A Doctor Thesis

博士論文

Pruned Labeling Algorithms: Unified Indexing Scheme for Graph Query Processing

（枝刈りラベリング法による大規模グラフ上の体系的なクエリ処理）

by

Takuya Akiba

秋葉 拓哉

Submitted to

the Graduate School of the University of Tokyo

on December 12, 2014

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Information Science and Technology

in Computer Science

Thesis Supervisor: Hiroshi Imai　今井 浩

Professor of Computer Science

ABSTRACT

Graph-shaped data are ubiquitous; whenever we handle relationship among any kinds of entities, a graph emerges as the most natural model. Owing to the recent popularization of the Internet and the World Wide Web, it is playing more and more important roles in real applications to extract beneficial information from large-scale graphs such as social networks and web graphs. One of the most fundamental and crucial building blocks there is *indexing methods* for answering shortest paths and their variants. These methods first construct a data structure called an *index* from a graph, and then they efficiently answer queries using the data structure. From both theoretical and empirical sides, the study of these indexing methods has been intensively conducted to seek for a good trade-off between scalability (i.e., index size and indexing time) and query performance (i.e., query time and accuracy).

However, this field has been still in its infancy. Theoretical methods with asymptotic complexity bounds for arbitrary graphs do not work well in practice. Therefore, state-of-the-art empirical methods are heuristics that highly depend on properties of a specific kind of queries and targeted real graphs. Hence, lines of research on different problems are almost independent, and totally different approaches have been developed for different popular problems such as shortest-path queries on complex networks, reachability queries on directed acyclic graphs, and shortest-path queries on road networks.

In this thesis, we address this issue by proposing *pruned labeling algorithms*, which is based on a new unified principle that can be widely applied to these path-related queries. We first give an indexing method named *pruned landmark labeling* for shortest-path queries on complex networks. Our method is an exact method, that is, it always answers correct distance between arbitrary two points. It precomputes *distance labels* for vertices by performing a breadth-first search from every vertex. Seemingly too obvious and too inefficient at first glance, the key ingredient introduced here is pruning during breadth-first searches. While we can still answer the correct distance for any pair of vertices from the labels, it surprisingly reduces the search space and sizes of labels. We experimentally demonstrate that the combination of these two techniques is efficient and robust on various kinds of large-scale real-world networks. In particular, our method can handle social networks and web graphs with hundreds of millions of edges, which are two orders of magnitude larger than the limits of previous exact methods, with comparable query time to those of previous methods.

Then, we show that efficient methods tailored to different kinds of queries can also be obtained based on the same pruning principle. Specifically, we design indexing methods for reachability queries on directed acycilic graphs, shortest-path queries on road networks, historical shortest-path queries on evolving networks, and top-k shortest-path queries on complex networks. We demonstrate that each of these methods is also comparable with state-of-the-art methods for each kind of query, thus showing exceptional generality of our unified approach.

Finally, we tackle another long-standing question in this field: what is the key factor besides network size that has a large effect on the performance of indexing methods? For example, when processing real-world networks, we sometimes see that indices constructed from the graphs by the same algorithm may be of quite different sizes, even if graphs are of similar size. We investigate the ways for measuring such *difficulty* of networks. We theoretically and empirically show that obtaining the width of a *tree decomposition* can take us closer to the answer.

論文要旨

物事の関係が現れるほぼあらゆる場面で，データはグラフとして表現され処理される．特に近年では，インターネット及びワールド・ワイド・ウェブの普及に伴い，ソーシャルネットワークやウェブグラフを始めとする非常に大規模なグラフデータが偏在している．そのため，大規模グラフデータから有用な情報を効率的に引き出すことは現代社会の様々な場面において重要な役割を担っている．そのような大規模グラフの処理の根幹を支える重要な部品の 1 つが，最短経路及び関連問題に対する索引付け手法である．それらの手法は，グラフから予め索引と呼ばれるデータ構造を前計算し，そのデータ構造を用いて 2 点間の最短経路などの問合せに効率的に応答する．スケーラビリティ（索引サイズや索引構築時間）と応答性能（応答時間や精度）の良好なトレードオフを達成することが索引付け手法の目的である．

しかし，最短経路関連問題に対する索引付け手法の研究は未だに未成熟な状況にあった．理論的な結果として，任意のグラフにおける漸近的保証を持つ手法の開発が長らく取り組まれてきているものの，現実的な性能は実用に足るものになっていない．一方，現実的なグラフにおいて高い性能を達成する手法は，対象とする現実的なグラフの性質に依存したヒューリスティクスとして独立に開発されてきた．従って，複雑ネットワーク上での最短経路クエリ，有向無閉路グラフ上での到達可能性クエリ，道路ネットワーク上での最短経路クエリというような異なる状況が，完全に異なる問題として扱われ，個別にアプローチが開発されており，各個撃破の状態にあった．

そこで，本論文では統一的なアルゴリズムの枠組みである枝刈りラベリング法の提案を行う．提案手法は距離ラベルと呼ばれるデータ構造を前計算し索引として保存する．距離ラベルに基づく手法は一般にメモリ参照の局所性により応答性能が極めて優れている．しかし，既存手法はいずれも距離ラベルの計算を異なる最適化問題への定式化を通じて間接的に行うため，スケーラビリティに問題があった．一方，枝刈りラベリング法は巧妙な枝刈りにより最短経路探索と同時に直接的に距離ラベルの計算を行うことができ，応答性能を犠牲にすることなくスケーラビリティを大幅に向上する．そして，最短経路探索を行う順番の変更により性質の異なるグラフの構造を活用でき，また異なる種類の距離ラベルに対しても同じアプローチでアルゴリズムを設計できるため，上記のように今まで異なる問題として扱われてきた問題に対して体系的にアルゴリズムを与えることができる．具体的な問題として，複雑ネットワーク上での最短経路クエリ，有向無閉路グラフ上での到達可能性クエリ，道路ネットワーク上での最短経路クエリ，複雑ネットワーク上での Top-k 最短経路クエリ，動的ネットワークにおける最短経路変化履歴クエリを扱う．実験によりそれぞれの問題における最新の手法との比較を行い，一部の問題では大きな性能改善を達成し，残りの問題でも少なくとも同程度の性能を達成することができることを示す．

さらに，提案手法を含むグラフ索引付け手法の性能を左右するグラフの性質を探る．例えほぼ同じサイズのグラフであっても，索引構築時間や索引サイズといった索引付け手法の性能がグラフによって大きく異なることがあり，その原因は今まで分かっていなかった．そこで，木分解という道具を用いることにより，このようなグラフに潜むサイズ以外の「難しさ」をある程度捉えることができることを理論的及び実験的に示す．

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Hiroshi Imai, for his precious advice and encouragement from his broad knowledge and experience. Without his support and guidance, my four years in graduate school could not be accomplished. I am also obliged to him for showing me how to conduct research effectively, how to present results in papers and talks, and how to communicate with other researchers. These skills would be definitely valuable in my whole research career.

I am also very honored to have Prof. Naoki Kobayashi as the chair, and Prof. Masami Hagiya, Prof. Reiji Suda, Prof. Tetsuo Shibuya, Prof. Kunihiko Sadakane, and Prof. Satoru Iwata as the members of the committee for this Ph.D. thesis. I highly appreciate them taking the time and effort to study and examine this thesis.

I am also profoundly grateful to my great collaborators so far, Christian Sommer, Ken-ichi Kawarabayashi, Yoichi Iwata, Yuichi Yoshida, Yosuke Yano, Yuki Kawata, Takanori Maehara, Nori Nozomi, and Takanori Hayashi. I really enjoyed working with these insightful and talented people, where I learned a lot from them.

I would also like to thank all members in Imai Laboratory, Akitoshi Kawamura, Francois Le Gall, Masato Edahiro, Mami Takahashi, Norie Fu, Vorapong Suppakitpaisarn, Kenta Takahashi, Hiroyuki Miyata, Toshihiro Tanuma, Takahiko Satoh, Jean-Francois Baffier, Yoshikazu Aoshima, Akihiro Hashikura, Shunichi Matsuda, Ly Nguyen, Yoichi Iwata, Hidefumi Hiraishi, Hiroyuki Ohta, Junya Fukawa, Chitchanok Chuengsatiansup, Alonso Gragera, Bingkai Lin, Keigo Oka, Akira Motoyama, Yuto Hirakuri, Yuki Kawata, Chihiro Komaki, Yosuke Yano, Naoto Ohsaka, Takuto Ikuta, Shuichi Hirahara, Takanori Hayashi, Makoto Soejima, Kentaro Yamamoto, Holger Thies, Jeremy Cohen, and Simon Klein.

In addition to the main laboratory, I also had the privilege of working with or visiting several prominent research groups. First, I have been a member of the complex network and map graph group at *JST ERATO Kawarabayashi Large Graph Project*. As people there have close research interest with me, I pretty enjoyed the weekly seminar with them, specifically, Ken-ichi Kawarabayashi, Yuichi Yoshida, Naoki Masuda, Takehisa Hasegawa, Kazuhiro Inaba, Yutaka Horita, Ryosuke Nishi, Junichi Teruyama, Taro Takaguchi, Ryohei Hisano, Tatsuro Kawamoto, Kodai Saito, Daiki Takeuchi, Yoshitake Murai, Leo Speidel, and members from Imai Laboratory. In addition to the team members, I also appreciate people in the other teams of the project, especially, Kazuo Imai, Naonori Kakimura, Yusuke Kobayashi, Kohei Hayashi, Yutaro Yamaguchi, Kensuke Otsuki, and Satoko Tsushima.

I also profited from several fruitful events organized by members of *JST ERATO Minato Discrete Structure Manipulation System Project*. I want to say thank you to people there, especially to Shin-ichi Minato, Hiroki Arimura, Koji Tsuda, Takeaki Uno and Yasuo Tabei.

It was also great to be involved with *Graph CREST (Advanced Computing and Optimization Infrastructure for Extremely Large-Scale Graphs on Post Peta-Scale Supercomputers)*. People there kindly taught me the state-of-the-art research on graph processing in the high performance computing community. Especially, I would like to thank Katsuki Fujisawa, Toyotaro Suzumura, Toshio Endo, Hitoshi Sato, Ken Wakita, Yuichiro Yasui and Koji Ueno.

I also did two valuable research internships at *Microsoft Research*. The first internship was at *Microsoft Research Asia* (in Beijing). I owe a big thanks to my mentor there, Tetsuya Sakai. I also appreciate researchers and colleagues there, Yuki Arase, Jun-ichi Tsujii, Mitsuo Yoshida, Jun Hatori, Hiroki Hanaoka, Tsuyoshi Takatani, Satoshi Ikehata, Takeshi Sakaki, Yasuhisa Yoshida, Kazuya Okada, Lin Meng, and Yatao Li.

The second internship was at *Microsoft Research Silicon Valley*, which was suddenly closed in September 2014, during my internship. I would never forget what happened there in front of me. I express my sincere appreciation to my mentor there, Daniel Delling, and group members, Robert E. Tarjan, Andrew V. Goldberg, Edith Cohen, Renato F. Werneck, Thomas Pajor, Daniel Fleischman and Ilya Razenshteyn.

Finally, I would like to express my heartfelt appreciation to my family members, my friends and all of those who have supported me in any aspect of graduate school life.

Takuya Akiba, December 2014

Contents

Chapter 1

Introduction

This thesis is about practical indexing methods for real graphs to efficiently answer path-related queries. The aim of this chapter is to reconfirm the importance of these practical graph indexing methods, explain the current research gaps, and give the overview of the contribution of this thesis.

1.1 Real-World Graph Data

Graph-shaped data are ubiquitous; whenever we handle relationship among any kinds of entities, a graph emerges as the most natural model. Formal definition of graphs and related terms will be given in Chapter 2.

The study of graphs is considered to be pioneered by Leonhard Euler in 1735. He proved that one cannot cross each bridge exactly once by walk at the famous *Seven Bridges of Königsberg*, which is currently known as the notion of *Euler tour* or *Euler path*. After that, graphs have been one of the main targets of study in discrete mathematics.

Nowadays, graphs are also data sources of crucial importance in many kinds of real systems such as web services, artificial intelligence, operations management and industrial engineering. In particular, owing to the recent popularization of the Internet and the World Wide Web, it is playing more and more critical roles in real applications to extract beneficial information from large-scale graphs such as social networks and web graphs.

In the following part of this section, we review the representative kinds of real-world graph data of our interest. Specifically, we study the definition, history and applications of them.

1.1.1 Social Networks

Generally, *social networks* are graphs where each vertex represents an individual and each edge represents some kind of relationship. Examples of relationship that is often modeled as social networks are friendship, collaboration (e.g., co-author graphs and co-starring graphs), communication (e.g., e-mail networks and instant-messaging networks).

Social networks have been definitely the graphs of the biggest interest to various research communities such as sociology and psychology because their properties are closely related to the nature of human beings. Study of social network is said to have started in around 1930 [Kar29]. Famous results include small-world phenomenon [Mil67, TM69]. In late 1990s, interesting structures of real networks stimulated the statistical physics community to build statistical modeling of these networks [WS98]. Together with other networks such as web

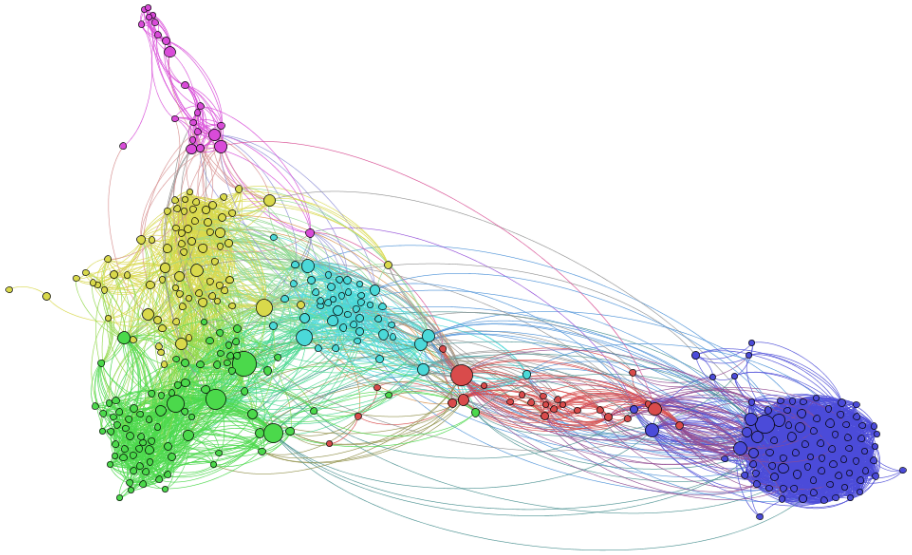


Figure 1.1: A social ego network of the author (i.e., the subgraph induced by the friends of the authors) extracted from Facebook by Netvizz[2].

graphs and biological networks, this led to the emergence of the new research community called *network science* or *complex network theory*.

Recently, many people, especially young people, have started using *social networking web services* (so-called *SNSs*). On these services, one creates a list of users with whom to connect (called *friends*), and they interact by sharing messages, pictures or videos with them. Indeed, the author is an active user of several social networking services at the moment such as Facebook, Twitter, Google+, LinkedIn, Instagram, Flickr, and Vine, to name but a few.

Through these services, the providers obtain the real, large-scale social networks. Figure 1.1 is an example of social network data extracted from a SNS. These network data enabled to confirm classic hypotheses that were hard to verify with small data and to find new common global properties [BBR$^+$12, BV12]. Moreover, mining beneficial information from these networks is getting a crucial task for these service providers to improve the quality of experience for these web services. The numbers of vertices in these social networks correspond to numbers of users of these services, thus ranges depending on their popularity. The largest social networking service at this moment is Facebook with 1.35 billion monthly active users[1].

1.1.2 Web Graphs

Web pages of the day may have *hyperlinks*, which help visitors to move to another web page by clicking them. *Web graphs* are graphs where vertices represent web pages and edges correspond to hyperlinks from the web pages.

Interestingly, web graphs also played a crucial role in the most notable change in the history of web search engines: the rise of Google. There were numerous search engines before Google, but they ranked search results just by keyword relevance, which were fragile to spam sites. Google is the first engine that introduced an idea of search result ranking using web graphs, and it significantly improved

[1] `http://newsroom.fb.com/company-info/`
[2] `https://apps.facebook.com/netvizz/`

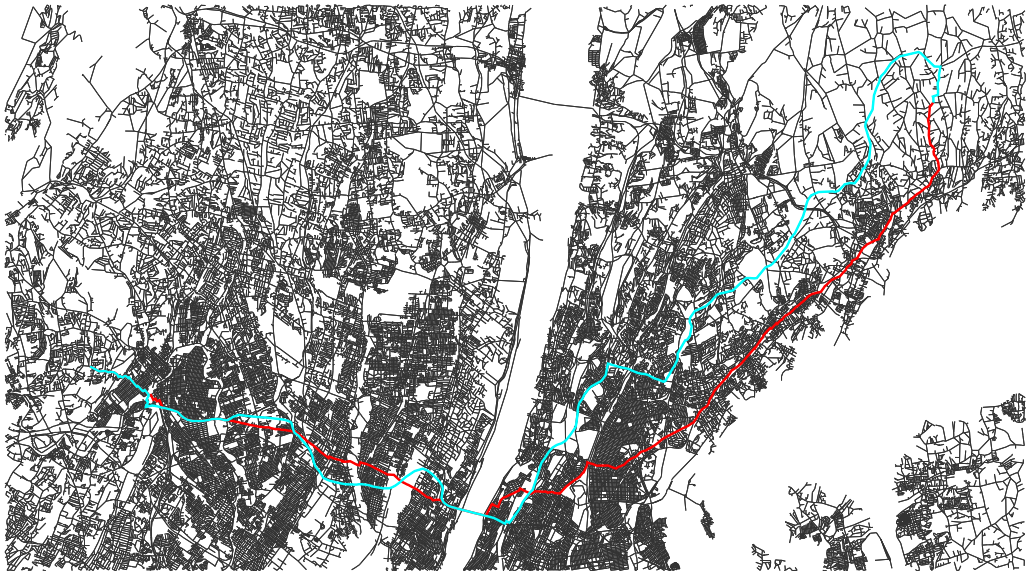


Figure 1.2: A part of a road network of the City of New York [DGJ09]. Red and blue paths illustrate the shortest paths between the same pair of vertices, where the red one optimizes distance and the blue one optimizes time.

web search experience [PBMW99]. This history indicates that web graphs might be really precious data sources.

As the notion of web graphs emerged after the popularization of the World Wide Web, they are newer than social networks. However, while it was very hard to obtain large-scale social networks before those social networking services mentioned above, web graphs can be automatically obtained by crawling, i.e., just conducting graph search on web pages. Therefore, large graph data made available earlier for web graphs than social networks. Famous results include the *bow-tie structure* [BKM+00].

Obtaining an accurate web graph is an important task both for scientists (who are interested in network properties) and practitioners (such as web search engine providers). Seemingly, crawling is an easy search task based on graph searching. However, it is actually very hard due to the enormous number of web pages and dynamic web services. Therefore, considerable engineering effort has been done to cope with these problems [BCSV04,BMSV14]. While the number of web pages cannot be soundly defined due to dynamic web services, it is said that there are at least 3.5 billion web pages and 128.7 billion edges [MVLB14].

1.1.3 Road Networks

Road networks are graphs where vertices and edges represent intersections and streets, respectively. That is, road networks correspond to graph representations of maps. Figure 1.2 shows an example of a road network. Road networks are obviously helpful for many applications such as route planning, transportation optimization, and evacuation planning.

Road networks are usually weighted graphs, where weight can be time, distance or cost. Figure 1.2 also illustrates two paths between the same pair of vertices that optimize distance and time, respectively,

Previously, road networks can be obtained only by buying map data from map vendors. However, some vendors kindly provided road network data for research purpose. Popular ones are the road network of United States provided by Tiger

and that of Europe provided by PTV [DGJ09]. Unfortunately, it has been pointed out that the former one contains several errors. Recently, there is a Wikipedia-style collaborative open map called *OpenStreetMap*[3]. Researchers can obtain large-scale road network data from OpenStreetMap. The road networks created from OpenStreetMap have hundreds of millions of vertices and edges [DGW13].

1.2 Path-Related Queries on Graphs

In this thesis, we focus on indexing methods for processing path-related queries on graphs. In this section, we briefly introduce the problems and explain applications of them. The formal definition of the problems will be given later.

1.2.1 Reachability Queries

Answering a *reachability query* is to determine whether there is a directed path from a vertex s to a vertex t on a given directed graph $G = (V, E)$. Reachability queries are ubiquitous as one of the most basic and important operations on graphs.

For example, in query engines such as SPARQL and XQuery, it is one of the fundamental building blocks for answering user queries [PAG09,Cha03,WED+08]. In computational biology, it is employed for representing and analyzing molecular and cellular functions [vHNM+00]. In program analysis, it enables precise interprocedural dataflow analysis [RHS95, Rep97].

1.2.2 Shortest-Path and Distance Queries

A *shortest-path query* asks the shortest path between two vertices in a graph, and a *distance query* asks the distance between two vertices in a graph. Answering these queries is also ones of the most fundamental operations on graphs, and has a wide range of applications.

For example, on transportation networks, computing a shortest path corresponds the route planning problem. On social networks, distance between two users is considered to indicate the closeness, and used in socially-sensitive search to help users to find more related users or contents [VFD+07, YBLS08], or to analyze influential people and communities [KKT03, BHKL06]. On web graphs, distance between web pages is one of indicators of relevance, and used in context-aware search to give higher ranks to web pages more related to the currently visiting web page [UCDG08, PBCG09]. Other applications of distance queries include top-k keyword queries on linked data [HWYY07, TWRC09], discovery of optimal pathways between compounds in metabolic networks [RAS+05, RS06], and management of resources in computer networks [PSV04, BLM+06].

1.3 Indexing Methods

In this section, we introduce *indexing methods* for graphs, and explain remaining challenges in this field.

First of all, we would like to confirm the practical importance of quickly answering the queries above (Figure 1.3). Suppose we have a real-time network-aware service (e.g., a network-aware search service [VFD+07, YBLS08]) on an online social network. To generate a response to a user (e.g., a search result), the total procedure may want to compute some kinds of fundamental metrics

[3] `http://www.openstreetmap.org/`

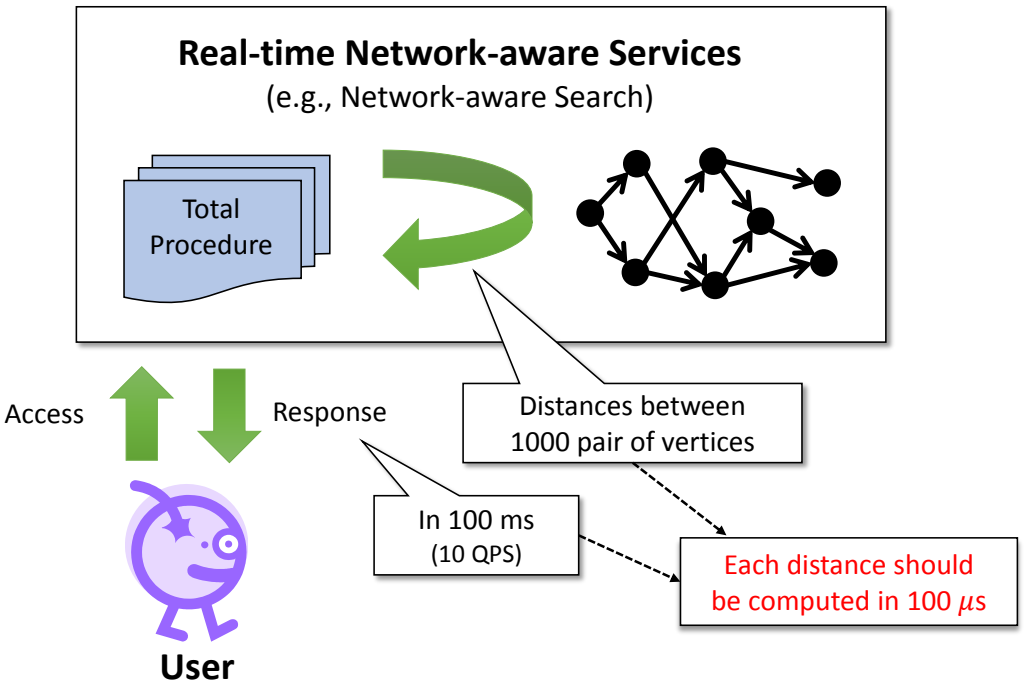


Figure 1.3: The necessity of graph indexing methods.

(e.g., distance) on the graph between many pair of vertices (e.g., for ranking the search result). On the other hand, as this system is a real-time service, a user may expect to get a result instantly, or we also have to process many queries in a second to keep low load average. Supposing distances between one thousand pairs of vertices are necessary for each result, and each result should be generated under ten milliseconds, distance between single pair need to be obtained under one hundred microseconds.

Therefore, to quickly obtain the answers to the path-related queries introduced above, *indexing methods* have been employed. Generally, graph indexing methods have two steps: indexing and query answering (Figure 1.4). First, it constructs a data structure called an *index* from the given graph. After obtaining an index, it answers *queries* between arbitrary pairs of vertices. The first step is also called *preprocessing* or *precomputation*.

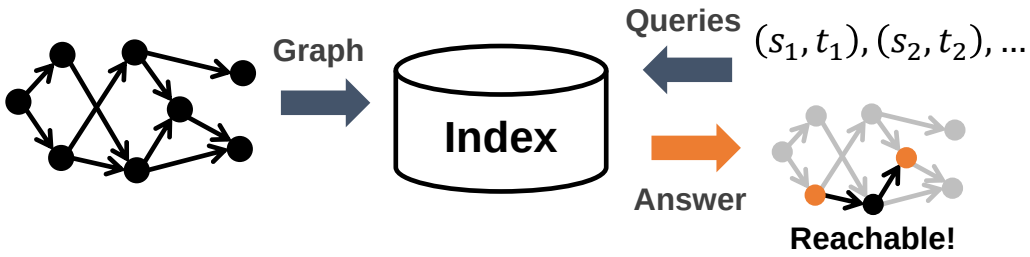


Figure 1.4: The overview of graph indexing methods.

The indexing methods are evaluated in terms of the trade-off between *scalability* and *query performance* (Figure 1.5). The term scalability means the applicability to larger graphs, and measured by indexing time (i.e., the time consumption for constructing an index) and index size (i.e., the data size of the constructed index). On the other hand, query performance is evaluated with regard to query time and precision. As evaluation of indexing methods is multi-criteria, methods that provide different trade-offs are of different importance.

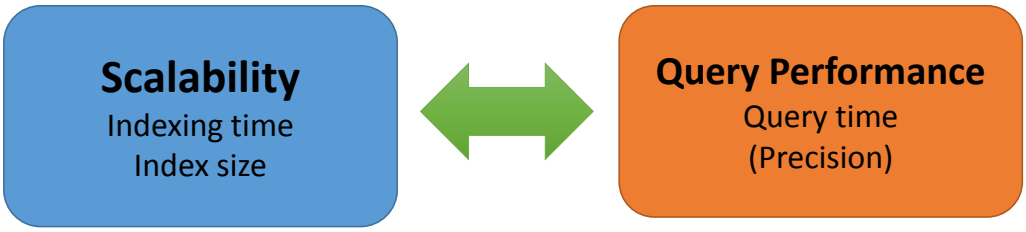


Figure 1.5: The performance trade-off of indexing methods between scalability and query performance.

There are two obvious extreme methods that optimize either of those two criteria: *non-indexing methods* and *full-indexing methods*. Non-indexing methods are those which conduct no preprocessing and answer queries solely by online computation. Obviously, both indexing time and index size of these non-indexing methods are optimal, while query performance is generally worse than other indexing methods and insufficient for practical applications mentioned above. In contrast, full-indexing methods precompute answers to all possible queries (i.e., every pair of vertices). For example, computing the answers to all the possible distance queries corresponds to the *all pairs shortest path* problem (see Section 2.2.3). The query performance of full-indexing methods is clearly optimal, but the scalability is highly limited. Thus, we need to develop more practical indexing methods that lie midway between no-indexing methods and full-indexing methods.

Practical graph indexing methods have been studied in the database community and experimental algorithmics community. In these communities, graph indexing methods are relatively recent topics of their interest, probably due to the recent emergence of large graph data.

Previously, indexing methods for different kinds of queries have been almost independently studied, and current state-of-the-art methods are apparently based on different approaches. Here we briefly explain the previous results for reachability queries, distance queries on complex networks, and distance queries on road networks. For details, please see Chapter 3.

Reachability Queries

One of the most classical approaches is to compress *transitive closure* [Sim88, vSdM11]. Whereas its query performance is excellent, even with compression, space complexity is still essentially quadratic, and thus this approach is not promising with regard to scalability.

