) < \theta$ in the procedure invalid. As a result, we derive a subgraph which satisfies the conditions of SC but possibly contains some vertices u with $f_{cn}(u) < \theta$. We show the detailed pseudocode of RCSC in Algorithm 4. Without the sorted data structure, the time complexity of RCSC reduces to $O(m)$, and the pruning effectiveness is at least the same as Line 2 and Line 4 in Enum.

3.6 Experiments

In this section, we evaluate the efficiency and effectiveness of our proposed algorithms. We implement our algorithms with four versions according to the pruning techniques, namely SigClique, SigClique-SC, SigClique-CSC, and SigClique*. SigClique is the Algorithm 1 with basic k-core and graph coloring reduction techniques. SigClique-SC utilizes the significant core technique to prune the unpromising vertices. SigClique-CSC is the algorithm with the colorful significant core pruning rule. SigClique* contains all the pruning techniques. We also im-

Algorithm 5: SigClique$^*(G(V,E),\theta,k)$

```
1  color $G$ based on the degeneracy order;
2  CSC$(G,\theta,k-1)$;
3  ST$(G,\theta,k)$;
4  Enum$^*(V,\emptyset,\theta,k)$;
1  Procedure Enum$^*(R,I,\theta,k)$ :
2      RCSC$(R,\theta,k-1)$;
3      if $R \cap I \neq I$ then return;
4      if $R$ is a $(k,\theta)$-significant clique then
5          if IsMax$(R,\bigcap_{v\in R} N(v),\theta)$ then output $R$;
6          return;
7      if $R = I$ then return;
8      pick a vertex $u$ from $R \setminus I$;
9      Enum$^*(I \cup N_R(u) \cup \{u\}, I \cup \{u\},\theta,k)$;
10     Enum$^*(R \setminus \{u\}, I,\theta,k)$;
```

plement kClist, a variation of the k-clique listing algorithm[36]. We add our significance computation and maximality testing to make it able to enumerate sigcliques. All the algorithms are implemented in C++, and all the experiments are conducted on a Linux machine with 3.7 GHz Xeon CPU and 64GB memory.

Datasets. We evaluate our algorithms with seven real-world datasets as showed in Table 3.2. EUmail is a communication network that contains the email sending and receiving information from a large European research institution. In EUmail, each vertex denotes an email address, and the edge between two vertices represents that at least one mail was sent between them. Amazon is a co-purchasing network from Amazon.com, where the vertices represent products, and the edges between them mean the two products are frequently purchased together. DBLP is a co-authorship network in which each vertex represents an author, and there will be an edge between two vertices if they have co-authored at least three papers. Both Youtube and Hyves are social networks. CiteSeer and Patent are citation networks. To generate the labels of the vertices on each graph, we ran-

Table 3.2: Network statistics (deg_{max} is the maximum degree, c is the core value)

Dataset	n	m	deg_{max}	c
EUmail	265,214	420,045	7,636	37
Amazon	334,863	925,872	549	6
CiteSeer	384,413	1,751,463	1739	15
DBLP	556,533	1,311,766	373	25
Youtube	1,134,890	2,987,624	28,754	51
Hyves	1,402,673	2,777,419	31,883	39
Patent	3,774,768	16,518,948	793	64

domly assign each vertex with 1-3 labels chosen from a 4-label alphabet. EUmail, Amazon, Youtube, and Patent are downloaded from the Stanford Large Network Dataset Collection [1]. CiteSeer and Hyves are downloaded from the website of the Koblenz Network Collection [2]. DBLP is extracted from the computer science bibliography DBLP [3].

Parameters. Our algorithms have two parameters, namely θ and k. The parameter θ is selected from the set $\{8, 10, 12, 14\}$ with the default value $\theta = 10$, and the parameter k is ranging from $\{3, 5, 7, 9\}$ with a default value $k = 5$. Unless otherwise specified, a parameter is set as the default value when we vary the other one. Regarding the efficiency evaluation, we prepare four labels $\Sigma = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$. We randomly and independently assign a set of labels in Σ for each vertex in the graph. We assign the same proportion for all labels, and the expected label distribution is $P = \{0.25, 0.25, 0.25, 0.25\}$.

3.6.1 Efficiency Evaluation

Exp-1: Overall Efficiency of maximal significant clique enumeration. Table 3.3 compares the running time of SigClique* with SigClique and kClist under

[1]`https://snap.stanford.edu/`
[2]`http://konect.cc/networks/`
[3]`https://dblp.uni-trier.de/`

Table 3.3: Overall running time of enumerating maximal significant cliques on all datasets and the number of corresponding maximal significant cliques

Dataset	SigClique	SigClique*	kClist	# sigcliques
EUmail	8s	**7s**	13s	1198
Amazon	362s	2s	1s	31
CiteSeer	340s	8s	2s	137
DBLP	224s	**74s**	204s	1388
Youtube	469s	**86s**	138s	6036
Hyves	1605s	**219s**	525s	998
Patent	-	**151s**	177s	1790

the default parameter setting. On the right side, we also show the number of corresponding maximal significant cliques. We can see that SigClique* is the fastest on most datasets. Compared to SigClique, the speedup in EUmail is small (from 8 seconds to 7 seconds) for the following reasons. First, EUmail has the smallest size in all datasets, and the improvement room is limited. Second, the average degree in EUmail is relatively small, which means the basic k-core pruning has been very effective. On Patent, SigClique* takes about 151 seconds while SigClique cannot finish in 5 hours. kClist is faster on Amazon and CiteSeer as both datasets have small core values and numbers of cliques, which means the listing can be fast. On other datasets with larger core values and numbers of cliques, our SigClique* is much faster.

Exp-2: Running time of different pruning strategies with varying parameters. We evaluate the running time of our algorithms by varying the input parameter k and θ. Given the default $k = 5$, we vary θ from 8 to 14 in Fig. 3.6 (a) and Fig. 3.6 (b). We only have the line for SigClique* on Patent since the algorithms cannot finish in 5 hours under other settings. We can see a downward trend for the algorithms on both Youtube and Patent when θ increases. On Patent, the running time of SigClique* is about 1.25 hours when $\theta = 8$ and drops to 25 seconds when $\theta = 14$. Fig. 3.6 (b) also reveals the effectiveness of our several pruning techniques. SigClique is the slowest algorithm under all θ values since it only uses the basic k-core and the graph coloring pruning rules to reduce the search space. SigClique-SC starts to consider statistical pruning, which is a little faster than SigClique. When θ is small, the running time of SigClique-SC is similar to SigClique, because the degree pruning dominates in the algorithm. The gap between SigClique-SC and SigClique proves the effectiveness of the significant core model. SigClique-CSC is the second fastest algorithm in the figure since it adopts a stronger pruning model than SigClique-SC. The gap between SigClique-CSC and SigClique-SC proves the effectiveness of the colorful significant core model. SigClique* contains all optimizations, which is the fastest. The gap between SigClique* and SigClique-CSC proves the effectiveness of the significant truss model.

Given the default $\theta = 10$, we vary k from 3 to 9 in Fig. 3.6 (c) and Fig. 3.6 (d). The results are similar to those when varying θ. For example, on Patent, SigClique* takes 175 seconds and 12 seconds when $k = 3$ and $k = 9$, respectively. In Fig. 3.6 (d), the running time of SigClique is almost the same as that of SigClique-SC when k is large, because θ does not change and the degree pruning is the dominating rule. Fig. 3.7 shows the running time of SigClique* on Patent and Youtube while varying the θ and k .

Exp-3: Number of vertices after reduction. To further evaluate the effectiveness of our pruning techniques, we report the number of vertices after performing the reduction rules (e.g., core, significant core, colorful significant core, and significant truss) and before invoking the Enum procedure. The result can be found in Fig. 3.8, where Fig. 3.8 (a) and (c) show the results when varying θ. Fig. 3.8 (b) and (d) show the results when varying k. When varying θ, we can see that the number of vertices for SigClique never changes since only the structural pruning is performed in SigClique. Note that even though the significant truss is an edge pruning model, many vertices become isolated and are removed during the edge removal. The results show that our final pruning techniques are extremely effective. For example, in Fig. 3.8 (a), the number of vertices is about 194 thousand when $\theta = 8$ but drops to only about 17 thousand when $\theta = 14$. We do not show the result for Amazon when $k = 9$ as there remain no vertices under that setting.

Exp-4: Number of maximal significant cliques. We report the number of maximal significant cliques under different parameters in Fig. 3.9. The number of results gradually decreases when θ increases from 8 to 14 on all datasets. For example, in Fig. 3.9 (a), we have over 12 thousand maximal $(5, 8)$-significant cliques and 154 maximal $(5, 14)$-significant cliques on Patent. In Fig. 3.9 (b), we see a sharp drop from $k = 7$ to $k = 9$. That is because there exist a larger number of $(7, 10)$-cliques or $(8, 10)$-cliques on Patent.

On Amazon, we do not show the results when k is 7 or 9 because there are zero cliques under such settings.

Exp-5: Scalability testing. We evaluate the scalability of our final algorithm SigClique* with the baseline SigClique as a comparison.

We generate a range of graph sizes and densities by randomly selecting vertices and edges, respectively, from 20% to 100%. To sample vertices, we generate the induced subgraph of the selected vertices, while for edges, we consider the vertex set of the corresponding incident edges. The outcomes of our experiments are depicted in Fig. 3.10. We do not have results of SigClique under several settings since it cannot finish in 5 hours. The results show that our algorithm is scalable to large graphs. In Fig. 3.10 (a), the time of SigClique* is 0.5 second when sampling 20% vertices and increases to 151 seconds finally.

3.6.2 Case study on DBLP

To evaluate the effectiveness of our model, we conduct a case study on DBLP and compare it with the maximal clique model. We generate a collaboration network for the DBLP, where an edge connects two researchers if they have at least three co-authored publications. Regarding the vertex labels, we select the publication data for each researcher from three major conferences in the database area — SIGMOD, PVLDB, and ICDE over the last decade. We assign each vertex the label that indicates the author has a published paper on the corresponding conference. Specifically, for each conference every year, given a researcher u, we assign u a conference name if u has a publication in the conference and an empty value $\emptyset$ otherwise. Note that the assigned labels can be repeated for each researcher.

For example, assume that u has one SIGMOD publication, two PVLDB publications, and no ICDE publication in one year. The added labels in such a year for u are $\{\mathsf{SIGMOD}, \mathsf{PVLDB}, \mathsf{PVLDB}, \emptyset\}$. Given the impact factor of these conferences, we expect the proportion of label frequencies for SIGMOD, PVLDB, ICDE, and $\emptyset$ (no paper published) as $1 : 2 : 3 : 30$. The rationale is that we give a relatively low expected frequency for the highest-ranked conference. If a researcher does not have any publication in the last decade, they will have thirty $\emptyset$ labels. Intuitively, a significant clique with a large chi-square value in our setting may represent a research group with many high-ranked conference publications.

In Fig. 3.11 (a)-(c), we show all the maximal significant cliques containing the Prof. Jiawei Han with the parameters $k = 5$ and $\theta = 500$. The parameter setting is application-oriented, where generally, the larger the k or θ, the fewer the resulting subgraphs. As we can see, there are three $(5, 500)$-cliques reported by our model. The chi-square statistics of the results in Fig. 3.11 (a), (b), and

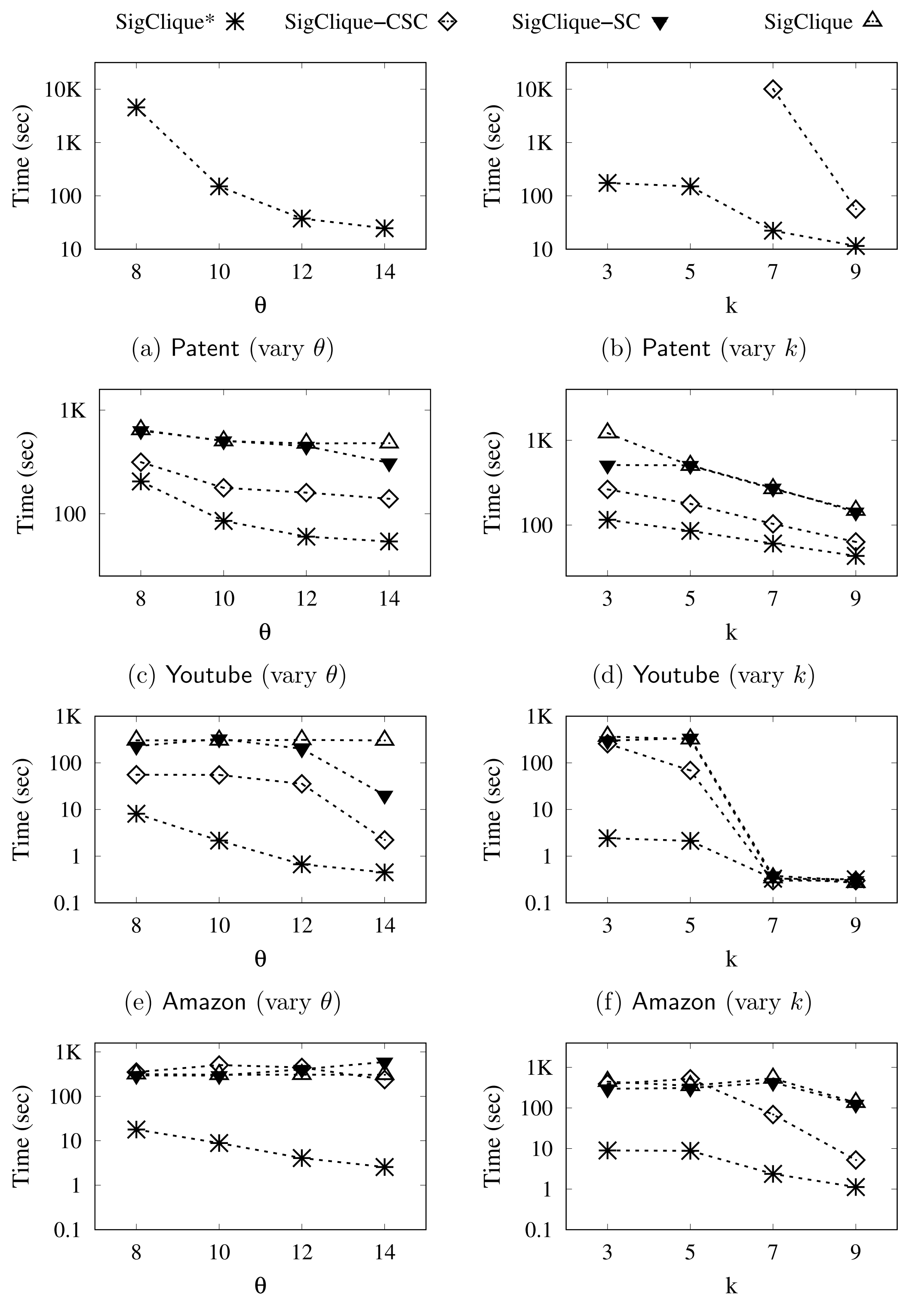
(c) are 828.639, 589.864, and 722.331, respectively.

In comparison, we also enumerate all the maximal cliques containing Prof. Jiawei Han with at least 5 vertices. There are 33 resulting subgraphs, including two 7-cliques, four 6-cliques, and 27 5-cliques. Due to the large result size, we show some representatives of them in Fig. 3.12. We can see that the number of maximal cliques is much more than that of the $(5, 500)$-cliques. We find that the main research interests of many resulting authors do not lay in the database area. For example, Prof. Tarek F. Abdelzaher, Prof. Su Lu, and Dr. Shaohan Hu may be mainly interested in the Internet of Things, Cyber-Physical Systems, and Mobile Computing since they have contributed a lot to the corresponding research area. However, they all have co-authored only one PVLDB paper in the past decade. According to historical publications, they should be excluded if we want to find significant database communities.

We have a consistent conclusion when considering the chi-square statistics of the resulting maximal cliques. In Fig. 3.13, we group all the 33 maximal cliques (with size at least 5) containing Prof. Han into five groups by corresponding chi-square statistics. We can see that the chi-square statistics of most results are relatively low, which means the resulting communities have rather less significance in the database area. For example, there are 25 results whose chi-square statistics lay in the interval $(0, 200]$. By contrast, all the resulting three $(5, 500)$-significant cliques reported by our model have much higher chi-square statistics. Note that the three maximal $(5, 500)$-significant cliques are also included in the results of maximal cliques.

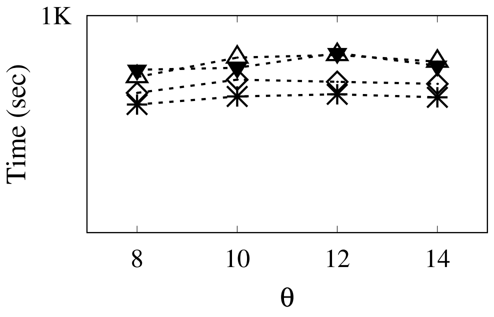
3.7 Chapter Summary

In this chapter, we formulate a new model, called maximal (k, θ)-significant clique, to capture statistically significant cohesive subgraphs in large labeled graphs. We propose a novel branch-and-bound algorithm and several effective pruning rules to enumerate all maximal (k, θ)-significant clique. Extensive experiments are conducted to show the efficiency of our algorithms. In future works, we are interested in extending our model to handle the graphs with both vertex labels and edge weights and explore certain parallel frameworks to enhance the scalability for large graphs.

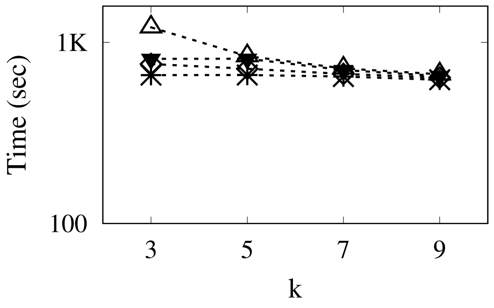


(a) Patent (vary θ)
(b) Patent (vary k)
(c) Youtube (vary θ)
(d) Youtube (vary k)
(e) Amazon (vary θ)
(f) Amazon (vary k)
(g) CiteSeer (vary θ)
(h) CiteSeer (vary k)

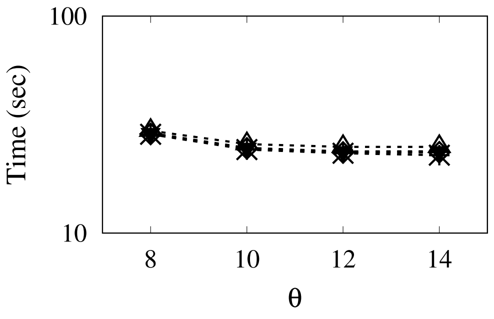
Figure 3.6: Running time of different algorithms by varying θ and k



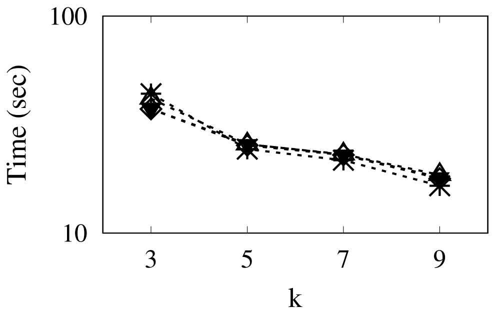
(i) DBLP (vary θ)



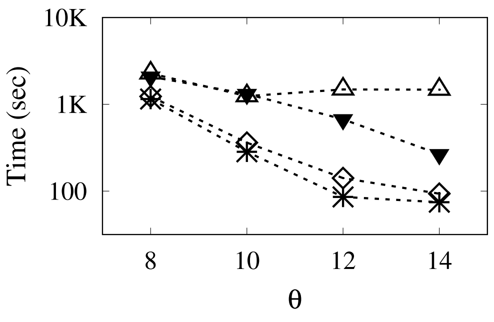
(j) DBLP (vary k)



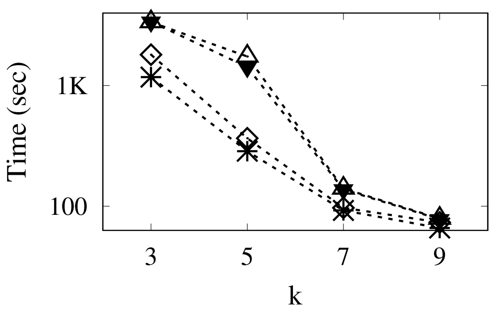
(k) EUmail (vary θ)



(l) EUmail (vary k)

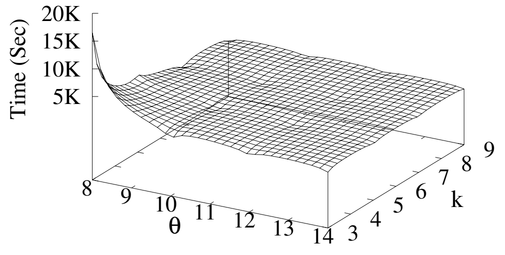


(m) Hyves (vary θ)

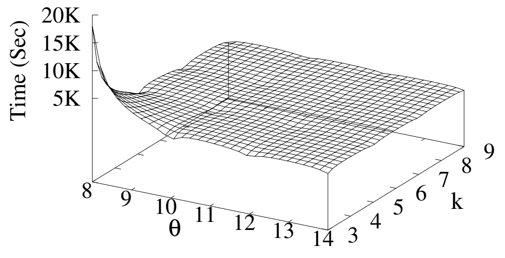


(n) Hyves (vary k)

Figure 3.6: Running time of different algorithms by varying θ and k (continued)

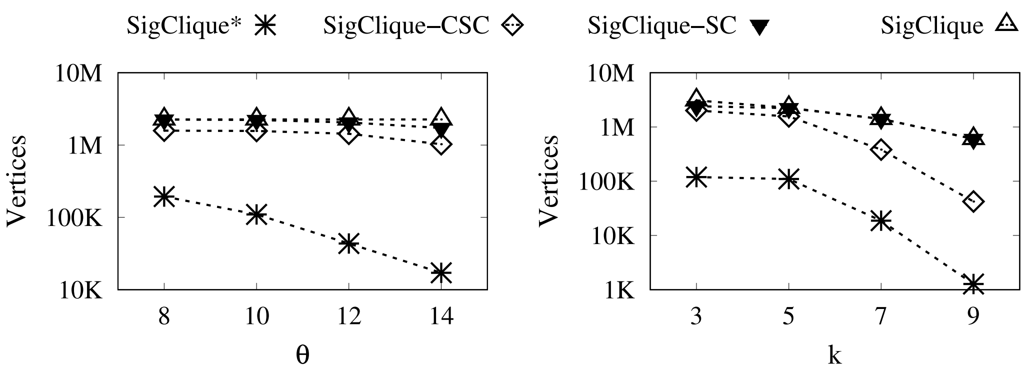


(a) Patent

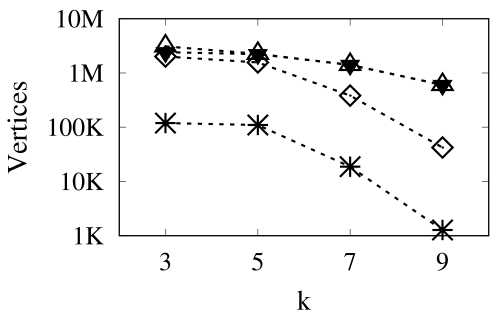


(b) Youtube

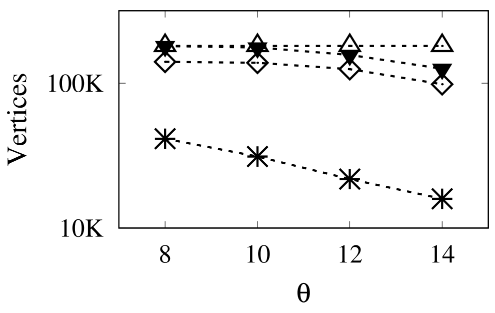
Figure 3.7: Running time of SigClique* by varying θ and k



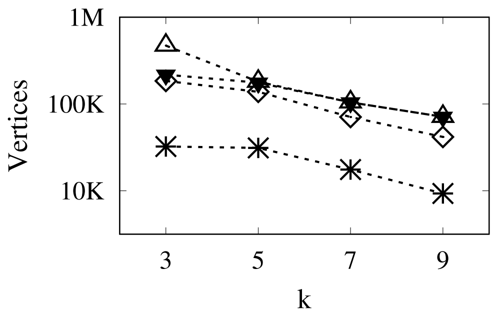
(a) Patent (vary θ)



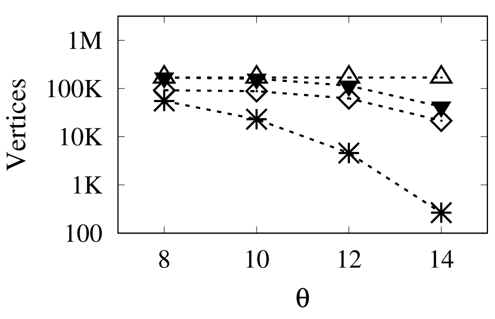
(b) Patent (vary k)



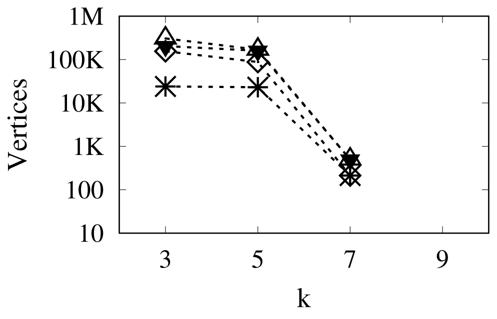
(c) Youtube (vary θ)



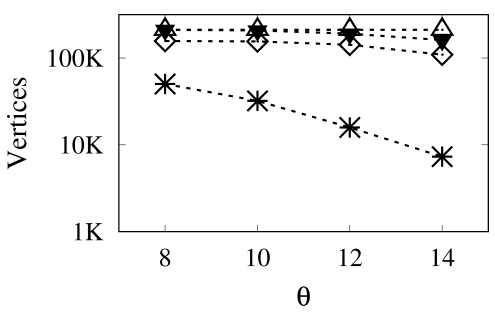
(d) Youtube (vary k)



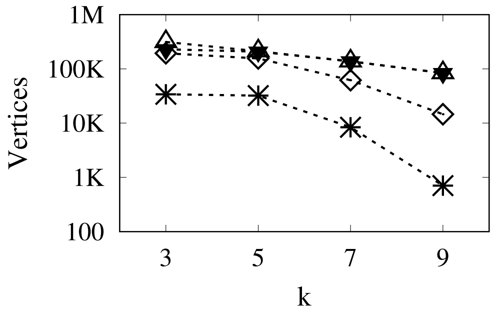
(e) Amazon (vary θ)



(f) Amazon (vary k)

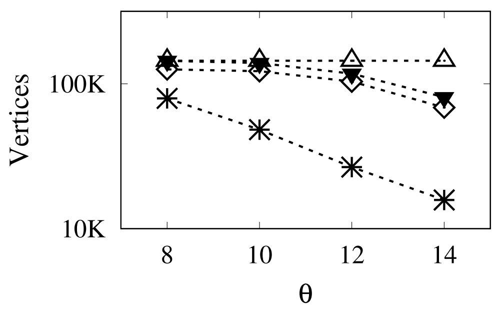


(g) CiteSeer (vary θ)

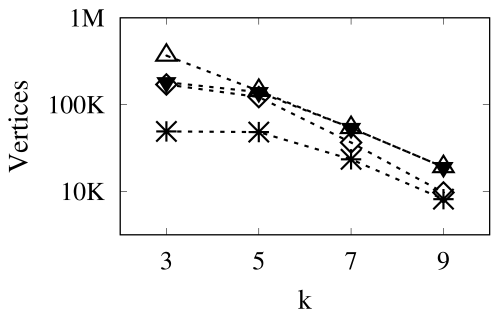


(h) CiteSeer (vary k)

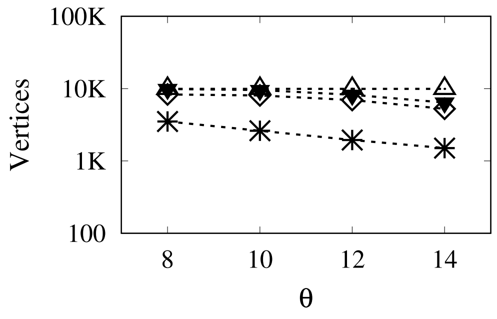
Figure 3.8: The number of vertices after pruning by varying parameters θ and k



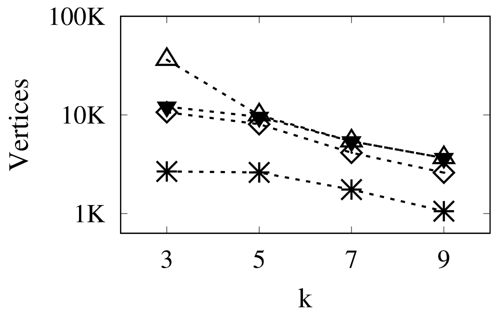
(i) DBLP (vary θ)



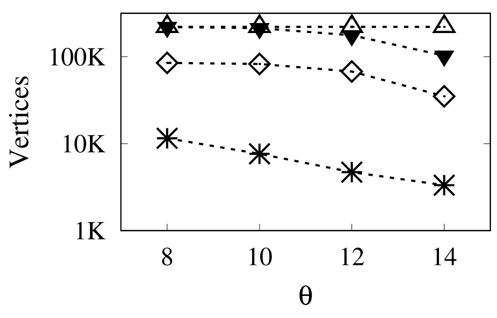
(j) DBLP (vary k)



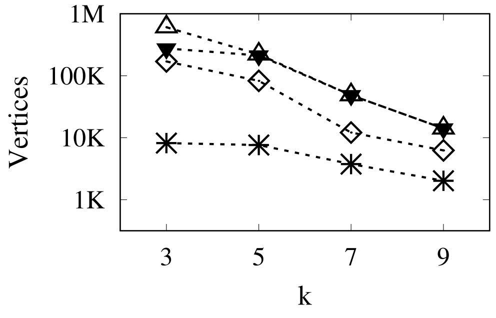
(k) EUmail (vary θ)



(l) EUmail (vary k)

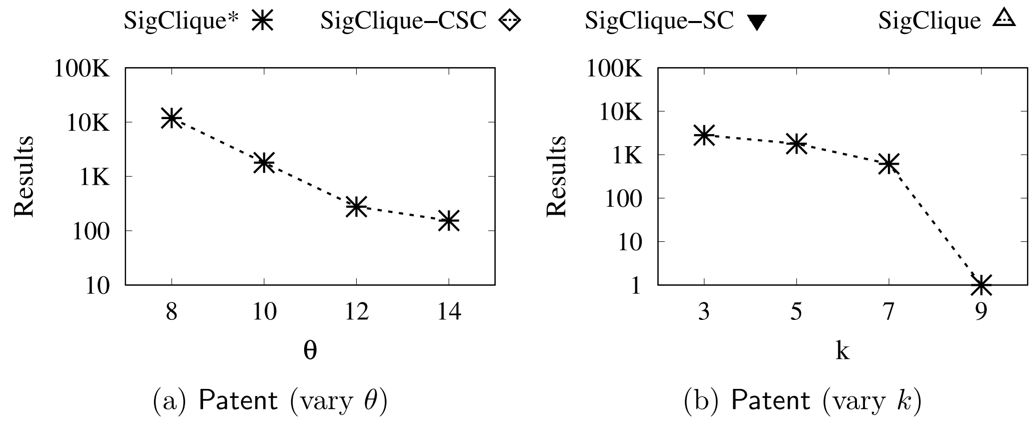


(m) Hyves (vary θ)

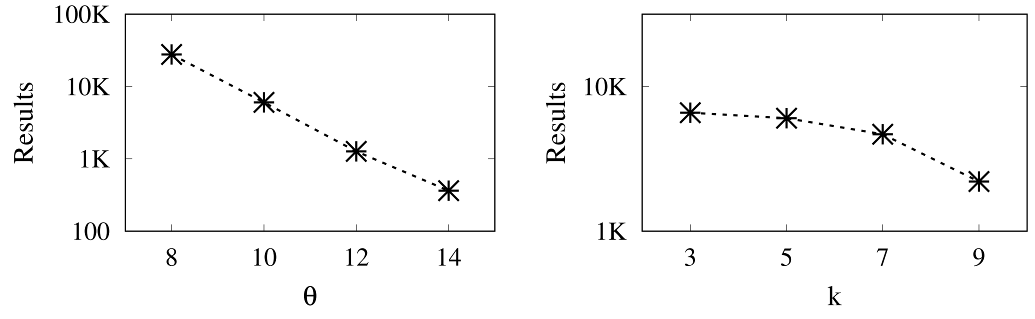


(n) Hyves (vary k)

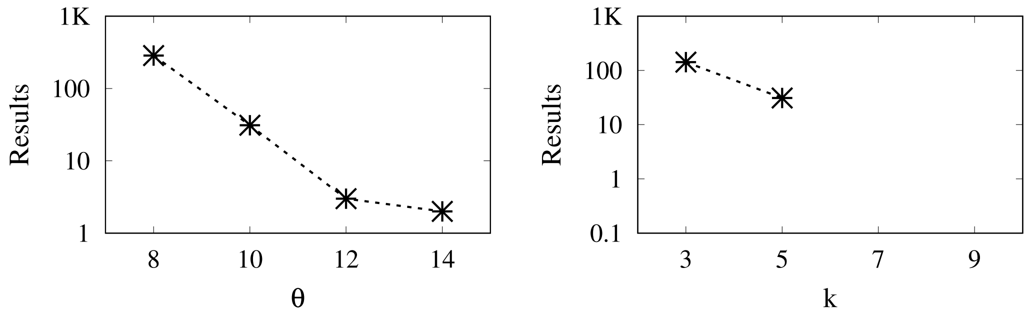
Figure 3.8: The number of vertices after pruning by varying parameters θ and k (continued)



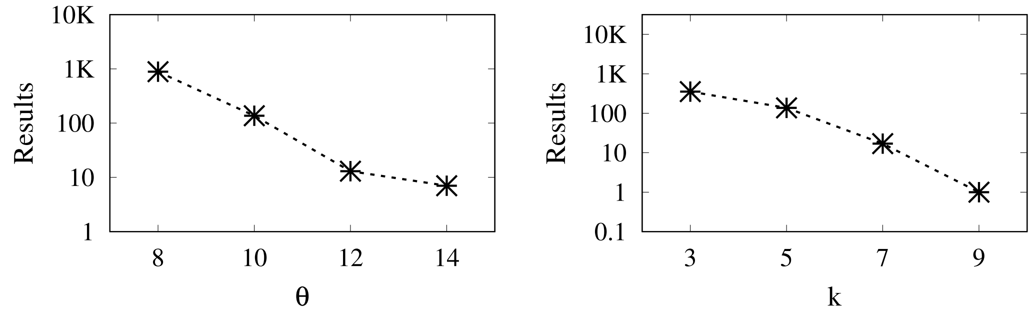
(a) Patent (vary θ) (b) Patent (vary k)



(c) Youtube (vary θ) (d) Youtube (vary k)

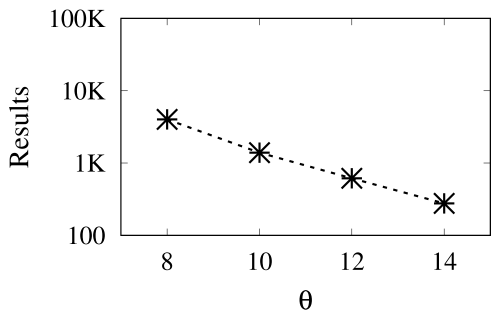


(e) Amazon (vary θ) (f) Amazon (vary k)

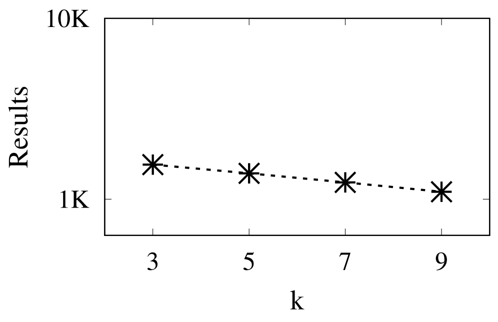


(g) CiteSeer (vary θ) (h) CiteSeer (vary k)

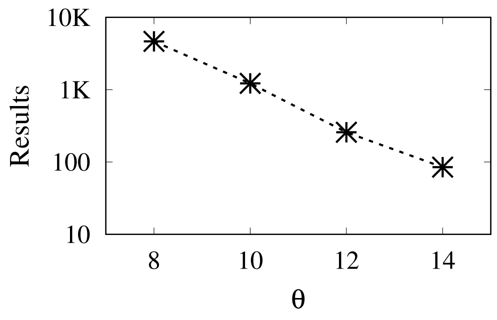
Figure 3.9: The number of maximal (θ, k)-cliques



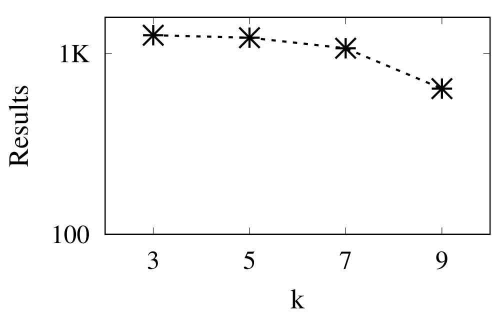
(i) DBLP (vary θ)



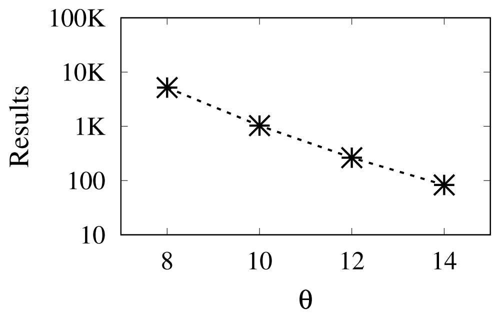
(j) DBLP (vary k)



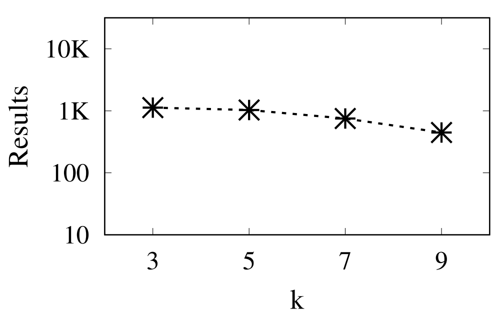
(k) EUmail (vary θ)



(l) EUmail (vary k)

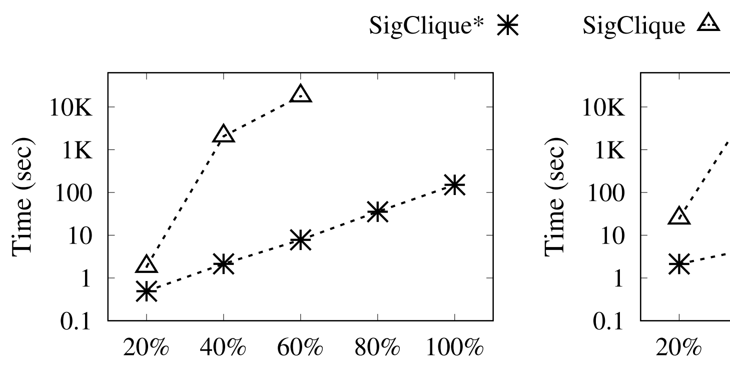


(m) Hyves (vary θ)

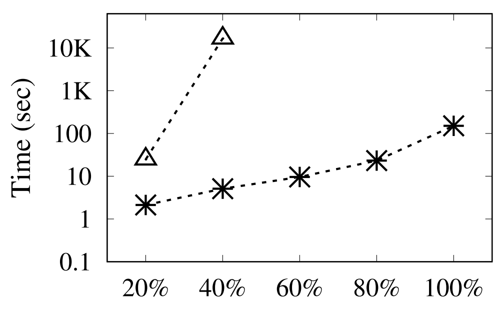


(n) Hyves (vary k)

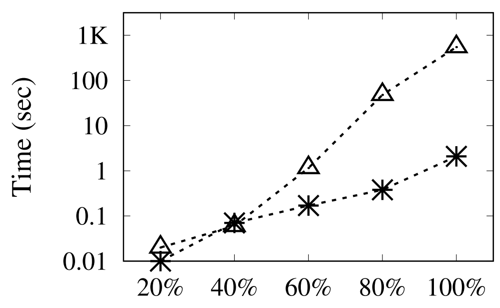
Figure 3.9: The number of maximal (θ, k)-cliques (continued)



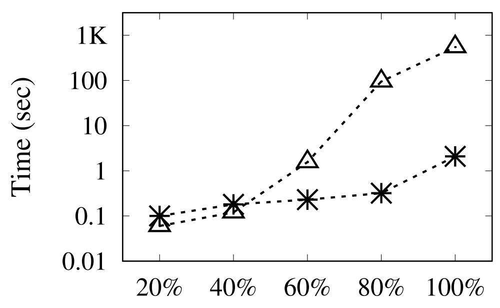
(a) Patent Vary $|V|$



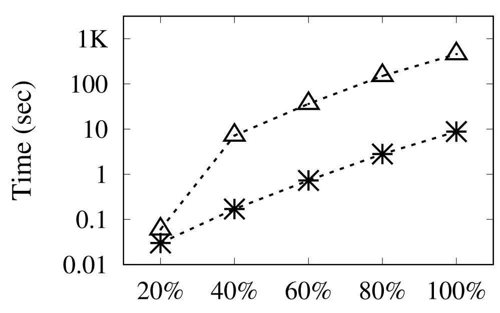
(b) Patent Vary $|E|$



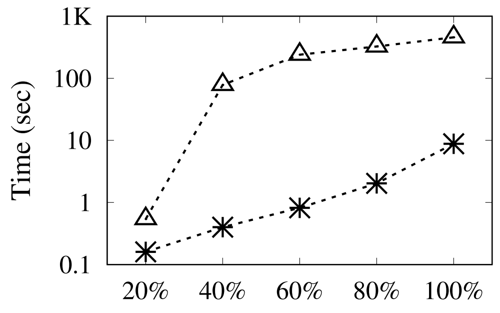
(c) Amazon Vary $|V|$



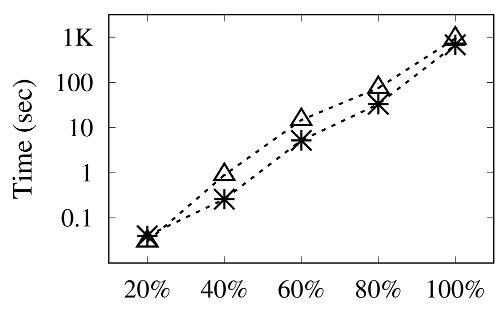
(d) Amazon Vary $|E|$



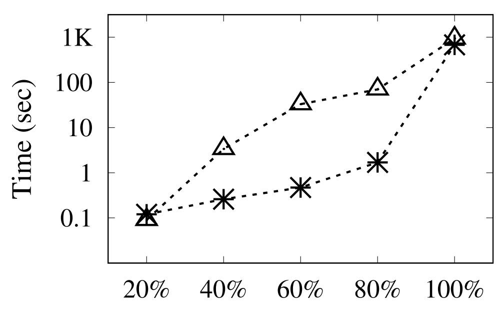
(e) CiteSeer Vary $|V|$



(f) CiteSeer Vary $|E|$

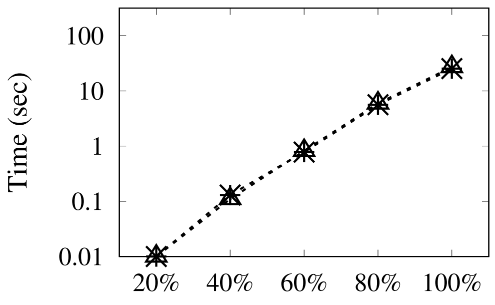


(g) DBLP Vary $|V|$

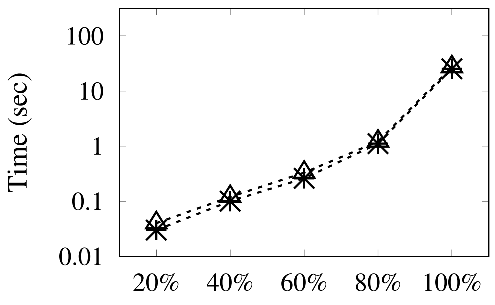


(h) DBLP Vary $|E|$

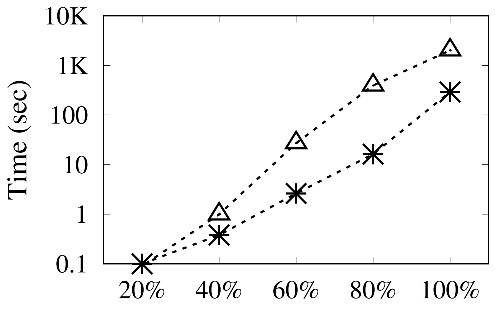
Figure 3.10: Scalability Testing.