In contrast, methods that conduct an online graph search guided by precomputed indices for answering each query achieve better scalability due to small indexing time and index size [CGK05, YCZ12, ZYQ$^+$12]. However, their query time is several orders of magnitude slower than other methods, which is critical for certain applications such as SPARQL engines and XQuery engines, as sometimes answers to thousands or millions of reachability queries are necessary to process one user query [YCZ12].

Methods based on *labeling* to vertices have also been studied for a long time [CHKZ03, STW04, CYL$^+$06, JXRF09]. They precompute a *label* for each vertex so that a reachability query can be answered from the labels of two endpoints. This approach is promising since, after obtaining small labels, they attain both fast query time and small index size. However, computing such labels has

been challenging and highly expensive, thus limiting the scalability of this approach.

Shortest-Path and Distance Queries on Complex Networks

Although the line of the research is almost independent, labeling-based approaches have been also studied for distance queries. However, as with reachability queries, for distance queries on complex networks, efficiently finding small labels is also a challenging and long-standing problem [CHKZ03, CY09, ADGW12]. One of the latest methods is *hierarchical hub labeling* [ADGW12], which is based on a method for road networks [ADGW11]. Another latest method related to 2-hop cover is *highway-centric labeling* [JRXL12].

An approach based on *tree decompositions* is also reported to be efficient [Wei10, ASK12]. It heuristically computes a tree decompositions and stores shortest-distance matrices for each bag. It is not hard to compute distances from this information.

Unfortunately, both of them highly suffer from drawback of scalability. They take at least thousands of seconds or tens of thousands of seconds to index networks with millions of edges [Wei10, ASK12, ADGW12, JRXL12].

Therefore, to handle larger complex networks, apart from these exact methods, approximate methods are also studied. That is, we do not always have to answer correct distances. They are successful in terms of much better scalability and very small average relative error for random queries. However, some of these methods take milliseconds to answer queries [GBSW10, TACGBn+11, QCCY12], which is about three orders of magnitude slower than other methods. Some other methods answer queries in microseconds [PBCG09, VFD+07], but it is reported that precision of these methods for close pairs of vertices is not high [QCCY12, ASK12]. This drawback might be critical for applications such as socially-sensitive search or context-aware search since, in these applications, distance queries are employed to distinguish close items.

Shortest-Path and Distance Queries on Road Networks

In comparison to reachability queries and distance queries on complex networks, shortest-path queries on road network has a larger body of research. Exhaustive survey is given in [BDG+14], and detailed experimental comparison of recent methods is given in [WXD+12].

Methods of an early date are based on the Dijkstra search and they decrease the number of visited vertices by *guiding* the search using precomputed data, which are called *goal-directed techniques*. Representative methods include *ALT* [IHI+94, GH05], *reach* [Gut04] and *arc flags* [KMS06].

Another newer group of methods is based on *hierarchical techniques*. Their indices are hierarchical, where higher levels capture more important parts (e.g., highways). Among several hierarchical methods, the notable one is the *contraction hierarchy* algorithm [GSSD08], which is simple, elegant and highly scalable. However, its query time is several orders of magnitude slower than other state of the art methods.

More recently, the labeling-based approach became successful for distance queries on road networks [ADGW11, ADGW12, DGW13]. Interestingly, again, the labeling algorithms are totally different from those for other queries. In [ADGW11], labels for road networks are computed through contraction hierarchies.

1.4 Contributions

As we have seen above, state-of-the-art empirical methods are heuristics that highly depend on properties of a specific kind of queries and targeted real graphs. Hence, though problems are related and somewhat close, lines of research on different problems are almost independent, and different approaches have been developed for different popular problems, even when limiting to labeling-based methods.

In this thesis, we address this issue by proposing *pruned labeling algorithms*, which is based on a new unified principle that can be widely applied to these path-related queries. Our emphasis is mainly on the practical impact. We summarize the contributions below.

1.4.1 Pruned Labeling Algorithms

The main contribution of this thesis is to present indexing methods for a wide range of important path-related queries based on our new notion of *pruned labeling*. As we observed above, previous state-of-the-art indexing methods for different queries are almost independently developed, and thus they are based on different approaches. Even limiting to labeling-based methods, their labeling algorithms are different among those for different kinds of queries. By contrast, the methods presented in this thesis are based on the same notion of pruned labeling. Specifically, we design indexing methods for reachability queries on directed acycilic graphs, shortest-path queries on road networks, historical shortest-path queries on evolving networks, and top-k shortest-path queries on complex networks. We demonstrate that each of these methods is also comparable with state-of-the-art methods for each kind of query, thus showing exceptional generality of our unified approach.

The pruned labeling algorithm is first devised for shortest-path distance queries, and that for distance queries (without any enhancement) is in the simplest form. This basic form is called the *pruned landmark labeling* algorithm. Therefore, we first present the basic form of the pruned labeling algorithm for distance queries in Chapter 4. As with previous labeling methods, it is an exact method, that is, it always answers correct distance between arbitrary two points. It precomputes distance labels for vertices by performing a breadth-first search from every vertex. Seemingly too obvious and too inefficient at first glance, the key ingredient introduced here is pruning during breadth-first searches. While we can still answer the correct distance for any pair of vertices from the labels, it surprisingly reduces the search space and sizes of labels.

Moreover, we also propose an efficient online incremental index update algorithm (Figure 1.6) for vertex and edge addition. To the best of our knowledge, our method is the first practical exact indexing method to efficiently process distance queries and dynamic graph updates. It basically conducts local pruned BFSs that visit only the vertices whose labels need to be updated. The idea behind it is to properly *resume* and *stop* BFSs.

Real-time index update would definitely improve user experience because these networks are highly dynamic and, more importantly, operations of users are *bursty* with regard to temporal locality [Bar05]. For example, on an online social networking service, when a user begins a friendship with another user, chances are high for the user to keep using the service for a few more minutes. Using our dynamic indexing method, the new friendship can be immediately reflected, which has never been possible with static methods that require periodic

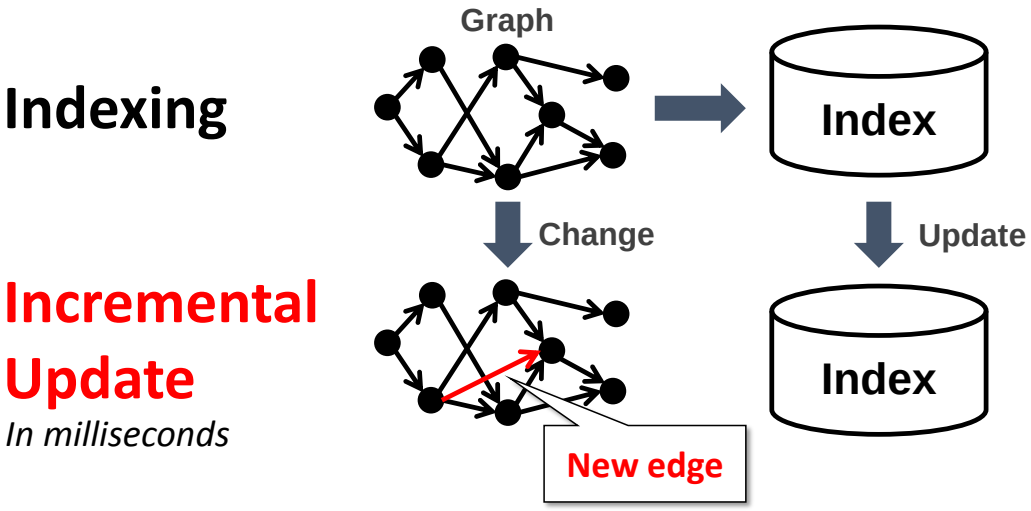


Figure 1.6: A general illustration of incremental index update.

index reconstruction.

This is joint work with Yoichi Iwata and Yuichi Yoshida. A part of this work was published in an extended abstract on Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD 2013; research track full paper) [AIY13]. The other part of this work was published in another extended abstract on the Proceedings of the 23rd International Conference on World Wide Web (WWW 2014; research track full paper) [AIY14].

1.4.2 Bit-parallel Labeling for Unweighted Complex Networks

Then, we apply our pruned labeling approach to distance queries on complex networks. While solely using the new pruned labeling algorithm is already competitive with previous methods, to gain further scalability, we propose another technique to exploit unweightedness of these real complex networks. We show that we can perform 32 or 64 breadth-first searches simultaneously exploiting bitwise operations, which enables more strong exploitation of the structure of these networks.

We experimentally demonstrate that the combination of these two techniques is efficient and robust on various kinds of large-scale real-world networks. In particular, our method can handle social networks and web graphs with hundreds of millions of edges, which are two orders of magnitude larger than the limits of previous exact methods, with comparable query time to those of previous methods.

This result was achieved in joint work with Yoichi Iwata and Yuichi Yoshida. An extended abstract was published in the same paper on the Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD 2013; research track full paper) [AIY13].

1.4.3 Path-based Labeling for Directed Acyclic Graphs

Next, we consider methods based on pruned labeling for reachability queries. Again, the original pruned labeling algorithm itself is already competitive with previous methods for reachability queries. However, we also present an extension of pruned landmark labeling algorithm named the *pruned path labeling* algorithm, which exploits the fact that we are only interested in reachability here.

We experimentally compared our method with other state-of-the-art scalable methods. Our experimental results show that they have good scalability and can be applied to graphs with tens of millions of vertices and edges, their query time is the fastest among the methods and two orders of magnitude faster than online-search-based methods, and their index size is an order of magnitude smaller than transitive-closure-based methods.

This is joint work with Yosuke Yano, Yoichi Iwata and Yuichi Yoshida. An extended abstract was also published in the Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM 2013; DB track short paper) [YAIY13]. In particular, the experiments in this part were done with considerable help from the first author.

1.4.4 Highway-based Labeling for Road Networks

Then, we move on to shortest-path distance queries on road networks. As road networks have quite a different structure from othe networks, we present a new framework (i.e. data structure and query algorithm) referred to as *highway-based labelings* and a preprocessing algorithm for it named *pruned highway labeling* based on pruned landmark labeling. Our proposed method has several appealing features from different aspects in the literature. Indeed, we take advantages of theoretical analysis of the seminal result by Thorup for distance oracles [Tho04], more detailed structures of real road networks, and the pruned labeling algorithm that conducts *pruned* Dijkstra's algorithm.

The experimental results show that the proposed method is comparable to the previous state-of-the-art labeling method in both query time and in data size, while our main improvement is that the preprocessing time is much faster.

This is joint work with Yuki Kawata, Yoichi Iwata and Ken-ichi Kawarabayashi. An extended abstract was also published in the Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX 2014) [AIKK14]. In particular, the experiments in this part were done with considerable help from the last author.

1.4.5 Historical Queries for Evolving Complex Networks

In addition to previous three popular kinds of graph querying problems, we propose new kinds of useful path-related queries that can also be efficiently processed by methods based on pruned labeling. We first consider *historical* shortest-path distance queries on time-evolving complex networks.

When analyzing historical networks, for which timestamps of vertices and edges are also available, in addition to the latest snapshot, the shortest paths and distances on previous snapshots or transition of them by time are also of interest. We call such queries about previous snapshots *historical queries*. In particular, we study two kinds of historical queries. A *snapshot query* asks the shortest path or distance on a specified previous snapshot, and a *change-point query* asks all the moments when the distance between two vertices has changed.

Indexing schemes supporting historical queries would be a powerful back-end for time-evolving network analysis, as it enables many new interesting studies. For example, from the transition of the shortest paths between two vertices, we can grasp the events that shortened the distance or important links that lie in the shortest paths for a long duration, which would provide valuable insights. Moreover, it would also enable distance-based analysis of influential people and communities [KKT03, BHKL06] on dynamic networks. In particular, transition

of *closeness centrality* and *distance distribution* can be efficiently computed with historical change-point queries.

Experimental results show the efficiency and robustness of our method based on pruned labeling. They can construct indices from large networks with millions of vertices, and their query time is very small and around microseconds, which are competitive with previous static methods. Meanwhile, the proposed methods can update their indices for single graph modification in around milliseconds, which is several orders of magnitude faster than reconstructing indices from scratch.

This result was achieved in joint work with Yoichi Iwata and Yuichi Yoshida. An extended abstract was published in the Proceedings of the 23rd International Conference on World Wide Web (WWW 2014; research track full paper) [AIY14].

1.4.6 Top-k Distance Queries on Complex Networks

Then, we also propose an indexing scheme for top-k shortest-path distance queries on graphs, which is useful in a wide range of important applications such as network-aware searches and link prediction. While many efficient methods for answering standard (top-1) distance queries have been developed, none of these methods are directly extensible to top-k distance queries. We develop a new framework for top-k distance queries and then present an efficient indexing algorithm based on our *pruned landmark labeling* scheme. The scalability, efficiency and robustness of our method is demonstrated in extensive experimental results.

This is joint work with Takanori Hayashi, Nozomi Nori, Yoichi Iwata and Yuichi Yoshida. An extended abstract is in press for the Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015; technical track paper) [AHN+15]. The experiments in this part were done with help from the second author.

1.4.7 Treewidth and Empirical Graph Tractability

The final part of this thesis contributes to the field of practical graph indexing methods in a different direction from the other contributions above. In this chapter, we tackle the long-standing question in this field: what is the key factor in addition to network size that has a large effect on the size of constructed indices for graph path queries? As discussed above, practical indexing methods exploits the structure of graphs, and thus their performance cannot be fully predicted only by sizes of networks.

We take the first big step in this direction by using *tree decomposition* [RS84]. We propose a new heuristic tree decomposition algorithm based on the new notion of *star-based representation*, that is more scalable than previous algorithms. Then, we apply our algorithm to various network instances and compare the results with performance numbers of state-of-the-art labeling algorithms. The results indicate that the difficulty of network instances for labeling-based indexing algorithms can be measured to some extent by heuristically obtaining tree decompositions.

This result was achieved in joint work with Yoichi Iwata, Takanori Maehara, and Ken-ichi Kawarabayashi. A part of this work is published as an extended abstract the in Proceedings of the VLDB Endowment, Vol. 7, No. 12 (PVLDB 2014; research track full paper) [MAIK14], and the other part is in preparation [AMK14].

1.5 Organization of This Thesis

This thesis is organized as follows. Figure 1.7 illustrates the structure of this thesis. In Chapter 2, we give preliminaries of this thesis such as definition from graph theory and basic graph algorithms. Chapter 3 explains the related work concerning path-related querying methods. Chapters 4–10 contain the contributions outlined in Section 1.4 above. Specifically, Chapter 4 explains the basic form of our pruned labeling algorithms (Section 1.4.1). In Chapter 5, we present the bit-parallel labeling scheme for further scalability on unweighted complex networks (Section 1.4.2). In Chapter 6, we describe our *pruned path labeling* method specialized for reachability queries (Section 1.4.3). Chapter 7 is devoted to explain the *pruned highway labeling* method for road networks (Section 1.4.4). In Chapter 8, the extension of pruned labeling for *historical queries* on time-evolving networks is presented (Section 1.4.5). Chapter 9 gives another extension of pruned labeling for top-k distance queries (Section 1.4.6). In Chapter 10, we present the relationship between treewidth and empirical graph tractability (Section 1.4.7). We conclude in Chapter 11.

1. Pruned labeling algorithms (Chapters 4–9)
   (a) Basic form [AIY13] (Chapter 4)
   (b) Further techniques for major graph queries (Chapters 5–7)
      i. Distance queries on complex networks [AIY13] (Chapter 5)
      ii. Reachability queries on DAGs [YAIY13] (Chapter 6)
      iii. Distance queries on road networks [AIKK14] (Chapter 7)
   (c) New graph queries and their usefulness (Chapters 8–9)
      i. Historical queries on evolving networks [AIY14] (Chapter 8)
      ii. Top-k queries on complex networks [AHN$^+$15] (Chapter 9)
2. Treewidth and empirical tractability [MAIK14, AMK14] (Chapter 10)

Figure 1.7: Organization of this thesis.

Chapter 2

Preliminaries

In this thesis, we focus on networks that are modeled as graphs. In this chapter, we first describe the necessary definitions from graph theory (Section 2.1), then review fundamental algorithms for graphs (Section 2.2), and finally explain common structural properties of real graphs (Section 2.3).

List of notations For better readability, please refer to Table 2.1 for the list of notations used throughout this thesis.

Table 2.1: Notations

Notation	Description
$G = (V, E)$	A graph.
$V(G)$	The vertex set of graph G.
$E(G)$	The edge set of graph G.
n	The number of vertices in graph G.
m	The number of edges in graph G.
$N_G^-(v), N_G^+(v)$	The in-neighbors and out-neighbors of vertex v in graph G.
$w(e)$	The weight of edge e.
$\mathcal{P}_{st}$	The set of all (not necessarily simple) paths from s to t.
$d_G(u, v)$	The shortest-path distance from vertex u to v in graph G.
$P_G(u, v)$	Set of all the vertices on the shortest paths between vertex u and v in graph G.

2.1 Definitions

2.1.1 Undirected and Directed Graphs

We consider two kinds of graphs: undirected graphs and directed graphs, defined below.

Definition 2.1 (Undirected Graph)**.** *An* undirected graph G *is a pair* $G = (V, E)$ *where* $E \subseteq \binom{V}{2}$.

Definition 2.2 (Directed Graph)**.** *A* directed graph G *is a pair* $G = (V, E)$ *where* $E \subseteq V \times V$.

V is called *vertices* and E is called *edges*. Vertices are also referred to as *nodes* or *points*. We use symbols n and m to denote the number of vertices $|V|$ and the number of edges $|E|$, respectively, when the graph is clear from the context. We also denote the vertex set and edge set of graph G by $V(G)$ and

$E(G)$, respectively. We often assume that vertices are uniquely represented by integers (i.e., $V = \{v_1, v_2, \ldots, v_n\}$), enabling natural comparisons of two vertices $u, v \in V$ by expressions such as $u < v$ or $u \leq v$.

We often explain algorithms assuming that given graphs are undirected. Specifically, algorithms are explained for undirected graphs in Chapters 4, 5, 7, 8, 9 and 10. This is for simplicity of exposition, and, as briefly discussed in each chapter, it is easy to obtain corresponding algorithms for directed graphs, On the other hand, in Chapter 6, we discuss algorithms for directed graphs from the beginning, as direction is essential for the problem setting of that chapter.

Following the definition above, the precise notation of an undirected edge between two vertices u and v in an undirected graph is set $\{u, v\}$. However, following the tradition in this field, pair (u, v) is also considered to describe the edge $\{u, v\}$ in this thesis. Note that, on undirected graphs, pairs (u, v) and (v, u) describe the same edge and indistinguishable. This is in order to make some portion of the discussion common among undirected and directed graphs. In what follows, unless mentioned (e.g., when we simply say "*a graph G*"), discussions apply both to undirected and directed graphs.

2.1.2 Adjacency, Neighbors and Degree

We then introduce the basic notion of *adjacency*, *neighbors* and *degree*.

Definition 2.3 (Adjacency)**.** *For a graph $G = (V, E)$, two vertices $u, v \in V$ are called* adjacent *if there is an edge between them, i.e., $(u, v) \in E$ or $(v, u) \in E$.*

Definition 2.4 (Neighbors)**.** *For a graph $G = (V, E)$ and two vertices $u, v \in V$, u is called an* in-neighbor *of v if $(u, v) \in E$. The set of the in-neighbors of vertex v is denoted by $N_G^-(v)$, i.e., $N_G^-(v) = \{u \in V \mid (u, v) \in E\}$. Similarly, for a graph $G = (V, E)$ and two vertices $u, v \in V$, u is called an* out-neighbor *of v if $(v, u) \in E$. The set of the out-neighbors of vertex v is denoted by $N_G^+(v)$, i.e., $N_G^+(v) = \{u \in V \mid (v, u) \in E\}$.*

Definition 2.5 (Degree)**.** *For a graph $G = (V, E)$ and a vertex v, the* in-degree *and* out-degree *of v are defined as the number of its in-neighbors and out-neighbors, that is, $\left|N_G^-(v)\right|$ and $\left|N_G^+(v)\right|$, respectively.*

When the graph is clear from the context, we omit the subscripts, e.g., in-neighbors of a vertex v may be described as $N^-(v)$. Please note that, on undirected graphs, for any vertex v, its in-neighbors and out-neighbors are the same (and thus its in-degree and out-degree are also the same). Therefore, for undirected graphs, we simply call them *neighbors* and *degree* and we omit the superscripts, e.g., neighbors of a vertex v may be described as $N_G(v)$ or $N(v)$.

2.1.3 Edge Weight, Paths and Distance

We sometimes consider *weighted* graphs to represent costs, time or geometric distance of an edge.

Definition 2.6 (Weighted Graph)**.** *A* weighted graph *is a graph $G = (V, E)$ associated with a weight function $w : E \to \mathbb{R}$. For an edge $e \in E$, we call $w(e)$ the* weight *of the edge.*

In this thesis, unless stated otherwise, we assume that the weight of any edge is positive, i.e., $w : E \to \mathbb{R}^+$. This is mainly due to fundamental technical reason, which will be discussed in Section 2.2.2. However, real-world networks rarely

have negative weights (as they come from costs, time and geometric distance), and thus this assumption seldom limits the applicability of our algorithms.

In contrast to weighted graphs, a graph that is not associated with a weight function is called an *unweighted graph*. For an unweighted graph $G = (V, E)$, we consider the weight of any edge as one, i.e., we consider a weight function w where $w(e) = 1$ for any $e \in E$.

Next, we define paths and their lengths.

Definition 2.7 (Path)**.** *For a graph $G = (V, E)$ and two vertices $s, t \in V$, a* path *from s to t is a sequence of edges $((u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k))$ where $u_0 = s$ and $u_k = t$.*

Definition 2.8 (Cycle)**.** *A* cycle *is a non-empty path whose both endpoints coincide.*

Definition 2.9 (Simple Path)**.** *A path P is called* simple *if it does not pass any vertex more than once.*

Definition 2.10 (Path Length)**.** *For a path P, its* length *$w(P)$ is defined as the sum of the weight of the edges, that is, $w(P) = \sum_{e \in P} w(e)$.*

In this thesis, for a pair of vertices (s, t), let $\mathcal{P}_{st}$ be the set of all (not necessarily simple) paths from s to t. If $\mathcal{P}_{st} \neq \emptyset$, we say that t is *reachable* from s, and otherwise we say that t is *unreachable* from s.

Then, we define *shortest paths* and *shortest-path distance*. In the following, we consider graphs without cycles with negative length, because shortest paths and distance cannot be defined otherwise.

Definition 2.11 (Shortest Path)**.** *For a graph $G = (V, E)$ and two vertices $s, t \in V$, a path P from s to t is called a* shortest path *if $w(P) = \min_{Q \in \mathcal{P}_{st}} w(Q)$.*

Definition 2.12 (Shortest-Path Distance)**.** *For a graph $G = (V, E)$ and two vertices $s, t \in V$, we define* distance *$d_G(s, t)$ from s to t as the length of a shortest path between them, i.e., $d_G(s, t) = \min_{Q \in \mathcal{P}_{st}} w(Q)$.*

For unreachable pairs, following the tradition in this field, we abuse the notation by defining $d_G(s, t) = \infty$, where ∞ is considered to be a very large number. Note that, on undirected graphs, distance is symmetry, i.e., $d_G(s, t) = d_G(t, s)$, whereas it is not always the case on directed graphs. As the shortest-path distance in graphs is a metric, it satisfies the *triangle inequalities*.

Lemma 2.1 (Triangle Inequality)**.** *For a graph $G = (V, E)$ and three vertices $s, u, t \in V$, the following inequalities hold.*

$$d_G(s, t) \leq d_G(s, v) + d_G(v, t), \tag{2.1}$$

$$d_G(s, t) \geq |d_G(s, v) - d_G(v, t)|. \tag{2.2}$$

We define $P_G(s, t) \subseteq V$ as the set of all vertices on the shortest paths between vertices s and t. In other words,

$$P_G(s, t) = \{v \in V \mid d_G(s, v) + d_G(v, t) = d_G(s, t)\}.$$

The *diameter* of a graph is defined below using shortest-path distance.

Definition 2.13 (Diameter)**.** *For a graph $G = (V, E)$, the diameter of G is defined as*

$$\max_{u, v \in V, d_G(u, v) \neq \infty} d_G(u, v).$$

2.1.4 Strongly and Weakly Connected Components

Definition 2.14 (Strongly Connected)**.** *For a graph $G = (V, E)$, a pair of vertices $s, t \in V$ are called* strongly connected *if both $d_G(s, t)$ and $d_G(t, s)$ are finite. A graph is called strongly connected if any pair of vertices are strongly connected.*

Definition 2.15 (Weakly Connected)**.** *For a graph $G = (V, E)$, a pair of vertices $s, t \in V$ are called* weakly connected *if there is a sequence of vertices $(u_1 = s, u_2, \ldots, u_{k-1}, u_k = t)$ where $(u_i, u_{i+1}) \in E$ or $(u_{i+1}, u_i) \in E$ for any $1 \leq i \leq k - 1$. A graph is called weakly connected if any pair of vertices are weakly connected.*

Both strong connectivity and weak connectivity are equivalence relations. Therefore, we often consider natural partitions defined by the relations, which are called *strongly* and *weakly connected components*.

Definition 2.16 (Strongly Connected Components)**.** *For a graph,* strongly connected components *are all the maximal vertex sets where any pair of vertices in each set is strongly connected.*

Definition 2.17 (Weakly Connected Components)**.** *For a graph,* weakly connected components *are all the maximal vertex sets where any pair of vertices in each set is weakly connected.*

For an undirected graph, strong connectivity and weak connectivity are equivalent, and thus we simply call such pairs or graphs *connected*. Similarly, we use the term *connected components* for undirected graphs.

Directed acyclic graphs (DAGs) are directed graphs which are opposite from strongly connected graphs, defined below.

Definition 2.18 (Directed Acyclic Graph)**.** *A directed graph is called a* directed acyclic graph *if no pair of vertices is strongly connected.*

In other words, directed acyclic graphs are graphs without cycles.

2.1.5 Trees and Shortest-Path Trees

Definition 2.19 (Undirected Tree)**.** *An undirected graph $G = (V, E)$ is called an* undirected tree *if it is connected and $|E| = |V| - 1$.*

In other words, an undirected tree is a connected graph without any cycles. Undirected trees are also called *unrooted trees*. Another equivalent definition of an undirected tree is a undirected graph on which there is exactly one simple path between any pair of vertices.

Definition 2.20 (Directed Tree)**.** *A directed graph $G = (V, E)$ is called a* directed tree *if it is weakly connected, $|E| = |V| - 1$, and there is a vertex r where any vertex $v \in V$ is reachable from r. Such a vertex r is called a* root*.*

In other word, a directed graph is a directed tree if and only if there is a vertex r (root vertex) where, for any vertex v, there is exactly one simple path from r to v. In a directed tree, there is exactly one root vertex.

For a directed tree T and two vertices $u, v \in V(T)$, if there is a directed edge $(u, v) \in E(T)$, u is called the *parent* of v, and v is called a *child* of u. Any vertex except the root has exactly one parent, and the root has no parent.

Definition 2.21 (Shortest-Path Tree)**.** *For a graph G and vertex s, a directed tree T is a* shortest-path tree *from s if its root is s, its vertex set $V(T)$ is the set of reachable vertices from s on G, and, for any vertex $u \in V(T)$, the path from s to u on T is a shortest path from s to u on G.*

As we will discuss in Section 2.2.2, computing a shortest-path tree from a vertex is called the *single source shortest path problem*. A shortest-path tree can be obtained in near-linear time.

2.1.6 Planar Graphs, Tree Decomposition and Minor-Closed Properties

Then, we introduce theoretical graph families that are relevant to our study. We start from introducing one of the most fundamental and natural graph families, *planar graphs*. As they are not necessary in the other parts of this thesis, we omit the detailed definition of notions such as *plane*, *drawing* and *intersection*.

Definition 2.22 (Planar Graph)**.** *A graph is* planar *if it can be drawn on the plane under the restriction that its edges intersect only at their endpoints.*

Then, we introduce another important notion called *tree decomposition* and its related notions, *width* and *treewidth*. Tree decomposition and the related notions were first proposed by Halin [Hal76], and then rediscovered by Robertson and Seymour [RS84] and, by Arnborg and Proskurowski [AP89], independently.

Definition 2.23 (Tree Decomposition)**.** *A* tree decomposition *of a graph G is a pair $(T, \mathscr{X})$, where T is a tree and $\mathscr{X} = \{X_t\}_{t \in T}$ is a family of subsets of $V(G)$, with the following properties:*

1. $\bigcup_{t \in V(T)} X_t = V(G)$.
2. *For every $(u, v) \in E(G)$, there exists $t \in V(T)$ such that $u, v \in X_t$.*
3. *For all $v \in V(G)$, the set $\{t \mid v \in X_t\}$ induces a subtree of T.*

In this thesis, we call the sets X_t *bags*. Figure 2.1 is an example of a graph and one of its tree decompositions. Rectangles denote the bags. Circles and blue lines denote the vertices and the edges of the original graph. Red lines connect the same vertices between adjacent bags.