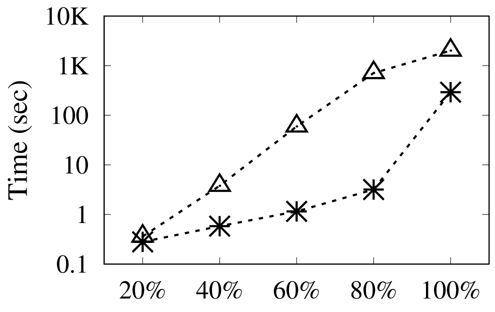
(i) EUmail Vary $|V|$



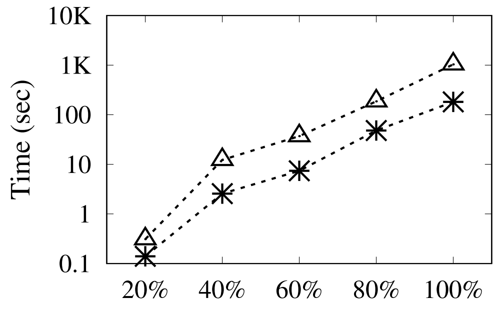
(j) EUmail Vary $|E|$



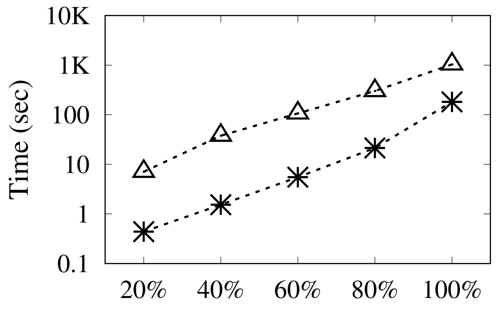
(k) Hyves Vary $|V|$



(l) Hyves Vary $|E|$



(m) Youtube Vary $|V|$



(n) Youtube Vary $|E|$

Figure 3.10: Scalability Testing (continued).

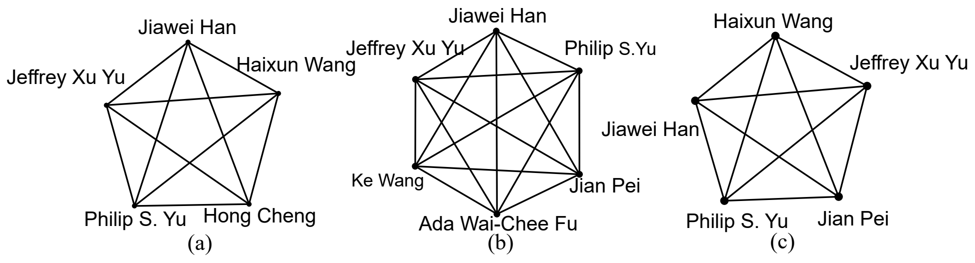


Figure 3.11: The maximal (5, 500)-significant cliques of Prof. Jiawei Han on DBLP graph

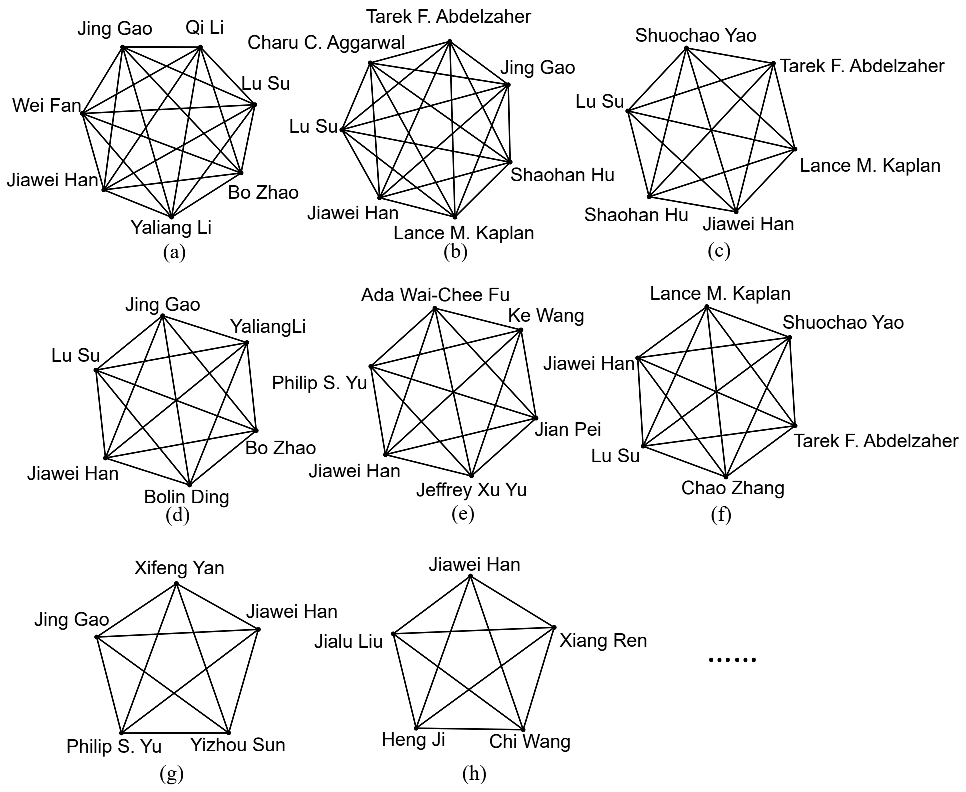


Figure 3.12: The maximal cliques containing Prof. Jiawei Han on DBLP graph with at least 5 vertices. (a) and (b) are 7-cliques. (c)-(f) are 6-cliques. (g) and (h) are 5-cliques

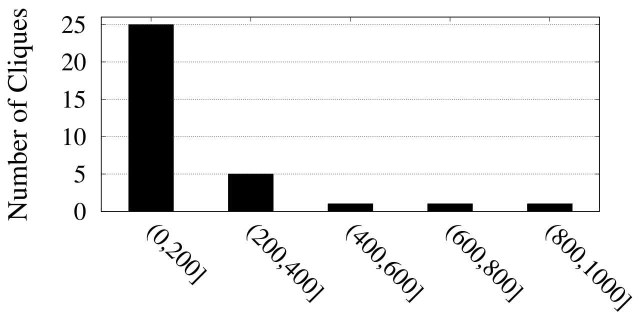


Figure 3.13: The chi-square statistic of all the maximal cliques containing Prof. Jiawei Han with at least 5 vertices

Chapter 4

EFFICIENT SHORTEST PATH COUNTING ON LARGE ROAD NETWORKS

4.1 Chapter Overview

In this chapter, we study the shortest-path counting query on road networks. This chapter is structured as follows. Section 4.2 provides the problem definition and the state-of-the-art algorithms. In Section 4.3, we show a novel index-based approach and its querying methods. Theoretical analysis confirms the advances of the new approach compared to the state-of-the-art methods. Section 4.4 first provides a non-trivial baseline index construction method to construct our index and then proposes an improved approach with optimization techniques which is much faster than the non-trivial baseline. Section 4.5 evaluates the proposed algorithms and Section 4.6 concludes this chapter.

4.2 Preliminaries

We first introduce the problem definition of shortest path counting in Section 4.2.1. We also provide a basic online method for the query. Section 4.2.2 describes the state-of-the-art method for shortest path counting. We analysis the limitation of the state-of-the-art method and discuss on the opportunities in Section 4.2.3. Table 4.1 lists the frequent notations used in this chapter.

Table 4.1: Frequent notations used in Chapter 4.

Notation	Meaning
$G = (V, E, \phi)$	a road network
$V(G)$	the set of vertices
$E(G)$	the set of edges
$\phi(e)$	the weight of an edge e
$\phi(p)$	the length of a path p
$N_G(u)$	the neighbor set of u in G
$\mathsf{sd}(u, v)$	the shortest distance between u and v
$\mathsf{spc}(u, v)$	the shortest path count between u and v
$\mathsf{cspc}(u, v)$	the convex shortest path count between u and v
$X(u)$	the tree node of u
$\mathsf{A}(u)$	the ancestor set of u
$\mathsf{CA}(u, v)$	the common ancestors of u and v
$\mathsf{T}(u)$	the subtree set of u

4.2.1 Problem Statement

We consider a road network $G = (V, E, \phi)$ which is usually a degree-bounded, connected, and weighted graph. $V(G)$ is a set of vertices, $E(G)$ is a set of edges, and $\phi : e \in E(G) \mapsto N^+$ is a weight function for each edge. We mainly focus on undirected graphs in this thesis. Our techniques can be easily extended to directed graphs and other sparse graphs. When it is clear from the context, we use V and E to denote $V(G)$ and $E(G)$, and use $n = |V|$ and $m = |E|$ to denote the numbers of vertices and edges, respectively. We denote all neighbors of a vertex v in G as $N_G(v) = \{u|(u,v) \in E(G)\}$. We use $N(v)$ to denote $N_G(v)$ when the context is obvious. We use $\phi(e)$ to denote the weight of an edge $e \in E$. On road networks, the edge weight may represent the actual length or the travel time of a road segment. A (simple) path[1] $p = (v_1, v_2, \dots, v_k)$ is a sequence of distinct vertices where $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. The length of a path p, denoted by $\phi(p)$, is the sum of weights of all edges on the path, i.e., $\phi(p) = \sum_{i=1}^{k-1} \phi(e(v_i, v_{i+1}))$. Given two vertices s and t, the shortest distance between s and t, denoted by $\mathsf{sd}(s,t)$, is the smallest length of all paths between s and t. A path p between s and t is a shortest path if $\phi(p) = \mathsf{sd}(s,t)$. We denote the set of all the shortest paths between s and t in the graph G by $P_G(s,t)$, and we use $P(s,t)$ when context is obvious. The number of shortest paths is denoted by $\mathsf{spc}(s,t)$, i.e., $\mathsf{spc}(s,t) = |P(s,t)|$.

Problem Definition Given a road network G and two query vertices $q = (s,t)$, the shortest path counting problem aims to efficiently compute the number of shortest paths between s and t.

Example 12. *Fig. 1.2 shows an example of a road network $G(V,E,\phi)$ with 20 vertices and 29 edges. The weight is marked on each edge. Considering vertices v_6 and v_{16}, there are many paths between them, such as $p_1 = (v_6, v_8, v_{14}, v_{16})$ and $p_2 = (v_6, v_4, v_3, v_1, v_{13}, v_{14}, v_{16})$. We have $\phi(p_1) = 9$ and $\phi(p_2) = 10$, and p_2 is not a shortest path. Given that there is no other path p with length less than p_1, p_1 is a shortest path, and the shortest distance between v_6 and v_{16} is 9. There are also 5 other paths with the same length 9 between v_6 and v_{16}. Therefore, the number of shortest paths between v_6 and v_{16} is 6 in G.*

A Basic Online Method. The shortest path count can be straightforwardly derived as a byproduct of the Dijkstra's algorithm in computing the shortest

[1] In this thesis, the term "path" always means a simple path.

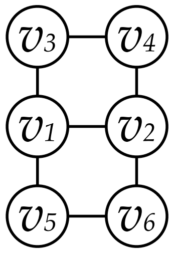
distance. Specifically, we use a queue to maintain all visited vertices with a priority of distance to the source vertex. For each visited vertex during the search, in addition to maintaining the distance from the source vertex, we store the corresponding shortest path count. Given a vertex v, let $D[v]$ and $C[v]$ be intermediate shortest distance and shortest path count, respectively. When exploring v from a neighbor u of v, if a shorter distance (i.e., $D[u] + \phi(u, v) < D[v]$) is found, we replace $D[v]$ and $C[v]$ with $D[u] + \phi(u, v)$ and $C[u]$, respectively. If $D[u] + \phi(u, v) = D[v]$, we add $C[u]$ to $C[v]$. We do not update $C[v]$ and $D[v]$ if $D[u] + \phi(u, v) > D[v]$. The online method works but may suffer from weak scalability in large graphs since all edges in the graph will be scanned in the worst case.

4.2.2 The State of the Art: Hub Labeling

Zhang and Yu [115] proposed a hub-labeling-based algorithm to count shortest paths efficiently. They first introduce the exact shortest path covering (ESPC) that guarantees to cover all shortest paths without redundancy and then propose a corresponding hub pushing algorithm to build the index. More specifically, for each vertex u, they precompute a collection of labels $L(u)$, and each label is a triplet $(w, \mathsf{sd}(u, w), \delta_{u,w})$, where $\mathsf{sd}(u, w)$ is the shortest distance between u and w, and $\delta_{u,w}$ is the number of a subset of shortest paths between u and w in the graph (i.e., $\delta_{u,w} \leq \mathsf{spc}(u, w)$). Given two query vertices u and v, the shortest path count is computed based on the following equation.

$$\mathsf{spc}(u, v) = \sum_{w \in L(u), w \in L(v), \mathsf{sd}(u,w)+\mathsf{sd}(v,w)=\mathsf{sd}(u,v)} \delta_{u,w} \cdot \delta_{v,w} \tag{4.1}$$

In Eq. 4.1, the shortest distance $\mathsf{sd}(u, v)$ can be derived by the formula $\min_{w \in L(u), w \in L(v)} \mathsf{sd}(u, w) + \mathsf{sd}(v, w)$.



Vertex	$L(\cdot)$
v_1	$(v_1, 0, 1)$
v_2	$(v_2, 0, 1), (v_1, 1, 1)$
v_3	$(v_3, 0, 1), (v_2, 2, 1), (v_1, 1, 1)$
v_4	$(v_4, 0, 1), (v_3, 1, 1), (v_2, 1, 1), (v_1, 2, 2)$
v_5	$(v_5, 0, 1), (v_2, 2, 1), (v_1, 1, 1)$
v_6	$(v_6, 0, 1), (v_5, 1, 1), (v_2, 1, 1), (v_1, 2, 2)$

Figure 4.1: A simple graph and its hub-labeling index given the vertex order $v_1 \leq v_2 \leq v_3 \leq v_4 \leq v_5 \leq v_6$.

For index construction, Zhang and Yu [115] assign a total order $\leq$ for all vertices. They process vertices in descending order. For each vertex $w \in V$, they perform a search from w in the induced subgraph of all the vertices whose orders are not higher than w. The shortest distance and shortest path count from w to each vertex v are collected in the search, and a corresponding label is added to $L(v)$. During the search, if a shorter distance between w and v is found based on the existing labels of w and v (i.e., a vertex with a high order than w covers the shortest path of w and v), they prune the search space and stop to search the neighbors of v. The algorithm finishes in n iterations.

Example 13. *We use a simple graph in Fig. 4.1 to illustrate the index structure of [115]. We assume that the total order is* $v_1 \leq v_2 \leq v_3 \leq v_4 \leq v_5 \leq v_6$. *The labels of all six vertices are presented on the right. Give a pair of query vertices* v_4 *and* v_5*, we have the shortest distance* $\mathsf{sd}(v_4, v_5) = \min\{\mathsf{sd}(v_4, v_2) + \mathsf{sd}(v_5, v_2), \mathsf{sd}(v_4, v_1) + \mathsf{sd}(v_5, v_1)\} = 3$*, and the number of shortest paths* $\mathsf{spc}(v_4, v_5) = \delta(v_4, v_2) \cdot \delta(v_5, v_2) + \delta(v_4, v_1) \cdot \delta(v_5, v_1) = 1 \times 1 + 2 \times 1 = 3$.

Several optimizations are also developed in [115] to reduce the index size. Given that both query efficiency and index size are closely related to the label size, they also analyze the maximum label size for each vertex for several types of graphs. We will compare their theoretical results with our method on road networks in Section 4.3.2.

4.2.3 Opportunities

We first analyze the limitations of the state-of-the-art method. In [115], the label size for a vertex can be very large, and we need to scan all the labels of two query vertices in the worst case. To precompute the labels for each vertex, their method scans the induced subgraph of all vertices with lower ranks. When processing the first vertex, the entire graph is scanned, and searching a large graph is costly.

The key to improving the efficiency of counting shortest paths is to reduce the number of label comparisons in query processing. To this end, we organize vertices in a tree structure called tree decomposition [39] and proposed a tree-based index structure. Even though tree decomposition has been used in existing works to compute the shortest distance on road networks, the main technical challenge in our problem is to avoid the redundancy of query results, which is quite different from existing studies.

A straightforward idea to construct our index is to adopt a similar framework in the hub-labeling method, where high-ranking vertices are first processed. We significantly improve the efficiency of index construction by adopting a reverse framework. We first process low-ranking vertices so that all intermediate information can be fully utilized when high-ranking vertices are processed. We propose graph reduction techniques to guarantee the correctness of our new computing framework. We also propose a rule to relax the index definition, which reduces the computational cost but still guarantees correctness.

4.3 Tree-based Shortest Path Counting

We propose a new labeling-based index carefully defined based on a tree structure of all vertices in the graph. Compared to the state of the art, our solution achieves higher efficiency for both query processing and index construction with a more compact index.

4.3.1 Tree Decomposition

Tree Decomposition has been used in many applications to speed up certain graph computational problems [39]. We give the formal definition as follows.

Definition 12. (TREE DECOMPOSITION) *Given a graph $G(V, E)$, a tree decomposition of G, denoted as T_G, is a tree in which every tree node $X \in T_G$ is a subset of V (i.e., $X \subseteq V$) such that the following conditions hold:*

1. $\bigcup_{X \in T_G} X = V$;
2. *for every $(u, v) \in E$, there exists $X \in T_G$ such that $u \in X$ and $v \in X$;*
3. *for every $u \in V$, $\{X | u \in X\}$ forms a connected subtree of T_G.*

Definition 13. (TREEWIDTH AND TREEHEIGHT) *Given a tree decomposition T_G of a graph G, the* treewidth *of T_G, denoted by $w(T_G)$, is one less than the maximum cardinality of all nodes in T_G, i.e., $w(T_G) = \max_{X \in T_G} |X| - 1$. The* treeheight*, denoted by $h(T_G)$, is the maximum depth of all nodes in T_G where the depth of a node X is the distance from X to the root node in T_G.*

We use w and h to denote treewidth and treeheight, respectively, when it is clear from the context. It is important to note that we can derive a tree decomposition of a road network with low treewidth and low treeheight values. For example, on the road network of New York City with 264,346 vertices and

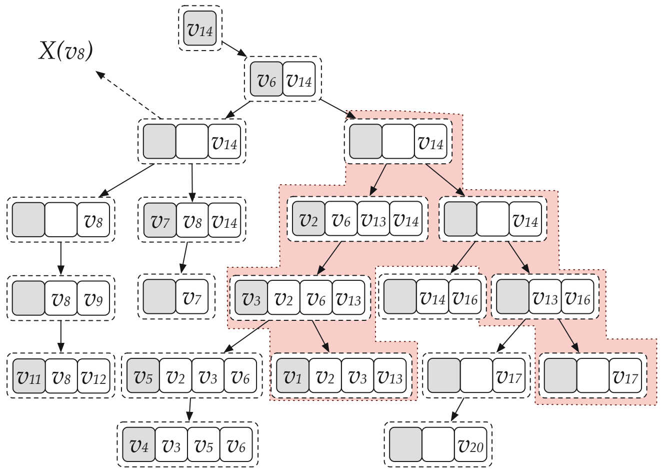


Figure 4.2: Tree decomposition T_G of G.

733,846 edges, we can construct a tree decomposition with a treeheight of 505 and a treewidth of 134. Detailed statistics for other datasets can be found in Table 4.2.

Tree Decomposition Construction. Given a graph G, there could be multiple tree decompositions, and it is NP-Complete to determine the minimized treewidth of all tree decompositions of G [8]. In this thesis, we adopt a sub-optimal tree decomposition method proposed in [61] with a time complexity of $O(n \cdot (w^2 + \log n))$. The method is relatively efficient in practice and has been used in several research works on road networks [76, 77, 27]. It processes each vertex in a greedy way. Specifically, in each iteration, the algorithm picks the vertex v with the smallest degree and creates a corresponding tree node $X(v)$ with all neighbors of v in the graph. Then, it removes v and updates the graph by adding an edge between every pair of unconnected neighbors of v. Assume u is the first removed neighbor of v after removing v, we set $X(u)$ as the parent of $X(v)$ in the tree decomposition. The algorithm terminates after removing all vertices, and we get the tree decomposition of the given graph.

It is straightforward to see that the derived tree decomposition contains n

nodes, and there is a one-to-one correspondence from graph vertices to tree nodes. In the rest, we assume this property holds, and tree decomposition is computed using the method in [61].

We always refer to each $v \in V$ in graphs as a vertex and refer to each $X \in T_G$ as a (tree) node. We use the vertex v instead of the tree node $X(v)$ for simplicity when it is clear from the context. The depth of a vertex v, denoted by $\mathsf{Depth}(v)$, is the number of edges from the tree node $X(v)$ to the root node. The ancestor set of a vertex v, denoted by $\mathsf{A}(v)$ is the set of vertices u such that $X(u)$ is an ancestor of $X(v)$ in the tree decomposition. The subtree set of a vertex v, denoted by $\mathsf{T}(v)$, is the set of vertices u such that $X(u)$ is in the subtree rooted by $X(v)$ in the tree decomposition.

Example 14. *Fig. 4.2 shows a tree decomposition T_G for the road network in Fig. 1.2. To construct such a tree decomposition, we first pick the vertex with the lowest degree. Suppose we pick v_{19} and create a tree node $X(v_{19})$ with its neighbors v_{20} and v_{17}. We then remove v_{19} and add an edge between v_{17} and v_{20}. We repeat the above process until all the vertices are removed. Assume v_{20} is the first removed neighbor of v_{19}, we set $X(v_{20})$ as the parent of $X(v_{19})$. We repeat the process and get a tree with 20 tree nodes.*

The corresponding tree node of the vertex v_3 is $X(v_3) = \{v_3, v_2, v_6, v_{13}\}$. The ancestors of v_3 are $\mathsf{A}(v_3) = \{v_{14}, v_6, v_{13}, v_2\}$, and the subtree set of v_3 is $\mathsf{T}(v_3) = \{v_3, v_5, v_1, v_4\}$. For the vertex v_{13}, all tree nodes containing v_{13} form a connected subtree and are marked in the red area in Fig. 4.2.

4.3.2 TL-Index

We propose a new index structure, named TL-Index, to count shortest paths based on tree decomposition. Compared to the state-of-the-art hub-labeling-based index, the only similar part in TL-Index is, conceptually, to store a set of labeling vertices with corresponding distance and count values to each vertex v. We carefully pick the labeling values by utilizing tree decomposition. We organize the labels in a structure that can avoid the merge-sort-like style query mechanism in [61] and achieve higher query efficiency. The details of query processing can be found in Section 4.3.3. Below, we introduce an important definition which is crucial to guarantee the correctness of the index.

Definition 14. (Convex Path) *Given a tree decomposition T_G of graph $G(V, E)$, a path $p = (s, v_1, v_2, \ldots, v_k, t)$ between two vertices s and t is a convex path if for every $1 \leq i \leq k$, the depth of $X(v_i)$ is larger than the smaller one of $X(s)$ and $X(t)$, i.e., $\forall 1 \leq i \leq k, \mathsf{Depth}(v_i) > \min(\mathsf{Depth}(s), \mathsf{Depth}(t))$.*

Example 15. *Given the graph G in Fig. 1.2 and its tree decomposition T_G in Fig. 4.2, the path (v_6, v_8, v_7, v_{10}) is a convex path. This is because $\mathsf{Depth}(v_7) > \mathsf{Depth}(v_8) > \min(\mathsf{Depth}(v_6), \mathsf{Depth}(v_{10}))$. The path $(v_6, v_8, v_{14}, v_{16})$ is not a convex path, as $\mathsf{Depth}(v_{14}) < \min(\mathsf{Depth}(v_6), \mathsf{Depth}(v_{16}))$.*

Let $\odot$ be the concatenation of two paths. We provide a support lemma followed by the key theorem motivating our index below.

Lemma 9. *Given a tree decomposition T_G and an arbitrary path p in G, there exists only one vertex in p with the lowest depth.*

Proof. We prove Lemma 9 by contradiction. Given a path p, suppose we have $u, v \in p$, and both u and v have the lowest depth in p. Thus, $T(u)$ and $T(v)$ must be two disjoint sub-trees in T_G. We denote the sub-path between u and v by $(u, w_1, w_2, \ldots, w_{i-1}, w_i, v)$. We first consider the u side. As $(u, w_1) \in E$, based on Definition 12 (2), we know $\exists X \in T_G, u \in X, w_1 \in X$. Based on Definition 12 (3), $X(u)$ and X should be in the same sub-tree, $X(w_1)$ and X should also be in the same sub-tree, i.e., $X(u)$ and $X(w_1)$ should be in the same

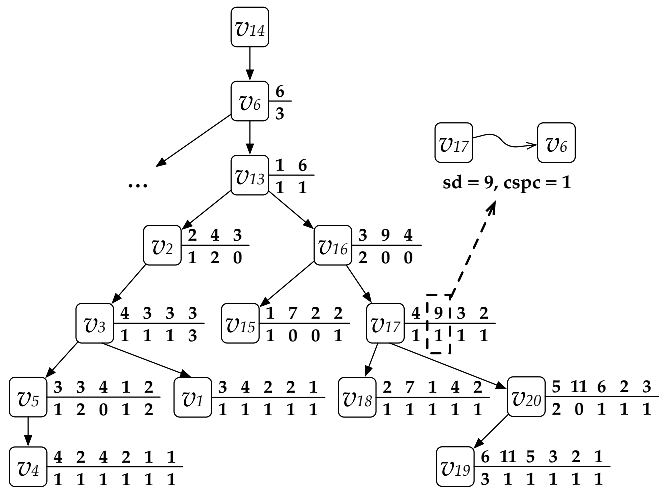

v_{14}
v_6 6 / 3
...
v_{13} 1 6 / 1 1
v_{17} v_6
sd = 9, cspc = 1
v_2 2 4 3 / 1 2 0
v_{16} 3 9 4 / 2 0 0
v_3 4 3 3 3 / 1 1 1 3
v_{15} 1 7 2 2 / 1 0 0 1
v_{17} 4 9 3 2 / 1 1 1 1
v_5 3 3 4 1 2 / 1 2 0 1 2
v_1 3 4 2 2 1 / 1 1 1 1 1
v_{18} 2 7 1 4 2 / 1 1 1 1 1
v_{20} 5 11 6 2 3 / 2 0 1 1 1
v_4 4 2 4 2 1 1 / 1 1 1 1 1 1
v_{19} 6 11 5 3 2 1 / 3 1 1 1 1 1

Figure 4.3: The TL-Index for G.

sub-tree. As u has the minimum depth, we have $w_1 \in T(u)$. We similarly have $w_2 \in T(u), \ldots, w_i \in T(u)$. On the v side, we also have $w_i \in T(v)$ where the contradiction exists. □

Theorem 9. *Given an arbitrary path $p(v_1, v_2, \ldots, v_k)$, either p is a convex path or there exists one and only one pair of convex paths p_1 and p_2 such that $p = p_1 \odot p_2$.*

Proof. Based on Lemma 9, given a non-convex path p between two vertices s and t, assume that v is the vertex with the smallest depth in p. We have $v \neq s$ and $v \neq t$. Otherwise, p is a convex path. Therefore, p can be divided to two sub-paths from v, and both sub-paths are convex paths. □

Based on Theorem 9, each shortest path is either a convex shortest path or a concatenation of two convex shortest paths, given that any sub-path of a shortest path is also a shortest path. Here, a convex shortest path is a convex path that has the same length as the shortest path between two terminal vertices in the graph. To compute the shortest path count for any pair of vertices, our idea is to precompute the distance and the count of all convex shortest paths between each possible pair of vertices. Note that the number of convex shortest paths is significantly smaller than that of all shortest paths, which is supported by the following lemma.

Lemma 10. *Given a tree decomposition T_G of a graph G and an arbitrary path p, let v be the vertex with the smallest depth in p. v is the ancestor of all other vertices in p.*

The shortest paths between two vertices s and t can be divided into two types. The first is the convex shortest path between s and t, and the other is the concatenation of two convex shortest paths from s and t, respectively. It is easy to see that for a non-convex shortest path p, two convex sub-paths join at the vertex with the smallest depth in p. Therefore, our index stores the count of each precomputed convex shortest path as a label of the terminal vertex with a larger depth. We formally define the index as follows.

Definition 15. *Given a road network G, TL-Index precomputes:*

1. *a tree structure of all vertices by tree decomposition;*
2. *the shortest distance from each vertex to all its ancestors;*
3. *the convex shortest path count from each vertex to all its ancestors.*

Note that by a tree structure in Definition 15, we discard all vertices except v in each tree node $X(v)$ and use v as a tree node instead of the original vertex set $X(v)$.

Example 16. *We show the TL-Index for the road network G (Fig. 1.2) in Fig. 4.3 based on the tree decomposition T_G in Fig. 4.2. For simplicity, we only show a part of the index. The index is based on the tree structure of the tree* *decomposition. For each vertex (index tree node), we store the shortest distance and the corresponding convex shortest path count to each ancestor. The label for the ancestor close to the root is arranged in the front. We take the vertex v_{17} as an example. As we mark in Fig. 4.3, the shortest distance between v_{17} and v_6 is 9, and there is only 1 convex shortest path. The shortest distance between v_{17} and v_{14} is 4, and its corresponding convex shortest path count is 1. Note that in the labels of v_{20}, the count value for v_6 is 0 since there is no convex path between them whose length is less than or equal to 11.*

Theorem 10. *The space complexity of TL-Index is $O(n \cdot h)$.*

Zhang and Yu [115] also analyze the space usage of their proposed hub-labeling index for graphs with small treewidth. They show that their index size can be bounded by $O(wn \log n)$ if the vertex order is carefully arranged, where w is the treewidth, and n is the number of vertices. Our index size is much smaller given that h is much smaller than $w \log n$. The statistics (e.g., w, h and n) of each dataset evaluated in experiments is provided in Table 4.2. We also compare the practical index size in Fig. 4.11.

4.3.3 Query Processing with TL-Index

We propose the query processing algorithm in this section. Our TL-Index is also a labeling-based index that satisfies the criteria of exact shortest path covering (ESPC) defined in [115]. Given two query vertices s and t, the general idea is to identify all common labeling vertices as shown in Eq. 4.1. We utilize the tree structure to bound the common labels of query vertices. In this way, we avoid the merge-sort-like strategy to process labels of two query vertices in Eq. 4.1 and speed up the query processing by visiting a limited number of common vertices directly. We reduce the visited labels in query processing by the following lemma.

Lemma 11. *Given two query vertices s and t, let $\mathsf{CA}(s,t)$ be the set of vertices v such that $X(v)$ is a common ancestor*[2] *of $X(s)$ and $X(t)$. For each shortest path p between s and t, let u be the vertex with the lowest depth in p. We have $u \in \mathsf{CA}(s,t)$.*

Based on Lemma 11, the shortest distance path count between s and t can be computed using the following equation.

$$\mathsf{spc}(s,t) = \sum_{v \in \mathsf{CA}(s,t), \mathsf{sd}(s,v)+\mathsf{sd}(v,t)=\mathsf{sd}(s,t)} \mathsf{cspc}_{s,v} \cdot \mathsf{cspc}_{v,t}, \tag{4.2}$$

where $\mathsf{cspc}_{s,v}$ denotes the number of all convex shortest paths between vertices s and v. The shortest distance between s and t can be computed as follows.

$$\mathsf{sd}(s,t) = \min_{v \in \mathsf{CA}(s,t)} \mathsf{sd}(s,v) + \mathsf{sd}(v,t) \tag{4.3}$$

We provide the query processing algorithm based on TL-Index in Algorithm 6. We first compute the LCA of s and t in line 1. Then, for each common ancestor of s and t, we initialize the shortest path count if a shorter distance is found (lines 5–7). We increase the existing count if the same distance value is found (lines 8–9).

Theorem 11. *The time complexity of Algorithm 6 is $O(h)$, where h is the* $\mathsf{treeheight}$ *of the tree decomposition T_G.*

Proof. In line 1 of Algorithm 6, it takes $O(1)$ time to find the LCA [14]. In line 3, the size of $\mathsf{A}(v) \cup \{v\}$ is bounded by the $\mathsf{treeheight}$ h of T_G. □

[2]Each tree node is also regarded as an ancestor of itself in Lemma 11.

```
Algorithm 6: TL-Query
  Input: the TL-Index and two query vertices $s, t$
  Output: the shortest distance $\mathsf{sd}(s,t)$, and corresponding count $\mathsf{spc}(s,t)$
1 $v \leftarrow$ the LCA of $s$ and $t$ in the tree;
2 $d \leftarrow \infty, c \leftarrow 0$;
3 foreach $u \in \mathsf{A}(v) \cup \{v\}$ do
4     $d' \leftarrow \mathsf{sd}(s,u) + \mathsf{sd}(u,t)$;
5     if $d' < d$ then
6         $d \leftarrow d'$;
7         $c \leftarrow \mathsf{cspc}_{s,u} \cdot \mathsf{cspc}_{u,t}$;
8     else if $d' = d$ then
9         $c \leftarrow c + \mathsf{cspc}_{s,u} \cdot \mathsf{cspc}_{u,t}$;
10 return $d$ and $c$
```

4.4 Index Construction

We propose algorithms for index construction in this section. Section 4.4.1 extends the framework in the state of the art and presents a non-trivial indexing algorithm. Sections 4.4.2, 4.4.3, 4.4.4, and 4.4.5 propose optimizations and our improved algorithm for index construction.

4.4.1 Basic Index Construction by Hub Pushing

In [115], the authors propose a hub-pushing algorithm to construct an order-based labeling index. We introduce a non-trivial baseline named TL-Construct by extending their framework.

The pseudocode of TL-Construct is provided in Algorithm 7. Given the tree decomposition, TL-Construct processes vertices in a top-down manner in the tree. The algorithm runs in n rounds. In each round (lines 2–20), we pick the vertex u with the smallest depth in the tree and break the tie by picking an arbitrary one. We search from u with a distance priority. The arrays $D[\cdot]$ and $C[\cdot]$ store the distance and the shortest path count, respectively, from u to each vertex

```
Algorithm 7: TL-Construct
Input: A road network $G(V, E, \phi)$
Output: The TL-Index of $G$
1  $T_G \leftarrow$ TreeDecomposition$(G)$;
2  foreach $X(u) \in T_G$ in a top-down manner do
3      foreach $v \in \mathsf{T}(u)$ do $D[v] = \infty$;
4      $D[u] = 0, C[u] = 1$;
5      $Q \leftarrow$ an empty queue prioritized by $D[\cdot]$;
6      $Q.enqueue(u)$;
7      while $Q$ is not empty do
8          $v \leftarrow Q.dequeue()$;
9          $d \leftarrow \min_{p \in \mathsf{A}(u)} \mathsf{sd}(u, p) + \mathsf{sd}(p, v)$;
10         if $d < D[v]$ then
11             $\mathsf{sd}(u, v) \leftarrow d$, $\mathsf{cspc}(u, v) \leftarrow 0$;
12             continue
13         else $\mathsf{sd}(u, v) \leftarrow D[v], \mathsf{cspc}(u, v) \leftarrow C[v]$ ;
14         foreach $v' \in N(v)$ do
15             $nd \leftarrow D[v] + \phi(v, v')$;
16             if $D[v'] > nd \wedge \mathsf{Depth}(v') > \mathsf{Depth}(u)$ then
17                 $D[v'] \leftarrow nd, C[v'] \leftarrow C[v]$;
18                 $Q.enqueue(v')$
19             else if $D[v'] = nd$ then
20                 $C[v'] \leftarrow C[v'] + C[v]$;
```

during the search.

The distance and the shortest path count to the source vertex u itself are initialized by 0 and 1 respectively in line 4. In each iteration, we pop the top vertex v from the priority queue in line 8, and v is the nearest unprocessed vertex to u.

Line 9 computes the distance between u and v based on the information computed in earlier rounds. For each common ancestor p of u and v, we compute the distance by using p as a bridging vertex given that the shortest distances from p to u and from p to v have been computed. If the distance based on earlier information is shorter than the current distance, we store the shorter distance and terminate further exploration from v in lines 10–12. Otherwise, we store the shortest distance $\mathsf{sd}(u, v)$ and the convex shortest path count $\mathsf{cspc}(u, v)$ for the TL-Index.

We extend the search space from v to each neighbor of v. In line 16, $D[v'] > nd$ means we find shorter distance from u to v'. In this case, we replace the existing distance and the convex shortest path count to v'. Note that $D[v']$ can be ∞ if v' is not visited in the search. If the distance is the same as that computed in earlier iterations (line 19), we increase the number of convex shortest paths in line 20. Lines 16 and 19 also guarantee that we only process vertices with a larger depth than the source vertex u.

Example 17. *We show a running example for Algorithm 7. Given a graph G (Fig. 1.2) and its tree decomposition T_G (Fig. 4.2), let us consider the shortest distance and the shortest path count between the vertices v_{14} and v_{16}. Algorithm 7 starts from v_{14} and adds all its neighbors into Q in lines 14–18. It also updates $D[\cdot]$ and $C[\cdot]$ in line 17. For instance, we have $D[v_{15}] = 1$, $C[v_{15}] = 1$, $D[v_{16}] = 3$, and $C[v_{16}] = 1$ after scanning all the neighbors of v_{14}. In the next iteration, suppose Q pops v_{15} in line 8, and Algorithm 7 expands its neighbor, i.e., v_{16} in line 15-20. We have $nd = D[v_{15}] + \phi(v_{16}, v_{15}) = 1 + 2 = 3$ which equals $D[v_{16}]$. Thus, we add $C[v_{16}]$ and $C[v_{15}]$ in line 20. When Q pops v_{16}, we store $D[v_{16}]$ and $C[v_{16}]$ to $\mathsf{sd}(v_{14}, v_{16})$ and $\mathsf{cspc}(v_{14}, v_{16})$, respectively. When Q is empty in line 7, we have computed the shortest distance and the shortest path count from v_{14} to each other vertex. The next vertex is v_6 in line 2. Consider a case that $u = v_6$ when Q pops v_{16} in line 8. We find $D[v_{16}] = 11$ which is greater than $\mathsf{sd}(v_6, v_{14}) + \mathsf{sd}(v_{14}, v_{16}) = 9$ (lines 9-10). Therefore, we set $\mathsf{sd}(v_6, v_{16}) = 9$, $\mathsf{cspc}(v_6, v_{16}) = 0$ (line 11) and terminate the exploration in line 12.*

Theorem 12. *The time complexity of Algorithm 7 is $O(nh^2 + nh \log n)$.*

Algorithm 8: Operator $\ominus$

```
  Input: A road network $G$ and a vertex $u \in V(G)$
  Output: The graph $G \ominus u$
1 foreach $v, w \in N(u)$ do
2     if $(v, w) \notin E \vee \phi(v, w) > \phi(v, u) + \phi(u, w)$ then
3         $E \leftarrow E \cup \{(v, w)\}$;
4         $\phi(v, w) \leftarrow \phi(v, u) + \phi(u, w)$;
5         $\varsigma(v, w) \leftarrow \varsigma(v, u) \times \varsigma(u, w)$;
6     else if $\phi(v, w) = \phi(v, u) + \phi(u, w)$ then
7         $\varsigma(v, w) \leftarrow \varsigma(v, w) + \varsigma(v, u) \times \varsigma(u, w)$;
8 remove $u$ and all incident edges from $G$;
```

Proof. For each vertex $u \in G$, we only visit the vertices $v \in T(u)$. Thus, we visit $O(n \cdot h)$ times in total. In each visit, we query $O(h)$ times in line 9. The maintenance of the priority queue Q costs $O(h \cdot m + h \cdot n \log n)$ using Fibonacci heap. Thus, the time complexity of Algorithm 7 is $O(nh^2 + nh \log n)$. $\square$

4.4.2 A New Upward Computing Framework

The index construction algorithm proposed in Section 4.4.1 essentially computes a distance value and a count value for each pair of vertices with an ancestor-descendant relationship. For each vertex u in each round, Algorithm 7 performs a priority-queue-based search for all vertices in the subtree rooted from u and computes values between u and all its descendants. However, the search space can be the whole graph in the worse case. In addition, for each visited vertex v, the algorithm scans all the ancestors of u to check if a shorter distance value exists, which incurs significant extra cost.

To improve the efficiency of index construction, we propose a novel index construction algorithm, which is called TL-Construct*. TL-Construct* adopts an upward computing framework. Specifically, for each vertex u in each round, we compute the distance value and the count value between u and all its ancestors in the tree. We propose a graph reduction technique in Section 4.4.3 to support the correctness of the upward computing framework. We relax the index definition in Section 4.4.4 to speed up the upward computation of the shortest distance and the shortest path count while guaranteeing the correctness simultaneously. We show the final index construction algorithm in Section 4.4.5.

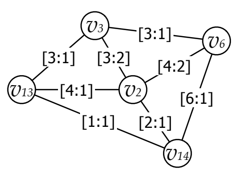
4.4.3 Graph Reduction

Preserving Shortest Path Count. For simplicity, we first introduce the DC-Graph, which is denoted by $G(V, E, \phi, \varsigma)$. Compared with the conventional road network, the DC-Graph contains an additional function ς to assign a count weight for each edge. Given a path p in a DC-Graph, we define the count value of p, denoted by $\varsigma(p)$, as the following equation.

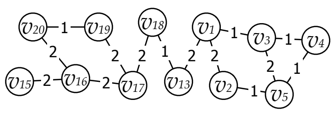
$$\varsigma(p) = \prod_{e \in p} \varsigma(e) \tag{4.4}$$

Given two vertices u and v in a DC-Graph G', to count the number of shortest paths $\mathsf{spc}_{G'}(u, v)$, we sum the count values of all their shortest paths instead of just calculating the number of paths. The DC-Graph is a generalized version of conventional road networks. Given $\varsigma(e) = 1$ for all edges e, counting paths in a DC-Graph is equivalent to counting those in the corresponding conventional graph with the same vertices and edges. Next, we define the DCP-Graph as follows.

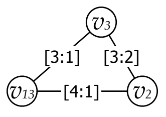
Definition 16. (DCP-Graph) *Given a road network $G(V, E, \phi)$, a DC-Graph $G'(V', E', \phi', \varsigma)$ is a DCP-Graph if $V' \subseteq V$ and for every pair $u, v \in V'$, $\mathsf{sd}_G(u, v) = \mathsf{sd}_{G'}(u, v)$ and $\mathsf{spc}_G(u, v) = \mathsf{spc}_{G'}(u, v)$.*



(a) A reduced graph G' of G



(b) Local graph for v_3 and v_{13} in the original graph G.



(c) Local graph for v_3 and v_{13} in the reduced graph G'.

Figure 4.4: Examples of optimizations in index construction based on the graph G in Fig. 1.2 and its tree decomposition in Fig. 4.2. For each edge in the subfigures (a) and (c), the label means [the distance weight ϕ: the count weight ς]. For the subfigure (b), the label means the distance of the edge.

Example 18. *Fig. 4.4 (a) shows a DCP-graph with five vertices from G in Fig. 1.2. All other vertices are reduced. Each edge has two weights namely ϕ and ς, we denote them by $[\phi : \varsigma]$. The reduced graph G' preserves the shortest distance and the shortest path count of all pairs of vertices in G'. For example, we have $\mathsf{sd}_G(v_2, v_6) = \mathsf{sd}_{G'}(v_2, v_6) = 4$ and $\mathsf{spc}_G(v_2, v_6) = \mathsf{spc}_{G'}(v_2, v_6) = 2$. We also have $\mathsf{sd}_G(v_{14}, v_6) = \mathsf{sd}_{G'}(v_{14}, v_6) = 6$ and $\mathsf{spc}_G(v_{14}, v_6) = \mathsf{spc}_{G'}(v_{14}, v_6) = 3$.*

Based on Definition 16, we propose a reduction operation for each vertex in a graph G and transform G to a DCP-Graph as shown in Algorithm 8. Note that for any edge e whose count weight $\varsigma(e)$ is not defined, we initialize it as 1 in the algorithm. For every pair of neighbors v, w of u in G, if 1) the edge (v, w) does not exist earlier, or 2) the edge (v, w) exists but a shorter edge appears (line 2), we create the edge and replace the edge distance weight and the edge count weight by $\phi(v, u) + \phi(u, w)$ and $\varsigma(v, u) \times \varsigma(u, w)$, respectively. If there is an existing edge (v, w) and the distances are the same (line 6), we increase the corresponding edge count weight in line 7.

Lemma 12. *Algorithm 8 returns a DCP-Graph of G.*

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

- Case 1: the shortest path from u to v in G does not pass through w. Thus, we know $\mathsf{sd}_G(u,v) = \mathsf{sd}_{G'}(u,v)$ and $\mathsf{spc}_G(u,v) = \mathsf{spc}_{G'}(u,v)$ as the shortest path in G is also the shortest path in G'.

- Case 2: the shortest path from u to v in G passes through w. In this case, suppose that the shortest path between u and v is $(u, \ldots, w_i, w, w_j, \ldots, v)$. As Algorithm 8 eliminates w in G', and inserts a new edge (w_i, w_j) with $\phi(w_i, w_j) = \phi(w_i, w) + \phi(w, w_j)$ and $\varsigma(w_i, w_j) = \varsigma(w_i, w) \times \varsigma(w, w_j)$ (if there is no edge $(w_i, w_j) \in G$ or $\phi(w_i, w_j) > \phi(w_i, w) + \phi(w, w_j)$ in G) or $\varsigma(w_i, w_j) \leftarrow \varsigma(w_i, w_j) + \varsigma(w_i, w) \times \varsigma(w, w_j)$ (if there is already $\phi(w_i, w_j) = \phi(w_i, w) + \phi(w, w_j)$ in G).