Then, we define the *width* of a tree decomposition and *treewidth* for a graph.

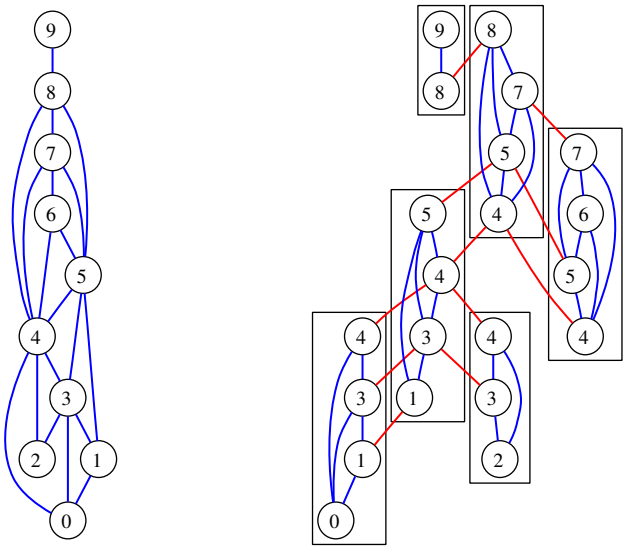
Definition 2.24 (Width)**.** *The* width *of a tree decomposition $(T, \mathscr{X})$ is*

$$\max_{t \in V(T)} \{|X_t| - 1\}.$$

Definition 2.25 (Treewidth)**.** *A graph G has* treewidth *w if w is the minimum such that G has a tree decomposition of width w.*

Intuitively, a tree decomposition with small width enables to deal with the graph like a tree, and thus, a graph with small treewidth is algorithmically *easy* like trees. We will use tree decomposition in the theoretical analysis of our algorithm (Section 4.3) and empirical study on real network properties (Chapter 10).

Finally, we introduce the definition of *minor-closed properties*. To that end, we first introduce *contraction* of an edge in a graph.



(a) A graph

(b) One of its tree decompositions

Figure 2.1: An example of tree decomposition.

Definition 2.26 (Contraction)**.** *Given a graph $G = (V, E)$ and an edge $e = (u, v) \in E$,* contracting *edge e converts the graph G to $G' = (V', E')$, where, letting v_e be a new vertex, $V' = (V \setminus \{u, v\}) \cup \{v_e\}$ and E' is obtained by removing e and replacing the occurrence of u and v by v_e.*

Roughly speaking, *contraction* of an edge $e = (u, v)$ is an operation of removing the edge e and merging u and v into a new vertex. Intuitively, contraction of an edge is to make the edge shorter and shorter until its endpoints meet.

Then, we define *minor* and *minor-closed property.*

Definition 2.27 (Minor)**.** *A graph H is called a* minor *of a graph G if H is obtained from G by removing vertices, removing edges, and contracting edges.*

Definition 2.28 (Minor-closed Property)**.** *A graph property P is* minor-closed *if any minor of a graph satisfying P satisfies P.*

As easily observed, minor-closed properties are generalization of planarity and bounded treewidth, That is, planarity is a minor-closed property, and having bounded treewidth is another minor-closed property.

Seemingly, minor closeness is a quite simple and natural class of graph properties, but actually it has turned out that the minor closeness of a graph family solely gives strong information on it. One of the most notable results on minor closeness is *Robertson-Seymour theorem* (also known as the *graph minor theorem*) [RS04], which states that any minor-closed property is determined by a finite set of *forbidden minors*. This is generalization of Kuratowski's theorem [Kur30] (or Wagner's theorem [Wag37]) for planar graphs.

We will also use some part of the fertile results on minor-closed properties in the theoretical analysis of our algorithm (Section 6.3).

2.1.7 Dynamic Graphs

In this thesis, we sometimes consider dynamic graphs. We basically model a dynamic graph as a time series of graphs, and use symbol G_τ to denote the

network at time τ. For simplicity, we assume time is described by positive integers (i.e., graph snapshots are $G_1, G_2, \ldots$), and we define G_0 as an empty graph, i.e., $G_0 = (\emptyset, \emptyset)$. We use symbol G to denote the latest network.

We denote the distance between vertices u, v in graph G_τ by $d_\tau(u, v)$. For edge $(u, v) \in G$, we define $t(u, v) = \tau$, where τ is the time when the edge appeared in the graph (i.e., $(u, v) \notin E(G_{\tau-1})$ and $(u, v) \in E(G_\tau)$).

2.2 Fundamental Graph Algorithms

2.2.1 Algorithm Evaluation Criteria

In this thesis, emphasis is largely put on practical performance. Therefore, our main evaluation criteria is the real performance numbers on modern computer systems against real network datasets.

We sometimes use synthetic networks, but we prefer real graphs. This is because state-of-the-art methods are designed to exploit the common structures on real graphs, and, unfortunately, current synthetic models of networks are not sufficiently similar to real networks. Experimental results on synthetic graphs are often different from those on real graphs.

On the other hand, we also sometimes discuss theoretical complexities. For measuring time and space complexities, we assume the *word RAM model*. In the word RAM model with *word length* b, each register can hold an integer ranging from 0 to $2^b - 1$. Operations among registers such as addition, subtraction, bit shifts and bit-wise operations can be done in constant time. Registers are managed like an array, and indirect addressing is capable.

2.2.2 Single Source Shortest Path Algorithms

The *single source shortest path (SSSP) problem* is, given a graph $G = (V, E)$ and a source vertex $s \in V$, to compute a shortest-path tree from s or distance from s to all other vertices.

For an unweighted graph, a *breadth-first search (BFS)* can compute them in $O(n + m)$ time. For a DAG, even if it is weighted, they can be obtained by conducting *dynamic programming* in $O(n + m)$ time.

For a weighted graph, *Dijkstra's algorithm* [Dij59] can be applied if weight of any edge is non-negative. It runs in $O(n^2)$ time when not using priority queues. If we use standard priority queues such as binary heaps, it works in $O(m \log n)$ time. In theory, sophisticated priority queues such as fibonacci heaps [FT87] improve the time complexity to $O(m + n \log n)$, but it is slow in practice due to hidden large constant factor. For an undirected graph with non-negative weight, *Thorup's algorithm* [Tho99] have the best time complexity of $O(n+m)$ time, but it is also slow in practice.

For a weighted graph that may contain negative weight, these algorithms do not work correctly, and thus *Bellman-Ford algorithm* [Bel58, For56] is often employed. Its worst-case time complexity is $O(nm)$ time, but implementations using queues work much faster in practice than what is expected from the time complexity.

2.2.3 All Pairs Shortest Path Algorithms

The *all pairs shortest path (APSP) problem* is, given a graph $G = (V, E)$, to compute shortest-path trees from all the vertices or distances from any pair of

vertices. The obvious solution to this problem is to conduct a SSSP algorithm for n times from each vertex.

On the other hand, there are some algorithms designed for the APSP problem. The *Warshall-Floyd algorithm* is a popular algorithm [War62, Flo62], which is based on very simple dynamic programming and runs in $O(n^3)$ time. While its asymptotic time complexity is the same as conducting Dijkstra's algorithm without priority queues, The Warshall-Floyd algorithm is much faster because of its simpleness. However, as the real graphs that we deal with in this thesis have much less edges than $\Theta(n^2)$, and thus the former approach based on SSSP is much more efficient.

In theory, APSP algorithms based on algebraic approach have been also proposed. In particular, for undirected graphs with *small* weights, the algorithm based on matrix multiplication runs in $O(M(n) \cdot \text{polylog}(n))$, where $M(n)$ is the time complexity of the multiplication of two $n \times n$ matrices [Car71]. The current fastest matrix multiplication algorithm is by Le Gall [LG14] with $M(n) = O(n^\omega)$ where $\omega < 2.3728639$. However, these algorithms are too complex to implement, and also their empirical performance is not believed to be promising because of huge hidden constant factors.

2.3 Common Structural Properties of Real-world Graphs

Finally, in this section, we review the common structural properties of real-world graphs of our interest. The graph data that we deal with in this thesis can be roughly classified into two families: complex networks and road networks, where complex networks comprise various but structurally similar networks such as social networks, web graphs and computer networks. We review the structural properties of these two families.

2.3.1 Complex Networks

The structural properties of complex networks have been intensively studied in the Web, data mining and network science communities. Among many findings so far, we introduce those that are representative or closely related to our subsequent discussion.

Power-law Degree Distribution The degree distribution of various complex networks roughly follow a *power-law* [BA99, FFF99, MMG+07]. That is, for some constant γ, the fraction $p(k)$ of the vertices with degree k is proportional to $k^{-\gamma}$, i.e., $p(k) \propto k^{-\gamma}$. A power-law means that there are likely a few vertices that have exceptionally high degree, and many vertices have degree below the average, leading to the existence of highly central vertices (or *hubs*). Parameter γ is called *power exponent*, where $2 < \gamma < 3$ in most networks.

Small Average Distance and Small Diameter Average distance and diameters of complex networks are much smaller than what is intuitively expected from the numbers of vertices. This is probably famous for social networks together with the phrase "six degrees of separation" [Mil67, TM69]. Recently, on Facebook social network data, average distance is measured as 4.74, which rewrites the phrase to "four degrees of separation" [BBR+12, BV12].

Large Clustering Coefficient Intuitive explanation of the clustering coefficient is the probability that "a friend of a friend" is a friend [HL71]. Global clustering coefficient considers any pair of "a friend of a friend", while local clustering coefficient considers only friends of a person. It is known that, for real complex networks, global clustering coefficient is much larger than what is expected from random edge connection [HL71, WS98, MMG+07]. Moreover, local clustering coefficient is quite high especially for vertices with small degree [MMG+07]. This indicates that real complex networks have some kind of *locality*.

Core-Fringe Structure Generally, the *core-fringe structure*, *core-periphery structure* or *core-whisker structure* of complex networks state that a network can be roughly classified into two parts: the core part and fringe part, where the core part is relatively dense and well connected, while the fringe part is tree-like [RPFM14, MMG+07, CSTW12, MAIK14]. With regard to formal definition of the core-fringe structure, several different formulations have been proposed. Among them, Maehara et al. proposed that a core is a expander-like subgraph and a fringe is a subgraph of small treewidth [MAIK14], which is quite useful for algorithm designers because these two notions are closely related to graph theory and graph algorithms.

2.3.2 Road Networks

In contrast to complex networks, the structural property and its modeling is not as popular for road networks. This is probably because the structure of road networks is much more intuitive and seemingly straightforward. However, for us, algorithm designers, since we need to design algorithms for road networks, the structural properties of road networks are also important.

Small Separators While road networks are not precisely planar, they are close to planar graphs. Therefore, they share several common properties to planar graphs. Among them, existence of *small separators* is quite useful when designing algorithms [Tho04, KMS06, ADGW11]. It indicates that road networks can be decomposed into parts by removing relatively small set of edges or vertices.

Highways In contrast to general planar graphs, there are *highways* in road networks. Highways usually have higher average speed than other roads, and thus, one likely pass through highways when traveling long distance. In other words, long shortest paths are likely pass through (parts of) paths that correspond highways. Therefore, compared with complex networks, while complex networks have highly central vertices (i.e., hubs), road networks may not have vertices that are as central as these vertices, but instead, they have very popular paths (i.e., highways). This is our motivation of our *highway-based labeling framework* introduced in Chapter 7, which enable explicitly exploiting these highway paths.

Chapter 3

Review of Graph Indexing Methods

In this chapter, we review previous graph indexing methods for path-related queries. As mentioned above, in this field, theoretical and practical methods are almost totally independent. Since we focus on practical indexing methods in this thesis, in this chapter, we also mainly review previous practical methods. For theoretical results, we refer to [Som14]. In particular, as our method is closely related to indexing methods for distance queries on complex networks, these methods are of main interest in this chapter.

3.1 Shortest-path and Distance Queries on Complex Networks

For simplicity, we explain methods for answering distance queries, but it is also easy for almost all the methods to simultaneously answer the shortest paths. Methods are classified into two groups: exact methods and approximate methods. Exact methods always answer correct distance, while answers of approximate methods may contain error. In this section, we assume that given graph is undirected and unweighted, but discussion can be naturally extended for directed and/or weighted graphs.

3.1.1 Labeling Methods

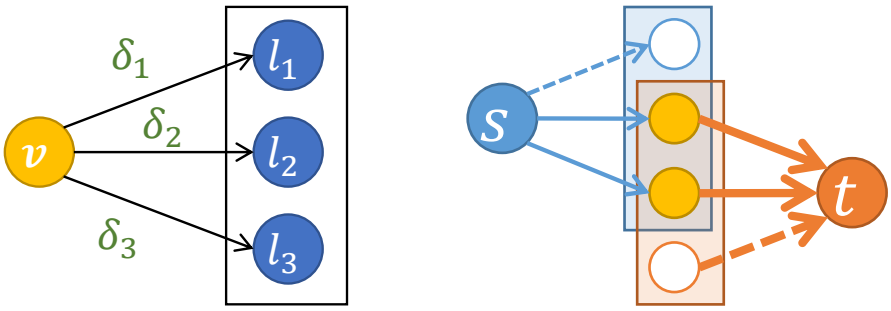
Large portion of these methods can be considered as *labeling methods*. Therefore, we first introduce labeling methods. Labeling methods are defined as indexing methods that precompute a *label* $L(v)$ for each vertex v, and answer a distance query between vertices s and t only by using two labels $L(s)$ and $L(t)$.

In this thesis, we refer to the data structure and query algorithm of a labeling method as a *labeling framework*. Most of the previous methods are based on the labeling framework called *2-hop cover* [CHKZ03]. Our method also follows this framework.

Index Data Structure

For each vertex v, we precompute a label denoted as $L(v)$, which is a set of pairs (u, δ_{uv}), where u is a vertex and $\delta_{uv} = d_G(u, v)$ (Figure 3.1a). We sometimes call the set of labels $\{L(v)\}_{v \in V}$ as an *index*. We call the index *correct* if it satisfies the following conditions.

Definition 3.1 (2-Hop Cover Index for Distance Queries [CHKZ03])**.** *Set of labels* $\{L(v)\}_{v \in V}$ *is a* (correct) 2-hop cover index for distance queries *of graph* $G = (V, E)$ *if* $L(v)$ *is a set of pairs* (u, δ_{uv})*, where* u *is a vertex and* $\delta_{uv} = d_G(u, v)$*, and, for any pair of vertices* $s, t \in V$*,*



(a) An illustration of a 2-hop label

(b) The query algorithm of the 2-hop cover framework

Figure 3.1: The index data structure and query algorithm of the 2-hop cover framework.

1. $L(s) \cap L(t) = \emptyset$ *if* $d_G(s,t) = \infty$, *and*
2. $\min\{\delta_{vs} + \delta_{vt} \mid (v, \delta_{vs}) \in L(s), (v, \delta_{vt}) \in L(t)\} = d_G(s,t)$ *if* $d_G(s,t) \neq \infty$.

For directed graphs, two labels are computed and stored for each vertex, called a *forward label* and *backward label*. *Hierarchical 2-hop covers* [ADGW12] (also called *hierarchical hub labelings*) are a natural special class of 2-hop covers defined below.

Definition 3.2 (Hierarchical 2-Hop Cover [ADGW12])**.** *A 2-hop cover index* $\{L(v)\}_{v \in V}$ *is a* hierarchical 2-hop cover index *of graph* $G = (V, E)$ *if the relationship "vertex* v *is in the label of vertex* w*" defines a partial order on the vertex set* V*.*

Moreover, *canonical 2-hop covers* [ADGW12] (also called *canonical hub labelings*) are a further special class of hierarchical 2-hop covers. Recent algorithms, including our pruned labeling algorithm, compute canonical 2-hop covers. Canonical 2-hop covers are defined as follows.

Definition 3.3 (Canonical 2-Hop Cover [ADGW12])**.** *For a graph* $G = (V, E)$*, a 2-hop cover index* $\{L(v)\}_{v \in V}$ *is a* canonical 2-hop cover index *of* G *if, for a total order* r *of vertices,* $L(s)$ *is the set of vertices that ranks the highest in* $P_G(s,t)$ *with respect to* r *for any vertex* t*.*

Note that, given a total order, the canonical 2-hop cover is unique.

Query Algorithm

To answer a distance query between vertices s and t, we compute and answer $\textsc{Query}(s, t, L)$ defined as follows (Figure 3.1b),

$$\textsc{Query}(s, t, L) = \min\{\delta_{vs} + \delta_{vt} \mid (v, \delta_{vs}) \in L(s), (v, \delta_{vt}) \in L(t)\}.$$

We define $\textsc{Query}(s, t, L) = \infty$ if $L(s)$ and $L(t)$ do not share any vertex. It is obvious that, if L is a correct 2-hop cover index, $\textsc{Query}(s, t, L) = d_G(s,t)$ for any pair of vertices s and t.

For each vertex v, we store the label $L(v)$ so that pairs in it are sorted by their vertices. Then, we can compute $\textsc{Query}(s, t, L)$ in $O(|L(s)| + |L(t)|)$ time using a merge-join-like algorithm.

Indexing Algorithms

As we have seen above, the indexing framework (i.e., the index data structure and query algorithm) of the 2-hop cover framework is simple. However, finding small 2-hop covers efficiently is a challenging and long-standing problem. As the minimization of the size of labels is proven to be NP-hard, approximate algorithms and heuristic algorithms have been studied.

In the original work [CHKZ03], Cohen et al. proposed an approximate labeling algorithm that guarantees the approximation ratio of $O(\log n)$ from the optimal 2-hop cover. However, the algorithm first computes the distance matrix, then reduces the label computation to an optimization problem. Therefore, the scalability is highly limited. A faster labeling algorithm built on this approach with the same approximation guarantee was recently presented by Delling et al., which is orders of magnitude better than previous algorithm [DGSF14]. Nevertheless, it can process networks with at most tens of thousands of edges.

More scalable algorithm that computes canonical 2-hop covers was proposed by Delling et al. [ADGW12]. It is based on a method for road networks [ADGW11], which will be introduced in later sections. It first constructs hierarchical 2-hop covers through *contraction hierarchies*, then it reduces the label sizes by making them canonical. It also requires $\Theta(n^2)$ space.

Extensions

A variant related to 2-hop cover is *highway-centric labeling* [JRXL12]. In this method, we first compute a spanning tree T and use it as a "highway". That is, when computing distance $d_G(u, v)$ between two vertices u and v, we output the minimum over $d_G(u, w_1) + d_T(w_1, w_2) + d_G(w_2, v)$ where w_1 and w_2 are vertices in labels of u and v, respectively, and $d_T(\cdot, \cdot)$ is the distance metric on the spanning tree T.

IS-Label is another method that combines *partial* 2-hop cover and online graph searches [FWCW13]. To create an index, it recursively computes a vertex cover and *reduce* vertices in the corresponding independent set. Simultaneously, it adds label entries to the labels of these removed vertices. The recursion is stopped when the graph gets sufficiently small. To answer queries, it reduces the problem to many-to-many distance computation on the reduced graph by using the partial 2-hop labels.

3.1.2 Tree-Decomposition-Based Methods

An approach based on heuristic tree decomposition is also reported to be efficient [Wei10, ASK12]. Tree decomposition and complex networks are seemingly unrelated at first glance, as intuitively treewidth of complex networks is not small. However, because of the *core–fringe structure* of these networks [CNSW00, NSW01], the fringe part is actually tree-like, and these methods exploit the tree-like structure by tree decomposition.

Index Data Structure

Since complex networks do *not* have small treewidth, to exploit the tree-like fringes, new notion of *relaxed tree decomposition* is used in these methods.

Definition 3.4 (Relaxed Tree Decomposition [Wei10, ASK12])**.** *A tree decomposition is called a* relaxed tree decomposition *with* width w *if all bags except for one have size at most* $w + 1$*.*

We call the one arbitrary large bag the *root bag*. As an index, these methods precompute and store a tree decomposition and a distance matrix for each bag of it. The index size is $O(nw^2 + r^2)$, where w is the relaxed width of the tree decomposition, and r is the size of the root bag.

Indexing Algorithm

They compute a tree decomposition by algorithms built on the *min-degree heuristic* algorithm [BHS03]. As we will also use the min-degree heuristic in Chapter 10, we will explain the algorithm later.

The original method proposed by Wei [Wei10] computes the distance matrices after obtaining tree decompositions (Figure 3.2). In contrast, Akiba et al. proposed to simultaneously compute the distance matrices on *reduced graphs* of min-degree heuristics [ASK12]. This technique improved the scalability by an order of magnitude.

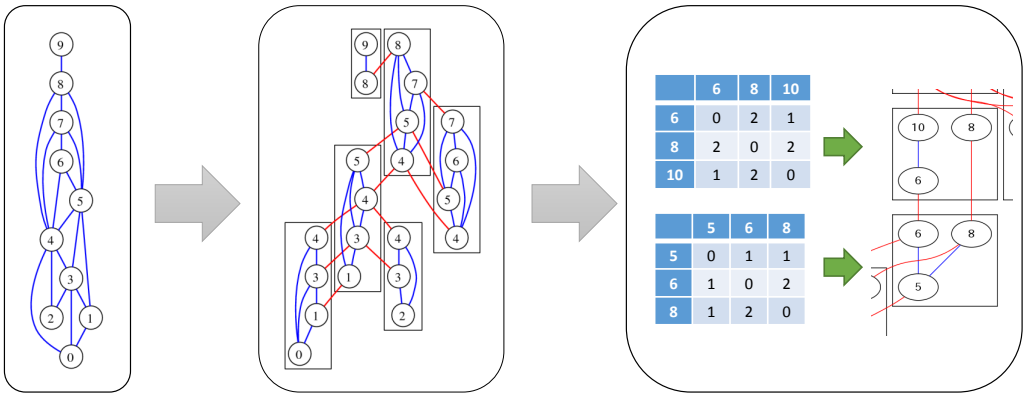


Figure 3.2: The index construction process of tree-decomposition-based approaches.

Query Algorithm

To answer queries, we conduct dynamic programming on the trees (Figure 3.3). The basic algorithm by Wei works as follows [Wei10]. We start from two bags that contain the specified endpoints. We ascend the bags by computing distances from endpoints to vertices in each bag. We stop the process at the lowest common ancestor of the two bags. It works in $O(w^2 h)$ time, where h is the height of the tree decomposition. Akiba, et al. proposed more sophisticated query algorithms that work in $O(w^2 \log h)$ time [ASK12].

3.1.3 Landmark-based Methods

To gain more scalability than these exact methods, approximate methods, which do not always answer correct distances, also have been studied. The major approach is the *landmark-based approach* [TC03, VFD+07].

The basic idea of these methods is to select a subset L of vertices as landmarks, and precompute the distance $d_G(\ell, u)$ between each landmark $\ell \in L$ and all the vertices $u \in V$. When the distance between two vertices u and v is queried, we answer the minimum $d_G(u, \ell) + d_G(\ell, v)$ over landmarks $\ell \in L$ as an estimate.

Generally, the precision for each query depends on whether actual shortest paths pass nearby the landmarks. Therefore, by selecting central vertices as landmarks, the accuracy of estimates becomes much better than selecting landmarks

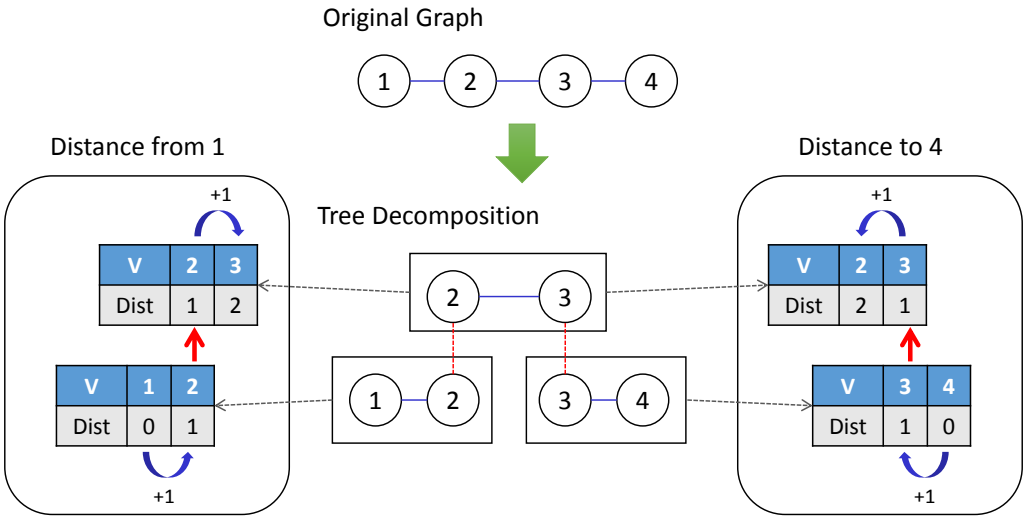


Figure 3.3: The query algorithm of tree-decomposition-based distance querying.

randomly [PBCG09, CSTW12]. However, for close pairs, the precision is still much worse than the average, since lengths of shortest paths between them are small and they are unlikely to pass nearby the landmarks [ASK12].

To further improve the accuracy, several techniques were proposed [GBSW10, TACGBn+11, QCCY12]. They typically store shortest-path trees rooted at the landmarks instead of just storing distances from the landmarks. To answer queries, they extract paths from the shortest-path trees as candidates of shortest-paths, and improve them by finding loops or shortcuts. While they significantly improve the accuracy, the query time becomes up to three orders of magnitude slower.

3.2 Shortest-path and Distance Queries on Road Networks

Shortest-path distance queries on road network has a larger body of research. In this section, we focus on two representative recent methods that are related with methods of our interest: contraction hierarchies and labeling methods. Exhaustive survey is given in [BDG+14], and detailed experimental comparison of recent methods is given in [WXD+12]. In this section, as we deal with road networks, we assume that given graph $G = (V, E)$ is weighted and has a weight function w.

3.2.1 Contraction Hierarchies

The idea based on contraction hierarchies is just to repeatedly *contract* vertices by increasing order of *importance* to obtain two DAGs (upward graph and downward graph), on which query algorithms can efficiently obtain shortest-paths [GSSD08].

Index Data Structure

From the given graph, indexing algorithms construct two weighted DAGs, *upward graph* $G_\uparrow = (V, E_\uparrow)$ and *downward graph* $G_\downarrow = (V, E_\downarrow)$. The two DAGs respect a total order of vertices called *importance* in a reverse way. That is, for any $(u, v) \in E_\uparrow$, u has a lower importance than v, and, for any $(u, v) \in E_\downarrow$, u has a higher importance than v. A weighted graph $G' = (V, E_\uparrow \cup E_\downarrow)$ is called a *contraction hierarchy* of the graph G.

Indexing Algorithm

We construct a contraction hierarchy basically by repeatedly *contracting* vertices (Figure 3.4). First, both of the upward and downward graph are initialized with the edges of original graph with the corresponding directions. Contraction of a vertex v is to remove vertex v and add edges that are necessary not to change the distance between at least one pair of remaining vertices.

When contracting less important vertices, as shortest paths are less likely to pass these vertices, new edges are less likely to be added. Therefore, vertices are contracted by increasing order of importance. The order is heuristically determined simultaneously with contraction process.

Query Algorithm

The query algorithm is just to compute a shortest path from two endpoints in the contraction hierarchy G'. We can conduct bidirectional search on it.

Relation to Tree-Decomposition-Based Methods

Interestingly, indexing algorithms of contraction hierarchies have much in common with tree-decomposition-based methods introduced above. As we will explain in Chapter 10 the min-degree heuristic algorithm computes tree decompositions by *reducing* vertices. This reduction process is quite similar to the contraction above. However, one crucial difference is that, while the tree decomposition process adds edges between every pair of neighbors, the contraction process only adds necessary edges for remaining shortest paths. If we add only necessary edges during tree decomposition, obtained decompositions would not be a tree decomposition. On the other hand, as the contraction hierarchy algorithm adds less edges, it seems to be more scalable.

However, there is also difference of index data structures; while the contraction hierarchies are union of DAGs, tree decompositions can be processed like trees and thus query algorithms can be more efficient. While these methods are developed independently in different communities (the database community and experimental algorithmics community), and also targeted graphs are different, they turned out to be interestingly related.

Contraction hierarchies have also interesting relation to labeling-based methods, which will be introduced in the next section.

3.2.2 Labeling Methods

The recent emergence of practical labeling methods for distance queries on road networks is quite interesting and indeed a rare example of practical impact of theoretical research in this field. In the seminal theoretical work by Abraham et al. [AFGW10], they theoretically analyzed the indexing algorithms for distance queries on road networks under their new notion of *highway dimension*. Interestingly, their theoretical result suggested that there should be good labels on road networks, whereas previously little effort has been done for labeling-based methods. Then, in their follow-up empirical work, Abraham et al. demonstrated that indeed the labeling approach is practical on continental-scale road networks of the day [ADGW11].