Hence, both the distance and the corresponding edge count are preserved, and Algorithm 8 returns a DCP-Graph of G. □

Graph Reduction in Tree Decomposition. We can reduce the graph in tree decomposition in a natural way, given that we need to connect every pair of neighbors when eliminating each vertex in tree decomposition. The revised tree decomposition is called DCP-Tree Decomposition and is shown in Algorithm 9. Lines 4–10 iteratively reduce each vertex from the graph. The vertex u is removed from G in line 8. $\pi(\cdot)$ is an array to record the removing order of all vertices. Performing the operator $\ominus$ for all vertices would not increase the time complexity of tree decomposition.

Lemma 13. *The time complexity of Algorithm 9 is $O(n \cdot w^2 + n \log n)$.*

Proof. In Algorithm 9, the dominant cost for each vertex is maintaining and selecting the vertex with the smallest degree in V', which costs $O(n \cdot \log n)$ time. In line 8, the $\ominus$ operator costs $O(w^2)$ time. Thus, the overall time complexity of Algorithm 9 is $O(n \cdot w^2 + n \log n)$. □

Algorithm 9: DCP-TreeDecomposition

```
   Input: $G(V, E, \phi)$
   Output: Tree decomposition $T_G$
1  $T_G \leftarrow \emptyset$
2  foreach $e \in E$ do $\varsigma(e) = 1$;
3  $V' \leftarrow V, i \leftarrow 1$;
4  while $V \neq \emptyset$ do
5      $u \leftarrow$ the vertex with the smallest degree in $V$;
6      $X(u) \leftarrow \{u\} \cup N(u)$;
7      create a tree node $X(u)$ in $T_G$;
8      $G \leftarrow G \ominus u$;
9      $\pi(u) = i$;
10     $i \leftarrow i + 1$;
11 foreach $u \in V'$ do
12     if $|X(u)| > 1$ then
13         $v \leftarrow \arg\min_{v \in X(u) \setminus \{u\}} \pi(v)$;
14         set $X(v)$ be the parent of $X(u)$ in $T_G$;
15 return $T_G$
```

4.4.4 Relaxing Convex Shortest Path

In this section, we focus on the phase of computing convex shortest path count. We simplify the logic of index construction by relaxing the convex shortest path count in the index definition. Given a graph G, its tree decomposition T_G and two vertices u, v with $\mathsf{Depth}(u) \neq \mathsf{Depth}(v)$, the local graph of u and v is the induced subgraph of $\mathsf{T}(u) \cup \mathsf{T}(v)$ in G. That is to say, the local graph contains all the vertices who have tree depths no smaller than both u and v.

Example 19. *Given graph G in Fig. 1.2 and the tree decomposition in Fig. 4.2, the local graph for v_2 and v_{13} in $V(G)$ is shown in Fig. 4.4 (b). As we can see from the figure, all the vertices are from the subtree rooted by v_{13}, given that $\mathsf{T}(v_2) \subset \mathsf{T}(v_{13})$.*

Based on the concept of the local graph, we relax the definition of the convex shortest path as follows.

Definition 17. (LOCAL SHORTEST DISTANCE AND LOCAL SHORTEST PATH COUNT) *Given a tree decomposition T_G of graph $G(V, E)$, the local shortest distance (resp. shortest path count) between two vertices u and v, denoted by $\mathsf{sd}(u, v)^-$ (resp. $\mathsf{cspc}(u, v)^-$), is the shortest distance (resp. shortest path count) of u and v in their local graph.*

Example 20. *Let us continue Example 19. In the local graph (Fig. 4.4(b)), for vertices v_2 and v_{13}, we can find the shortest path $p_1 = (v_2, v_1, v_{13})$. The shortest distance $\mathsf{sd}(v_2, v_{13})^- = 4$, and the local shortest path count is $\mathsf{cspc}(v_2, v_{13})^- = 1$. By contrast, When we look at the full graph G in Fig. 1.2, we find the shortest path $p_2 = (v_2, v_{14}, v_{13})$ which has a smaller distance than the local shortest path p_1. We can also see that $\mathsf{Depth}(v_{14}) < \min(\mathsf{Depth}(v_2), \mathsf{Depth}(v_{13}))$ in T_G and p_2 is not a convex shortest path. Therefore, the shortest distance between v_2 and v_{13} in the full graph G is $\mathsf{sd}(v_2, v_{13}) = 3$, and the convex shortest path count is $\mathsf{cspc}(v_2, v_{13}) = 0$,*

Lemma 14. *Given a convex shortest path p between two vertices u and v, p is a local shortest path in the local graph of u and v.*

Lemma 15. *Algorithm 6 is correct if we replace the shortest distance and the convex shortest path count in TL-Index (Definition 15) by the local shortest distance and the local shortest path count, respectively.*

Proof. Given vertices u and v, $w \in \mathsf{CA}(u, v)$. We first consider the shortest paths between u and w. If there is any convex shortest path between u and w, we have $\mathsf{sd}(u, w) = \mathsf{sd}(u, w)^-$ and $\mathsf{cspc}(u, w) = \mathsf{cspc}(u, w)^-$ based on Lemma 14. If there is no convex shortest paths between u and w, we have $\mathsf{sd}(u, w) < \mathsf{sd}(u, w)^-$. Based on Theorem 9, there must be at least one vertex $w' \in \mathsf{A}(w)$ that satisfies $\mathsf{sd}(u, w) = \mathsf{sd}(u, w')^- + \mathsf{sd}(w', w)^-$ and $\mathsf{spc}(u, w) =$

$\sum_{w'} \mathsf{cspc}(u, w')^{-} \cdot \mathsf{cspc}(w', w)^{-}$. We have the same result for the shortest paths between w and v. Thus, Algorithm 6 is correct. □

Based on Lemma 15, we compute the local shortest distance and the local shortest path count from each vertex to its ancestors. Compared to computing the exact shortest distance and convex shortest path count, the local values reduce significant search space in the index construction. Recall that based on the graph reduction techniques in Section 4.4.3, we have a reduced graph preserving the shortest distance and the shortest path count during the index construction. By combing the reduction and the local computation ideas, computing the local shortest distance and local shortest path count is conducted in a local reduced subgraph.

Example 21. *An example to compute values from v_3 to v_{13} is provided in Fig. 4.4 (c). When processing v_3, many other vertices have been reduced as shown in G' of Fig. 4.4 (a). Based on Lemma 15, we only care the induced subgraph of all vertices in G' without a depth smaller than v_{13}. The final search space is shown in Fig. 4.4 (c).*

4.4.5 The Final Algorithm

As shown in Algorithm 10, we first compute the tree decomposition and the count weight for all new edges (shortcuts) inserted to the graph. Then, for each vertex u, we compute the local shortest distance and the local shortest path count from u to its ancestors in lines 3–12. Specifically, $X(u) \setminus \{u\}$ in line 4 are neighbors of u in the DCP-Graph after performing the reduction operation $\ominus$ for all subtree vertices of u. Note that for every vertex u' in $X(u) \setminus \{u\}$, the values have been computed in earlier rounds. Therefore, we use each neighbor u' to compute the distance and the count from u to each v (line 6 and line 7), since

Algorithm 10: TL-Construct*

```
Input: A road network $G(V, E, \phi)$
Output: The TL-Index of $G$
1  $T_G \leftarrow$ DCP-TreeDecomposition($G$);
2  foreach $X(u) \in T_G$ in a top-down manner do
3      foreach $v \in \mathsf{A}(u)$ do
4          foreach $u' \in X(u) \setminus \{u\}$ do
5              if Depth($u'$) < Depth($v$) then continue;
6              $d \leftarrow \phi(u, u') + \mathsf{sd}(u', v)^-$;
7              $c \leftarrow \varsigma(u, u') \cdot \mathsf{cspc}(u', v)^-$;
8              if $d < \mathsf{sd}(u, v)^-$ then
9                  $\mathsf{sd}(u, v)^- \leftarrow d$;
10                 $\mathsf{cspc}(u, v)^- \leftarrow c$;
11             else if $d = \mathsf{sd}(u, v)^-$ then
12                 $\mathsf{cspc}(u, v)^- \leftarrow \mathsf{cspc}(u, v)^- + c$;
```

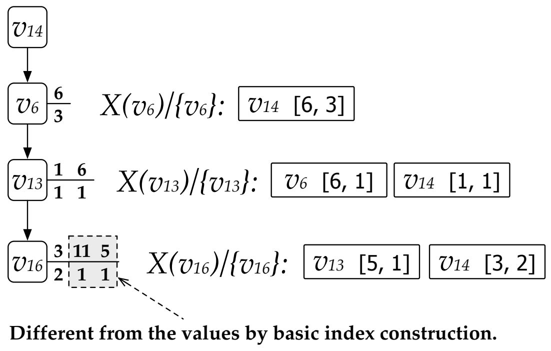


Figure 4.5: An example of TL-Construct*.

the shortest path from u to v must pass at least one of the neighbors. We do not need to consider the neighbor u' with a smaller depth than the target vertex v in line 5 since u' is not in the local graph. During computing distance via neighbors, we replace the shortest distance and shortest path count if a shorter distance is found (lines 8–10). We increase the count if the same shortest distance is found (lines 11–12).

Example 22. *We show a running example with a partial TL-Index in Fig. 4.5.*

On the right side, we list the distance weight and the corresponding count weight derived by the DCP-TreeDecomposition *(Algorithm 9). Given* $u = v_6$ *in line 2, we need to check its ancestor* v_{14}*. In line 6 and 7, we have* $d = \phi(v_6, v_{14}) + \mathsf{sd}(v_{14}, v_{14})^- = 6{+}0 = 6$ *and* $c = \varsigma(v_6, v_{14}) {\cdot} \mathsf{cspc}(v_{14}, v_{14})^- = 1{\times}1 = 1$*. Given that* $\mathsf{sd}(v_6, v_{14})^-$ *is initialized as* ∞*, we update* $\mathsf{sd}(v_6, v_{14})^- = 6$ *and* $\mathsf{cspc}(v_6, v_{14})^- = 1$*. For the iteration of* $u = v_{16}$ *in line 2, when* $v = v_{14}$ *in line 3, we update* $\mathsf{sd}(v_{16}, v_{14})^- = 3$ *and* $\mathsf{cspc}(v_{16}, v_{14})^- = 2$*, which is straightforwardly derived by* DCP-TreeDecomposition*. Then, for the labels from* v_{16} *to* v_6 *(*$v = v_6$ *in line 3), we calculate* $d = \phi(v_{16}, v_{13}) + \mathsf{sd}(v_{13}, v_6)^- = 5 + 6 = 11$ *and* $c = \varsigma(v_{16}, v_{13}) \cdot \mathsf{cspc}(v_{13}, v_6)^- = 1$ *in lines 6–7. Then, we update the labels correspondingly in line 9-10. Note that the labels from* v_{16} *to* v_6 *and* v_{13} *differ from those in the TL-index in Fig. 4.3. This is because we only store the local shortest distance and the local shortest path count by our final algorithm.*

Theorem 13. *The time complexity of Algorithm 10 is* $O(n \log n + nhw)$*, where* n *is the number of vertices,* h *is the* treeheight*, and* w *is the* treewidth*.*

Proof. In line 1, Algorithm 9 takes $O(n \cdot w^2 + n \log n)$. From line 2 to line 12, there are three loops and the time complexity is $O(nhw)$. The overall time complexity is $O(n \log n + nhw)$.

□

4.5 Experiments

We conduct extensive experiments to evaluate our methods against the state-of-the-art approach. All the algorithms are implemented in C++ with -O3 optimization, and the experiments are conducted on a Linux machine with an Intel Xeon Gold 6248 2.5GHz CPU and 768GB RAM. We evaluate all the algorithms on fourteen real-world graphs as shown in Table 4.2. GRD is the power grid network of the western states in the USA[3]. SYD is a public transportation network containing all the public transportation stops in Sydney [62]. All the rest are road networks from DIMACS [4]. The statistics are reported in Table 4.2.

Compared Algorithms. In our experiments, we compare our algorithms with the state-of-the-art solution HL-Index [115] for the shortest path counting query processing on real-world networks. We obtain the C++ code from the authors and revise their index construction algorithm to handle weighted graphs by replacing the Breadth-First Search with the Dijkstra's Search, given that only unweighted graphs are considered in their implementation. We compare the following algorithms in experiments.

- HL-Index: The hub-labeling index in [115].
- HL-Query: The query processing algorithm in [115].
- HL-Construct: The index construction algorithm in [115].
- TL-Index: Our index structure (Definition 15).
- TL-Query: Our query processing algorithm (Algorithm 6).
- TL-Construct: Our basic indexing algorithm (Algorithm 7).
- TL-Construct*: Our optimized indexing algorithm (Algorithm 10).

Note that the hub-labeling method cannot finish indexing the USA dataset within 24 hours, thus we do not report the results.

Exp-1: Query Time. We compare the average query time between TL-Query and HL-Query. For each dataset, we randomly generate one million queries. We

[3]http://konect.cc/

[4]http://www.diag.uniroma1.it//challenge9/download.shtml

Table 4.2: Statistics of road networks.

Name	Description	n	m	h	w
GRD	US Power Grid	4,941	6,594	72	25
SYD	Public Transport	24,063	28,695	194	79
NY	NYC	264,346	733,846	505	134
BAY	Bay Area	321,270	800,172	403	108
COL	Colorado	435,666	1,057,066	465	146
FLA	Florida	1,070,376	2,712,798	520	136
NW	Northwest US	1,207,945	2,840,208	548	146
NE	Northeast US	1,524,453	3,897,636	828	219
CAL	CA and NV	1,890,815	4,657,742	713	215
LKS	Great Lakes	2,758,119	6,885,658	1325	370
EUS	Eastern US	3,598,623	8,778,114	1022	272
WUS	Western US	6,262,104	15,248,146	1041	326
CUS	Central US	14,081,816	34,292,496	2433	660
USA	Full US	23,947,347	58,333,344	2564	693

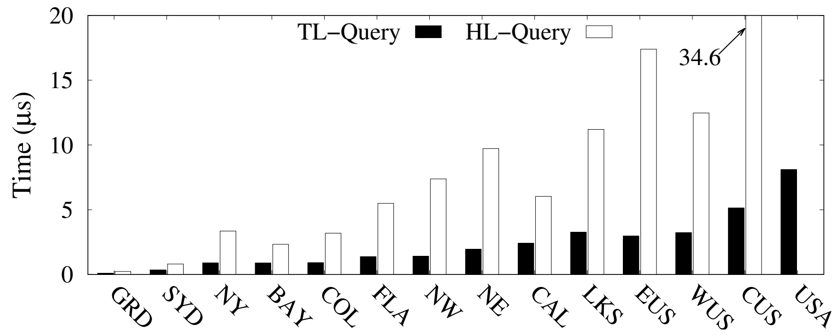


Figure 4.6: Query time.

record the query time of each algorithm and report the average time in Fig. 4.6. We also show the speedup of TL-Query over HL-Query in Fig. 4.7. As we can see from Fig. 4.6 and Fig. 4.7, TL-Query is significantly faster than HL-Query on all datasets. This is mainly because TL-Query only needs to visit a small number of labels compared with HL-Query in counting shortest paths. For example, in the CUS dataset, TL-Query takes 5.14 µs on average while HL-Query requires 34.59 µs. TL-Query is 6.73 times faster than HL-Query.

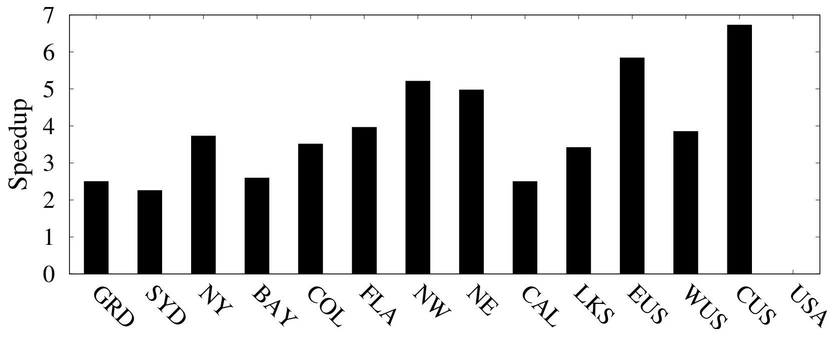


Figure 4.7: TL-Query speedup over HL-Query.

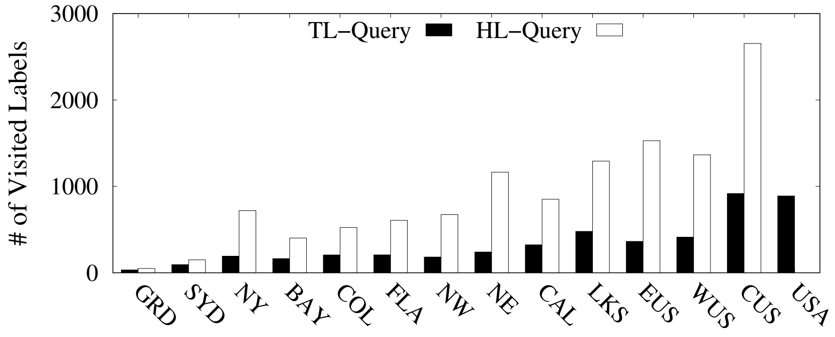
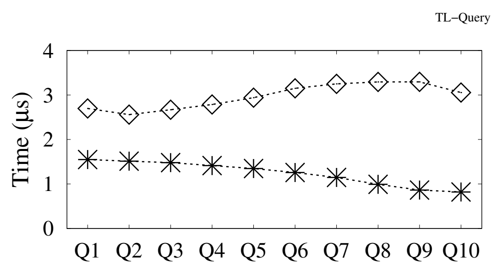
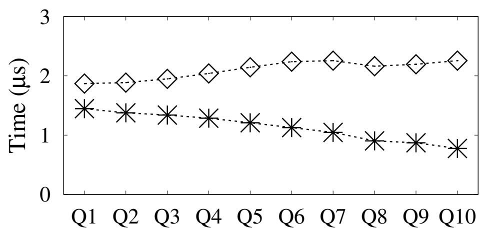


Figure 4.8: Number of visited labels in query processing.

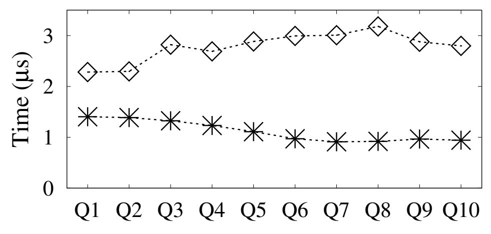
Exp-2: Visited Label Size in Query Processing. When processing queries, both TL-Query and HL-Query need to check and compare a number of labels to derive the final result. We evaluate the number of visited labels in query processing in this evaluation. Similar to the query processing time, we report the average value of one million random queries. The results are shown in Fig. 4.8. We can see that the average label size of TL-Query is significantly smaller than HL-Query on all the datasets. For example, on the NE dataset, the average size of TL-Query is 240, and that of HL-Query is 1164. That means, on average, HL-Query needs to visit 1164 labels to finish the query, while TL-Index only needs to visit 240 labels. This result is consistent with our query performance evaluation in Fig. 4.6.



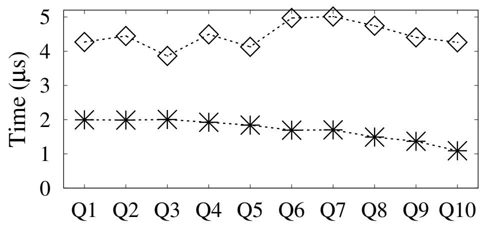
(a) NY



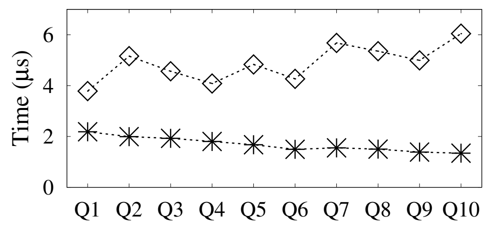
(b) BAY



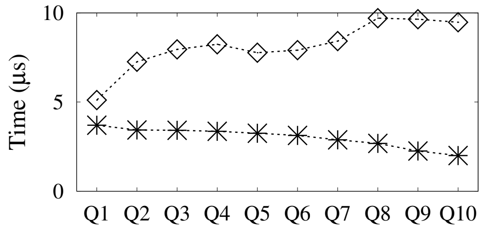
(c) COL



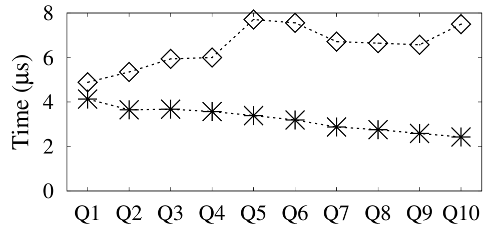
(d) FLA



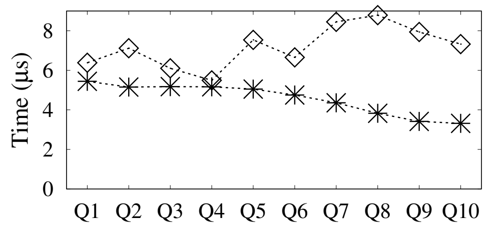
(e) NW



(f) NE

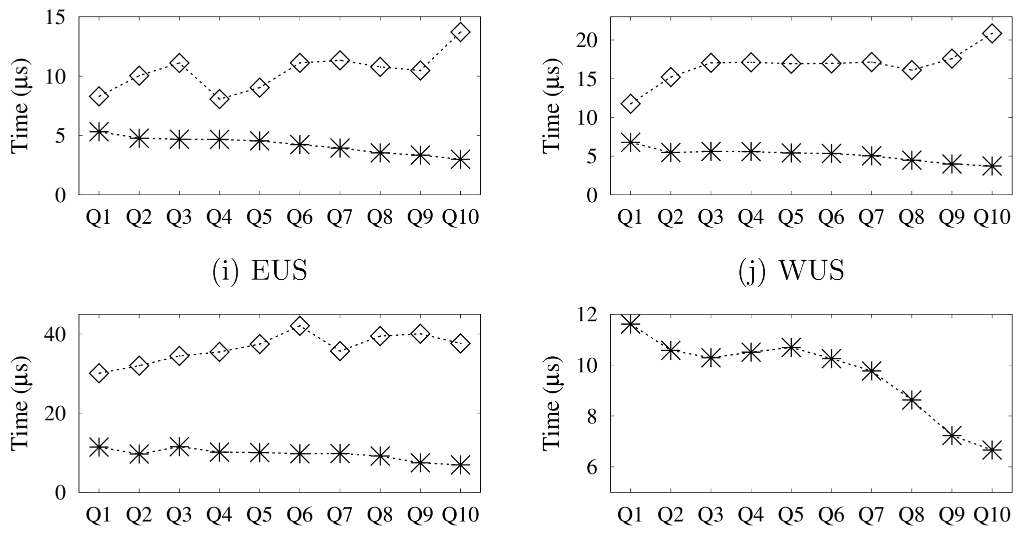


(g) CAL



(h) LKS

Figure 4.9: Query processing time varying query distance.