The labeling framework (i.e., the index data structure and query algorithm) used for distance queries on road networks is exactly the 2-hop cover

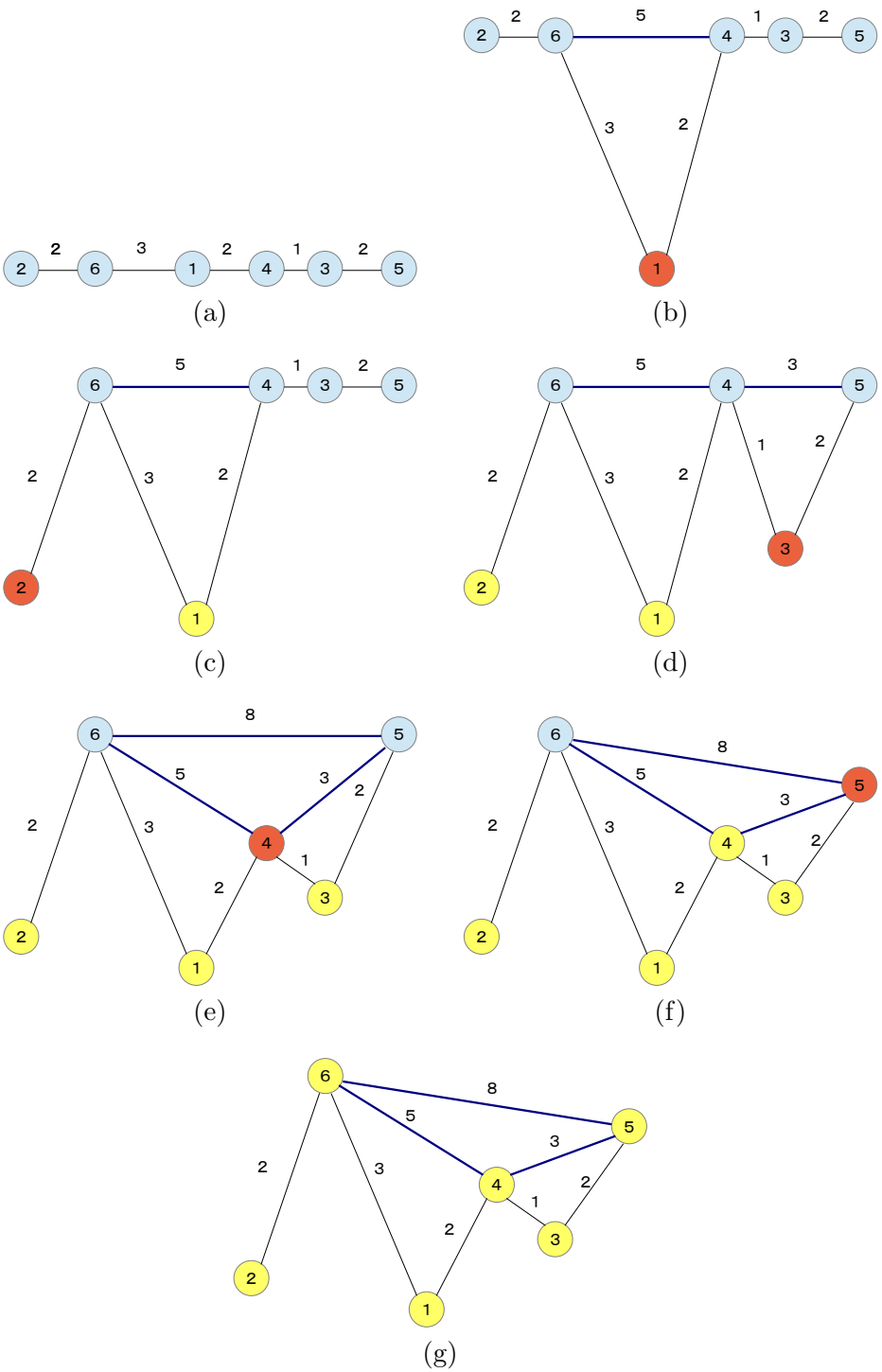


Figure 3.4: The index construction process of contraction hierarchies. Blue, yellow and red vertices denote those which are not yet contracted, already contracted and just being contracted, respectively.

framework introduced above. However, please note that it is called as *hub-based labeling* or *hub labeling* in the field of distance queries on road networks [ADGW11, ADGW12].

Hierarchical Labels through Contraction Hierarchies

The first practical indexing algorithm by Abraham et al. computes labels by using contraction hierarchies [ADGW11]. Their labels are hierarchical labels (see Definition 3.2), which respects the order of *importance* of the corresponding contraction hierarchies. The basic idea is to construct the forward label of v as the vertices that can be reached from v in the upward graph $G_{\uparrow}$, and to construct the backward label of v as the vertices that can reach v in the downward graph $G_{\downarrow}$. It is easy to see that the labels are correct because the query algorithm corresponds the bidirectional search on the contraction hierarchy graph. Surprisingly, their algorithm proved that road networks with tens of millions of vertices and edges had 2-hop covers with average label sizes of about one hundred. As a result, the labeling-based approach became the indexing method for distance queries on road networks with the fastest query time.

Canonical Labeling by Pruning

Then, in their next paper, they proposed the notion of canonical labeling (Definition 3.3). They also proposed a pruning algorithm to reduce the label sizes of hierarchical labels by converting them to canonical labels. For a road network of Western Europe, their new indexing algorithm using the technique above reduced the label size from 85 to 69.

Recall that canonical labeling is unique under the same vertex ordering. Therefore, both their algorithm and our algorithm compute the exactly same canonical labeling when they are given the same vertex ordering, and both are based on pruning. However, the algorithms are totally different and our algorithm has much better scalability. Their algorithm first computes hierarchical labeling and then obtains canonical labeling (the latter step is called pruning), while ours directly computes canonical labeling during pruned graph searches. Indeed, in their recent work [DGPW14], they also started using our labeling algorithm (see Chapter 11).

Label Compression

Because the largest disadvantage of labeling-based methods is space consumption, they proposed label compression schemes to reduce the space consumption [ADGW11, DGW13, DGPW14]. The basic idea of their compression scheme is to represent a label as a tree and reuse the common parts of different labels. Therefore, the total set of labels is represented as a DAG.

In particular, the underlying thought in [DGW13] is, again, interestingly connected with contraction hierarchies. Their finding is that, while they construct hierarchical 2-hop cover labels from contraction hierarchies, contraction hierarchies can also be easily obtained from any hierarchical 2-hop cover labels by considering *label trees*. This is quite interesting because the labeling-based approach is one of the *heaviest* indexing approaches, while the contraction hierarchy approach is one of the most *lightweight* indexing approaches, in the sense of the trade-off between index size and query time. Therefore, their compression scheme is designed to combine the contraction-hierarchy-like index structure into the labeling-based approach.

3.3 Reachability Queries

In this section, we review previous methods on reachability queries. They can be roughly classified into three approaches: transitive closure compression, online search, and labeling.

3.3.1 Transitive-closure-based Methods

The most classical approach is to compress *transitive closure*, i.e., the $|V| \times |V|$ table describing reachability for all pairs. In *Tree Cover* [ABJ89], transitive closure is represented as an interval label of a spanning tree, and the minimization of the label size is considered. The *interval list (IL)* method compresses transitive closure by using run-length-like compression [Nuu95]. *Path-Tree* [JXRW08] decomposes a graph into paths, and then creates an interval label on a graph of paths. In [vSdM11], transitive closure is compressed by a data structure called *PWAH* after sorting vertices by topological order. Generally, queries can be answered quickly in these methods, just by looking up the compressed transitive closure. However, the size of full transitive closure can be quadratic, which causes the scalability problem.

3.3.2 Online-search-based Methods

There are several online-search-based methods, which basically conduct a depth-first search (DFS) guided by indices for each query [CGK05, YCZ12, ZYQ$^+$12]. *GRIPP* [CGK05] exploits an index called an *interval label* on a spanning tree, and answers a query by conducting a DFS efficiently by reducing search space using an interval label. *GRAIL* [YCZ12] is one of state-of-the-art method in this approach. This method speeds up DFS using several randomly created interval labels. The advantages of these methods are short indexing time and memory-efficiency since an interval label is created by a single DFS. However, query time is slow compared with other methods since these methods traverse the input graph to answer each query. An on-disk version of GRAIL has also been proposed [ZYQ$^+$12].

3.3.3 Labeling-based Methods

Another popular approach is labeling-based methods. For reachability queries, 2-hop cover indices have also been often used [CHKZ03,STW04,CYL$^+$06], whereas there are a few extensions that are specialized for reachability queries.

2-Hop Cover Framework for Reachability Queries

Given a DAG $G = (V, E)$, we create two types of labels $L_{\mathrm{OUT}}(v), L_{\mathrm{IN}}(v) \subseteq V$ for each vertex v, defined below.

Definition 3.5 (2-Hop Cover Index for Reachability Queries [CHKZ03])**.** *Set of labels $\{L(v)\}_{v \in V}$ is a* (correct) 2-hop cover index for reachability queries *of graph $G = (V, E)$ if $L(v)$ is a set of vertices, and, for any pair of vertices $s, t \in V$, $L(s) \cap L(t) \neq \emptyset$ if and only if $d_G(s, t) \neq \infty$.*

As with 2-hop covers for distance queries, upon a query (s, t), we return $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}})$ defined as follows.

$$\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}}) = \begin{cases} \textbf{true} & \text{if } L_{\mathrm{OUT}}(s) \cap L_{\mathrm{IN}}(t) \neq \emptyset, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

We construct labels L_{OUT} and L_{IN} so that, for every $s, t \in V$, $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}})$ matches the reachability on G from s to t. Naive calculation of $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}})$ needs $O(l_s \cdot l_t)$ time for $l_s = |L_{\mathrm{OUT}}(s)|$ and $l_t = |L_{\mathrm{IN}}(t)|$. However, if $L_{\mathrm{OUT}}(s)$ and $L_{\mathrm{IN}}(t)$ are sorted, it can be done in $O(l_s + l_t)$ time by scanning both labels from their heads to tails simultaneously.

While the framework is highly common to methods for distance queries introduced above, labeling algorithms are again designed independently and almost independent [STW04,CYL+06]. For example, the approach proposed by Schenkel et al. is based on divide-and-conquer algorithm. The algorithm is tailored to graphs that are quite close to trees such as XMLs.

Extensions of 2-Hop Covers

3-hop cover [JXRF09] is a generalization of 2-hop cover, where labels are created to satisfy that a vertex can reach its descendants via two vertices by using chain decomposition. These hop cover methods usually achieve small query time and index size, but computing such good labels is prohibitively expensive on large-scale graphs.

3.3.4 General Improving Techniques

There are some general techniques to improve the performance of reachability indices. The idea of query preserving compression [FLWW12] is to make a compressed graph with which we can correctly answer reachability queries on the original graph. In [JRDX12], a reachability computation framework named SCARAB was proposed. In the framework, a smaller graph called a reachability backbone is constructed from an input graph and a reachability query on the original graph can be answered by issuing several reachability queries on the reachability backbone. These techniques can be combined with any methods, including ours, and make them more scalable.

3.4 Theoretical Results

As mentioned in Chapter 1, most of the results from the theoretical algorithm community are of independent interest from practical efficiency. By contrast, there are a few notable theoretical results that offer valuable insight also into empirical algorithmics on graph indexing. In this section, we introduce these results.

3.4.1 Highway Dimension

Real road networks are close to planar graphs, but they are not exactly planar. For example, there are grade separated crossings. Moreover, even if we assume road networks are planar, still, it is not sufficient to theoretically explain the efficiency of practical indexing methods for distance queries on road networks. Intuitively, these practical indexing models exploit more specific structures of road networks, e.g., existence of *popular* roads such as highways.

To theoretically discuss the efficiency of algorithms on such road networks, Abraham et al. proposed a new theoretical notion called *highway dimension* [AFGW10]. They modeled road networks as graphs with small highway dimension. Intuitively, a graph with small highway dimension is one that has *small* sets of vertices where any *long* shortest paths pass through at least one of

them. This assumption coincides our intuitive understanding of the structure of real road networks.

Under the assumption of small highway dimension, those practical indexing methods are theoretically proved to be efficient for the first time. Moreover, as we mentioned above, the results also motivated the use of the labeling approach for distance queries on road networks, which has been indeed demonstrated to be prominent.

3.4.2 Power-Law Random Graphs

As with the discussion above, to discuss distance querying on complex networks such as social and web graphs, there also should be assumptions for graphs. As we explained above, practical methods exploit the common structures of these networks, and thus analysis on general graphs are unfortunately not very interesting.

To address this issue, Chen et al. presented the first theoretically interesting results about distance queries on graphs with power-law degree distribution [CSTW12]. They assumed fixed degree random graph, and they conducted rigorous average case analysis on the theoretical indexing method by Thorup and Zwick [TZ05]. The results theoretically confirmed our intuition that highly central vertices (i.e., hubs) in these networks make shortest path computation easier.

Chapter 4

Basic Form of Pruned Landmark Labeling Algorithm

In this chapter, we present the basic form of the *pruned landmark labeling* algorithm. Here, we assume that the given graph $G = (V, E)$ is undirected and unweighted. We only consider basic distance queries, i.e., we construct an index to efficiently answer distances between arbitrary pairs of vertices. As the index data structure, we use the original 2-hop cover framework explained in Chapter 3.

We first explain the offline labeling algorithm and prove its correctness (Section 4.1). Then, we discuss *vertex ordering* for empirically exploiting the structures of real networks (Section 4.2). Next, we look into theoretical properties of our labeling algorithm to further see that, with proper vertex ordering strategies, it can exploit various structures of real networks (Section 4.3). In Section 4.4, common engineering techniques for efficient implementation are presented. Finally, we discuss online update algorithms of 2-hop indices for dynamic graph update based on pruned labeling (Section 4.5).

In this chapter, we do not present experimental results. Experimental evaluation will be presented in the following chapters, which deal with more concrete problem settings on real instances. This chapter presents the concept of pruned labeling that is common to the methods tailored to each problem setting.

4.1 Labeling Algorithm

We explain our labeling algorithm (i.e., an indexing algorithm for the 2-hop cover framework) in two steps. We start with the following naive method, and then we introduce pruning to the method.

4.1.1 Naive Landmark Labeling

Our naive landmark labeling algorithm conducts a BFS from each vertex and store distances between all pairs as an index. Though this method is too obvious and inefficient, for the exposition of the next method, we explain the details.

Let $V = \{v_1, v_2, \ldots, v_n\}$. We start with an empty index L_0, where $L_0(u) = \emptyset$ for every $u \in V$. Suppose we conduct BFSs from vertices in the order of $v_1, v_2, \ldots, v_n$. After the k-th BFS from a vertex v_k, we add distances from v_k to labels of reached vertices, that is, $L_k(u) = L_{k-1}(u) \cup \{(v_k, d_G(v_k, u))\}$ for each $u \in V$ with $d_G(v_k, u) \neq \infty$. We do not change labels for unreached vertices, that is, $L_k(u) = L_{k-1}(u)$ for every $u \in V$ with $d_G(v_k, u) = \infty$.

L_n is the final index. Obviously $\textsc{Query}(s, t, L_n) = d_G(s, t)$ for any pair of vertices s and t, and therefore, L_n is a correct 2-hop cover for exact distance

Algorithm 4.1 Pruned BFS from $v_k \in V$ to create index L'_k.

```
1: procedure PrunedBFS(G, v_k, L'_{k-1})
2:     Q ← a queue with only one element v_k.
3:     P[v_k] ← 0 and P[v] ← ∞ for all v ∈ V(G) \ {v_k}.
4:     L'_k[v] ← L'_{k-1}[v] for all v ∈ V(G).
5:     while Q is not empty do
6:         Dequeue u from Q.
7:         if Query(v_k, u, L'_{k-1}) ≤ P[u] then
8:             continue
9:         L'_k[u] ← L'_{k-1}[u] ∪ {(v_k, P[v_k])}
10:        for all w ∈ N_G(v) s.t. P[w] = ∞ do
11:            P[w] ← P[u] + 1.
12:            Enqueue w onto Q.
13:    return L'_k
```

queries. This is because, if s and t are reachable, then $(s, 0) \in L_n(s)$ and $(s, d_G(s,t)) \in L_n(t)$ for example.

This method can be considered as a variant of landmark-based approximate methods, which we mentioned in Section 3. The standard landmark-based method can be regarded as a method that precomputes L_l instead of L_n and estimates distance between s and t by $\textsc{Query}(s, t, L_l)$, where $l \ll n$ is a parameter expressing the number of landmarks.

4.1.2 Pruned Landmark Labeling

Then, we introduce *pruning* to the naive method. Similarly to the method above, we conduct *pruned* BFSs from vertices in the order of $v_1, v_2, \ldots, v_n$. We start with an empty index L'_0 and create an index L'_k from L'_{k-1} using the information obtained by the k-th pruned BFS from vertex v_k.

We prune BFSs as follows. Suppose that we have an index L'_{k-1} and we are conducting a BFS from v_k to create a new index L'_k. Suppose that we are visiting a vertex u with distance δ. If $\textsc{Query}(v_k, u, L'_{k-1}) \leq \delta$, then we prune u, that is, we do not add (v_k, δ) to $L'_k(u)$ (i.e. $L'_k(u) = L'_{k-1}(u)$) and we do not traverse any edge from vertex u. Otherwise, we set $L'_k(u) = L'_{k-1}(u) \cup \{(v_k, \delta)\}$ and traverse all the edges from the vertex u as usual. As with the previous method, we also set $L'_k(u) = L'_{k-1}(u)$ for all vertices $u \in V$ that were not visited in the k-th pruned BFS. This algorithm, performing pruned BFSs, is described as Algorithm 4.1, and the whole preprocessing algorithm is described as Algorithm 4.2.

Figure 4.1 shows examples of pruned BFSs. The first pruned BFS from vertex 1 visits all the vertices (Figure 4.1a). During the next pruned BFS from vertex 2 (Figure 4.1b), when we visit vertex 6, since $\textsc{Query}(2, 6, L'_1) = d_G(2,1) + d_G(1,6) = 3 = d_G(2,6)$, we prune vertex 6 and we do not traverse edges from it. We also prune vertices 1 and 12. As the number of performed BFSs increases, we can confirm that the search space gets smaller and smaller.

4.1.3 Proof of Correctness

In the following, we prove that this method computes a correct 2-hop cover index, that is, $\textsc{Query}(s, t, L'_n) = d_G(s, t)$ for any pair of vertices s and t.

Theorem 4.1. *For any $0 \leq k \leq n$ and for any pair of vertices s and t,* $\textsc{Query}(s, t, L'_k) = \textsc{Query}(s, t, L_k)$.

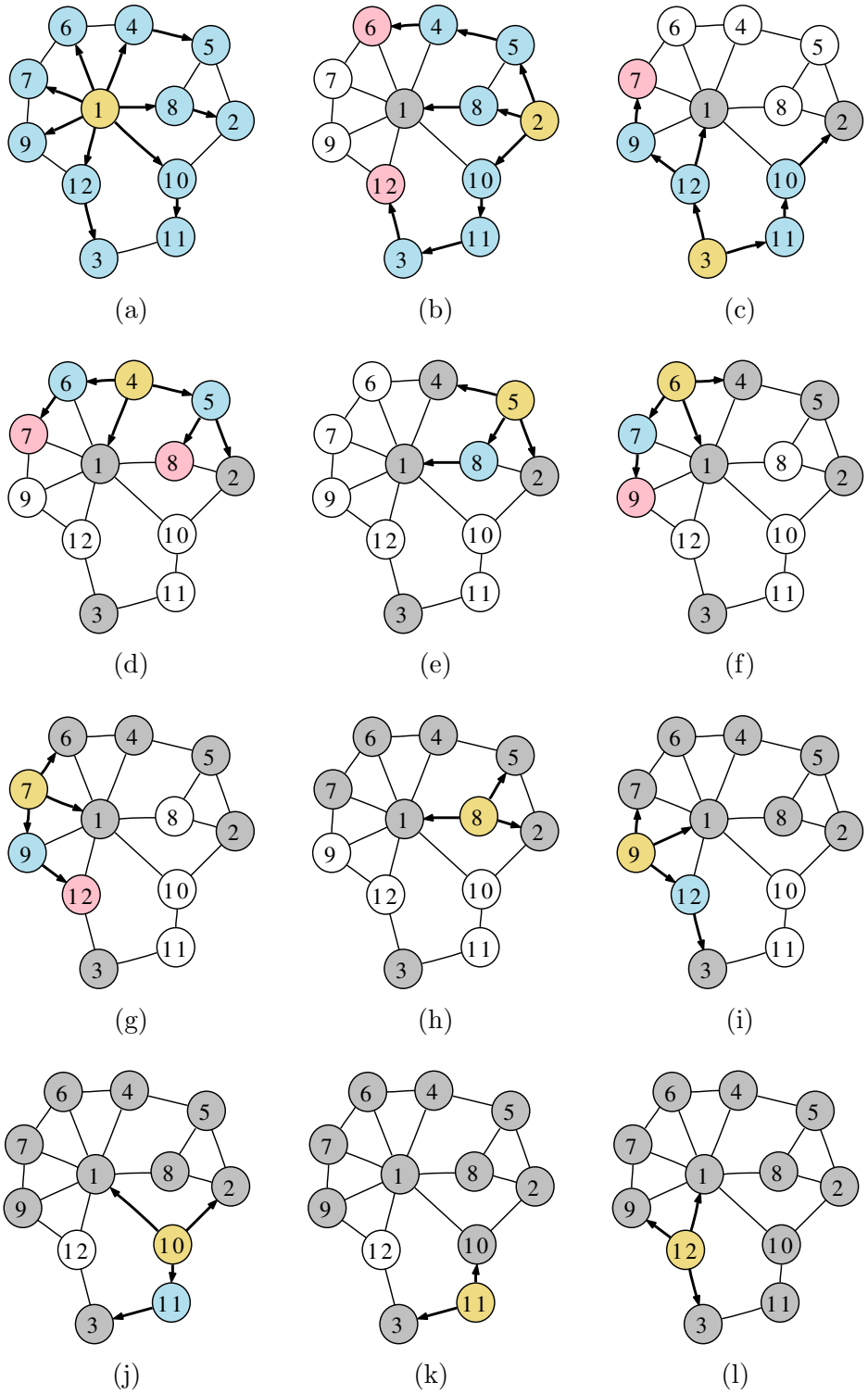


Figure 4.1: Examples of pruned BFSs. Yellow vertices denote the roots, blue vertices denote those which we visited and labeled, red vertices denote those which we visited but pruned, and gray vertices denote those which are already used as roots.

Algorithm 4.2 Compute a 2-hop cover index by pruned BFS.

```
1: procedure PREPROCESS(G)
2:     L'_0[v] ← ∅ for all v ∈ V(G).
3:     for k = 1, 2, ..., n do
4:         L'_k ← PRUNEDBFS(G, v_k, L'_{k-1})
5:     return L'_n
```

Proof. We prove the theorem by mathematical induction on k. Since $L'_0 = L_0$, it is true for $k = 0$. Now we assume it holds for $0, 1, \ldots, k-1$ and prove it also holds for k.

Let s, t be a pair of vertices. We assume these vertices are reachable in G, since otherwise the answer ∞ can be obviously obtained. Let j be the smallest number such that $(v_j, \delta_{v_j s}) \in L_k(s), (v_j, \delta_{v_j t}) \in L_k(t)$ and $\delta_{v_j s} + \delta_{v_j t} = \textsc{Query}(s, t, L_k)$. We prove that $(v_j, \delta_{v_j s})$ and $(v_j, \delta_{v_j t})$ are also included in $L'_k(s)$ and $L'_k(t)$. This immediately leads to $\textsc{Query}(s, t, L'_k) = \textsc{Query}(s, t, L_k)$. Due to the symmetry between s and t, we prove $(v_j, \delta_{v_j s}) \in L'_k(s)$.

First, for any $i < j$, we prove by contradiction that $v_i \notin P_G(v_j, s)$. If we assume $v_i \in P_G(v_j, s)$, from Inequality 2.1

$$\begin{aligned}\textsc{Query}(s, t, L_k) &= d_G(s, v_j) + d_G(v_j, t) \\ &= d_G(s, v_i) + d_G(v_i, v_j) + d_G(v_j, t) \\ &\geq d_G(s, v_i) + d_G(v_i, t).\end{aligned}$$

Since $(v_i, d_G(s, v_i)) \in L_k(s)$ and $(v_i, d_G(t, v_i)) \in L_k(t)$, this contradicts to the assumption of the minimality of j. Therefore, $v_i \notin P_G(v_j, s)$ holds for any $i < j$.

Now we prove that $(v_j, d_G(v_j, s)) \in L'_k(s)$. Actually, we prove a more general fact: $(v_j, d_G(v_j, u)) \in L'_k(u)$ for all $u \in P_G(v_j, s)$. Note that $s \in P_G(v_j, s)$. Suppose that we are conducting the j-th pruned BFS from v_j to create L_j. Let $u \in P_G(v_j, s)$. Since $P_G(v_j, u) \subseteq P_G(v_j, s)$ and $v_i \notin P_G(v_j, s)$ for any $i < j$, we have $v_i \notin P_G(v_j, u)$ for any $i < j$. Therefore, $\textsc{Query}(v_j, u, L'_{j-1}) > d_G(v_j, u)$ holds. Thus, we visit all vertices $u \in P_G(v_j, s)$ without pruning, and it follows that $(v_j, d_G(v_j, u)) \in L'_j(u) \subseteq L'_k(u)$. □

As a corollary, our method is proved to be an exact distance querying method by instantiating the theorem with $k = n$.

Corollary 4.1. *For any pair of vertices s and t,*

$$\textsc{Query}(s, t, L'_n) = d_G(s, t).$$

4.2 Vertex Ordering Strategies

In the algorithm description above, we conducted pruned BFSs from vertices in the order of $v_1, v_2, \ldots, v_n$. We can freely choose the order, and moreover it turns out that the order is crucial for the performance of this method as we will see in the experimental results presented in Section 5.2.3.

To decide the order of vertices, we should select *central* vertices first in the sense that many shortest paths pass through these vertices. Since we would like to prune later BFSs as much as possible, we want to *cover* larger part of pairs of vertices by earlier BFSs. That is, the earlier labels should offer correct distances for as many pairs of vertices as possible, and therefore the earlier vertices should be those who many shortest paths passes through.

This problem is quite similar to the problem of selecting good landmarks for landmark-based approximate methods, which is discussed well in [PBCG09]. In that problem, we also want to select good landmarks so that many shortest path passes through these vertices or nearby vertices.

This part of our algorithm highly depends on the structures of graphs. Therefore, we leave this part until we consider concrete problem settings. In fact, on networks with skewed degree distribution (e.g., complex networks), simple strategies work surprisingly well, as we will see in Section 5.2.3. On the other hand, for road networks, we need different strategies, as the degree gives little information about each vertex on these networks.

4.3 Theoretical Properties

4.3.1 Minimality

Theorem 4.2. *Let L'_n be the index defined in Section 4.1.2. L'_n is minimal in the sense that, for any vertex v and for any pair $(u, \delta_{uv}) \in L'_n(v)$, there is a pair of vertices (s, t) such that, if we remove (u, δ_{uv}) from $L'_n(v)$, we cannot answer the correct distance between s and t.*

Proof. Let $v_i \in V$ and $(v_j, \delta_{v_j v_i}) \in L'_n(v_i)$. This implies $j < i$. We show that if we remove $(v_j, \delta_{v_j v_i})$ from $L'_n(v_i)$ then we cannot answer the correct distance between v_i and v_j. We claim that, for any $k \neq j$, either (i) $(v_k, \delta_{v_k v_i}) \notin L'_n(v_i)$ or $(v_k, \delta_{v_k v_j}) \notin L'_n(v_j)$ holds, or (ii) $d_G(v_i, v_k) + d_G(v_k, v_j) > d_G(v_i, v_j)$ holds. Suppose $k < j$ and assume that (ii) does not hold. Then, (i) must hold since otherwise the j-th BFS should have pruned vertex v_i and $(v_j, \delta_{v_j v_i}) \notin L'_n(v_i)$. Suppose $k > j$ and assume that (ii) does not hold. Then, $v_k \in P_G(v_i, v_j)$ and therefore $(v_j, \delta_{v_j v_k}) \in L'_j(v_k)$, thus the k-th BFS prunes vertex v_j, leading to $(v_k, \delta_{v_k v_j}) \notin L'_n(v_j)$. □

4.3.2 Canonicality of Labels

As we mentioned in Chapter 3, our labels are hierarchical (Definition 3.2).

Lemma 4.1 ([DGPW14])**.** *The labels computed by the pruned landmark labeling algorithm are canonical.*

Among hierarchical 2-hop indices that respect a vertex ordering, the canonical labeling with respect to the vertex ordering is the only one minimal index (and thus the index is also minimum) [ADGW12]. Therefore, together with the discussion on minimality, our labels are also canonical with respect to the vertex ordering.

4.3.3 Exploiting Existence of Highly Central Vertices

Then, we compare our method with landmark-based methods to show that our method also can exploit the existence of highly central vertices. We consider the standard landmark-based method [PBCG09,VFD$^+$07], which do not use any path heuristics. As we stated in Chapter 3, by selecting central vertices as landmarks, it attains remarkable average precision for real-world networks. From the following theorem, we can observe that our method is efficient for networks whose distance can be answered by landmark-based methods with such high precision, and our method also can exploit the existence of these central vertices.

Theorem 4.3. *If we assume that the standard landmark-based approximate method can answer correct distances to $(1-\epsilon)n^2$ pairs (out of n^2 pairs) using k landmarks, then the pruned landmark labeling method gives an index with average label size $O(k+\epsilon n)$.*

Proof. After conducting pruned BFSs from the k landmark vertices first, at most ϵn^2 pairs are added to the index in total, since we never add pairs whose distance can be answered from current labels. □

4.3.4 Exploiting Small Treewidth

Finally, we show a theoretical evidence that our method can also exploit tree-like fringes efficiently. As we explained in Chapter 3, methods based on tree decompositions were proposed for distance queries [Wei10, ASK12]. Both of them extend methods which work efficiently for graphs of small treewidth, and they exploit low treewidth of fringes in real-world networks by tree decompositions. Interestingly, though we do not use tree decompositions explicitly, we can prove that our method can efficiently process graphs of small treewidth. Thus, our method implicitly exploits this property of real-world networks too. For definitions of treewidth and tree decompositions, see Chapter 2.

Theorem 4.4. *Let w be the treewidth of G. There is an order of vertices with which the pruned landmark labeling method takes $O(wm \log n + w^2 n \log^2 n)$ time for preprocessing, stores an index with $O(wn \log n)$ space, and answers each query in $O(w \log n)$ time.*

Proof. We consider the centroid decomposition [Jor69] of the tree decomposition. First we conduct pruned BFSs from all the vertices in a centroid bag. Then, later pruned BFSs never go beyond the bag. Therefore, we can consider as we divided the tree decomposition into disjoint components, each having at most half of the bags. We recursively repeat this procedure. The number of recurrences is at most $O(\log n)$. Since we add at most w pairs to each vertex at each depth of recursion, the number of pairs in each label is $O(w \log n)$. At each depth of recursion, the total time consumed by pruned BFSs from the current components is $O(wm + w^2 n \log n)$, where $O(wm)$ is the time for traversing edges and $O(w^2 n \log n)$ is the time for pruning tests. □

4.4 Common Techniques for Efficient Implementation

4.4.1 Preprocessing

Index: First, in the description above, we treated L'_{k-1} and L'_k separately and explained as if we copy L'_{k-1} to L'_k for simplicity of explanation. However, this copy can be easily avoided by keeping only one index and adding labels to it after each pruned BFS.

Initialization: Another important note is to avoid $O(n)$ time initialization for each pruned BFS. The reason why this method is efficient is that the search space of pruned BFSs gets much more smaller than the whole graph. However if we spend $O(n)$ time for initialization, it would be the bottleneck. What we want to do in the initialization is to set all values in the array storing tentative distances as ∞ (Line 3). We can avoid $O(n)$ time initialization as follows. Before we conduct the first pruned BFS, we set all values in the array P as ∞. (This takes $O(n)$ time but we do this only once.) Then, during each pruned BFS, we

store all vertices we visited, and after each pruned BFS, we set $P[v]$ as ∞ for all each vertex v we have visited.

Arrays: For the array storing tentative distances, it is better to use 8-bit integers. Since networks of our interest are small-world networks, 8-bit integers are enough to represent distances. Using 8-bit integers, the array fits into low-level cache memories of recent computers, resulting in the speed up by reducing cache misses.

Querying: To evaluate queries for pruning (Line 7), it is faster to use an algorithm different from the normal one since we can exploit the fact here that we issue many queries whose one endpoint is always v_k. Before starting the k-th pruned BFS from v_k, we prepare an array T of length n initialized with ∞ and set $T[w] = \delta_{wv_k}$ for all $(w, \delta_{wv_k}) \in L'_{k-1}(v_k)$. To evaluate $\textsc{Query}(v_k, u, L'_{k-1})$, for all $(w, \delta_{wu}) \in L'_{k-1}(u)$, we compute $\delta_{wu} + T[w]$ and return the minimum. Though normal querying algorithm takes $O(\left|L'_{k-1}(v_k)\right| + \left|L'_{k-1}(u)\right|)$ time, this algorithm runs in $O(\left|L'_{k-1}(u)\right|)$ time. As Line 7 is the bottleneck of the algorithm, this technique speeds up preprocessing by about twice. Note that T should be represented by 8-bit integers as the same reason described above, and $O(n)$ time initialization for array T should be avoided in the same way for array P.

Prefetching: Unfortunately, we cannot fit the index and the adjacency lists into the cache memory for large-scale networks. However, we can manually prefetch them to reduce the cache misses, since vertices which we will access soon are in the queue. Manual prefetching speeds up preprocessing by about 20%.

Thread-Level Parallelism: As with parallel BFS algorithms [APPB10], the pruned BFS algorithm can be also parallelized. However, for simple experimental analysis and fair comparison to previous methods, we did not parallelize our implementation in the experiments.

Sorting Labels: When applying merge-join-like algorithms to answer queries, pairs in labels need to be sorted by vertices. However, actually we do not need to sort explicitly by storing ranks of vertices instead of vertices. That is, when adding a pair (u, δ) in the i-th pruned BFS from vertex u, we add a pair (i, δ) instead. Then, since pairs are added from vertices with lower rank to those with higher rank, all the labels are automatically sorted.

4.4.2 Querying

Sentinel: We add a dummy entry, (n, ∞), to the label $L(v)$ for each $v \in V$. This dummy entry ensures that we find the same vertices, n, in the end when scanning two labels. Thus we can avoid to separately test whether we have scanned till the end.

Arrays: For each label $L(v)$, it is faster to store the array for vertices and the array for distances separately since distances are only used when vertices match [ADGW11]. We also align arrays to cache lines.

4.5 Incremental Update Algorithm

Finally, we present our algorithm that incrementally updates the current index to reflect graph changes. The idea behind our efficient update algorithm is to carefully *resume* and *stop* pruned BFSs.

4.5.1 Supported Updates

As we mentioned in Chapter 1, we focus on two kinds of graph updates: vertex additions and edge additions. We do not consider removal. This is due to the following reasons.

1. As we can see that no previous methods support any incremental updates, supporting only additions is already quite technically challenging.
2. Supporting removals is even much harder and it seems impossible to efficiently support removals without making big compromise on performance such as index size and query time.
3. Removals never happen in certain kinds of real-world dynamic networks such as interaction networks in social media and instant messaging services, co-author networks, co-starring networks, e-mail networks, telephone networks, and so on.
4. In other kinds of real dynamic networks, still, additions are much more frequent than removals [Mis09].
5. For these reasons, it is quite common to ignore removals when analyzing and modeling dynamic networks [BA99, LKF07, Mis09, VMCG09]. As an evidence, widely used public datasets of time-evolving graphs do not contain any removal[1][2].

Please note that we can assume the newly inserted vertex is isolated since otherwise we can process it by first inserting an isolated vertex and then inserting edges incident to it. Similarly, if multiple edges are inserted simultaneously, we process them one by one. Under the 2-hop framework, inserting a new isolated vertex v can be easily done by setting a new empty label $L(v) = \emptyset$. Thus, in what follows, we focus on edge additions.

Since we only consider vertex and edge additions, for any $\tau > 0$, $V(G_{\tau-1}) \subseteq V(G_\tau)$ and $E(G_{\tau-1}) \subseteq E(G_\tau)$. As we only consider vertex and edge additions, the following lemma is a key for designing algorithms.

Lemma 4.2. *Let $G_\tau, G_{\tau'}$ be graphs where $E(G_\tau) \subseteq E(G_{\tau'})$. For any pairs of vertices s and t, $d_\tau(s,t) \geq d_{\tau'}(s,t)$.*

That is, for any pairs of vertices, distance between them never increases by adding vertices or edges.

Proof. The path that was the shortest path in G_τ is also present in $G_{\tau'}$. □

Please note that this lemma does not tell that distance on a dynamic graph without removals is uninteresting. For example, diameter or average distance do not necessarily always decrease because of vertex additions.

Suppose that we are maintaining index M for dynamic graph G and we want to update M to reflect a newly inserted edge (a, b) that was previously absent. We refer to the old graph without the new edge as $G_{\tau-1}$ and the new graph with the new edge as G_τ. Similarly, we refer to the old and new index as $M_{\tau-1}$ and M_τ, respectively.

From Lemma 4.2, it is sound to keep outdated distances in the index since we never underestimate distances because of them. Therefore in our method, we

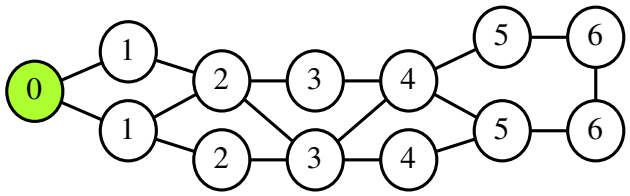
[1] `http://socialnetworks.mpi-sws.org/`
[2] `http://konect.uni-koblenz.de/`

do not remove outdated label entries since detecting them is too costly. We only add new label entries or rewrite distances of existing label entries. Under this strategy, the minimality of the index M_τ as a whole is broken after updates, but we will later see that the set of newly added label entries is minimal to answer correct distances in G_τ. Moreover, we will see that the increase of label sizes is satisfyingly small in practice.

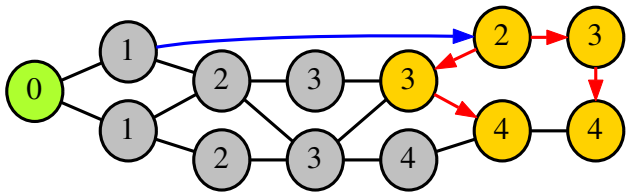
4.5.2 Update Algorithm for Naive Labeling

We first describe an update algorithm for the naive landmark labeling method. The obvious way is to conduct full BFSs from every vertex again and update labels, which is not different from the indexing algorithm at all. To update labels more efficiently, let us reduce the search space of each BFS. The two key insights here are the following. First, if the distance from vertex v_k to vertex u has changed, then all the new shortest paths between them pass through the new edge (a, b). Second, suppose the shortest path P between v_k and $u \neq a, b$ has changed. Then the distance between v_k and w has changed, where w is the penultimate vertex in P.

We can assume $d_{\tau-1}(v_k, a) \leq d_{\tau-1}(v_k, b)$ without loss of generality. Based on these facts, for every v_k, it suffices to *resume* the BFS from b originally rooted at v_k and *stop* at unchanged vertices. That is, instead of inserting $(v_k, 0)$ to the initial queue, we insert $(b, d_{\tau-1}(v_k, a)+1)$ to the initial queue, which corresponds to the position after passing through the new edge (a, b), and we do not traverse edges from vertex u if $\delta \geq d_{\tau-1}(v_k, u)$, where δ is the tentative distance for u drawn from the queue. Figure 4.2 illustrates an example.



(a) A graph



(b) The graph after inserting a new edge (drawn in blue)

Figure 4.2: A running example for the update algorithm. The green vertex is the root, and the distance to the root is written in each vertex.

4.5.3 Update Algorithm for Pruned Labeling

Now we explain our update algorithm for the pruned landmark labeling. We introduce pruning to the previous update algorithm. Let s, t be vertices and k

Algorithm 4.3 Update index M for newly added edge (a, b)

```
1: procedure InsertEdge(G, a, b, M)
2:     for all v_k ∈ M(a) ∪ M(b) from lower k do
3:         ResumePBFS(G, v_k, b, d(v_k, a) + 1, M) if v_k ∈ M[a]
4:         ResumePBFS(G, v_k, a, d(v_k, b) + 1, M) if v_k ∈ M[b]
5: procedure ResumePBFS(G, v_k, u, δ_ru, M)
6:     Q ← a queue with only one element (u, δ_ru).
7:     while Q is not empty do
8:         Dequeue (v, δ) from Q.
9:         if PrefixalQuery(v_k, v, M, k) ≤ δ then
10:            continue
11:        M[v] ← M[v] ∪ {(v_k, δ)}
12:        for all w ∈ N(v) do
13:            Enqueue (w, δ + 1) onto Q.
```

be an integer. We define a new function PrefixalQuery as follows,

$$\begin{aligned}&\textsc{PrefixalQuery}(s, t, M, k) = \\ &\quad \min\{\delta_s + \delta_t \mid (v_i, \delta_s) \in M(s), (v_i, \delta_t) \in M(t), i \le k\}.\end{aligned}$$

That is, $\textsc{PrefixalQuery}(s, t, M, k)$ is the answer to the query between vertices s and t computed from the index M only using distances to vertices whose IDs are at most k. We define $\textsc{PrefixalQuery}(s, t, M, k) = \infty$ if $M(s)$ and $M(t)$ do not share any vertex whose ID is at most k. Using this function, suppose we are conducting a resumed BFS originally rooted at v_k and visiting vertex u with distance δ, we prune u if $\textsc{PrefixalQuery}(v_k, u, M, k) \le \delta$.

However, one crucial question is left: for which roots do we need to resume BFSs? The obvious solution is resuming BFSs no matter what their roots are as with the previous algorithm, but it is too inefficient since it takes at least $\Omega(|V|)$ time. Interestingly, the answer is exactly what we have in $M(a)$ and $M(b)$. That is, it suffices to conduct resumed BFSs originally rooted at v_k if $v_k \in M(a) \cup M(b)$. This is because, if $v_k \notin M(a) \cup M(b)$, both a and b are pruned or unreached during previous (resumed) BFSs rooted at v_k, and since the shortest path between v_k and them has not changed from the last snapshot, the situation does not change at all. The total algorithm is described as Algorithm 4.3.

Time complexity To roughly estimate the time complexity, we assume $|L(v)| = O(l)$ and the number of vertices visited during each resumed BFS is $O(s)$, where l and s are some integers. Then, since we conduct resumed BFSs $O(l)$ times and each pruning test takes $O(l)$ time, in total, each update can be done in $O(l^2 s)$ time. In our experiments, l was around hundreds and s was around tens on average.

4.5.4 Proof of Correctness

We prove the correctness of the proposed incremental update algorithm. For vertices s, t and $0 \le k \le n$, we define the *restricted distance* between s and t with respect to k as $d'(s, t, k) = \min_{i \le k}\{d(s, v_i) + d(v_i, t)\}$. We define $d'(s, t, k) = \infty$ if $k = 0$ or v_i is unreachable from s or t for all $i \le k$. As with d_τ, we denote the restricted distance at time τ by $d'_\tau(s, t, k)$. The following notion of correctness is important.

Definition 4.1 (Prefixal Correctness)**.** *Let M be a 2-hop cover index for graph G. Index M is* prefixally correct *if* $\textsc{PrefixalQuery}(s, t, M, k) = d'(s, t, k)$ *for any $s, t \in V$ and $0 \leq k \leq n$.*

Note that prefixal correctness is stronger than normal correctness. Due to Theorem 4.1, the initial index constructed by the *full* pruned landmark labeling algorithm satisfies prefixal correctness. In what follows, we prove that prefixal correctness of an index is maintained by the incremental update algorithm.

In what follows, let $M_{\tau-1}$ be a prefixally correct index for graph $G_{\tau-1}$, let G_τ be the graph created by inserting edge $(a, b) \notin E(G_{\tau-1})$ to $G_{\tau-1}$, and M_τ be the index updated by the Algorithm 4.3 from $M_{\tau-1}$ for the edge addition.

Lemma 4.3. *For any pair of vertices s and t and $0 < k \leq n$, we have* $\textsc{PrefixalQuery}(s, t, M_\tau, k-1) = d'_\tau(s, t, k-1)$*. For any vertex u and $0 < k \leq n$, if $d_\tau(v_k, u) < d_{\tau-1}(v_k, u)$ and $d_\tau(v_k, u) < d'_\tau(v_k, u, k-1)$, then we have $(v_k, d_\tau(v_k, u)) \in M_\tau(u)$.*

Proof. Since $d_\tau(v_k, u) < d_{\tau-1}(v_k, u)$, all the shortest paths from v_k to u in the new snapshot G_τ pass through the new edge (a, b). We assume $d_\tau(v_k, a) < d_\tau(v_k, b)$ without loss of generality, and suppose that one of the shortest paths is of the form $(v_k, \ldots, a, b = w_0, w_1, w_2, \ldots, u = w_l)$. We can observe that, not only u but also for any w_i, $d_\tau(v_k, w_i) < d_{\tau-1}(v_k, w_i)$ holds. Moreover, since $d_\tau(v_k, u) < d'_\tau(v_k, u, k-1)$, none of the shortest paths between v_k and u in G_τ goes through the vertex v_j for any $j < k$. This also holds for any w_i, that is, none of the shortest paths between v_k and w_i in G_τ passes through v_j for any $j < k$, and thus $d_\tau(v_k, w_i) < d'_\tau(v_k, w_i, k-1)$.

During the resumed BFS originally rooted at v_k, we have $\textsc{PrefixalQuery}(v_k, w_i, M_\tau, k) \geq \min\{d'_\tau(v_k, w_i, k-1), d_{\tau-1}(v_k, w_i)\}$ for any w_i. Therefore, $\textsc{PrefixalQuery}(v_k, w_i, M_\tau, k) \geq d_\tau(v_k, w_i)$, and w_i is not pruned. Thus, the BFS reaches vertex u with the correct distance $d_\tau(v_k, u)$, and pair $(v_k, d_\tau(v_k, u))$ is newly added to the label $M_\tau(u)$. □

Theorem 4.5. *M_τ is a prefixally correct index for G_τ.*

Proof. We prove the prefixal correctness of M_τ by mathematical induction on k. For $k = 0$, it is true as, for any pair of vertices s and t with $s \neq t$, $\textsc{PrefixalQuery}(s, t, M_\tau, 0) = d'(s, t, 0) = \infty$. Now we assume $k > 0$ and $\textsc{PrefixalQuery}(s, t, M_\tau, k-1) = d'(s, t, k-1)$ for any pairs of vertices s and t, and prove $\textsc{PrefixalQuery}(s, t, M_\tau, k) = d'(s, t, k)$ for any pairs of vertices s, t.

Let $\delta' = d'_\tau(s, t, k)$. If $\delta' = d'_\tau(s, t, k-1)$, then we have nothing to show due to the assumption of the mathematical induction. Otherwise, since $\delta' < d'_\tau(s, t, k-1)$, $\delta' = d_\tau(s, v_k) + d_\tau(v_k, t)$. Therefore, it suffices to show that $(v_k, d_\tau(v_k, s)) \in M_\tau(s)$ and $(v_k, d_\tau(v_k, t)) \in M_\tau(t)$. Due to the symmetry between s and t, we only show $(v_k, d_\tau(v_k, s)) \in M_\tau(s)$.

First, we consider the case $d_\tau(v_k, s) = d_{\tau-1}(v_k, s)$. From $d'_\tau(s, t, k) = d_\tau(s, v_k) + d_\tau(v_k, t) < d'_\tau(s, t, k-1)$, there is no vertex v_i with $i \leq k-1$ on any shortest path between s and v_k in G_τ. This is the case in $G_{\tau-1}$ since $d_\tau(v_k, s) = d_{\tau-1}(v_k, s)$. It follows that $d'_{\tau-1}(v_k, s, k-1) > d'_{\tau-1}(v_k, s, k)$. Thus if $(v_k, d_\tau(v_k, s)) \notin M_{\tau-1}(s)$, then $\textsc{PrefixalQuery}(v_k, s, M_{\tau-1}, k) = d'_{\tau-1}(v_k, s, k-1) > d'_{\tau-1}(v_k, s, k)$, which contradicts to the prefixal correctness of $M_{\tau-1}$. Therefore, $(v_k, d_\tau(v_k, s)) \in M_{\tau-1}(s) \subseteq M_\tau(s)$ holds.

Otherwise, we can assume $d_\tau(v_k, s) < d_{\tau-1}(v_k, s)$. As $\delta' < d'_\tau(s, t, k-1)$, none of the shortest paths from v_k to s goes through the vertex v_i for any $i < k$, and thus $d_\tau(v_k, s) < d'_\tau(v_k, s, k-1)$. From Lemma 4.3, $(v_k, d_\tau(v_k, s)) \in M_\tau(s)$. □

Corollary 4.2. *Let M_1 be the index constructed by the offline indexing algorithm for graph G_1. For $2 \leq i \leq \tau$, let M_i be the index updated by Algorithm 4.3 from M_{i-1} for the edge addition to make G_i from G_{i-1}. Then, the index M_τ is a correct 2-hop index for G_τ.*

Moreover, the sufficient condition for a pair to be added shown in Lemma 4.3 is actually also a necessary condition. Therefore, the minimality of newly added pairs is derived.

Theorem 4.6. *The label entries added by the update algorithm are minimal. That is, for any vertex u and any pair in $M_\tau(u) \setminus M_{\tau-1}(u)$, if we remove the pair from $M_\tau(u)$, then M_τ becomes an incorrect 2-hop index for G_τ.*

4.5.5 Efficient Implementation

Unnecessary BFSs: When entries (r, δ_{ra}) and (r, δ_{rb}) are contained in both labels $M(a)$ and $M(b)$, we actually do not need to resume the BFS twice from a and b. Instead, we compare the distance in these labels. We can assume $\delta_{ra} \leq \delta_{rb}$ without loss of generality. If $\delta_{rb} \leq \delta_{ra} + 1$, then the distance $d(r, b)$ has not changed, and thus nothing happened. Therefore, we need to do nothing. Otherwise, we resume the BFS only from b with distance $\delta_{ra} + 1$.

Rewriting Labels: Suppose we are conducting a BFS originally rooted at r and visiting vertex v with distance δ. If $(r, \delta_{rv}) \in M(v)$ where $\delta_{rv} > \delta$, instead of adding a new label entry (r, δ), we rewrite the distance in the pair above as δ.

Visited Bitset: To avoid evaluating the same prefixal query on a pruned vertex more than once, we use a bitset to manage visited vertices.

Chapter 5

Bit-parallel Labeling for Unweighted Complex Networks

As we explained in Chapter 1, distance queries on complex networks are used in important applications such as network-aware searching and network analysis. In this chapter, we focus on distance queries on unweighted complex networks such as social networks and web graphs. Specifically, we propose the *bit-parallel labeling* technique tailored for these networks, and present the experimental results on unweighted complex networks.

5.1 Bit-parallel Labeling Technique

To further speed up both preprocessing and querying, we propose an optimizing method which exploits bit-level parallelism. *Bit-parallel* methods (also known as *broadword* algorithms) are those that perform different calculations on different bits in the same word to exploit the fact that computers can perform bitwise operations on a word at once. The word length is commonly 32 or 64 in computers of the day.

In the following, we denote the number of bits in a computer word as b and assume bitwise operations on bit vectors of length b can be done in $O(1)$ time. We propose an algorithm to conduct BFSs and compute labels from $b+1$ roots simultaneously in $O(m)$ time. Moreover, we also propose a method to answer distance queries for any pair of vertices via one of these $b+1$ vertices in $O(1)$ time.

5.1.1 Bit-parallel Labels

To describe the preprocessing algorithm and the querying algorithm, we first define what we store in the index. As we explain in the next subsection, we conduct bit-parallel BFSs from a vertex r and a subset of its neighbors $S_r \subseteq N_G(r)$ with size at most b. We define

$$S_r^i(v) = \{u \in S_r \mid d_G(u,v) - d_G(r,v) = i\}.$$

The key insight of this bit-parallel labeling scheme is as follows (Figure 5.1). Since vertices in S_r are neighbors of r, for any vertex $u \in S_r$ and any vertex $v \in V$, $|d_G(u,v) - d_G(r,v)| \leq 1$. Therefore, for each $v \in V$, S_r can be partitioned into $S_r^{-1}(v), S_r^0(v)$, and $S_r^{+1}(v)$. That is, $S_r^{-1}(v) \cup S_r^0(v) \cup S_r^{+1}(v) = S_r$.

We compute *bit-parallel labels* and store them in the index. For each vertex $v \in V$, we precompute a bit-parallel label denoted as $L_{\mathrm{BP}}(v)$. $L_{\mathrm{BP}}(v)$ is a set of quadruples $(u, \delta_{uv}, S_u^{-1}(v), S_u^0(v))$, where $u \in V$ is a vertex, $\delta_{uv} = d_G(u,v)$ and $S_u^i(v) \subseteq S_u$ is defined above. We store $S_u^{-1}(v)$ and $S_u^0(v)$ by bit vectors of b bits.

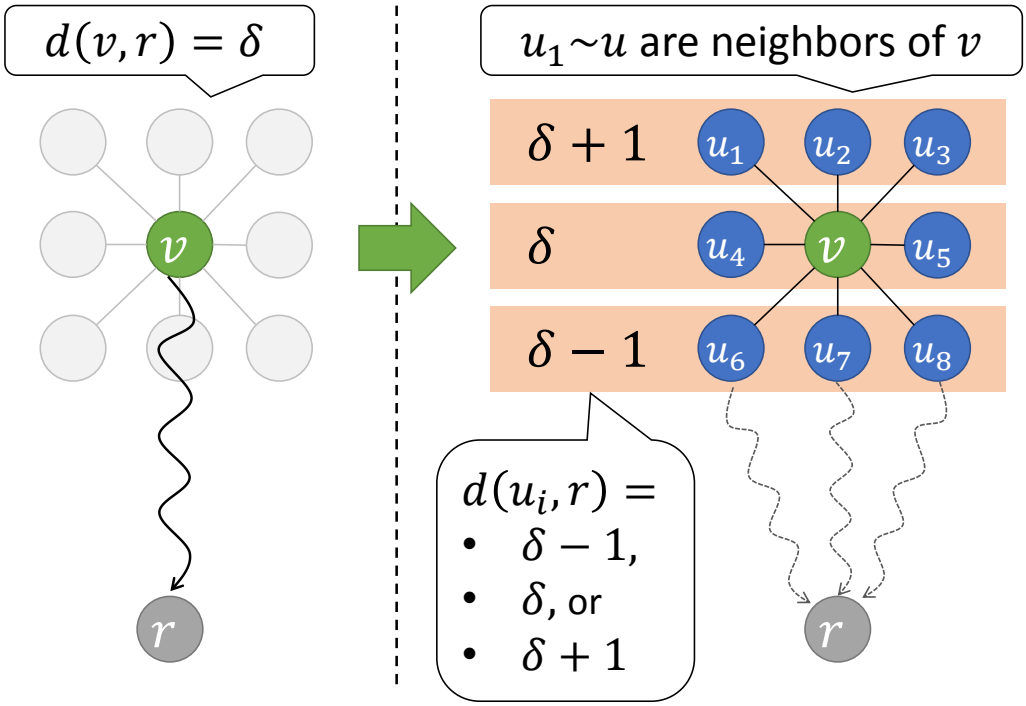


Figure 5.1: The key insight of the bit-parallel labeling scheme.

Note that $S_u^{+1}(v)$ can be obtained as $S_u \setminus (S_u^{-1}(v) \cup S_u^0(v))$, but actually we do not use $S_u^{+1}(v)$ in the querying algorithm.

In order to describe subsets of S_r by bit vectors of b bits, we assign an unique number between one and $|S_r|$ to each vertex in S_r, and express presence of the i-th vertex by setting the i-th bit.

5.1.2 Bit-parallel BFS

We once put aside the pruning discussed in Section 4.1.2 and we make a bit-parallel version of the naive labeling method discussed in Section 4.1.1. We introduce pruning later in Section 5.1.4.

Let $r \in V$ be a vertex and $S_r \subseteq N_G(r)$ be a subset of neighbors of r with size at most b. We explain an algorithm to compute $d_G(r, v)$, $S_r^{-1}(v)$ and $S_r^0(v)$ for all $v \in V$ that are reachable from $\{r\} \cup S_r$. The algorithm is described as Algorithm 5.1. Basically we conduct a BFS from r computing sets S^{-1} and S^0.

Let v be a vertex. Suppose that we have already computed $S_r^{-1}(w)$ for all w such that $d_G(r, w) < d_G(r, v)$. We can compute $S_r^{-1}(v)$ as follows,

$$S_r^{-1}(v) = \left\{ u \in S_r \mid u \in S_r^{-1}(w), w \in N_G(v), d_G(r, w) = d_G(r, v) - 1 \right\},$$

since if u is in $S_r^{-1}(v)$, $d_G(u, v) = d_G(r, v) - 1$ and therefore u is on one of the shortest paths from r to v. Similarly, assuming that we have already computed $S_r^{-1}(w)$ for all w such that $d_G(r, w) \leq d_G(r, v)$ and $S_r^0(w)$ for all w such that $d_G(r, w) < d_G(r, v)$, we can compute $S_r^0(v)$ as follows,

$$\begin{aligned} S^0(v) = &\left\{ u \in S_r \mid u \in S_r^0(w), w \in N_G(v), d_G(r, w) = d_G(r, v) - 1 \right\} \\ &\cup \left\{ u \in S_r \mid u \in S_r^{-1}(w), w \in N_G(v), d_G(r, w) = d_G(r, v) \right\}. \end{aligned}$$

Therefore, along with the breadth-first search, we can compute S_r^{-1} and S_r^0 alternately by dynamic programming in the increasing order of distance from r.