(i) EUS (j) WUS

(k) CUS (l) USA

Figure 4.9: Query processing time varying query distance (continued).

Exp-3: Varying Query Distance. We further test the query efficiency of the algorithms by varying the query distance. Following Ouyang et al.'s experimenatl setting [76], we generate ten groups of queries, $Q_1, Q_2, \ldots, Q_{10}$, by distances for each dataset. Specifically, we set l_{min} to be 1,000 (1 kilometer) and l_{max} to be the maximum resulting distance of any pair of vertices in the graph. Let $x = (l_{max}/l_{min})^{1/10}$. A randomly generated query belongs to Q_i if its distance between the source and target vertices falls in the range $(l_{min} \times x^{i-1}, l_{min} \times x^{i}]$. We randomly generate 10,000 queries for each group and each dataset. We record the average time of TL-Query and HL-Query for the 10,000 queries and report the results by varying query distance from Q_1 to Q_{10} in Fig. 4.9.

We can see that our solution TL-Query outperforms HL-Query in all cases. We also make the following observations. First, when the query distance increases, the time cost of HL-Query increases. For example, on the BAY dataset, HL-Query takes 1.869 µs on average to process Q_1 queries. When handling Q_{10} queries, HL-Query takes 2.256 µs. This is because, in HL-Index, two faraway vertices may have more common labels than two close vertices. Second, in contrast to HL-Query, when the query distance increases, the time cost of TL-Query decreases. On Q_1 queries, TL-Index takes 1.447 µs. While on Q_{10} queries, it only requires 0.776 µs. This is because, in TL-Query, the dominant cost is querying the label from the LCA to the root. Intuitively, when the distance between two vertices is long, their LCA is more likely to have a lower Depth in the tree decomposition. Therefore, our proposed method visits fewer labels when the query distance is longer. Third, when the distance between the source and the target vertices is relatively large, our method TL-Query is significantly faster than HL-Query.

Exp-4: Index Construction. The index construction time for the algorithms HL-Construct, TL-Construct*, and TL-Construct is shown in Fig. 4.10. When the dataset size increases, the indexing time of all the algorithms also increases. Among all three algorithms, our TL-Construct* is the most efficient in indexing. Our solution TL-Construct* is 8–20 times faster than HL-Construct on large road networks. For example, on EUS, WUS, and CUS, our TL-Construct* is 19.41, 15.98, and 20.19 times faster than HL-Construct, respectively. Note that HL-Construct cannot finish indexing USA within 24 hours. Compared to TL-Construct, our proposed TL-Construct* is also several times faster. For example, on EUS, WUS, CUS, and USA, our TL-Construct* is 9.34, 11.61, 14.27, and 13.92 times faster than TL-Construct. Meanwhile, our baseline algorithm is also faster than HL-Construct for most datasets, thanks to the tree structure. The results show the advance of our indexing algorithm.

Exp-5: Index Size. We report the index size for the HL-Index and our proposed TL-Index. The results are shown in Fig. 4.11. When the size of the network increases, the size of the index also increases for both algorithms. The index size of our TL-Index is smaller than that of HL-Index on most datasets.

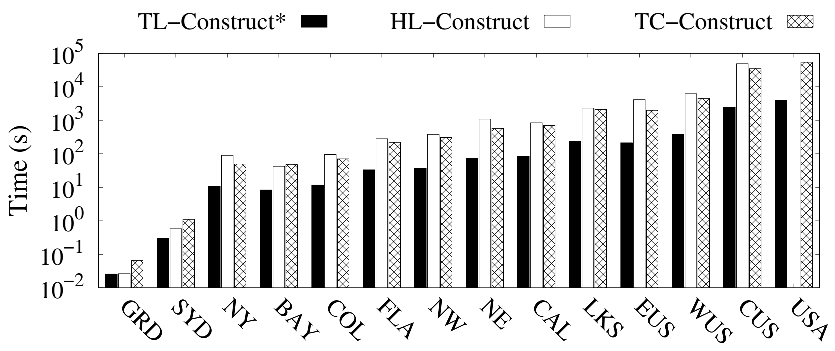


Figure 4.10: Index construction time.

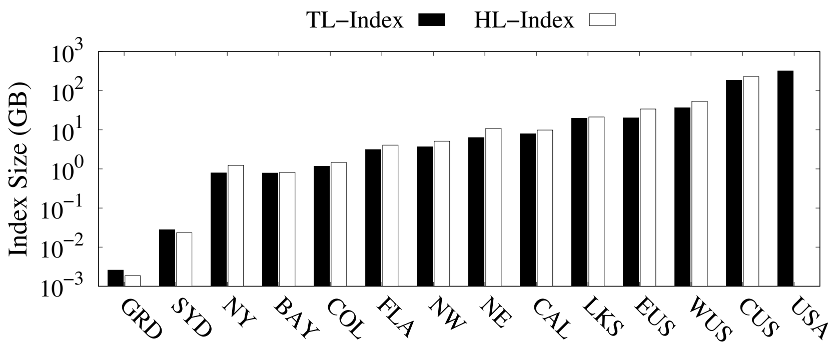
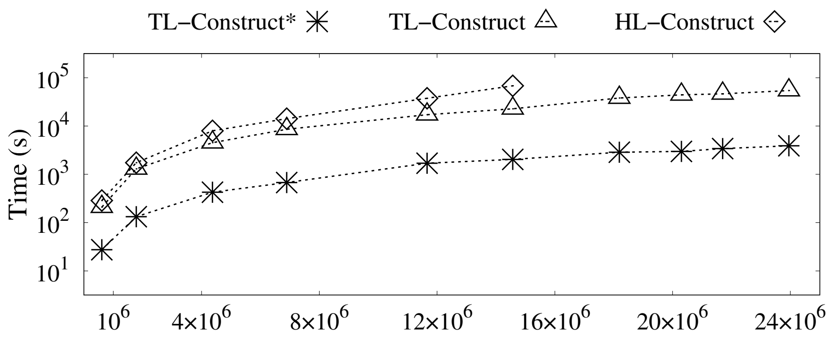


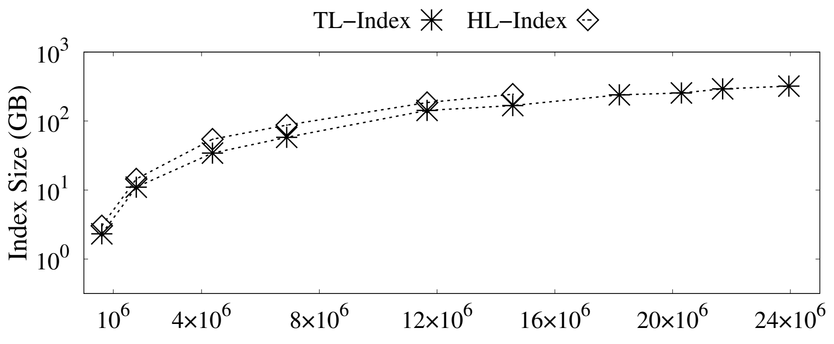
Figure 4.11: Index size (GB).

Specifically, on most of the datasets, the index size of TL-Index is 20%–40% smaller than that of HL-Index. This is because HL-Index is designed based on a total vertex order. Many labels are precomputed for each vertex.

Exp-6: Indexing Scalability. We test the indexing time and the index size when varying the dataset size (the number of vertices in the road networks) from 10^6 to 24×10^6, following Ouyang et al.'s work [76]. To conduct the study, we partition the map of the United States into 10×10 grids, from which we generate ten road networks (G_1 to G_{10}) by using the $1 \times 1, 2 \times 2, \ldots, 10 \times 10$ grids situated in the central region of the map. We report the indexing time and the index size in Fig. 4.12 (a) and Fig. 4.12 (b), respectively. When the dataset increases from 10^6 to 24×10^6, the indexing time increases stably for



(a) Indexing time.



(b) Index size.

Figure 4.12: Scalability testing.

all algorithms. Our TL-Construct* always outperforms the other two algorithms. HL-Construct cannot finish the datasets whose sizes are greater than 18×10^6 within 24 hours. Therefore, they are not reported in the figure.

Exp-7: The Number of Shortest Paths. We report the average and the maximum shortest path count in Fig. 4.13 and Fig. 4.14. Fig. 4.13 shows that the larger the road network, the greater the shortest paths count. This is because the shortest paths on larger road networks may have more hops, and we are more likely to get the shortest paths with the same shortest distance. Generally, for small networks, like NY, BAY, and COL, the average shortest path count is around 1.5. For medium networks, like FLA and WUS, the average shortest path count is about 3–7. For large networks, like CUS and USA, the average shortest path count is 52 and 97, respectively. Fig. 4.14 shows the average and maximum

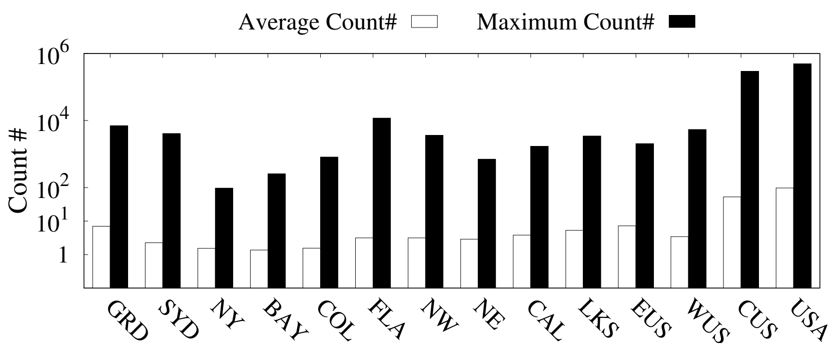


Figure 4.13: Shortest path count on different graphs.

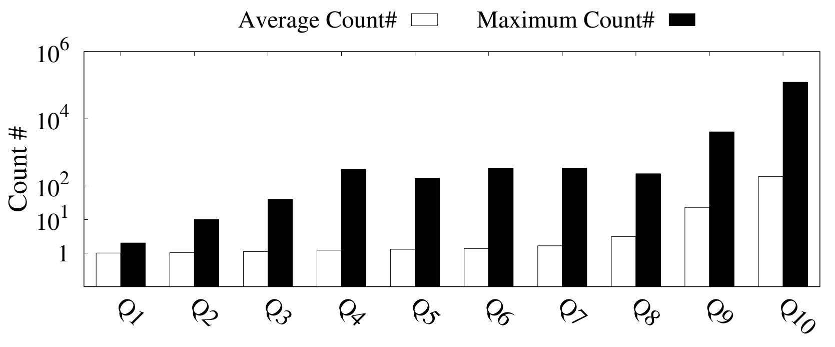


Figure 4.14: Shortest path count varying distance.

shortest path count number varying the shortest distance on the USA graph. The results on other graphs show a similar trend. The ten groups of queries are the same as those in Exp-3. The shortest path count number increases with the increase of the shortest distance, which is in accordance with the results above.

4.6 Chapter Summary

In this chapter, we study the shortest path counting query problem on road networks. We devise a novel index structure named TL-Index based on the tree decomposition method. The TL-Index supports efficient shortest path counting queries that outperform the state-of-the-art method. Moreover, we also present efficient index construction algorithms. The experimental results demonstrate the efficiency of our proposed algorithms.

Chapter 5

SHORTEST PATH COUNTING FOR DYNAMIC ROAD NETWORKS

5.1 Chapter Overview

In this chapter, we continue our study on the shortest-path counting query on road networks. Specifically, we aim to analyze the index maintenance issue in the index proposed in Chapter 4 when the weight on the road network changes. This chapter is structured as follows. Section 5.2 provides the problem definition and a general updating framework. Then, in Section 5.3, we show improved updating techniques for weight decreasing and increasing, respectively. We also discuss the design and impact of the local distance in TL-Index. Section 5.4 evaluates the proposed algorithms and Section 5.5 concludes this chapter.

5.2 TL-Index Maintenance

We first introduce the problem definition of TL-Index maintenance in Section 5.2.1. We then provide a general framework that locates the elements impacted by the updates in the TL-Index.

5.2.1 Problem Statement

Following Chapter 4, we still consider a road network $G = (V, E, \phi)$ which is a degree-bounded, connected, and weighted graph. $V(G)$ is a set of vertices, $E(G)$ is a set of edges, and $\phi : e \in E(G) \mapsto N^+$ is a weight function for each edge. We also have a TL-Index T which is already built by the construction methods proposed in Chapter 4. For each vertex $v \in V$ in graph G, there is a corresponding tree node $X(v)$ in $T(G)$ that holds the distance and count information from v to all its ancestors in $T(G)$. When changes are made to the weights on some edges in G, the task is to determine which nodes in T have labels that are impacted by the updates and update the corresponding label values. The problem of maintaining the TL-Index is formally defined as follows.
Problem Definition Given a road network G and its corresponding TL-Index T, the TL-Index maintenance problem aims to efficiently update the labels in T when G is dynamically updated.

5.2.2 An Up-and-Down Framework

In this section, we first review the TL-Index construction process, and then propose an up-and-down updating framework.

In Chapter 4, the tree decomposition T_G of a graph G is constructed by removing the vertex with the smallest degree. This construction method ensures that the structure of T_G is dependent solely on the graph structure, and not on the weights of the edges in G. Therefore, any changes made to the weights of the edges in G will not affect the structure of T_G. Instead, only the labels on the nodes of T_G, specifically the shortest distance and convex shortest path count labels, need to be updated.

Let us first reconsider the graph reduction process in Section 4.4.3. Given a road network G, a DCP-graph G' of G, which is a reduced graph of G, is generated in the graph reduction process. In each reduction step, we eliminate a vertex W from the DCP-graph G'. When a vertex w is removed from G', the tree node $X(w)$ is generated in T_G that contains all the edges connecting w to its neighbors in G' before the removal. Suppose the vertices u and v are w's two arbitrary neighbors before removing w, i.e., $u, v \in N(w)$, there are two possible conditions after removing w. First, if there is no edge between u and v in G', i.e., $(u, v) \notin G'$, we generate a new edge (u, v) with $\phi(u, v)$ and $\varsigma(u, v)$ according to the ϕ and ς values of (u, w) and (w, v), and insert (u, v) to the reduced graph. Second, if there is already the edge between u and v in G', we compare the weights of $\phi(u, v)$ and $\varsigma(u, v)$ between the current (u, v) and the newly generated values according to the ϕ and ς values of (u, w) and (w, v), and update $\phi(u, v)$ and $\varsigma(u, v)$ accordingly.

In the index maintenance process, whenever the values of (w, u) or (w, v) change, we need to update $\phi(u, v)$ and $\varsigma(u, v)$ for edge (u, v). This is because $\phi(u, v)$ and $\varsigma(u, v)$ are dependent on the values of (u, w) and (w, v). Additionally, changes to $\phi(u, v)$ or $\varsigma(u, v)$ will also affect other edges that depend on (u, v), leading to a chain of updates that continues until no further updates are necessary. In the following, we denote the edges on the DCP-graph G' as shortcuts for clearer illustration. Thus, when an edge in G changes its weight, our aim is to iteratively update shortcuts in G', until 1. there is no more change of ϕ and ς of all the affected nodes; 2. there is no more edge dependent on the changed edge, which means either node of the changed edge reaches the root of T_G.

During such an iterative update process, it is necessary to locate and recompute the values of the affected shortcuts. To facilitate the recalculation, we define the following shortcut table ST.

Definition 18. *Given two vertices u and v, the shortcut table ST records the eliminated vertices that affect the shortcut between u and v, i.e., $\mathsf{ST}(u,v) = \{w_1, w_2, \ldots, w_n\}$, where w_i affects (u,v) when it is eliminated in the DCP-Tree Decomposition.*

We can efficiently include the construction of the shortcut table ST into the DCP-Tree Decomposition process by inserting the eliminated vertex w into $\mathsf{ST}(u,v)$ for each pair of its neighbors $u, v \in N(w)$, at the time of vertex elimination.

With the help of the shortcut table ST, the process of recalculating the values of potentially impacted shortcuts is made more efficient. The table enables us to promptly recompute the values of $\phi(u,v)$ or $\varsigma(u,v)$ for a shortcut (u,v) when it is affected by the previous iteration by examining the eliminated vertices in ST that have an impact on (u,v).

In the updating process, when the weight of an edge (u,v) is changed, we utilize the shortcut table ST to recompute the ϕ and ς values of the shortcut between u and v, denoted as $\phi'(u,v)$ and $\varsigma'(u,v)$, and compare them to the original values of $\phi(u,v)$ and $\varsigma(u,v)$. If the distance or count values of the shortcut (u,v) are changed, then it is possible that the shortcuts between each vertex $w \in X(u)$ and v, supposing $\pi(u) < \pi(v)$, may be impacted. In such cases, we mark the shortcut between v and w that needs to be checked for each $w \in X(u)$. Then, following a bottom-up order along the tree structure, we check and update the marked shortcut iteratively.

After updating all the shortcuts that need to be checked, we have the nodes that have been affected by the update and the up-to-date distance and count values for all the shortcuts. The next step is to recompute the shortest distance labels and the convex shortest path count labels for each node that was impacted by the update and any further nodes that may have been affected as a result. This process is also performed iteratively.

Following the update order of shortcuts and labels, we propose an up-and-down maintenance framework, which is shown in Algorithm 11.

Algorithm 11: TL-Update

```
Input: Graph $G$ and the TL-Index $T$, the edge $(u, v)$ with $\pi(u) < \pi(v)$,
       and its new weight $\phi'_G(u, v)$
Output: The updated TC-index
1  $\phi_G(u, v) \leftarrow$ the original weight of edge $(u, v) \in G$;
2  $\mathsf{AFF} \leftarrow \emptyset$
3  $Q \leftarrow$ an empty priority queue
4  push $(u, v)$ into $Q$
5  while $Q$ is not empty do
6      $(u, w) \leftarrow Q.pop()$ with minimal $\pi(u)$
7      $\mathsf{AFF} \leftarrow \mathsf{AFF} \cup \{u\}$
8      recompute $\phi(u, w)$ and $\varsigma(u, w)$ with $\mathsf{ST}$
9      if $\phi(u, w)$ or $\varsigma(u, w)$ changes then
10         foreach $v \in X(u)$ do
11             if $\pi(v) > \pi(w)$ then
12                 push $(w, v)$ into $Q$
13             else if $\pi(v) < \pi(w)$ then
14                 push $(v, w)$ into $Q$
15 Let $p \in \mathsf{AFF}$ s.t. $\pi(p) = \max \pi(v) \forall v \in \mathsf{AFF}$, $UpdateLabels(p)$
16 Procedure UpdateLabels(p)
17     Update $\mathsf{sd}(p, a)$ and $\mathsf{cspc}(p, a)$ for each $a \in A(p)$
18     foreach child $X_c$ of $X_p$ do
19         $UpdateLabels(c)$;
```

In Algorithm 11, we first initialize an empty priority queue Q that prioritizes the minimal $\pi(u)$. We utilize the priority queue to realize the bottom-up updates of the shortcuts. Each time we pop up and check the shortcut with the minimal π, which indicates the shortcut is at the bottom of all the possibly affected shortcuts. In line 3, we push the edge (u,v) whose weight has changed into Q.

Lines 4-12 show the bottom-up process of updating shortcuts. Each time, Q pops the shortcut that could be affected with the lowest $\pi(u)$. We mark u affected by putting it into AFF in line 7. Then, we recompute the $\phi(u, w)$ and $\varsigma(u, w)$ with ST in line 8. If either $\phi(u, w)$ or $\varsigma(u, w)$ changes, then for each $v \in X(u)$, the values between v and w need to be checked. Hence, we push them into Q.

Finally, we update the shortest distance and convex shortest path count labels of all the affected nodes with the help of ϕ, ς and AFF. The process starts from the vertex $p \in \mathsf{AFF}$ with the largest π in T_G. We iterate over the sub-tree rooted at $N(p)$ in a top-down manner. For each $N(v) \in T_G(p)$, we recompute the sd and cspc labels from v to all its ancestors.

5.3 An Up-forward Updating Approach

Although Algorithm 11 avoids recomputing the TL-Index from scratch, it may still be inefficient due to excessive, unnecessary computations.

Firstly, in real-world applications, we may have multiple edge changes, and it is inefficient to update the whole index every time an edge is changed. Suppose updating one edge costs $O(\mathit{update})$ time, and then it costs $k \times O(\mathit{update})$ time when we update k edges. When k is large, such an updating process may cost a large amount of time. However, during the k times' updating process, there could be many overlapping areas that are repeatedly updated. Thus, to avoid redundant computation, we can devise a batched updating algorithm that updates the index only once with a batch of edge changes.

Secondly, for the shortcut updates, we first record the impacted shortcuts and check them one by one by recomputing the ϕ and ς values with ST. However, it is unnecessary to recompute each shortcut with ST. For many cases, each time we judge whether a shortcut could be affected, we can derive its values after imposing the change by carefully designing the value updating strategy and up-forward the values to the check step.

Thirdly, for the label updates, Algorithm 11 updates the labels of the whole subtree of the highest found node in the tree, which is inefficient. When the found node is high up in the tree, the whole tree will be examined. To avoid such redundant calculations, it is advisable to adopt an up-forward updating technique for the label values.

Considering the updating framework depicted in Algorithm 11, the updating process involves the transmission of change information among edges, shortcuts, and labels. Changes in edges lead to changes in some shortcuts, which in turn affect further shortcuts. The changes in all shortcuts result in alterations to some labels, and ultimately, changes in some labels trigger changes in other labels. This updating process can be divided into four steps.

Based on this process, we propose the optimized index update methods. To differentiate between the effects of weight increase and decrease, we have developed separate methods for each case. The term shortcut is also referred to as super edge. In the following, these two terms will be used interchangeably.

5.3.1 Index Maintenance for Weight Decrease

To address the edge weight decrease case, we introduce an index maintenance technique consisting of four steps: Edge-Super Decrease, Super-Super Decrease, Super-Label Decrease, and Label-Label Decrease.

Edge-Super Decrease The algorithm Algorithm 12 presents a method for batch edge weight decrease processing. It takes as input a list of edges whose weights have decreased and handles them collectively. Similar to Algorithm 11, the algorithm utilizes a priority queue P to record the shortcuts whose weights have changed. This difference is that we store the edge along with the original distance and count values, and if a shortcut (u, v) has already been added to P, we do not add it again.