Algorithm 5.1 Bit-parallel BFS from $r \in V$ and $S_r \subseteq N_G(r)$.

```
1: procedure BP-BFS(G, r, S_r)
2:     (P[v], S_r^{-1}[v], S_r^0[v]) ← (∞, ∅, ∅) for all v ∈ V
3:     (P[r], S_r^{-1}[r], S_r^0[r]) ← (0, ∅, ∅)
4:     (P[v], S_r^{-1}[v], S_r^0[v]) ← (1, {v}, ∅) for all v ∈ S_r
5:     Q_0, Q_1 ← an empty queue
6:     Enqueue r onto Q_0
7:     Enqueue v onto Q_1 for all v ∈ S_r
8:     while Q_0 is not empty do
9:         E_0 ← ∅ and E_1 ← ∅
10:        while Q_0 is not empty do
11:            Dequeue v from Q_0.
12:            for all u ∈ N_G(v) do
13:                if P[u] = ∞ ∨ P[u] = P[v] + 1 then
14:                    E_1 ← E_1 ∪ {(v, u)}
15:                    if P[u] = ∞ then
16:                        P[u] ← P[v] + 1
17:                        Enqueue u onto Q_1.
18:                else if P[u] = P[v] then
19:                    E_0 ← E_0 ∪ {(v, u)}
20:        for all (v, u) ∈ E_0 do
21:            S_r^0[u] ← S_r^0[u] ∪ S_r^{-1}[v]
22:        for all (v, u) ∈ E_1 do
23:            S_r^{-1}[u] ← S_r^{-1}[u] ∪ S_r^{-1}[v]
24:            S_r^0[u] ← S_r^0[u] ∪ S_r^0[v]
25:        Q_0 ← Q_1 and Q_1 ← ∅
26:    return (P, S_r^{-1}, S_r^0)
```

That is, first we compute $S_r^{-1}(v)$ for all $v \in V$ such that $d_G(r, v) = 1$, next we compute $S_r^0(v)$ for all $v \in V$ such that $d_G(r, v) = 1$, then we compute $S_r^{-1}(v)$ for all $v \in V$ such that $d_G(r, v) = 2$, next we compute $S_r^0(v)$ for all $v \in V$ such that $d_G(r, v) = 2$, and so on. Note that operations on sets can be done in $O(1)$ time by representing sets by bit vectors and using bitwise operations.

5.1.3 Bit-parallel Distance Querying

To process a distance query between a pair of vertices s and t, as with normal labels, we scan bit-parallel labels $L_{\mathrm{BP}}(s)$ and $L_{\mathrm{BP}}(t)$. For each pair of quadruples that share the same root vertex, $(r, \delta_{rs}, S_r^{-1}(s), S_r^0(s)) \in L_{\mathrm{BP}}(s)$ and $(r, \delta_{rt}, S_r^{-1}(t), S_r^0(t)) \in L_{\mathrm{BP}}(t)$, from these quadruples we compute distance between s and t via one of vertices in $\{r\} \cup S_r$. That is, we compute $\delta = \min_{u \in \{r\} \cup S_r} \{d_G(s, u) + d_G(u, t)\}$. A naive way is to compute $d_G(s, u)$ and $d_G(u, t)$ for all u and take the minimum, which takes $O(|S_r|)$ time. However, we propose an algorithm to compute δ in $O(1)$ time by exploiting bitwise operations.

Let $\tilde{\delta} = d_G(s, r) + d_G(r, t)$. Since $\tilde{\delta}$ is an upper bound on δ and $d_G(s, u) \geq d_G(s, r) - 1, d_G(u, t) \geq d_G(r, t) - 1$ for all $u \in S_r$, $\tilde{\delta} - 2 \leq \delta \leq \tilde{\delta}$. Therefore, what we have to do is to judge whether the distance δ is $\tilde{\delta} - 2$, $\tilde{\delta} - 1$ or $\tilde{\delta}$.

This can be done as follows. If $S_r^{-1}(s) \cap S_r^{-1}(t) \neq \emptyset$, then $\delta = \tilde{\delta} - 2$. Otherwise,

if $S_r^0(s) \cap S_r^{-1}(t) \neq \emptyset$ or $S_r^{-1}(s) \cap S_r^0(t) \neq \emptyset$, then $\delta = \tilde{\delta} - 1$, and otherwise $\delta = \tilde{\delta}$. Note that computing intersections of sets can be done by bitwise AND operations. Therefore, all these operations can be done in $O(1)$ time. Thus, the distance δ can be computed in $O(1)$ time, and, in total, we can answer each query in $O(|L_{\mathrm{BP}}(s)| + |L_{\mathrm{BP}}(t)|)$ time.

5.1.4 Introducing to Pruned Labeling

Now we discuss how to combine this bit-parallel labeling methods and the pruned labeling method discussed in Section 4.1.2. We propose a simple and efficient way as follows. First we conduct bit-parallel BFSs without pruning for t times, where t is a parameter. Then, we conduct pruned BFSs using both the bit-parallel labels and normal labels for pruning.

This method exploits different strength of the pruned labeling method and the bit-parallel labeling method. In the beginning, pruning does not work much and pruned BFSs visits large portion of the vertices. Therefore, instead of pruned labeling, we use bit-parallel labeling without pruning to efficiently cover a larger part of pairs of vertices. Skipping the overhead of vain pruning tests also contributes the speed-up.

As roots and neighbor sets for bit-parallel BFSs, we propose to greedily use vertices with the highest priority: we select a vertex with the highest priority as the root r among remaining vertices, and we select up to b vertices with the highest priority as the set S_r among remaining neighbors.

As we see in the experimental results in Section 5.2, this method improves the preprocessing time, the index size and the query time. Moreover, as we also confirm in the experiments, if we do not set too large value as t, at least it does not spoil the performance. Therefore we do not have to be too serious about finding a proper value for t, and our method is still easy to use.

5.1.5 Online Update

As with static pruned landmark labeling, the dynamic method can be combined with bit-parallel labeling to further improve the performance. Bit-parallel labels can be also updated incrementally. For each root r, we resume the BFS from one of the endpoints of the new edge. The main difference is that, for any visited vertex v, even if $d(r, v)$ has not changed, we push v to the queue if the associated bitsets are changed.

5.2 Experiments

5.2.1 Setup

Environment

We conducted experiments on a Linux server with Intel Xeon X5670 (2.93 GHz) and 48GB of main memory. The proposed method was implemented in C++. We used 8-bit integers to represent distances, 32-bit integers to represent vertices, and 64-bit integers to conduct bit-parallel BFSs. For vertex ordering, we mainly use the Degree strategy and we do not specify the vertex ordering strategy unless we use other strategies. For query time, we generally report the average time for 1,000,000 random queries.

Static Graph Datasets

To show the efficiency and robustness of our method, we conducted experiments on various real-world networks: five social networks, three web graphs and three computer networks. We treated all the graphs as undirected, unweighted graphs. Basically we used five smaller datasets to compare the performance between the proposed method and previous methods and to analyze the behavior of these methods, and used larger six datasets to show the scalability of the proposed method. The types of networks, the numbers of vertices and edges are presented in Table 5.1. The detailed description of each network is as follows.

Table 5.1: Static complex network datasets

Dataset	Network	$\|V\|$	$\|E\|$
Gnutella	Computer	63 K	148 K
Epinions	Social	76 K	509 K
Slashdot	Social	82 K	948 K
Notredame	Web	326 K	1.5 M
WikiTalk	Social	2.4 M	4.7 M
Skitter	Computer	1.7 M	11 M
Indo	Web	1.4 M	17 M
MetroSec	Computer	2.3 M	22 M
Flickr	Social	1.8 M	23 M
Hollywood	Social	1.1 M	114 M
Indochina	Web	7.4 M	194 M

Gnutella: This is a graph created from a snapshot of the Gnutella P2P network in August 2002 [RIF02].

Epinions: This graph is the on-line social network in Epinions (`www.epinions.com`), where each vertex represents a user and each edge represents a trust relationship [RAD03].

Slashdot: This is the on-line social network in Slashdot (`slashdot.org`) obtained in February 2009. Vertices correspond to users and edges correspond to friend/foe links between the users [LLDM09].

NotreDame: This is a web graph between pages from University of Notre Dame (domain `nd.edu`) collected in 1999 [AJB99].

WikiTalk: This is the on-line social network among editors of Wikipedia (`www.wikipedia.org`) created by communication on edits on talk pages by till January 2008 [LHK10b, LHK10a].

Skitter: This is an Internet topology graph created from traceroutes run in 2005 by Skitter [LKF05].

Indo: This is a web graph between pages in `.in` domain crawled in 2004 [BV04, BRSV11].

MetroSec: This is a graph constructed from Internet traffic captured by MetroSec. Each vertex represents a computer and two vertices are linked if they appear in a packet as sender and destination [MLH09].

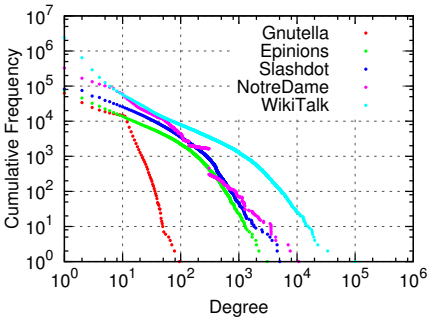
Flickr: This is the on-line social network in a photo-sharing site, Flickr (`www.flickr.com`) [MMG+07].

Indochina: This is a web graph of web pages in the country domains of Indochina countries, crawled in 2004 [BV04, BRSV11].

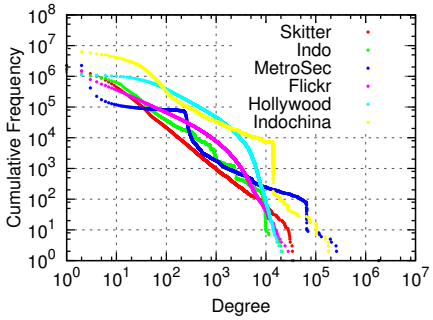
Hollywood: This is a social network of movie actors. Two actors are linked if they appeared in a movie together by 2009 [BV04, BRSV11].

First, we investigated the degree distribution of the networks, since degrees of vertices play important roles in our method when we use DEGREE strategy for vertex ordering. Figures 5.2a and 5.2b are the log-log plot of degree complementary cumulative distribution. As expected, we can confirm that all these networks generally exhibit power-law degree distributions.

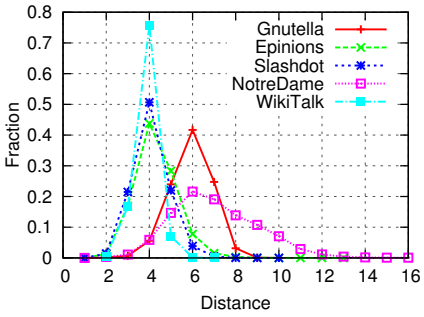
Then, we also examined the distribution of distances. Figures 5.2c and 5.2d show distribution of distances for 1,000,000 random pairs of vertices. As we can observe from these figures, these networks are also small-world networks, in the sense that the average distance is very small.



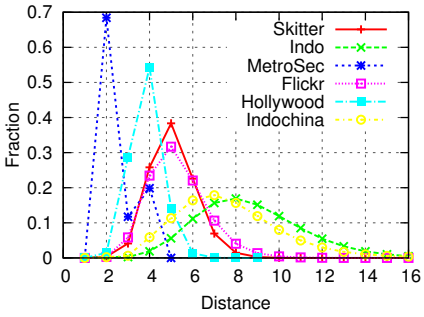
(a) Degree distribution of smaller five datasets.



(b) Degree distribution of larger six datasets.



(c) Distance distribution of smaller five datasets.



(d) Distance distribution of larger six datasets.

Figure 5.2: Properties of the static complex network datasets.

Dynamic Graph Datasets

We treated all the graphs as undirected graphs. Statistics of the datasets are given in Table 5.2. The detailed description of each dataset is as follows.

Epinions: This is the on-line social network in Epinions (`www.epinions.com`) [MA05].

Enron: This is an e-mail network among employees of *Enron* between 1999 and 2003 [KY04]. Vertices are employees and edges are created by e-mails between two employees.

P2P: This is a graph constructed from a log of an *eDonkey* server where vertices describe users and a link between two users appears when one provided a file to the other [ALM09].

Table 5.2: Dynamic Complex Network Datasets

Dataset	Network	$\|V\|$	$\|E\|$	Snapshots
`Epinions`	Social	132 K	831 K	421 K
`Enron`	Social	87 K	1.1 M	574 K
`P2P`	Computer	1.1 M	6.3 M	3.1 M
`YouTube`	Social	3.2 M	9.4 M	4.7 M
`Wikipedia`	Web	1.9 M	36.5 M	18.3 M
`DMS`	Synthetic	1.0 M	10.5 M	5.2 M
`Hyperbolic`	Synthetic	1.0 M	10.5 M	5.2 M
`ForestFire`	Synthetic	1.0 M	7.6 M	3.8 M

YouTube: This is the online social network among users of YouTube (`www.youtube.com`) crawled daily in 2007 [Mis09]. Vertices and edges correspond to users and friend links.

Wikipedia: is a web graph between English Wikipedia pages (`en.wikipedia.org`) constructed from edit history in 2007 [Mis09]. Each vertex represents a page and each edge represents a link.

DMS: This is a synthetic graph constructed under the Dorogovtsev-Mendes-Samukhin model [DMS00], which is a simple growth model based on preferential attachment that exhibits power-law degree distribution with configurable power-law exponent. We generally set the power-law exponent as about 2.3.

Hyperbolic: This is also a growth model based on preferential attachment. However, in addition to power-law degree distribution, it takes into account similarity of vertices by using distance on a hyperbolic space, leading to large clustering coefficients.

ForestFire: This is a graph generated by the forest fire model [LKF07], which exhibits not only standard static properties but also common properties on dynamic networks like densification power laws and shrinking diameter. For the `DMS` network, we set the power-law exponent as around 2.3. We generated the `ForestFire` network by Stanford Network Analysis Platform with the default parameters.

When using dynamic graph datasets, we conducted the experiments as follows.

1. We first construct an index from a graph with all the edges except last 10,000 edges by the offline indexing algorithm.
2. Then, we measure average update time by inserting the last 10,000 edges.
3. Finally, we measure the average query time with 1,000,000 random queries after reflecting all the dynamic updates.

As a baseline, the average time of BFSs for 1,000 random pairs is also reported.

5.2.2 Performance on Static Networks

First we present the performance of our method on the real-world datasets to show the efficiency and robustness of our method. Table 5.3 shows the performance of our method for the datasets. IT denotes preprocessing time, IS denotes index size, QT denotes average query time for 1,000,000 random queries, and LN denotes the average label size for each vertex, in the format of the size of normal

labels (left) plus the size of bit-parallel labels (right). We set the number of times we conduct bit-parallel BFSs as 16 for first five datasets and 64 for the rest.

In Table 5.3, we also listed the performance of two of the state-of-the-art existing methods. One is *hierarchical hub labeling* [ADGW12], which is also based on distance labeling. The other one is based on tree decompositions [ASK12], which is an improved version of TEDI [Wei10]. For these previous methods, we used the implementations by the authors of these methods, both in C++. Experiments for hierarchical hub labeling were conducted on a Windows server with two Intel Xeon X5680 (3.33GHz) and 96GB of main memory. Experiments for the tree-decomposition-based method were conducted on our environment described above. We also described the average time to compute distance by breadth-first search for 1,000 random pairs of vertices. Among these four methods including the proposed method, only the preprocessing of hierarchical hub labeling [ADGW12] was parallelized to use all the 12 cores. All the other timing results are sequential.

Preprocessing Time and Scalability

Our emphasis is particularly on the large improvement in the preprocessing time, leading to much better scalability. First, we successfully preprocessed the largest two datasets Hollywood and Indochina with millions of vertices and hundreds of millions of edges in moderate preprocessing time of 15,164 seconds and 6,068 seconds. This is improvement of two orders of magnitude on the graph size we can handle since other existing exact distance querying methods take thousands or tens of thousands of seconds to preprocess graphs with millions of edges.

For next four datasets with tens of millions of edges, it took less than one thousand seconds, while the previous methods did not finish after one day or ran out of memory. For smaller six datasets, they took at most one minute, and about at least 50 times faster than the previous methods for the most of them.

Query Time

The average query time was generally microseconds and at most 16 microseconds. For almost all the smaller five datasets, the query time of the proposed method is faster than the query time of the previous methods.

Index Size

for three datasets, index sizes were smaller than or almost equal to the previous methods, and for the other two datasets, index sizes were about 2.5 times larger than the tree-decomposition-based method, which is not big difference and still acceptable. For next four networks, index sizes range from two gigabytes to four gigabytes, and for the largest two networks, index sizes are 12GB and 22GB. For networks of these sizes, the graph data themselves have sizes of gigabytes, and nowadays computers with tens of gigabytes of memory are neither rare nor expensive, therefore it would be acceptable. However, even though nowadays computers with tens of gigabytes of memory are neither rare nor expensive, reducing the index size can be an important next research issue.

Table 5.3: Performance comparison between the proposed method and previous methods for the real-world datasets. IT denotes indexing time, IS denotes index size, QT denotes query time, and LN denotes average label size for each vertex. DNF means it did not finish in one day or ran out of memory.

Dataset	Pruned Landmark Labeling (This work)				Hierarchical Hub Labeling [ADGW12]				Tree Decomposition [ASK12]			BFS
	IT	IS	QT	LN	IT	IS	QT	LN	IT	IS	QT	
Gnutella	54 s	209 MB	5.2 μs	644+16	245 s	380 MB	11 μs	1,275	209 s	68 MB	19 μs	3.2 ms
Epinions	1.7 s	32 MB	0.5 μs	33+16	495 s	93 MB	2.2 μs	256	128 s	42 MB	11 μs	7.4 ms
Slashdot	6.0 s	48 MB	0.8 μs	68+16	670 s	182 MB	3.9 μs	464	343 s	83 MB	12 μs	12 ms
NotreDame	4.5 s	138 MB	0.5 μs	34+16	10,256 s	64 MB	0.4 μs	41	243 s	120 MB	39 μs	17 ms
WikiTalk	61 s	1.0 GB	0.6 μs	34+16	DNF	-	-	-	2,459 s	416 MB	1.8 μs	197 ms
Skitter	359 s	2.7 GB	2.3 μs	123+64	DNF	-	-	-	DNF	-	-	190 ms
Indo	173 s	2.3 GB	1.6 μs	133+64	DNF	-	-	-	DNF	-	-	150 ms
MetroSec	108 s	2.5 GB	0.7 μs	19+64	DNF	-	-	-	DNF	-	-	150 ms
Flickr	866 s	4.0 GB	2.6 μs	247+64	DNF	-	-	-	DNF	-	-	361 ms
Hollywood	15,164 s	12 GB	15.6 μs	2,098+64	DNF	-	-	-	DNF	-	-	1.2 s
Indochina	6,068 s	22 GB	4.1 μs	415+64	DNF	-	-	-	DNF	-	-	1.5 s

5.2.3 Analysis

Next we analyze the behavior of our method to investigate why our method is efficient.

Pruned BFS

First we study how labels are computed and stored. Figure 5.3a shows the number of distances added to labels in each pruned BFS, and Figure 5.3b shows the cumulative distribution of it, that is, the ratio of the distances stored no later than each step to all the distances stored in the end. We did not use bit-parallel BFSs for these experiments.

From these figures, we can confirm the large impact of the pruning. Figure 5.3a shows that the number of distances added to labels in each BFS decreases so rapidly. For example, after 1,000 times of BFSs, for all the three datasets distances are added to the labels of only less than 10% of the vertices, and after conducting 10,000 times of BFSs, for all the three datasets distances are added to the labels of only less than 1% of the vertices. Figure 5.3b also shows that large portion of the labels are computed in the beginning.

Sizes of Labels

Figure 5.3c shows the distribution of the sizes of labels after the whole preprocessing, sorted in the ascending order of sizes. We can observe that the size of a label each vertex has do not differ much for different vertices, and few vertices have much larger labels than the average. This shows that the query time of our method is quite stable.

If you are anxious about vertices with unusually large labels, you can precompute the distance between these vertices and all the vertices and answer it directly, since the number of such vertices are few as shown in Figure 5.3c.

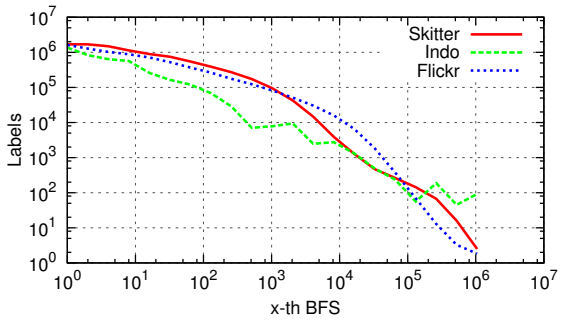
Pair Coverage

Figure 5.4a illustrates the ratio of the *covered* pairs of vertices, that is, the pairs of vertices whose distances can be answered correctly by current labels, at each step. We used 1,000,000 random pairs to estimate these ratios. We can observe that most pairs are covered in the beginning. This shows that such a large portion of pairs have the shortest paths that pass such a small portion of central vertices, which are selected by the Degree strategy. This is the reason why landmark-based approximate methods have good precision, and also the reason why our pruning works so effectively.

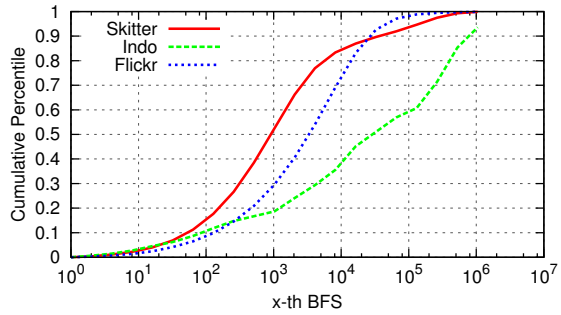
Figures 5.4b, 5.4c and 5.4d illustrate the ratio of the covered pairs of vertices at each step with pairs classified by distance. They show that generally distant pairs are covered earlier than close pairs. This is the reason why the precision of landmark-based approximate methods for close pairs are far worse than the precision for distant pairs. On the other hand, our method aggressively exploits this property: because distant pairs are covered in the beginning, we can prune distant vertices when processing other vertices, which results in fast preprocessing.

Vertex Ordering Strategies

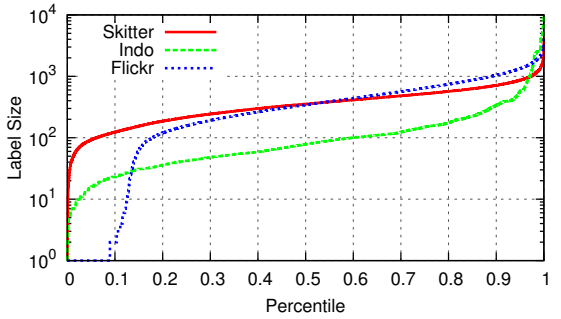
Based on the results on landmark-based methods [PBCG09], we propose and examine the following three strategies.



(a) Number of vertices labeled in each pruned BFS.



(b) Cumulative distribution of the number of vertices labeled in each pruned BFS.



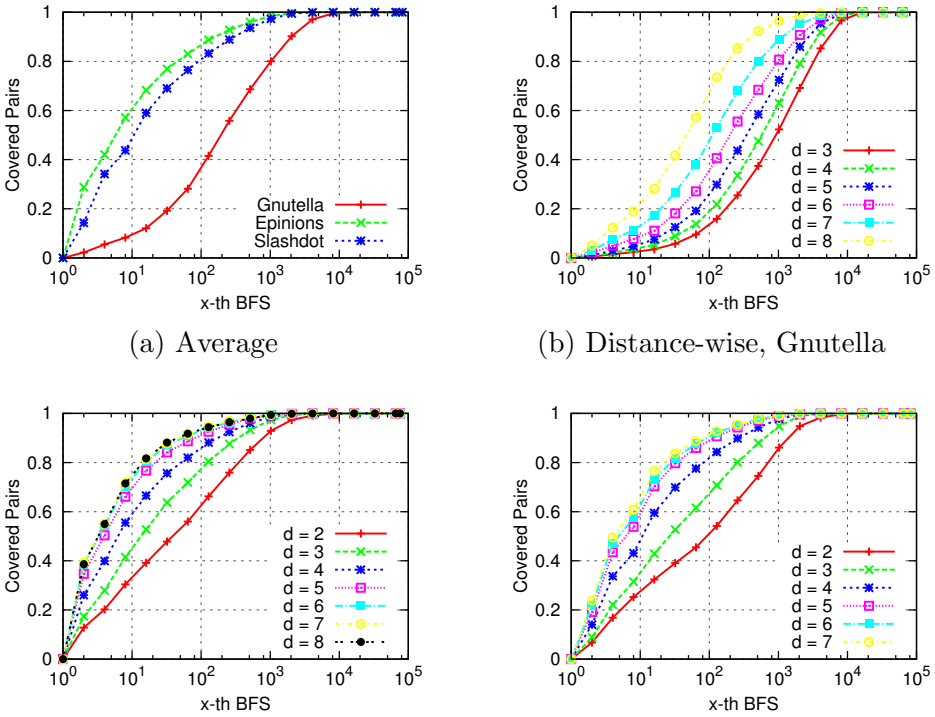
(c) Distribution of the sizes of labels.

Figure 5.3: Effect of pruning and sizes of labels.

- RANDOM: We order vertices randomly. We use this method as a baseline to show the significance of other strategies. As we show in the experiments, this strategy is much worse than other strategies.

- DEGREE: We order vertices from those with higher degree. The idea behind this strategy is that vertices with higher degree have stronger connection to many other vertices and therefore many shortest paths would pass through them.

- CLOSENESS: We order vertices from those with the highest closeness centrality. Since computing exact closeness centrality for all vertices costs $O(nm)$ time, which is too expensive for large-scale networks, we approximate closeness centrality by randomly sampling a small number of vertices and computing distances from those vertices to all vertices.



(a) Average

(b) Distance-wise, Gnutella

(c) Distance-wise, Epinions

(d) Distance-wise, Slashdot

Figure 5.4: Fraction of pairs of vertices whose distance can be answered by index, against number of performed pruned BFS.

Table 5.4: Average size of a label for each vertex against different vertex ordering strategies.

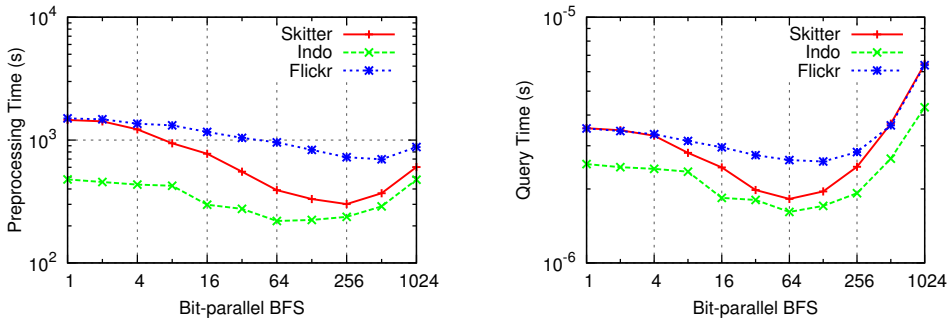
Dataset	Random	Degree	Closeness
Gnutella	6,171	781	865
Epinions	7,038	124	132
Slashdot	8,665	216	234
NotreDame	DNF	60	82
WikiTalk	DNF	118	158

We see the effect of vertex ordering strategies. Table 5.4 describes the average size of a label for each vertex using different vertex ordering strategies. We did not use bit-parallel BFSs for these experiments. For the Closeness strategy, we randomly sampled 50 vertices and conducted BFSs from these vertices to estimate closeness centrality. For the NotreDame dataset and the WikiTalk dataset, using the Random strategy our program did not finish because it ran out of memory.

As we can see, results are not so different between the Degree strategy and the Closeness strategy. The Degree strategy might be slightly better. On the other hand, the result of the Random strategy is much worse than other two strategies. This is because it is critical for our method that the first vertices are not central since we fail to prune later BFSs efficiently. This shows that by the Degree and Closeness strategies we can successfully capture central vertices.

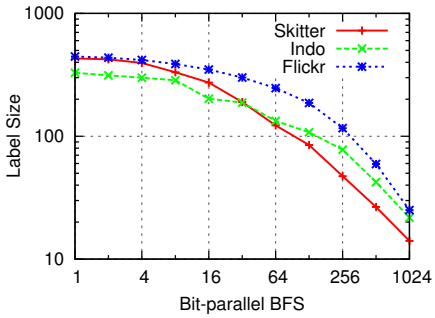
Bit-parallel BFS

Finally, we see the effect of bit-parallel BFSs discussed in Section 5.1. Figure 5.5 shows the performance of our method against different number of times we conduct bit-parallel BFSs.

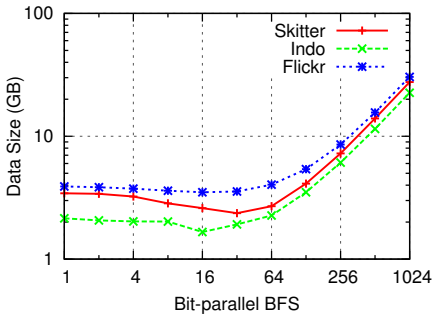


(a) Preprocessing time

(b) Query time



(c) Average size of a normal label



(d) Index size

Figure 5.5: Performance varing number of bit-parallel BFSs.

Figure 5.5a illustrates preprocessing time. It shows that, with a proper number of bit-parallel BFSs, preprocessing time gets two to ten times faster, resulting in the further enhancement to the scalability of our method. Figure 5.5b illustrates query time. We can confirm that query time also gets faster.

Figure 5.5c shows the average size of a normal label for each vertex. As we increase the number of bit-parallel BFSs, many pairs are covered by special labels computed by bit-parallel BFSs, and the size of normal labels decreases. Figure 5.5d shows the index size. With a proper number of bit-parallel BFSs, index size also decreases.

Another important finding from these figures is that the performance of our method is not too sensitive to the parameter of the number of bit-parallel BFSs. As they show, the performance of our method does not become worse much unless we choose a too big number. The proper parameters seem to common between different networks. Therefore, our method still is easy to use with this bit-parallel technique.

Table 5.5 lists the performance results without bit-parallel BFSs. We confirm that, even without bit-parallel BFSs, our pruned labeling scheme itself is powerful and indeed outperforms previous exact indexing methods. On the other hand, we also confirm that the bit-parallel technique significantly improves the overall performance.

Table 5.5: Performance results without bit-prallel BFSs.

Dataset	IT	IS	QT	LN
Skitter	1495 s	3.6 GB	3.8 μs	456
Indo	430 s	2.1 GB	2.6 μs	332
Flickr	1586 s	4.1 GB	3.8 μs	468

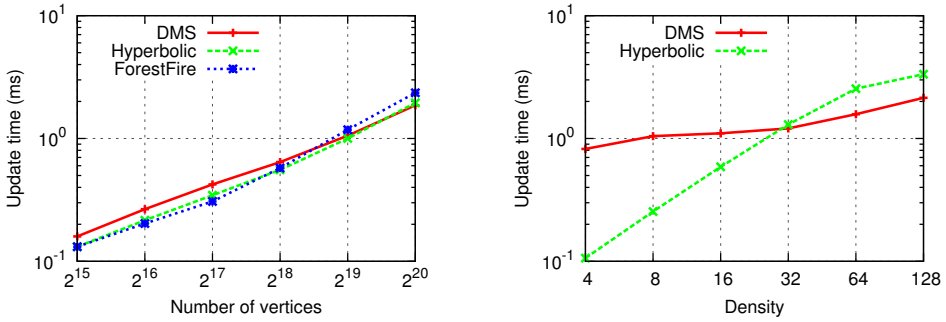
5.2.4 Performance on Dynamic Graphs

Finally, we examine the performance of update algorithm by using the dynamic graph datasets. Average update time reported in Table 5.6 is measured by inserting the last 10,000 edges of each dataset. We observe that the average update time is generally milliseconds, which is four or five orders of magnitude faster than full index reconstruction. Thus, we can confirm that our incremental update algorithm is also practical. As we can expect that queries are much more frequent than updates in real dynamic networks, although updates are slower than queries, updates are also sufficiently fast.

Table 5.6: Experimental results of our online update algorithm.

Dataset	Update time	Label increase	Visited vertices
`Epinions`	0.2 ms	1.0×10^{-4}	5.7
`Enron`	0.5 ms	3.2×10^{-4}	4.6
`P2P`	4.0 ms	1.5×10^{-4}	2.9
`YouTube`	4.3 ms	1.6×10^{-4}	11.4
`Wikipedia`	6.1 ms	4.0×10^{-5}	5.4
`DMS`	1.9 ms	1.8×10^{-4}	8.2
`Hyperbolic`	1.9 ms	2.5×10^{-7}	3
`ForestFire`	2.6 ms	3.7×10^{-4}	9.8

We further study update time in Figure 5.6, which illustrates the average update time on synthetic networks with different graph size or density. We define the density of graph $G = (V, E)$ as $|E| / |V|$. Figure 5.6a illustrates the average update time on synthetic networks with different numbers of vertices. For the DMS and Hyperbolic model, we set the density as ten. Note that the Forest Fire model cannot solely configure density and these generated networks follow the densification power law (i.e., $|E| \propto |V|^{1.2}$). From the figure, we can observe the average update time grows slowly with graph sizes. Similarly, Figure 5.6b describes the average update time on synthetic networks with 2^{17} vertices and different density. Though the patterns are different between the two models, we can confirm that the query time does not get too slow for denser networks.



(a) Varying graph size (b) Varying density

Figure 5.6: Update time on synthetic networks with different size and density.

Average label increase in Table 5.6 describes the average difference of the average label size before and after inserting an edge. As the average label increase is also small, the label size grows slowly. From small update time and label increase, we can conclude that our incremental update algorithm is practical.

Table 5.6 also lists the average number of vertices visited during each BFS

for dynamic index updates. More precisely, we define that a vertex is *visited* if it is inserted in the queue. It shows the numbers of visited vertices are very small, that is, the dynamic update is certainly done quite locally.

Chapter 6

Path-based Labeling for Directed Acyclic Graphs

In this chapter, we focus on another kind of queries called *reachability queries*. As reachability querying is a specialized problem of distance querying, it is easy to see that we can use our pruned landmark labeling algorithm also for reachability queries. In this chapter, after examining application of pruned landmark labeling to reachability queries (Section 6.1), we propose another method named *pruned path labeling*, which is specialized for reachability queries (Section 6.2). Then, we study the performance of these methods theoretically (Section 6.3) and empirically (Section 6.4).

In this chapter, let $G = (V, E)$ be a directed graph. For two vertices $s, t \in V$, we define reach(s, t) as **true** if there is a path from s to t and **false** otherwise. A reachability query (s, t) asks whether reach(s, t) is **true** or not. Let $v \in V$ be a vertex, we denote by anc(v) the set of vertices that can reach v, and by dec(v) the set of vertices that can be reached from v. We denote the children of v by children(v) and the parents of v by parents(v).

We can safely assume that the input graph is always a directed acyclic graph (DAG). To see this, let G be a directed graph. Note that all vertices in a strongly connected component (SCC) of G is equivalent in terms of reachability since they are reachable each other. Thus, G can be converted into a DAG by SCCs, preserving the information of reachability among vertices. By Tarjan's algorithm [Tar72], we can construct the DAG in $O(n+m)$ time, which is much shorter than indexing time in most cases. Therefore, in this chapter, we assume that the given graph is a DAG.

6.1 Pruned Landmark Labeling for Reachability Queries

First, we explain how to apply the pruned landmark labeling algorithm for reachability queries. Specifically, we focus on the difference between distance querying and reachability querying.

6.1.1 Labeling Algorithm

For simplicity, as with the presentation of pruned labeling algorithm for distance queries, we first describe a naive algorithm to construct L_{OUT} and L_{IN} without pruning, and then proceed to an algorithm with pruning.

Naive Landmark Labeling

Let $V = \{v_1, \ldots, v_n\}$ be the vertex set. We incrementally construct labels by processing $v_1, \ldots, v_n$ in this order. To make exposition easier, we define L^k_{OUT} and L^k_{IN} as L_{OUT} and L_{IN}, respectively, right after processing v_k. In the beginning, we start with $L^0_{\mathrm{OUT}}(v) = L^0_{\mathrm{IN}}(v) = \emptyset$ for every $v \in V$. Suppose that we have constructed L^{k-1}_{OUT} and L^{k-1}_{IN}. Then, we construct L^k_{OUT} and L^k_{IN} by processing v_k. First we describe how to construct L^k_{IN}. We conduct a BFS from v_k, and add v_k to the labels of vertices that are visited during the BFS. Specifically, we set

$$L^k_{\mathrm{IN}}(v) = \begin{cases} L^{k-1}_{\mathrm{IN}}(v) \cup \{v_k\} & \text{if } v \text{ is visited during the BFS,} \\ L^{k-1}_{\mathrm{IN}}(v) & \text{otherwise.} \end{cases}$$

To construct L^k_{OUT}, we conduct a *reversed* BFS from v_k, for which we traverse edges backwards. That is,

$$L^k_{\mathrm{OUT}}(v) = \begin{cases} L^{k-1}_{\mathrm{OUT}}(v) \cup \{v_k\} & \text{if } v \text{ is visited during the reversed BFS} \\ L^{k-1}_{\mathrm{OUT}}(v) & \text{otherwise.} \end{cases}$$

We use L^n_{OUT} and L^n_{IN} to answer reachability queries. Obviously, they satisfy that $\textsc{Query}(s, t, L^n_{\mathrm{OUT}}, L^n_{\mathrm{IN}}) = \mathrm{reach}(s, t)$ since every vertex has all information about which vertices it can reach and it can be reached from. We also note that $\textsc{Query}(s, t, L^k_{\mathrm{OUT}}, L^k_{\mathrm{IN}}) = \textbf{true}$ if and only if there is a path from s to t passing through one of $v_1, \ldots, v_k$.

Pruned Landmark Labeling

In pruned landmark labeling, we stop BFSs by pruning vertices whose reachability can be answered correctly from the labels constructed so far. The key observation is the following: Suppose that we are visiting a vertex v during the (forward) BFS from a vertex v_k, and that v can be shown to be reachable from v_k by existing labels, that is, $\textsc{Query}(v_k, v, L^{k-1}_{\mathrm{OUT}}, L^{k-1}_{\mathrm{IN}})$ is **true**. Then, we prune the vertex v and do not search descendants of v.

Similarly, when we visit v during the reversed BFS from v_k and $\textsc{Query}(v, v_k, L^{k-1}_{\mathrm{OUT}}, L^{k-1}_{\mathrm{IN}})$ is **true**, then we prune the vertex v. Though we have pruned vertices, L^k_{OUT} and L^k_{IN} still satisfy the property that $\textsc{Query}(s, t, L^k_{\mathrm{OUT}}, L^k_{\mathrm{IN}}) = \textbf{true}$ if and only if there is a path from s to t passing through one of $v_1, \ldots, v_k$.

6.2 Pruned Path Labeling

In this section, we first propose the pruned path labeling method, which is based on pruned landmark labeling. Then, we show the correctness of pruned path labeling. As we have already mentioned, an important part of our method is selecting paths for which many pairs of vertices are connected via these paths. We discuss heuristics to select such paths in Section 6.2.4.

6.2.1 Index Data Structure and Query Algorithm

The idea of pruned path labeling is iteratively selecting paths and conducting BFSs from these paths. The main difference from pruned landmark labeling is that we use paths instead of vertices to start BFSs with. Then, we store which vertices can reach these paths or can be reached from these paths. If a query

Algorithm 6.1 Answer a query in pruned path labeling

```
1: procedure QUERY(s, t, L_OUT, L_IN)
2:     i, j ← 1
3:     while i ≤ |L_OUT(s)| and j ≤ |L_IN(t)| do
4:         (p_s, v_s) ← the i-th element in L_OUT(s)
5:         (p_t, v_t) ← the j-th element in L_IN(t)
6:         if p_s = p_t then
7:             if v_s ≤ v_t then return true
8:             else
9:                 i ← i + 1
10:                j ← j + 1
11:        else if p_s < p_t then
12:            i ← i + 1
13:        else
14:            j ← j + 1
15:    return false
```

(s, t) is given, we find a path we have selected with two vertices u, v such that there is a path of the form $s - u - v - t$. In this sense, our method can be seen as a special case of 3-hop cover [JXRF09]. The detail is given in the following.

For a given DAG $G = (V, E)$, we take l paths $P_1, P_2, \ldots, P_l$ such that $\bigcup_{k=1}^{l} P_k = V$. Let $\{v_{k,1}, v_{k,2}, \ldots, v_{k,p_k}\}$ denote the sequence of vertices that forms the path P_k, where $p_k = |P_k|$. We construct two types of labels $L_{\mathrm{OUT}}(v), L_{\mathrm{IN}}(v) \subseteq \mathbb{N} \times \mathbb{N}$. It is supposed that, if $(i, j) \in L_{\mathrm{OUT}}(v)$ for some vertex $v \in V$, then v can reach $v_{i,j}$. Similarly, it is supposed that, if $(i, j) \in L_{\mathrm{IN}}(v)$ for a vertex $v \in V$, then v can be reached from $v_{i,j}$.

We note that, for any vertex $v \in V$ and i, we only have to store at most one pair (i, j) in $L_{\mathrm{OUT}}(v)$ and $L_{\mathrm{IN}}(v)$ to answer reachability. To see this, suppose that v can reach $v_{i,j}$ for some i and j. Then, v can reach every $v_{i,j'}$ for $j \leq j' \leq p_i$ since $v_{i,j}$ can reach $v_{i,j'}$ through the path P_i. Thus, we can choose an integer $j_{\min}$ such that v can reach $v_{i,j}$ if and only if $j_{\min} \leq j \leq p_i$. Therefore, we only have to store the pair $(i, j_{\min})$ in $L_{\mathrm{OUT}}(v)$ to answer the reachability of vertices in P_i from v. Conversely, for each $v \in V$ and i, we only have to store one pair (i, j) in $L_{\mathrm{IN}}(v)$.

Upon a query (s, t), we return $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}})$ defined as follows.

$$\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}}) = \begin{cases} \textbf{true} & \text{if } \exists i, j, j' \in \mathbb{N} \text{ } s.t. \text{ } j \leq j', \\ & \quad (i, j) \in L_{\mathrm{OUT}}(s), \\ & \quad (i, j') \in L_{\mathrm{IN}}(t), \\ \textbf{false} & \text{otherwise.} \end{cases}$$

In words, $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}})$ is **true** if and only if there are a path P_i and two integers j, j' with $j \leq j'$ such that s can reach $v_{i,j}$ and t can be reached from $v_{i,j'}$. We emphasize again that $v_{i,j}$ can reach $v_{i,j'}$ through P_i. We can compute $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}})$ in $O(|L_{\mathrm{OUT}}(s)| + |L_{\mathrm{IN}}(t)|)$ time if $L_{\mathrm{OUT}}(s)$ and $L_{\mathrm{IN}}(t)$ are sorted by path index (See Algorithm 6.1).

6.2.2 Labeling Algorithm

Now we describe how to construct labels L_{OUT} and L_{IN}. Again, we start with a naive algorithm. We basically conduct BFSs from paths $P_1, P_2, \ldots, P_l$ in this

order. Since labels L_{OUT} and L_{IN} grow gradually during the algorithm, we define L^i_{OUT} and L^i_{IN} as L_{OUT} and L_{IN} obtained right after processing the i-th path P_i. In particular, we define $L^0_{\mathrm{OUT}}(v)$ and $L^0_{\mathrm{IN}}(v)$ as $\emptyset$ for all $v \in V$, and the pair $L^l_{\mathrm{OUT}}(v)$ and $L^l_{\mathrm{IN}}(v)$ is the label finally output by the algorithm. Suppose that we have already constructed L^{k-1}_{OUT} and L^{k-1}_{IN}. Then, we construct L^k_{OUT} and L^k_{IN} as follows.

First, we conduct BFSs from vertices in P_k in descending order, that is, from v_{k,p_k} to $v_{k,1}$. In the BFS from the vertex $v_{k,j}$, we update L^k_{IN} as follows.

$$L^k_{\mathrm{IN}}(v) = \begin{cases} L^{k-1}_{\mathrm{IN}}(v) \cup \{(k,j)\} & \text{if } v \text{ is visited during the BFS from } v_{k,j}, \\ L^{k-1}_{\mathrm{IN}}(v) & \text{otherwise.} \end{cases}$$

When performing a BFS from $v_{k,j}$, we do not have to visit vertices that are already visited in previous BFSs since they already have a pair (k, j') for some $j' \geq j$.

After BFSs to construct L^k_{IN} are finished, we conduct reversed BFSs by traversing edges backwards from vertices in P_k in ascending order, that is, from $v_{k,1}$ to v_{k,p_k}. In the reversed BFS from the vertex $v_{k,j}$, we update L^k_{OUT} as follows.

$$L^k_{\mathrm{OUT}}(v) = \begin{cases} L^{k-1}_{\mathrm{OUT}}(v) \cup \{(k,j)\} & \text{if } v \text{ is visited during the} \\ & \quad \text{reversed BFS from } v_{k,j}, \\ L^{k-1}_{\mathrm{OUT}}(v) & \text{otherwise.} \end{cases}$$

Similarly to the previous case, we do not have to visit vertices that are previously visited by reversed BFSs.

Now we improve the naive algorithm by introducing pruning. The idea is the same as pruned landmark labeling. Suppose that we are processing a vertex v in the BFS from a vertex $v_{k,j}$ for some k and j. Then, we issue a query $\textsc{Query}(v_{k,j}, v, L^{k-1}_{\mathrm{OUT}}, L^{k-1}_{\mathrm{IN}})$. If the answer is **true**, we prune v, that is, we stop the BFS at v. When we are processing a vertex v in the reversed BFS from a vertex $v_{k,j}$ for some k and j, we issue a query $\textsc{Query}(v, v_{k,j}, L^{k-1}_{\mathrm{OUT}}, L^{k-1}_{\mathrm{IN}})$ instead. If the answer is **true**, we prune v. A pseudocode for constructing L_{OUT} and L_{IN} is shown in Algorithm 6.2.

Figure 6.1 shows an example of pruned path labeling. Three paths $P_1 = \{v_{1,1}, v_{1,2}, v_{1,3}, v_{1,4}\}$, $P_2 = \{v_{2,1}, v_{2,2}, v_{2,3}\}$, and $P_3 = \{v_{3,1}, v_{3,2}, v_{3,3}\}$ are selected. Then, BFSs are conducted from $v_{1,4}$, $v_{1,3}$, $v_{1,2}$, and $v_{1,1}$ in this order (Figure 6.1a) to obtain L^1_{IN}. We add a pair $(1,2)$ to $L^1_{\mathrm{IN}}(v_{2,2})$, $L^1_{\mathrm{IN}}(v_{2,3})$, and $L^1_{\mathrm{IN}}(v_{3,3})$. Also, we add $(1,1)$ to $L^1_{\mathrm{IN}}(v_{2,1})$. No pruning occurs during BFSs from P_1. Then similarly, reversed BFSs are conducted from $v_{1,1}$, $v_{1,2}$, $v_{1,3}$, and $v_{1,4}$ in this order (Figure 6.1b) to obtain L^1_{OUT}. For example, $L_{\mathrm{OUT}}(v_{2,3})$ obtains a pair $(1,3)$. Next, we conduct BFSs from vertices in P_2 in an appropriate order. When $v_{1,3}$ is visited during the BFS from $v_{2,3}$, we issue the query $\textsc{Query}(v_{2,3}, v_{1,3}, L^1_{\mathrm{OUT}}, L^1_{\mathrm{IN}})$. The query is **true** since $(1,3) \in L^1_{\mathrm{OUT}}(v_{2,3})$ and $(1,3) \in L^1_{\mathrm{IN}}(v_{1,3})$. Therefore, $v_{1,3}$ is pruned and we no longer continue the search from $v_{1,3}$ (Figure 6.1c). We continue this process until we finish performing (reversed) BFSs from all the paths (Figure 6.1d,6.1e,6.1f).

At this point, we note that pruned path labeling is equivalent to pruned landmark labeling when we always take paths of length zero, that is, each path consists of a single vertex.

A potential drawback of adopting paths instead of vertices is that it may increase the index size. This is because that each element in a label is a pair of integers (a path index and an index of a vertex in the path) instead of one

Algorithm 6.2 Conduct pruned BFSs from P_k

```
1: procedure PrunedBFS(G, P_k, L_OUT^{k-1}, L_IN^{k-1})
2:     p ← the number of vertices in P_k
3:     L_OUT^k[v] ← L_OUT^{k-1}[v] for all v ∈ V
4:     L_IN^k[v] ← L_IN^{k-1}[v] for all v ∈ V
5:     Q ← an empty queue
6:     U ← ∅
7:     for i ← p ... 1 do
8:         s ← P_k[i]
9:         Enqueue s onto Q
10:        while Q is not empty do
11:            Dequeue v from Q
12:            U ← U ∪ {v}
13:            if Query(s, v, L_OUT^{k-1}, L_IN^{k-1}) is false then
14:                L_IN^k[v] ← L_IN^k[v] ∪ {(k, i)}
15:                for all u ∈ children(v) do
16:                    if u ∉ U then
17:                        Enqueue u onto Q
18:    U ← ∅
19:    for i ← 1 ... p do
20:        s ← P_k[i]
21:        Enqueue s onto Q
22:        while Q is not empty do
23:            Dequeue v from Q
24:            U ← U ∪ {v}
25:            if Query(s, v, L_OUT^{k-1}, L_IN^{k-1}) is false then
26:                L_OUT^k[v] ← L_OUT^k[v] ∪ {(k, i)}
27:                for all u ∈ parents(v) do
28:                    if u ∉ U then
29:                        Enqueue u onto Q
30:    return (L_OUT^k, L_IN^k)
```

integer (a vertex number) as opposed to pruned landmark labeling. Therefore, we do not have any benefit if we cannot find long paths. Also, it is practically difficult to cover all the vertices by long paths. To address these issues, we combine the two methods. That is, for some constant $a \geq 0$, we perform pruned path labeling from a paths and then perform pruned landmark labeling from remaining vertices. Furthermore, we stop taking paths if the length of the path is shorter than b. From preliminary experiments, we decided to choose $a = 50$ and $b = 10$.

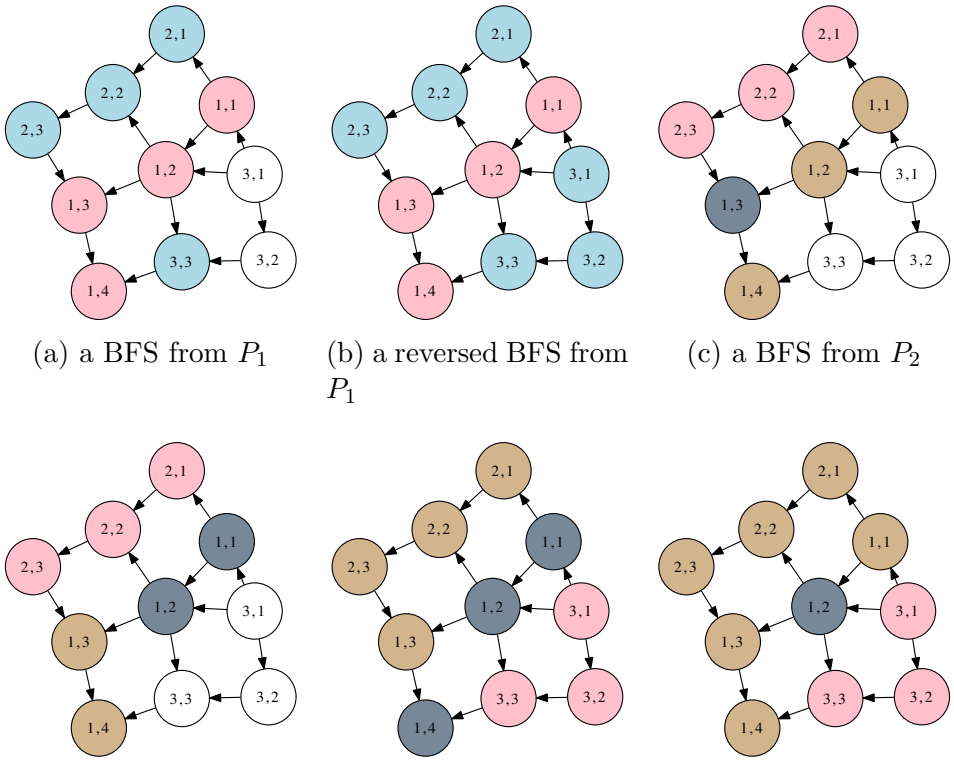
The effectiveness of pruned path labeling largely depends on how we select paths. We will discuss this issue in Section 6.2.4.

6.2.3 Correctness

We prove the correctness of pruned path labeling in the following two steps:

1. First we prove that the labels computed by the naive algorithm correctly answer queries.

2. Then we show that the pruned labels and the naive labels return the same answer for any query.



(a) a BFS from P_1 (b) a reversed BFS from P_1 (c) a BFS from P_2

(d) a reversed BFS from P_2 (e) a BFS from P_3 (f) a reversed BFS from P_3

Figure 6.1: An example of pruned path labeling. Color of a vertex indicates its status: Red is a start point of BFSs, blue is a vertex being searched, gray is a pruned vertex, and brown is a vertex already used as a start point.

Before the proof, we review definitions. Given a DAG $G = (V, E)$, let $P_1, P_2, \ldots, P_l$ be a set of l paths such that $\bigcup_{k=1}^{l} P_k = V$. Let $\{v_{k,1}, v_{k,2}, \ldots, v_{k,p_k}\}$ denote the sequence of vertices that forms the path P_k. Let L_{OUT}^k and L_{IN}^k be the labels obtained right after processing the k-th path in the naive algorithm, and let L_{OUT} and L_{IN} be L_{OUT}^l and L_{IN}^l, respectively.

Theorem 6.1. *For all* $s, t \in V$, $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}}) = \mathrm{reach}(s, t)$

Proof. First we show that $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}}) = \textbf{true}$ implies $\mathrm{reach}(s, t) = \textbf{true}$. Assume that $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}}) = \textbf{true}$, then there exist i, j, j' such that $(i, j) \in L_{\mathrm{OUT}}(s)$, $(i, j') \in L_{\mathrm{IN}}(t)$, and $j \leq j'$. From the algorithm, $s \in \mathrm{anc}(v_{i,j})$ and $t \in \mathrm{dec}(v_{i,j'})$ holds, and it follows that $\mathrm{reach}(s, v_{i,j}) = \textbf{true}$ and $\mathrm{reach}(v_{i,j'}, t) = \textbf{true}$. $\mathrm{reach}(v_{i,j}, v_{i,j'}) = \textbf{true}$ also holds from $j \leq j'$. Therefore, $\mathrm{reach}(s, t) = \textbf{true}$ because of the path $s - v_{i,j} - v_{i,j'} - t$.

Next we show that $\mathrm{reach}(s, t) = \textbf{true}$ implies $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}}) = \textbf{true}$. Clearly, $s \in \mathrm{anc}(t)$ holds. Since $\bigcup_{k=1}^{l} P_k = V$, there exist i, j such that $s \in P_i$, $s = v_{i,j}$. From the algorithm, there exists j' with $j \leq j'$ such that $(i, j') \in L_{\mathrm{IN}}(t)$ since at least $v_{i,j}$ can reach t in the BFS from $v_{i,j}$. Obviously $L_{\mathrm{OUT}}(s)$ has a pair (i, j), thus $\textsc{Query}(s, t, L_{\mathrm{OUT}}, L_{\mathrm{IN}}) = \textbf{true}$. □

Let M_{OUT}^k and M_{IN}^k be the labels obtained right after processing the k-th path in the pruned algorithm, and let M_{OUT} and M_{IN} be M_{OUT}^l and M_{IN}^l, respectively. Then, we prove the following theorem.

Theorem 6.2. *For all* $s,t \in V$ *and* $k \in \mathbb{N}$ *with* $0 \leq k \leq l$, $\textsc{Query}(s,t,L^k_{\mathrm{OUT}},L^k_{\mathrm{IN}}) = \textsc{Query}(s,t,M^k_{\mathrm{OUT}},M^k_{\mathrm{IN}})$

Proof. We prove the following propositions for all $0 \leq k \leq l$ by induction.

(A) For all $s,t \in V$ and $k \in \mathbb{N}$ such that $0 \leq k \leq l$, $\textsc{Query}(s,t,L^k_{\mathrm{OUT}},L^k_{\mathrm{IN}}) = \textsc{Query}(s,t,M^k_{\mathrm{OUT}},M^k_{\mathrm{IN}})$.

(B) For all $s,t \in V$,$\textsc{Query}(s,t,M^k_{\mathrm{OUT}},M^k_{\mathrm{IN}}) =$ **true** if and only if there exist i,j such that $i \leq k$ and $v_{i,j} \in \mathrm{dec}(s) \cap \mathrm{anc}(t)$.