Each time it pops an edge whose weight has decreased in line 3. Then it compares the new weight to the distance value ϕ of the shortcut between them. If the new weight is smaller, which means we find the shortest path with a shorter distance. We update the shortcut and mark it as updated by pushing it as well as its old distance and count values into the priority queue P. If the new weight is found to be equal to the distance value of the shortcut, it indicates the existence of a new route with an equivalent shortest distance. In this scenario, the previous count value is incremented by one, and the shortcut is marked as updated by inserting the previous distance and count values into the priority queue P. As the distance remains unchanged, it is straightforward to deduce that only the count value has been altered when processing this shortcut. Note that for an edge (u, v), we always assume $\pi(u) < \pi(v)$.

Super-Super Decrease After executing Algorithm 12, all the shortcuts that are affected by the list of edges whose weights have decreased are stored in the priority queue P. Subsequently, the algorithm Algorithm 13 calculates the impact of the change in these shortcuts on other related shortcuts. This process continues until no more shortcuts require modification, with each iteration adding further influenced shortcuts to the priority queue P. Additionally, another priority queue Q is utilized to keep track of all the changes made to the shortcuts, which will later have an impact on the changes to the labels. In line 3, a shortcut with its original distance and count values is extracted from the

Algorithm 12: EdgeSuperDec

```
1  $P \leftarrow$ an empty priority queue, minimizes $\pi$
2  while $\Delta G$ is not empty do
3      $(u, v, \phi'_G(u, v), 1) \leftarrow \Delta G.pop()$
4      if $\phi'_G(u, v) < \phi(u, v)$ then
5          $P$.insert$(u, v, \phi(u, v), \varsigma(u, v))$
6          $\phi(u, v) \leftarrow \phi'_G(u, v)$
7          $\varsigma(u, v) \leftarrow 1$
8      else if $\phi'_G(u, v) = \phi(u, v)$ then
9          $P$.insert$(u, v, \phi(u, v), \varsigma(u, v))$
10         $\varsigma(u, v) \leftarrow \varsigma(u, v) + 1$
11 return $P$
```

priority queue. Subsequently, for each vertex w that belongs to $X(u)$, it is determined if the values of shortcut (v, w) will be impacted. Here, it is assumed that $\pi(w) > \pi(v)$, indicating that node $X(w)$ is higher than node $X(v)$. If $\pi(w) < \pi(v)$, the vertices w and v are simply swapped and the process is carried out accordingly. This swap has been omitted for the sake of simplicity in the illustration.

Line 6 checks if the shortcut (u, w) has already been added to the priority queue P. If it has, we store the previous count value of (u, v) and leave the update process for (v, w) to be handled by (u, w). This is a crucial step to avoid double updates to the count value of (v, w), which could lead to incorrect count values. Each time a shortcut's count value is updated, it may be altered once, so it is important to ensure that each shortcut is updated only once. This technique of deferring the update skips the first shortcut and delegates the update to the second shortcut when both shortcuts have been changed. The old count value of (u, v) is stored so that it can be used to update the values of (v, w) at a later stage. This information is recorded for (u, w) for later use.

In lines 7 and 8, we calculate the new distance and count values of the shortcut

(v, w) by considering u as an intermediate node and denote them as d and c, respectively, Line 9 judges whether the distance value of (u, v) has decreased. If it decreases, we treat it as a weight-decreasing case and handle it accordingly. On the other hand, if the distance value remains the same, it means that only the count value of (u, v) has changed, and we handle it as a count-change case.

In the weight-decrease case, the new distance value is checked against $\phi(v, w)$. If the new distance is smaller, the distance and count values are updated. If the new distance is equal to $\phi(v, w)$, the count value of $\varsigma(v, w)$ is updated by adding the count value through u in line 18.

For the count-change case, the presence of the shortest paths passing through u is evaluated in line 19. If so, the count value $\varsigma(v, w)$ is updated by adding the new count value through u and subtracting the old count value through u, as shown in line 23. Line 22 ensures that $\varsigma_{u,w}$ always holds the previous count value for the shortcut (u, w).

Every time we discover updates to either the distance or count values, we push the altered shortcuts into both P and Q. We repeat the process of examining the shortcuts in P until it is empty and keep track of all the changed shortcuts by storing them in Q.

Super-Label Decrease The next step after generating Q with Algorithm 13 is to compute the labels that are impacted by the edge changes. This is done by using the Algorithm 14 algorithm. At the beginning of Algorithm 14, we initialize an empty priority queue R to record the labels that have been impacted by the weight changes of the shortcuts. Then, in line 3, we retrieve the shortcut changes one by one from Q. Note that Q maximizes π, which means we start from the top of the tree. From line 4 to line 19, we check for each ancestor a of u. We suppose $\pi(a) > \pi(v)$, which means $X(a)$ is on top of $X(v)$. When $\pi(a) < \pi(v)$, we simply swap the vertices a and v and carry out the process accordingly. This

Algorithm 13: SuperSuperDec

```
$Q \leftarrow P$, $Q$ maximizes $\pi$, $S \leftarrow \emptyset$
while $P$ is not empty do
    $(u, v, \phi, \varsigma) \leftarrow P.pop()$
    foreach $w \in X(u)$ do
        suppose $\pi(w) > \pi(v)$;
        if $(u, w) \in P$ then $S \leftarrow S \cup (u, v, \varsigma)$ and continue ;
        $d \leftarrow \phi(u, v) + \phi(u, w)$
        $c \leftarrow \varsigma(u, v) \cdot \varsigma(u, w)$
        if $\phi(u, v) < \phi$ then
            if $d < \phi(v, w)$ then
                $P.insert(v, w, \phi(v, w), \varsigma(v, w))$
                $Q.insert(v, w, \phi(v, w), \varsigma(v, w))$
                $\phi(v, w) \leftarrow d$
                $\varsigma(v, w) \leftarrow c$
            else if $d = \phi(v, w)$ then
                $P.insert(v, w, \phi(v, w), \varsigma(v, w))$
                $Q.insert(v, w, \phi(v, w), \varsigma(v, w))$
                $\varsigma(v, w) \leftarrow \varsigma(v, w) + c$
        else if $\phi(u, v) = \phi$ and $d = \phi(v, w)$ then
            $P.insert(v, w, \phi(v, w), \varsigma(v, w))$
            $Q.insert(v, w, \phi(v, w), \varsigma(v, w))$
            get $\varsigma_{u,w}$ from $S$ otherwise $\varsigma_{u,w} \leftarrow \varsigma(u, w)$
            $\varsigma(v, w) \leftarrow \varsigma(v, w) + c - \varsigma \cdot \varsigma_{u,w}$
return $Q$
```

swap has been omitted for simplicity in the illustration.

If both the shortcut (u, v) and the label (v, a) have changed, we may encounter a similar double-update issue. Therefore, in line 6, we record the shortcut (u, v) and postpone the update to avoid the double-update. Next, in lines 7 and 8, we compute the new distance label value and the count label value. We then check the weight-decrease case and count-change case for the shortcut (u, v). Depending on the case, we update $sd(u, a)$ and $spc(u, a)$ accordingly. Whenever a label (u, a) is changed, we add the update to R, which will be delivered to Algorithm 15 to further check other labels that would be affected by the labels in R.

Algorithm 14: SuperLabelDec

```
1  $R \leftarrow$ an empty priority queue, maximizes $\pi$, $S \leftarrow \emptyset$
2  while $Q.is\ not\ empty$ do
3      $(u, v, \phi, \varsigma) \leftarrow Q.pop()$
4      foreach $a \in A(u)$ do
5          suppose $\pi(a) > \pi(v)$;
6          if $(v, a) \in R$ then  $S \leftarrow S \cup (u, v, \varsigma)$ and continue ;
7          $d \leftarrow \phi(u, v) + \mathsf{sd}(v, a)$
8          $c \leftarrow \varsigma(u, v) \cdot \mathsf{spc}(v, a)$
9          if $\phi(u, v) < \phi$ then
10             if $d < \mathsf{sd}(u, a)$ then
11                 $R.insert(u, a, \mathsf{sd}(u, a), \mathsf{spc}(u, a))$
12                 $\mathsf{sd}(u, a) \leftarrow d$
13                 $\mathsf{spc}(u, a) \leftarrow c$
14             else if $d = \mathsf{sd}(u, a)$ then
15                 $R.insert(u, a, \mathsf{sd}(u, a), \mathsf{spc}(u, a))$
16                 $\mathsf{spc}(u, a) \leftarrow \mathsf{spc}(u, a) + c$
17         else if $\phi(u, v) = \phi$ and $d = \mathsf{sd}(u, a)$ then
18             $R.insert(u, a, \mathsf{sd}(u, a), \mathsf{spc}(u, a))$
19             $\mathsf{spc}(u, a) \leftarrow \mathsf{spc}(u, a) + (\varsigma(u, v) - \varsigma) \cdot \mathsf{spc}(v, a)$
20 return $R$, $S$
```

Label-Label Decrease To calculate the influence between labels, we first give the definition of reverse tree node as follows.

Definition 19. (REVERSE TREE NODE) *Given a graph $G(V, E)$ and its tree decomposition T_G, a vertex $u \in V$ is said to be in a reverse tree node for a vertex $v \in V$, denoted by $u \in X^{-1}(v)$, if $v \in X(u)$.*

The use of the *reverse tree node* $X^{-1}(v)$ facilitates the identification of the shortcut (x, u) and enables the determination of whether (x, v) requires updating. The algorithm Algorithm 15 provides a detailed procedure for checking and updating the labels that are possibly affected by the updates made to the labels in R.

Each time, we fetch a label pair (u, v) from R, and check for each vertex x in v's reverse tree node if we need to update the labels between x and v. We still assume $\pi(u) > \pi(x)$ to simplify the illustration. Lines 5 and 6 calculate the new label values for x, v via u. We judge if the new distance label is smaller. If so, we update the sd and spc label values. If the new distance value equals the old one, we update the spc value accordingly.

5.3.2 Index Maintenance for Weight Increase

To maintain the index with the edge weight increase case, we similarly have four steps, including Edge-Super Increase, Super-Super Increase, Super-Label Increase, and Label-Label Increase.

Edge-Super Increase Similar to the algorithm in Algorithm 12, we also use a priority queue to store the shortcuts that are affected by the weight increase of the edges. In line 4, we compare the original weight of edge (u, v) and the distance value of its shortcut $\phi(u, v)$. If they have the same length, it means that after the increase in weight, the edge (u, v) is no longer the shortest path

Algorithm 15: LabelLabelDec

```
1  while $R$ is not empty do
2      $(u, v, \phi, \varsigma) \leftarrow R.pop()$
3      foreach $x \in X^{-1}(v)$ do
4          suppose $\pi(u) > \pi(x)$
5          $d \leftarrow \phi(x, u) + \mathsf{sd}(u, v)$
6          $c \leftarrow \varsigma(x, u) \cdot \mathsf{spc}(u, v)$
7          if $\mathsf{sd}(u, v) < \phi$ then
8              if $d < \mathsf{sd}(x, v)$ then
9                  $R.insert(x, v, \mathsf{sd}(x, v), \mathsf{spc}(x, v))$
10                 $\mathsf{sd}(x, v) \leftarrow d$
11                 $\mathsf{spc}(x, v) \leftarrow c$
12             else if $d = \mathsf{sd}(x, v)$ then
13                 $R.insert(x, v, \mathsf{sd}(x, v), \mathsf{spc}(x, v))$
14                 $\mathsf{spc}(x, v) \leftarrow \mathsf{spc}(x, v) + c$
15         else if $\mathsf{sd}(u, v) = \phi$ and $d = \mathsf{sd}(x, v)$ then
16             $R.insert(x, v, \mathsf{sd}(x, v), \mathsf{spc}(x, v))$
17             get $\varsigma_{x,u}$ from $S$ otherwise $\varsigma_{x,u} \leftarrow \varsigma(x, u)$
18             $\mathsf{spc}(x, v) \leftarrow \mathsf{spc}(x, v) + c - \varsigma_{x,u} \cdot \varsigma)$
```

between u and v. Therefore, in line 6, we subtract one from $\varsigma(u, v)$ to account for this change. In lines 7-8, if $\varsigma(u, v)$ has become zero, we recompute $\phi(u, v)$ and $\varsigma(u, v)$ with the help of ST.

Algorithm 16: EdgeSuperInc

```
1 $P \leftarrow$ an empty priority queue, minimizes $\pi$
2 while $\Delta G$ is not empty do
3     $(u, v, \phi'_G(u, v), \varsigma) \leftarrow \Delta G.pop()$
4     if $\phi_G(u, v) = \phi(u, v)$ then
5         $P$.insert$(u, v, \phi(u, v), \varsigma(u, v))$
6         $\varsigma(u, v) \leftarrow \varsigma(u, v) - 1$
7         if $\varsigma(u, v) = 0$ then
8             recompute $\phi(u, v)$ and $\varsigma(u, v)$ with $\mathsf{ST}$
9 return $P$
```

Super-Super Increase

After executing Algorithm 16, we have recorded all shortcuts that are directly influenced by the weight increase of edges in the priority queue P. Then, for each shortcut in P, we compute the shortcuts that may be indirectly affected by it.

In line 3, we extract the shortcut (u, v) from P, and for each $w \in X(u)$, we check if the paths passing through u were the shortest paths before the edge weight was increased. If they were, we judge if $\phi(u, v)$ increases or simply the count changes and adjust the count values accordingly. If $\varsigma(v, w)$ becomes zero in line 15, we recompute the labels with the help of ST.

Similar to Algorithm 13, we add all the changed shortcuts to both P and Q. The priority queue P is used to iterate over the affected shortcuts, while the queue Q records all the shortcuts that are affected for the label influence computation.

Super-Label Increase

Algorithm 17: SuperSuperInc

```
1  $Q \leftarrow P$, $Q$ maximizes $\pi$, $S \leftarrow \emptyset$
2  while $P$ is not empty do
3      $(u, v, \phi, \varsigma) \leftarrow P.pop()$
4      foreach $w \in X(u)$ do
5          suppose $\pi(w) > \pi(v)$;
6          if $(u, w) \in P$ then $S \leftarrow S \cup (u, v, \phi, \varsigma)$ and continue ;
7          get $\phi_{u,w}, \varsigma_{u,w}$ from $S$ otherwise $\phi_{u,w} \leftarrow \phi(u, w)$, $\varsigma_{u,w} \leftarrow \varsigma(u, w)$
8          $d \leftarrow \phi + \phi_{u,w}$
9          $c \leftarrow \varsigma \cdot \varsigma_{u,w}$
10         if $d = \phi(v, w)$ then
11             if $\phi(u, v) > \phi$ then
12                 $P.insert(v, w, \phi(v, w), \varsigma(v, w))$
13                 $Q.insert(v, w, \phi(v, w), \varsigma(v, w))$
14                 $\varsigma(v, w) \leftarrow \varsigma(v, w) - c$
15                 if $\varsigma(v, w) = 0$ then
16                     recompute $\phi(v, w)$ and $\varsigma(v, w)$ with ST
17             else if $\phi(v, w) = \phi$ then
18                 $P.insert(v, w, \phi(v, w), \varsigma(v, w))$
19                 $Q.insert(v, w, \phi(v, w), \varsigma(v, w))$
20                 $\varsigma(v, w) \leftarrow \varsigma(v, w) - c + \varsigma(u, w) \cdot \varsigma(u, v)$
21 return $Q$
```

Algorithm 18 computes the labels that are affected by the weight increase of shortcuts in Q. The algorithm first checks the shortcuts from Q one by one, and for each ancestor a of u, it checks if the weight change of the shortcut (u, v) affects the distance and count labels between u and a.

In lines 7-8, the algorithm computes the original distance and count values of u, a via v, and compares them with $\mathsf{sd}(u, a)$. If they are equal, then the weight change of shortcut (u, v) will affect the weight of the labels between u and a. The algorithm then updates the labels between u and a according to the change in the shortcut weight. If the count value $\mathsf{spc}(u, a)$ becomes zero after the update, the algorithm recomputes the labels between u and a by visiting all the ancestors of u.

After iterating over all the shortcuts in Q, the label pairs that have been changed are stored in R.

Label-Label Increase After computing all the label pairs that are directly affected by the changes of the shortcuts, we calculate the further affected label pairs. For each label pair (u, v), we check each vertex x in v's reverse tree node $X^{-1}(v)$ to see if the labels for (x, v) will be affected by the changes of the label for (u, v). If it is affected, we update the labels accordingly.

To calculate the further affected label pairs, we first iterate through all the label pairs in R, and for each pair (u, v), we examine all the vertices x in $X^{-1}(v)$ and check whether the label pair (x, v) needs to be updated.

If the shortest distance between x and v equals the sum of the shortest distances between x and u and between u and v, then the label pair (x, v) needs to be updated. We update the labels and check if any further label pairs need to be updated as a result. If the shortest path count of the label pair (x, v) becomes zero, we recompute the labels between x and v by visiting all the ancestors of x.

Algorithm 18: SuperLabelInc

```
1  $R \leftarrow$ an empty priority queue, maximizes $\pi$
2  while $Q.is\ not\ empty$ do
3      $(u, v, \phi, \varsigma) \leftarrow Q.pop()$
4      foreach $a \in A(u)$ do
5          suppose $\pi(a) > \pi(v)$;
6          if $(v, a) \in R$ then $S \leftarrow S \cup (u, v, \phi, \varsigma)$ and continue ;
7          $d \leftarrow \phi + \mathsf{sd}(v, a)$
8          $c \leftarrow \varsigma \cdot \mathsf{spc}(v, a)$
9          if $d = \mathsf{sd}(u, a)$ then
10             if $\phi(u, v) > \phi$ then
11                 $R.insert(u, a, \mathsf{sd}(u, a), \mathsf{spc}(u, a))$
12                 $\mathsf{spc}(u, a) \leftarrow \mathsf{spc}(u, a) - c$
13                 if $\mathsf{spc}(u, a) = 0$ then
14                     recompute $\mathsf{sd}(u, a)$ and $\mathsf{spc}(u, a)$ with $A(u)$
15             else if $\phi(u, v) = \phi$ then
16                 $R.insert(u, a, \mathsf{sd}(u, a), \mathsf{spc}(u, a))$
17                 $\mathsf{spc}(u, a) \leftarrow \mathsf{spc}(u, a) - c + \varsigma(u, v) \cdot \mathsf{spc}(v, a)$
18 return $R, S$
```

Algorithm 19: LabelLabelInc

```
1  while $R$ is not empty do
2      $(u, v, \phi, \varsigma) \leftarrow R.pop()$
3      foreach $x \in X^{-1}(v)$ do
4          suppose $\pi(u) > \pi(x)$
5          get $\phi_{x,u}, \varsigma_{x,u}$ from $S$ otherwise $\phi_{x,u} \leftarrow \phi(x, u)$, $\varsigma_{x,u} \leftarrow \varsigma(x, u)$
6          $d \leftarrow \phi_{x,u} + \phi$
7          $c \leftarrow \varsigma_{x,u} \cdot \varsigma$
8          if $d = \mathsf{sd}(x, v)$ then
9              if $\mathsf{sd}(u, v) > \phi$ then
10                 $R.insert(x, v, \mathsf{sd}(x, v), \mathsf{spc}(x, v))$
11                 $\mathsf{spc}(x, v) \leftarrow \mathsf{spc}(x, v) - c$
12                 if $\mathsf{spc}(x, v) = 0$ then
13                     recompute $\mathsf{sd}(x, v)$ and $\mathsf{spc}(x, v)$ with $A(x)$
14             else if $\mathsf{sd}(u, v) = \phi$ then
15                 $R.insert(x, v, \mathsf{sd}(x, v), \mathsf{spc}(x, v))$
16                 $\mathsf{spc}(x, v) \leftarrow \mathsf{spc}(x, v) - c + \varsigma(x, u) \cdot \mathsf{spc}(u, v)$
```

5.3.3 An Optimized Priority Queue

Although the algorithms presented in Section 5.3.1 and Section 5.3.2 significantly reduce unnecessary recomputations of shortcuts and labels, the update process is still computationally intensive. This is because we use a priority queue that prioritizes π to record and arrange the update order of the changed shortcuts and labels. Since many shortcuts and labels will be affected during the maintenance process, managing the priority queue becomes a significant computational bottleneck.

To address this issue, we can leverage the tree decomposition property and the updating order to devise an optimized priority queue. Specifically, our updating order follows a top-down order along the tree branch. For example, in Algorithm 19, we need to ensure that the labels between u and v are updated before we update the labels from x to v. The priority queue R uses $\pi(u)$ to ensure this order. However, since the label change only affects nodes on the same branch, the processing order of nodes on different branches does not matter.

Therefore, we can use a relaxed order Depth to arrange the updating order within each branch. The Depth order only considers nodes on the same branch, and it is always bounded by the tree height h of the tree decomposition. We can use a bucket-like structure to store all affected label pairs according to the Depth of their lower nodes. When we push a changed label pair, we store it in the corresponding Depth. When we pop a changed label pair, we start from the smallest Depth. Both the push and pop operations can be done in $O(1)$ time, which is much faster than using a traditional priority queue.

5.3.4 Local Graph Revisited

In Section 4.4.4, we relaxed the convex shortest path to the local shortest path, which reduces unnecessary computation while maintaining query result correctness. This property not only benefits construction efficiency but also facilitates index updating, as we only need to check the local shortest paths in our index maintenance.

To compute updates of the local shortest distance and path count, we only need to modify the Super-Label and Label-Label updating strategies to consider only vertices in the local graph.

Specifically, in Algorithm 14 and Algorithm 18, we replace $a \in A(u)$ with $a \in A(v)$ in line 4. This ensures that we only update (u, a) when $X(a)$ is on top of $X(v)$ in the tree.

Similarly, in Algorithm 15 and Algorithm 19, we modify line 3 to change $x \in X^{-1}(v)$ to $x \in X^{-1}(u)$. This ensures that node $X(x)$ is always under node $X(u)$ in the tree.

5.4 Experiments

We conduct experiments to evaluate the proposed updating methods. All the algorithms are implemented in C++ with -O3 optimization, and the experiments are conducted on a Linux machine with an Intel Xeon Gold 6248 2.5GHz CPU and 768GB RAM. We evaluate all the algorithms on twelve real-world road networks as shown in Table 5.1. All the datasets are from DIMACS [1]. For each dataset, we randomly select 100 to 500 edges and change the weight of these edges. In the edge weight increase case, we double the edge weight for each selected edge, i.e., change the weight from $\phi(e)$ to $2 \times \phi(e)$. In the edge weight decrease case, we half the edge weight for each selected edge, i.e., change the weight from $\phi(e)$ to $\phi(e)/2$.

Compared Algorithms. We compare the following algorithms in experiments.

- TL-Dec: Our basic index updating algorithm (Algorithm 11) with edge weight decrease.
- TL-Inc: Our basic index updating algorithm (Algorithm 11) with edge weight increase.
- TL-Dec*: Our optimized index updating algorithm for edge weight decrease.
- TL-Inc*: Our optimized index updating algorithm for edge weight increase.

Exp-1: Updating Time. We compare the average updating time between the proposed maintenance algorithms. For each dataset, we execute the proposed methods in the edge weight decreasing and increasing cases, respectively.

We measured the update time of each algorithm and presented the results in Fig. 5.1. The graph shows that all algorithms experience an increase in updating

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

Table 5.1: Statistics of road networks.

Name	Description	n	m	h	w
NY	NYC	264,346	733,846	505	134
BAY	Bay Area	321,270	800,172	403	108
COL	Colorado	435,666	1,057,066	465	146
FLA	Florida	1,070,376	2,712,798	520	136
NW	Northwest US	1,207,945	2,840,208	548	146
NE	Northeast US	1,524,453	3,897,636	828	219
CAL	CA and NV	1,890,815	4,657,742	713	215
LKS	Great Lakes	2,758,119	6,885,658	1325	370
EUS	Eastern US	3,598,623	8,778,114	1022	272
WUS	Western US	6,262,104	15,248,146	1041	326
CUS	Central US	14,081,816	34,292,496	2433	660
USA	Full US	23,947,347	58,333,344	2564	693

time as the number of changed edges increases. However, our optimized updating methods, TL-Dec* and TL-Inc*, consistently outperform the basic updating methods. For instance, in the CAL dataset, when 500 edges have changed, TL-Dec and TL-Inc take 1385 seconds and 457 seconds on average, respectively, while TL-Dec* only requires 81 seconds and TL-Inc* costs 155 seconds. The significant improvement is attributed to the reduced computation of unnecessary labels in our optimized updating methods. Moreover, we observed that TL-Dec* is faster than TL-Inc*. This is because, during updating, TL-Inc* needs to visit other labels to compute the changed label, whereas TL-Dec* can directly deduce its value.

Exp-2: Proportion of Changed Labels. To gain a better understanding of the index maintenance process during updates, we recorded the proportion of labels that were updated for each dataset. We selected four representative graphs for clear illustration, and similar results were obtained for other graphs. As shown in Fig. 5.2, the proportion of labels that were updated increases with the number of changed edges, which is consistent with the increase in time cost

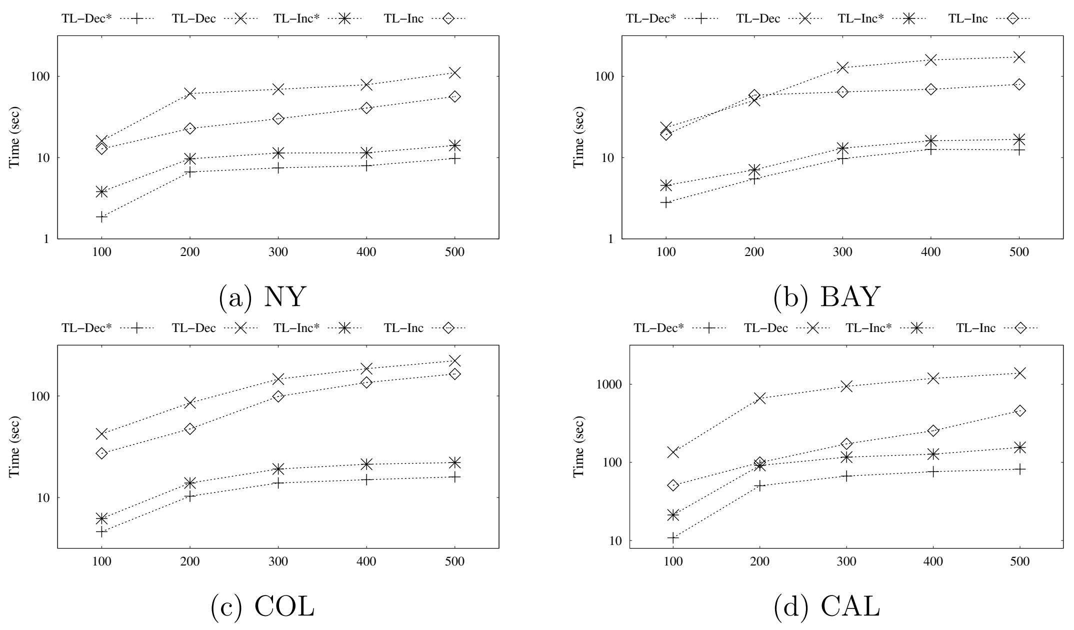


Figure 5.1: Index Updating Time With Varying Weight Change Size

shown in Exp-1. Additionally, we observed that the proportion of changed labels is relatively high when more edges' weights have changed. For instance, on the BAY dataset, when 500 edges' weights changed, more than half of the labels in the index also changed. This nature of high proportions of changed labels makes the index maintenance process more challenging and limits the potential acceleration.

5.5 Chapter Summary

In this chapter, we study the index maintenance problem for the index introduced in Chapter 4 for efficiently counting the shortest paths in road networks in the presence of changes in the weight of road segments. We start by developing a comprehensive updating framework that identifies the tree nodes whose labels may have changed and updates them accordingly. Next, we examine op-

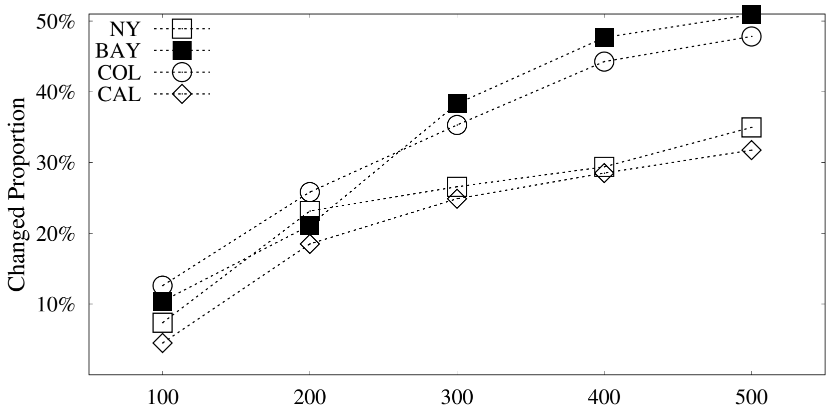


Figure 5.2: Changed Label Proportion With Vary Weight Change Size.

timized strategies for handling decreases and increases in road segment weights separately. The experimental results showcase the effectiveness of the proposed updating algorithms.

Chapter 6

EPILOGUE

In this thesis, we provide a comprehensive study of the problems relating to two special substructures in two types of large graphs. We first study the cohesive subgraph structure and the problem of mining statistically significant cliques in large labelled graphs. Then, we study the shortest path structure and the shortest path counting problem in large road networks.

For the significant clique mining problem, we propose an efficient branch-and-bound algorithm with sophisticated pruning technologies to efficiently enumerate all the maximal statistically significant cliques. We conduct extensive experiments to evaluate the proposed algorithms. The results show the effectiveness and efficiency of our algorithms.

For the shortest path counting problem, we devise a novel index structure based on the tree decomposition method. We provide efficient construction and query algorithms for the index structure. Extensive experiments are conducted to evaluate the efficiency of our proposed methods.

Real-world graphs are dynamic and constantly changing. To address this, we have developed efficient methods for maintaining the index when the graph updates. Our basic updating approach locates the changed region in the index and updates the labels without recomputing from scratch. To further accelerate the computation, we propose improved algorithms. To evaluate the effectiveness of our proposed methods, we conduct experiments on various graphs.

The following are the open problems that need further studies in our future research.

- **Pair-wise Shortest Path Enumeration on Road Networks.** In this study, we present a method for computing the shortest distance and path count between two nodes in a road network. Once these values are obtained, a natural follow-up task is to enumerate all the shortest paths between the nodes.
- **Parallel and distributed index construction for large graphs.** While our proposed index and construction methods outperform the state-of-the-art approach, constructing the index on large graphs can still be time-consuming. In our future work, we aim to investigate parallel or distributed index construction methods to address this issue for large graphs.
- **Other types of significant substructures computation on labeled graphs.** In addition to the clique model, there exist other cohesive subgraph models such as cores and trusses, which can effectively capture community structures in graphs. In our future work, we plan to investigate the statistical significance of these models for improved community modeling.

BIBLIOGRAPHY

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, pages 230–241, 2011.

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

[3] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Proceedings of the sixteenth workshop on algorithm engineering and experiments (ALENEX)*, pages 147–154, 2014.

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

[5] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM J. Comput.*, 2(1):1–6, 1973.

[6] M. Al Hasan and V. S. Dave. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2):e1226, 2018.

[7] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

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

[9] A. Arora, M. Sachan, and A. Bhattacharya. Mining statistically significant connected subgraphs in vertex labeled graphs. In *SIGMOD*, pages 1003–1014, 2014.

[10] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Algorithms and Models for the Web-Graph*, volume 4863, pages 124–137, 2007.

[11] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP*, pages 539–550, 2006.

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

[13] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[14] M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Latin American Symposium on Theoretical Informatics*, pages 88–94, 2000.

[15] M. Berlingerio, F. Pinelli, and F. Calabrese. ABACUS: frequent pattern mining-based community discovery in multidimensional networks. *DMKD*, 27(3):294–320, 2013.

[16] D. Berlowitz, S. Cohen, and B. Kimelfeld. Efficient enumeration of maximal k-plexes. In *SIGMOD*, pages 431–444, 2015.

[17] I. Bezáková and A. Searns. On counting oracles for path problems. In *International Symposium on Algorithms and Computation*, volume 123, pages 56:1–56:12, 2018.

[18] H. L. Bodlaender. Treewidth: characterizations, applications, and computations. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 1–14, 2006.

[19] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth, and minimum elimination tree height. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 1–12. Springer, 1991.

[20] A. S. Bozkır, S. Güzin Mazman, and E. Akçapınar Sezer. Identification of user patterns in social networks by data mining techniques: Facebook case. In *International symposium on information management in a changing world*, pages 145–153. Springer, 2010.

[21] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Soc. Networks*, 30(2):136–145, 2008.

[22] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Counting graphlets: Space vs time. In *WSDM*, pages 557–566, 2017.

[23] M. Bressan, S. Leucci, and A. Panconesi. Motivo: Fast motif counting via succinct color coding and adaptive sampling. *PVLDB*, 12(11):1651–1663, 2019.

[24] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[25] L. Chang. Efficient maximum clique computation over large sparse graphs. In *KDD*, pages 529–538. ACM, 2019.

[26] L. Chang, J. X. Yu, and L. Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 66(1):173–186, 2013.

[27] Z. Chen, A. W. Fu, M. Jiang, E. Lo, and P. Zhang. P2H: efficient distance querying on road networks by projected vertex separators. In *SIGMOD*, pages 313–325, 2021.

[28] H. Cheng, X. Yan, and J. Han. Mining graph patterns. In *Frequent pattern mining*, pages 307–338. Springer, 2014.

[29] J. Cheng, Y. Ke, A. W. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD*, pages 447–458. ACM, 2010.

[30] J. Cheng, Y. Ke, A. W. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.*, 36(4):21:1–21:34, 2011.

[31] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985.

[32] L. Chu, Z. Wang, J. Pei, Y. Zhang, Y. Yang, and E. Chen. Finding theme communities from database networks. *PVLDB*, 12(10):1071–1084, 2019.

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

[34] A. Conte, R. De Virgilio, A. Maccioni, M. Patrignani, and R. Torlone. Finding all maximal cliques in very large social networks. In *EDBT*, volume 2016, pages 173–184. OpenProceedings. org, 2016.

[35] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *SIGMOD*, pages 991–1002. ACM, 2014.

[36] M. Danisch, O. D. Balalau, and M. Sozio. Listing k-cliques in sparse real-world graphs. In *WWW*, pages 589–598. ACM, 2018.

[37] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017.

[38] A. Denise, M. Régnier, and M. Vandenbogaert. Assessing the statistical significance of overrepresented oligonucleotides. In *WABI*, volume 2149 of *Lecture Notes in Computer Science*, pages 85–97. Springer, 2001.

[39] R. Diestel. *Graph theory.* Springer, 2016.

[40] E. W. Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[41] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM Journal of Experimental Algorithmics*, 18:3.1–3.21, 2013.

[42] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *PVLDB*, 9(12):1233–1244, 2016.

[43] J. Flum and M. Grohe. The parameterized complexity of counting problems. *SIAM Journal on Computing*, 33(4):892–922, 2004.

[44] L. Freeman. *The Development of Social Network Analysis: A Study in the Sociology of Science*. Empirical Press, 2004.

[45] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85–112, 2004.

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

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

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

[49] H. He and A. K. Singh. Graphrank: Statistical modeling and mining of significant subgraphs in the feature space. In *ICDM*, pages 885–890. IEEE, 2006.

[50] H. He and A. K. Singh. Efficient algorithms for mining significant substructures in graphs with quality guarantees. In *ICDM*, pages 163–172. IEEE, 2007.

[51] M. He and S. Kazi. Data structures for categorical path counting queries. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, pages 15:1–15:17, 2021.

[52] Z. He, H. Liang, Z. Chen, C. Zhao, and Y. Liu. Computing exact p-values

for community detection. *Data Mining and Knowledge Discovery*, pages 1–37, 2020.

[53] A. Himmel, H. Molter, R. Niedermeier, and M. Sorge. Enumerating maximal cliques in temporal graphs. In *ASONAM*, pages 337–344, 2016.

[54] X. Huang and L. V. Lakshmanan. Attribute-driven community search. *PVLDB*, 10(9):949–960, 2017.

[55] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 1–10, 1977.

[56] S. Jain and C. Seshadhri. The power of pivoting for exact clique counting. In *WSDM*, pages 268–276, 2020.

[57] M. B. Jdidia, C. Robardet, and E. Fleury. Communities detection and analysis of their dynamics in collaborative networks. In *2007 2nd International Conference on Digital Information Management*, volume 2, pages 744–749. IEEE, 2007.

[58] Y. Jin, B. Xiong, K. He, Y. Zhou, and Y. Zhou. On fast enumeration of maximal cliques in large graphs. *Expert Systems with Applications*, 187:115915, 2022.

[59] R. M. Karp. Reducibility among combinatorial problems. In *CCC*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[60] A. Kirkley, H. Barbosa, M. Barthelemy, and G. Ghoshal. From the betweenness centrality in street networks to structural invariants in random planar graphs. *Nature communications*, 9(1):1–12, 2018.

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

[62] R. Kujala, C. Weckström, R. K. Darst, M. N. Mladenović, and J. Saramäki. A collection of public transport network data sets for 25 cities. *Scientific data*, 5(1):1–14, 2018.

[63] R. Li, S. Gao, L. Qin, G. Wang, W. Yang, and J. X. Yu. Ordering heuristics for k-clique listing. *PVLDB*, 13(11):2536–2548, 2020.

[64] R.-H. Li, Q. Dai, L. Qin, G. Wang, X. Xiao, J. X. Yu, and S. Qiao. Efficient signed clique search in signed networks. In *ICDE*, pages 245–256. IEEE, 2018.

[65] X. Li, M. Wu, C.-K. Kwoh, and S.-K. Ng. Computational approaches for detecting protein complexes from protein interaction networks: a survey. *BMC genomics*, 11(1):S3, 2010.

[66] G. Liu, L. Wong, and H. N. Chua. Complex discovery from weighted ppi networks. *Bioinformatics*, 25(15):1891–1897, 2009.

[67] C. Lu, J. X. Yu, H. Wei, and Y. Zhang. Finding the maximum clique in massive graphs. *PVLDB*, 10(11):1538–1549, 2017.

[68] C. Ma, R. Cheng, L. V. Lakshmanan, T. Grubenmann, Y. Fang, and X. Li. Linc: a motif counting algorithm for uncertain graphs. *PVLDB*, 13(2):155–168, 2019.

[69] S. Maniu, P. Senellart, and S. Jog. An experimental study of the treewidth of real-world graph data. In *ICDT*, volume 127, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[70] R. J. Mokken et al. Cliques, clubs and clans. *Quality & Quantity*, 13(2):161–173, 1979.

[71] J. W. Moon and L. Moser. On cliques in graphs. *Israel journal of Mathematics*, 3(1):23–28, 1965.

[72] S. A. Moosavi, M. Jalali, N. Misaghian, S. Shamshirband, and M. H. Anisi. Community detection in social networks using user frequent pattern mining. *KAIS*, 51(1):159–186, 2017.

[73] F. Moser, R. Colak, A. Rafiey, and M. Ester. Mining cohesive patterns from graphs with feature vectors. In *SDM*, pages 593–604, 2009.

[74] A. P. Mukherjee, P. Xu, and S. Tirthapura. Mining maximal cliques from an uncertain graph. In *ICDE*, pages 243–254. IEEE Computer Society, 2015.

[75] E. Otte and R. Rousseau. Social network analysis: a powerful strategy, also for the information sciences. *Journal of information Science*, 28(6):441–453, 2002.

[76] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu. When Hierarchy Meets 2-Hop-Labeling: Efficient Shortest Distance Queries on Road Networks. In *SIGMOD*, pages 709–724, 2018.

[77] D. Ouyang, D. Wen, L. Qin, L. Chang, Y. Zhang, and X. Lin. Progressive top-k nearest neighbors search in large road networks. In *SIGMOD*, pages 1781–1795, 2020.

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

[79] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li, et al. Trust: Triangle counting reloaded on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 32(11):2646–2660, 2021.

[80] G. A. Pavlopoulos, M. Secrier, C. N. Moschopoulos, T. G. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. G. Bagos. Using graph theory to analyze biological networks. *BioData mining*, 4(1):1–27, 2011.

[81] J. Pei, D. Jiang, and A. Zhang. Mining cross-graph quasi-cliques in gene expression and protein interaction data. In *ICDE*, pages 353–354. IEEE Computer Society, 2005.

[82] M. Pontecorvi and V. Ramachandran. A faster algorithm for fully dynamic betweenness centrality. *CoRR*, abs/1506.05783, 2015.

[83] A. Prado, M. Plantevit, C. Robardet, and J. Boulicaut. Mining graph topological patterns: Finding covariations among vertex descriptors. *TKDE*, 25(9):2090–2104, 2013.

[84] A. Prateek, A. Khan, A. Goyal, and S. Ranu. Mining top-k pairs of correlated subgraphs in a large network. *PVLDB*, 13(9):1511–1524, 2020.

[85] N. Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[86] R. Puzis, Y. Elovici, and S. Dolev. Fast algorithm for successive computation of group betweenness centrality. *Physical Review E*, 76(5):056709, 2007.

[87] S. Ranu and A. K. Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *ICDE*, pages 844–855. IEEE, 2009.

[88] T. R. Read and N. A. Cressie. *Goodness-of-fit statistics for discrete multivariate data.* Springer Science & Business Media, 2012.

[89] Y. Ren, A. Ay, and T. Kahveci. Shortest path counting in probabilistic biological networks. *BMC bioinformatics*, 19(1):1–19, 2018.

[90] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *WSDM*, pages 413–422, 2014.

[91] B. Roberts and D. P. Kroese. Estimating the number of st paths in a graph. *J. Graph Algorithms Appl.*, 11(1):195–214, 2007.

[92] R. A. Rossi, D. F. Gleich, and A. H. Gebremedhin. Parallel maximum clique algorithms with applications to network analysis. *SIAM J. Scientific Computing*, 37(5), 2015.

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

[94] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

[95] J. Shi, L. Dhulipala, and J. Shun. Parallel clique counting and peeling algorithms. In *SIAM Conference on Applied and Computational Discrete Algorithms*, pages 135–146, 2021.

[96] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, pages 939–948. ACM, 2010.

[97] D. Surian, N. Liu, D. Lo, H. Tong, E.-P. Lim, and C. Faloutsos. Recommending people in developers' collaboration network. In *2011 18th Working Conference on Reverse Engineering*, pages 379–388. IEEE, 2011.

[98] M. Tomassini and L. Luthi. Empirical analysis of the evolution of a scientific collaboration network. *Physica A: Statistical Mechanics and its Applications*, 385(2):750–764, 2007.

[99] E. Tomita. Efficient algorithms for finding maximum and maximal cliques and their applications. In *WALCOM*, volume 10167 of *Lecture Notes in Computer Science*, pages 3–15, 2017.

[100] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.

[101] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846, 2009.

[102] K. Tsuda. Entire regularization paths for graph data. In *ICML*, pages 919–926, 2007.

[103] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[104] S. Vishveshwara, K. Brinda, and N. Kannan. Protein structure: insights from graph theory. *Journal of Theoretical and Computational Chemistry*, 1(01):187–211, 2002.

[105] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[106] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu. I/o efficient core graph

decomposition: Application to degeneracy ordering. *IEEE Trans. Knowl. Data Eng.*, 31(1):75–90, 2018.

[107] K. Wongpanya, K. Sripimanwat, and K. Jenjerapongvej. Simplification of frequency test for random number generation based on chi-square. In *AICT*, pages 305–308. IEEE Computer Society, 2008.

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

[109] Y. Xu, J. Cheng, A. W.-C. Fu, and Y. Bu. Distributed maximal clique computation. In *International Congress on Big Data*, pages 160–167. IEEE, 2014.

[110] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *SIGMOD*, pages 433–444, 2008.

[111] N. Ye and Q. Chen. An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems. *Quality and Reliability Engineering International*, 17(2):105–112, 2001.

[112] C. Zhang, Y. Zhang, W. Zhang, L. Qin, and J. Yang. Efficient maximal spatial clique enumeration. In *ICDE*, pages 878–889. IEEE, 2019.

[113] M. Zhang, L. Li, W. Hua, R. Mao, P. Chao, and X. Zhou. Dynamic Hub Labeling for Road Networks. In *ICDE*, pages 336–347, Chania, Greece, Apr. 2021.

[114] P. Zhang and C. Moore. Scalable detection of statistically significant communities and hierarchies, using message passing for modularity. *Proceedings of the National Academy of Sciences*, 111(51):18144–18149, 2014.

[115] Y. Zhang and J. X. Yu. Hub labeling for shortest path counting. In *SIGMOD*, pages 1813–1828, 2020.

[116] Y. Zhang and J. X. Yu. Relative subboundedness of contraction hierarchy and hierarchical 2-hop index in dynamic road networks. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1992–2005, 2022.

[117] Z. Zhang, X. Huang, J. Xu, B. Choi, and Z. Shang. Keyword-centric community search. In *ICDE*, pages 422–433. IEEE, 2019.

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