For notational simplicity, we denote $\textsc{Query}(s,t,L^k_{\mathrm{OUT}},L^k_{\mathrm{IN}})$ by $q(s,t,k)$ and $\textsc{Query}(s,t,M^k_{\mathrm{OUT}},M^k_{\mathrm{IN}})$ by $q'(s,t,k)$.

Base case: $L^0_{\mathrm{OUT}}(v) = M^0_{\mathrm{OUT}}(v) = \emptyset$, $L^0_{\mathrm{IN}}(v) = M^0_{\mathrm{IN}}(v) = \emptyset$ for all $v \in V$ by definition. Therefore, $q(s,t,0) = q'(s,t,0) =$ **false** for all $s,t \in V$. From this, (A) and (B) follow.

Inductive case: Suppose (A) and (B) hold for $0,1,\ldots,k-1$. First, we prove (A) by contradiction. Assume that there exist $s,t \in V$ such that $q(s,t,k) \neq q'(s,t,k)$. We can assume that $q(s,t,k-1) = q'(s,t,k-1) =$ **false** since $q(s,t,k) = q'(s,t,k) =$ **true** holds if $q(s,t,k-1) = q'(s,t,k-1) =$ **true**. Moreover, we can assume that $q(s,t,k) =$ **true**, $q'(s,t,k) =$ **false**, and $\mathrm{reach}(s,t) =$ **true** since $q'(s,t,k) =$ **true** implies $q(s,t,k) =$ **true** from $M^k_{\mathrm{OUT}}(v) \subseteq L^k_{\mathrm{OUT}}(v)$ and $M^k_{\mathrm{IN}}(v) \subseteq L^k_{\mathrm{IN}}(v)$ for all $v \in V$. Since $q(s,t,k) =$ **true**, there exist j',j'' such that $(k,j') \in L^k_{\mathrm{OUT}}(s)$, $(k,j'') \in L^k_{\mathrm{IN}}(t)$, and $j' \leq j''$. From (B), we have $v_{i,j} \notin \mathrm{dec}(s) \cap \mathrm{anc}(t)$ for all $i \leq k-1$ and j since $q'(s,t,k-1) =$ **false**. Note that any path between $v_{k,j''}$ and t is contained in $\mathrm{dec}(v_{k,j''}) \cap \mathrm{anc}(t)$. There is no vertex $u \in \mathrm{dec}(v_{k,j''}) \cap \mathrm{anc}(t)$ such that $q'(u,t,k-1) =$ **true** since $\mathrm{dec}(v_{k,j''}) \cap \mathrm{anc}(t) \subseteq \mathrm{dec}(s) \cap \mathrm{anc}(t)$ and $q'(s,t,k-1) =$ **false** hold. Therefore, t is not pruned during the BFS from $v_{k,j''}$, thus $(k,j'') \in M^k_{\mathrm{IN}}(t)$ holds. Similarly $(k,j') \in M^k_{\mathrm{OUT}}(s)$ holds. However, this contradicts the assumption that $q'(s,t,k) =$ **false** since $(k,j') \in M^k_{\mathrm{OUT}}(s)$ and $(k,j'') \in M^k_{\mathrm{IN}}(t)$. Therefore, (A) holds for k.

Next, we prove (B). ($\Rightarrow$): Assume that $q'(s,t,k) =$ **true**. From the definition of q', there exist i,j,j' such that $(i,j) \in M^k_{\mathrm{OUT}}(s)$, $(i,j') \in M^k_{\mathrm{IN}}(t)$, $1 \leq i \leq k$, and $j \leq j'$. The vertex $v_{i,j} \in \mathrm{dec}(s) \cap \mathrm{anc}(t)$ holds since $v_{i,j} \in \mathrm{dec}(s)$, $v_{i,j} \in \mathrm{anc}(v_{i,j'})$, and $v_{i,j'} \in \mathrm{anc}(t)$. ($\Leftarrow$): Assume that there exist i,j such that $i \leq k$ and $v_{i,j} \in \mathrm{dec}(s) \cap \mathrm{anc}(t)$. Then, there exists the minimum $j' \leq j$ with $(i,j') \in \mathrm{dec}(s)$. Similarly there exists the maximum j'' such that $j'' \geq j$, $(i,j'') \in \mathrm{anc}(t)$. From the algorithm, $(i,j') \in L^k_{\mathrm{OUT}}(s)$ and $(i,j'') \in L^k_{\mathrm{IN}}(t)$ hold. Therefore, from (A) and the definition of q, $q'(s,t,k) = q(s,t,k) =$ **true**. The theorem holds when $k = l$. □

Based on the two theorems above, the correctness of the pruned path labeling is proved as the following corollary.

Corollary 6.1. *For any pair of vertices* $s,t \in V$, $\textsc{Query}(s,t,M_{\mathrm{OUT}},M_{\mathrm{IN}}) = \mathrm{reach}(s,t)$

Proof. From Theorem 1 and Theorem 2, $\textsc{Query}(s,t,M_{\mathrm{OUT}},M_{\mathrm{IN}}) = \textsc{Query}(s,t,M^l_{\mathrm{OUT}},M^l_{\mathrm{IN}}) = \textsc{Query}(s,t,L^l_{\mathrm{OUT}},L^l_{\mathrm{IN}}) = \textsc{Query}(s,t,L_{\mathrm{OUT}},L_{\mathrm{IN}}) = \mathrm{reach}(s,t)$. □

6.2.4 Path Selection

As with pruned landmark labeling, vertex ordering strategies largely influence the performance of pruned landmark labeling. Correspondingly, effectiveness of pruning should depend on how to select paths in pruned path labeling. We discuss these problems and propose path selection strategies. We empirically compare these strategies in Section 6.4.4, along with vertex ordering strategies.

We propose three path selection strategies: Longest, DPDeg, and DPInOut. In all strategies, we first assign a value to each vertex. The value assigned to a vertex v is 1, $d(v)$, or $(d_{\mathrm{IN}}(v)+1) \times (d_{\mathrm{OUT}}(v)+1)$ in each method, if the vertex is not selected as a part of a path before. Otherwise, we assign 0 to the vertex. Then, we select the path that maximizes the sum of the value of vertices in it by dynamic programming on the DAG.

After selecting 50 paths in Longest or DPInOut, we order remaining vertices by InOut. In DPDeg, we use Degree to order remaining vertices. The idea behind Longest is that selecting long paths would contribute to the pruning. Similarly, DPDeg and DPInOut are intended to select good vertices as many as possible.

6.3 Theoretical Properties

In [YAIY13], theoretical evidence that our methods perform well on real-world networks is given as follows. Note that analysis given in Section 4.3 applies to pruned landmark labeling even when dealing with reachability queries. In this section, we further see that our pruned path labeling algorithm can efficiently process graphs satisfying a minor-closed property.

Please note that examples of minor-closed properties include having bounded treewidth, planarity and bounded genus. That is, minor-closed graphs are generalization of bounded treewidth graphs. Thus, interestingly, pruned path labeling is not only practically but also theoretically stronger than pruned landmark labeling. We note that minor-closed properties also often appear in real-world networks. For example, consider a directed graph used for program analysis given as follows. The vertex set corresponds to the variable set used in the input program, and edges represent dependency between variables. If the input program executes a dynamic programming on a 2-dimensional array, the dependency among variables will form a grid graph, which is planar. Our analysis implies that pruned path labeling perform well on such networks.

The main result is the following.

Theorem 6.3 ([YAIY13]). *Let P be a minor-closed property and G be a digraph whose underlying graph satisfies P. Then, there is a strategy of choosing paths for which pruned path labeling on G outputs a label of size $O(\log n)$ for each vertex. (Constants depending on P are hidden in the $O(\cdot)$ notations.)*

Again, the theorem implies that index size is $O(n \log n)$ and query time is $O(\log n)$. Unfortunately, we can only prove that the running time of finding the strategy is polynomial (due to Theorem 6.4 below). We leave it as an open problem to improve it to (quasi-)linear time.

We introduce notions describing how many paths we need to decompose a graph into smaller components. In a rooted tree, a *monotone path* is a path with one endpoint at the root.

Definition 6.1 ([AG06, BGJ+12]). *Let G be a connected undirected graph of n vertices. G is (s,k)-path separable (for $k \geq n/2$) if for any rooted spanning tree T*

of G either (1) there exists a set P of at most s monotone paths in T so that each connected component of $G \setminus P$ is of size at most k, or (2) for some $s' < s$, there exists a set P of s' monotone paths in T so that the largest connected component of $G \setminus P$ is $(s - s', k)$-path separable.

G is said to be s-path separable if G is $(s, n/2)$-path separable. The set of paths P is called an s-path separator of G if each connected component of $G \setminus S$ has size at most $n/2$.

In the above definition, the number of vertices in the path separator is left unspecified. Trees are 1-path separable, since S can be taken to be the centroid. Similarly, graphs of treewidth w are $(w + 1)$-path separable. Thorup [Tho04] showed that every planar graph is 3-path separable and we only need the case (1) in the definition. The more general case of minor-closed properties is also known:

Theorem 6.4 ([AG06, BGJ+12])**.** *Let P be a minor-closed property. Then, there exists $s = s(P)$ such that every graph satisfying P is s-path separable. Furthermore, an s-path separator can be computed in polynomial time (for any choice of spanning trees).*

The last component in our proof is the following decomposition lemma.

Lemma 6.1 ([BGJ+12])**.** *Given a digraph G, we can construct in linear time a series of digraphs $G_1, \ldots, G_k$ so that*

(i) G_i is a subgraph of G.

(ii) $\sum_i |V(G_i)| \leq 2|V(G)|$.

(iii) There exists a spanning tree T_G for the underlying graph of G with the following property. Any monotone path in T_G restricted to G_i for any $i \in \{1, \ldots, k\}$ is a concatenation of at most two dipaths, and all vertices reachable from and to such a dipath are contained in G_{i-1}, G_i, or G_{i+1}.

Proof of Theorem 6.3. The first part of this proof is to recursively separate the given graph with directed paths. This can be done in essentially the same way with the *recursive graph fragmentation* [BGJ+12] as follows.

We first apply Lemma 6.1 to $G^0 := G$. As a result, a family of subgraphs $G^0_1, G^0_2, \ldots$ and a spanning tree T_{G^0} are obtained. Due to the path separability, there is a set of monotone paths P^0 on T_{G^0} that satisfies one of the following two conditions:

1. the sizes of all the connected components of $G^0 \setminus P^0$ are at most $n/2$, or
2. the largest connected component of $G^0 \setminus P^0$ has size greater than $n/2$ but is path separable.

Let us assume that P^0 satisfies the second condition. We denote the largest connected component in $G^0 \setminus P^0$ by G^1. By applying Lemma 6.1 to G^1, a family of subgraphs $G^1_1, G^1_2, \ldots$ and a spanning tree T_{G^1} are obtained. Then, similarly, there is a set P^1 of monotone paths in T_{G^1}, and, if necessary, we can recurse on the largest connected component of $G^1 \setminus P^1$. We end the recursion when the current connected component has size at most $n/2$. The number of paths in $S = P^0 \cup P^1 \cup \cdots$ is at most $s = \Theta(1)$ in total, and the number of times we recurse is a constant.

We now conduct pruned BFSs from each dipath in S to construct the index. Then as an outer recursion, we recurse on the subgraph induced by C for each

connected component C of $G \setminus S$. Note that C is also s-path separable since C satisfies the property P.

The correctness is clear since we are just applying pruned path labeling on a specific strategy of choosing dipaths. We now consider the efficiency of the strategy. From the property (iii), when performing pruned BFSs from dipaths in G^i_j, each vertex in $G^i_{j-1}, G^i_j, G^i_{j+1}$ will store a constant number of paths to its label. Since $\sum_j |V(G^i_j)| \leq 2n$ and the number of G^i's is $O(1)$, the total number of paths added to the label of a vertex in each step of the outer recursion is $O(1)$.

The size of the remaining connected components halves after each step. Thus, the depth of the outer recursion is $O(\log n)$. □

6.4 Experiments

We conducted two kinds of experiments: performance comparison and analysis. We first compare proposed methods and existing methods. Then, we present the comparison of vertex ordering strategies. These methods are evaluated in terms of query time, index size, and indexing time. As query time, we report the average time over one million random queries.

6.4.1 Experimental Setup

Environment

We conducted all the experiments on a Linux server with Intel Xeon X5675 3.07GHz and 288GB memory. We used only one core on all the experiments. Pruned landmark labeling (PLL) and pruned path labeling (PPL) are compared with three state-of-the-art existing methods, GRAIL [YCZ12], interval list (IL) [Nuu95] and PWAH [vSdM11]. GRAIL is a graph traversal method exploiting labels created by random DFSs, and one of the most memory efficient methods for reachability queries. IL and PWAH are methods that construct compressed transitive closure and they were shown to be the fastest methods for answering reachability queries on large graphs. The implementations of GRAIL and PWAH are by their authors, and the implementation of IL is by the authors of PWAH. In experiments in Section 6.4.2 and Section 6.4.3, we used Degree as the vertex ordering strategy for PLL and DPInOut as path selection strategy for PPL. All algorithms are implemented in C++ using standard template library (STL).

Datasets

We used real-world network datasets with more than a million vertices that have been used in the literature on reachability queries [vSdM11,YCZ12]. The numbers of vertices and edges (after contracting SCCs) are shown in Table 6.1. The detailed description is as follows.

ff/successors: This is a graph used for source code analysis of Firefox [vSdM11].

citeseerx, cit-patents: These are citation networks from CiteSeerX[1] and US patents[2] [YCZ12].

go-uniprot: This is the joint graph of Gene Ontology terms and annotation files from UniProt[3] [YCZ12].

[1] `http://citeseer.ist.psu.edu/`
[2] `http://snap.stanford.edu/data/`
[3] `http://www.uniprot.org/`

Table 6.1: Real-world datasets for reachability queries

Dataset	$\|V\|_{\text{SCC}}$	$\|E\|_{\text{SCC}}$
ff/successors	1,858,504	2,009,541
citeseerx	6,540,399	15,011,259
cit-patents	3,774,768	16,518,948
go-uniprot	6,967,956	34,770,235
uniprot22m	1,595,444	1,595,442
uniprot100m	16,087,295	16,087,293
uniprot150m	25,037,600	25,037,598

uniprot22m, uniprot100m, and uniprot150m: These are RDF graphs from UniProt database [YCZ12]. We note that underlying graphs of these graphs are very close to trees.

We also conducted experiments on even larger synthetic graphs to show the scalability of our methods. These graphs are created as follows. We first randomly determine the topological order of 10 million vertices. Then we randomly connect two non-adjacent vertices $|E|$ times, where $|E|$ is chosen as a parameter. Note that the direction of each edge is uniquely determined by the topological order.

6.4.2 Performance on Real-World Networks

First, we compared PLL and PPL with existing methods on real-world networks. Tables 6.2, 6.3, and 6.4 list the results of our experiments.

Query Time

Table 6.2 shows the average query time on real-world networks. PLL and PPL outperform all the other methods in general. IL also performs quite well especially on citeseerx. In many cases, however, PLL is about twice faster than IL. This is possibly because of compactness of labels and simplicity of the query processing procedure of PLL. PPL is slightly slower than PLL since answering queries by PPL is a little more complicated than PLL. PWAH and GRAIL are comparable on very sparse graphs, but they get very slow on the other graphs.

Table 6.2: Average query time (μs)

	PLL	PPL	GRAIL	IL	PWAH
Dataset	(This work)		[YCZ12]	[Nuu95]	[vSdM11]
ff/successors	0.085	0.133	0.279	0.154	0.202
citeseerx	0.124	0.164	27.946	0.103	0.214
cit-patents	0.253	0.296	11.591	0.292	15.451
go-uniprot	0.156	0.194	0.520	0.233	0.521
uniprot22m	0.083	0.122	0.403	0.173	0.243
uniprot100m	0.133	0.197	0.743	0.292	0.361
uniprot150m	0.153	0.223	0.776	0.248	0.351

Index Size and Indexing Time

Table 6.3 suggests that the index size of PLL and PPL are reasonable, though there is no doubt that GRAIL is the most memory-efficient method. On

uniprot22m, uniprot100m, and uniprot150m, PLL and PPL perform the best, but the difference on these datasets is not very significant. This may be due to the sparseness of these graphs, which makes it easier to compress the transitive closure by using IL or PWAH. On these graphs, the index size of PLL and that of PPL are exactly the same since the length of the first path in PPL is no longer than ten and PPL does not use paths at all. IL and PWAH perform better than PLL and PPL on ff/firefox. On the other hand, PLL and PPL outperform IL and PWAH on citeseerx and cit-patents. The index size of PLL and PPL is about 3% of IL and 12% of PWAH on cit-patents. We can say that PLL and PPL are robust in the sense that it only takes moderate space, less than 1GB, on all graphs in the experiments. As for the difference between PLL and PPL, PLL is slightly more space-efficient than PPL in most cases since we need two integers to represent each element in a label PPL whereas we only need one integers in PLL. However, the result on cit-patents shows that PLL has a potential to represent reachability in a more efficient way than PPL in some cases.

Table 6.3: Index size (MB)

Dataset	PLL (This work)	PPL (This work)	GRAIL [YCZ12]	IL [Nuu95]	PWAH [vSdM11]
ff/successors	122.3	91.6	29.7	40.0	34.1
citeseerx	122.0	126.7	104.6	441.3	156.0
cit-patents	664.6	691.2	60.4	22444.5	5593.1
go-uniprot	263.1	273.5	111.5	792.7	255.9
uniprot22m	19.4	19.4	25.5	19.6	19.5
uniprot100m	206.8	206.8	257.4	223.0	218.8
uniprot150m	334.0	334.0	400.6	373.8	366.2

Then we look at Table 6.4, which shows indexing time on real-world networks. GRAIL constantly shows great performance in indexing time since the number of elements in labels is linear in the number of vertices. Still, indexing time of PLL and PPL is acceptable, while they are relatively slow. They are even faster than IL and PWAH on cit-patents. This suggests that PLL and PPL work well on large and mildly dense graphs. IL performs quite well except on cit-patents, and PWAH needs approximately 1.5 to 2.5 times longer time than IL.

Table 6.4: Indexing time (sec)

Dataset	PLL (This work)	PPL (This work)	GRAIL [YCZ12]	IL [Nuu95]	PWAH [vSdM11]
ff/successors	10.46	8.19	1.08	7.84	5.02
citeseerx	23.13	45.42	7.65	6.70	16.03
cit-patents	192.05	239.95	8.24	397.04	847.83
go-uniprot	26.60	29.74	5.78	18.33	31.10
uniprot22m	2.82	3.02	0.96	0.96	1.23
uniprot100m	30.80	32.99	12.39	10.64	14.39
uniprot150m	49.48	53.56	20.52	17.21	24.16

6.4.3 Performance on Synthetic Graphs

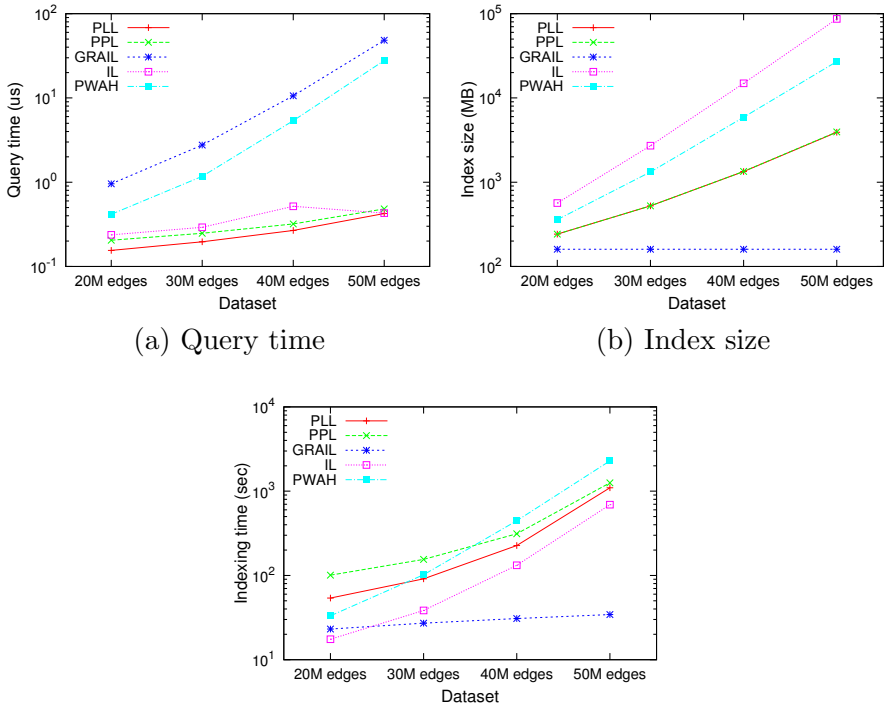
Second, we compared PLL and PPL with existing methods on synthetic graphs. Query time, index size, and indexing time on synthetic graphs are shown in

Figure 6.2. These synthetic graphs have ten million vertices and number of edges ranges from twenty million to fifty million. Note that these graphs are drawn with logarithmic-scale y-axis.

Figure 6.2a shows that PLL and PPL achieve very fast query time. The query time of PLL, PPL and IL increase very slowly as the number of edges becomes larger, within a microsecond even on the graph with 50 million edges. On the other hand, the query time of GRAIL and PWAH grows fast and exceeds 10 microseconds on that graph.

In Figure 6.2b, the index size of IL and PWAH become larger drastically as the graph becomes dense. The index size of PLL and PPL grow relatively slowly, and that of GRAIL does not change by the number of edges.

Figure 6.2c shows that GRAIL outperforms other methods in indexing time, especially on relatively dense graphs. PLL and PPL are relatively slow on very sparse graphs. However, these two methods overtake IL and PWAH as the graph becomes dense.



(a) Query time

(b) Index size

(c) Indexing time

Figure 6.2: Performance comparison of reachability queries on synthetic graphs.

As a whole, we can say that PLL and PPL outperform other methods on relatively dense graphs, achieving very fast query time and moderate index size. The index size of IL and PWAH becomes very large on dense graphs, and the query time of GRAIL and PWAH becomes very slow on these graphs. These experimental results show that PLL and PPL has a potential to handle real-world networks larger than those we used in the experiments.

6.4.4 Comparison of Vertex Ordering Strategies

Finally, we compare the performance of vertex selection and path selection strategies. We only show index size since query time and indexing time are almost

proportional to index size. The results are shown in Tables 6.5 and 6.6. *DNF* in Table 6.5 means that indexing did not finish in 20 minutes.

Table 6.5 shows that InOut performed the best among three vertex selection strategies. Random did not finish on any graph, while others finished indexing within 20 minutes on all graphs. This shows that vertex ordering strategies significantly influence the performance, as expected. InOut outperformed Degree on cit-patents and citeseerx. Especially, the index size of InOut is half as large as that of Degree on cit-patents. On the other graphs, these two strategies demonstrated almost the same performance.

Table 6.5: Comparison of index size of PLL using different vertex ordering strategies (MB)

Dataset	Random	Degree	InOut
ff/successors	DNF	123.4	122.3
citeseerx	DNF	168.0	122.0
cit-patents	DNF	1555.0	664.6
go-uniprot	DNF	270.3	263.1
uniprot22m	DNF	19.4	19.4
uniprot100m	DNF	206.8	206.8
uniprot150m	DNF	334.0	334.0

Then we compared the performance of path selection strategies (Table 6.6). DPInOut is quite stable and it outperforms the other strategies, though Longest performed slightly better than DPInOut in some cases. Again, the index size is the same in all strategies on uniprot22m, uniprot100m, and uniprot150m since we cannot find any path of length at least ten. DPDeg is inferior to other two strategies on citeseerx and cit-patents. The result shows that path selection strategies may drastically affect the performance of PPL.

Table 6.6: Comparison of index size of PPL using different path selection strategies (MB)

Dataset	Longest	DPDeg	DPInOut
ff/successors	91.0	92.7	91.6
citeseerx	126.1	166.5	126.7
cit-patents	678.1	1536.6	691.2
go-uniprot	428.0	279.6	273.5
uniprot22m	19.4	19.4	19.4
uniprot100m	206.8	206.8	206.8
uniprot150m	384.1	334.0	334.0

Chapter 7

Highway-based Labeling for Road Networks

Next, we deal with shortest-path and distance queries on road networks. As we explained in Chapter 1, structural properties of road networks are quite different from those of complex networks. Moreover, as road networks are weighted graphs, the bit-parallel labeling technique cannot be used. Therefore, we propose another method named *pruned highway labeling*, which is tailored to road networks.

We first propose another indexing framework (i.e., index data structure and query algorithm) named *highway-based labeling framework* (Section 7.1), which is another extension of the 2-hop cover framework. Then, the overview of our pruned highway labeling algorithm is presented (Section 7.2). Next, we present the detailed description of the algorithm (Section 7.3). Finally, results of our experimental evaluation is explained (Section 7.4).

7.1 Highway-based Labeling Framework

In this section, we propose a new labeling framework (i.e., data structure and query algorithm) referred to as the *highway-based labeling framework*. We first introduce the notion of a *highway decomposition* and explain what we store for labels (Section 7.1.1). This concept has some similarity to the decomposition used in Thorup [Tho04]. We then present our query algorithm (Section 7.1.2).

7.1.1 Highway Decomposition and Index Data Structure

First, we define a *highway decomposition*, which is a key for our proposed framework. In what follows, we identify a path with an ordered set of vertices.

Definition 7.1 (highway decomposition)**.** *A* highway decomposition *of a given graph G is a family of ordered sets of vertices $\mathscr{P} = \{P_1, P_2, ..., P_N\}$ such that,*

1. *$P_i = (p_{i,1}, p_{i,2}, \ldots, p_{i,l_i})$ is a shortest path between two vertices $p_{i,1}$ and p_{i,l_i},*
2. *$P_i \cap P_j = \emptyset$ for any i and j $(i \neq j)$, and*
3. *$P_1 \cup P_2 \cup \ldots \cup P_N = V$.*

In Section 7.3 we will describe how to compute a highway decomposition of a graph. In the following, we assume that we are given such a decomposition. In the highway-based labeling framework, a label $L(v)$ for a vertex v is a set of triples $(i, d(p_{i,1}, p_{i,j}), d(v, p_{i,j}))$, that is, for each vertex v, we store an index i of

a path P_i, distance from the starting point $p_{i,1}$ of the path to a vertex $p_{i,j}$ on the path, and distance from the vertex v to the vertex $p_{i,j}$[1].

In the hub-based labeling framework (i.e., the 2-hop cover framework), a candidate for the distance between s and t are computed from pairs $(v, d(s,v)) \in L(s)$ and $(v, d(t,v)) \in L(t)$ for the same vertex v. On the other hand, in the highway-based labeling framework, we can compute a candidate for the distance between s and t even for the case that the vertices stored in two triples are different. Indeed, suppose that the label $L(s)$ contains a triple $(i, d(p_{i,1}, p_{i,j}), d(s, p_{i,j}))$ and the label $L(t)$ contains a triple $(i, d(p_{i,1}, p_{i,k}), d(t, p_{i,k}))$. Then we can obtain a candidate distance by looking at $d(s, p_{i,j}) + d(p_{i,j}, p_{i,k}) + d(p_{i,k}, t) = d(s, p_{i,j}) + |d(p_{i,1}, p_{i,j}) - d(p_{i,1}, p_{i,k})| + d(p_{i,k}, t)$. This observation is crucial in our experiment. It allows us to make the label for each vertex to be smaller size. This is a clear advantage of the highway-based labeling framework. In particular, a large part of distances between pairs of far vertices can be answered by storing a few distances to central paths in real road networks (e.g. highways).

7.1.2 Query Algorithm

For an s-t query, we search for the triples that minimize the candidate distance. That is, we define the answer to an s-t query using labels L as

$$\begin{aligned}&\textsc{Query}(s,t,L)\\ &= \min\{d(s,p_{i,j}) + d(p_{i,j},p_{i,k}) + d(p_{i,k},t) \mid \\ &\qquad (i, d(p_{i,1},p_{i,j}), d(s,p_{i,j})) \in L(s), (i, d(p_{i,1},p_{i,k}), d(t,p_{i,k})) \in L(t)\}.\end{aligned}$$

The distance $d(p_{i,j}, p_{i,k})$ itself is not contained in labels $L(s)$ and $L(t)$, but can be computed by using $d(p_{i,1}, p_{i,j})$ and $d(p_{i,1}, p_{i,k})$ as mentioned above.

Naively computing the function above takes $\Theta(|L(s)|\,|L(t)|)$ time, but we can obtain a linear-time algorithm based on the following lemma.

Lemma 7.1. *There exist triples $(i, d(p_{i,1}, p_{i,j}), d(s, p_{i,j})) \in L(s)$ and $(i, d(p_{i,1}, p_{i,k}), d(t, p_{i,k})) \in L(t)$ that achieve the minimum candidate distance and satisfy the following: for any vertex $p_{i,l}$ with $\min(j,k) < l < \max(j,k)$, $(i, d(p_{i,1}, p_{i,l}), d(s, p_{i,l})) \notin L(s)$ and $(i, d(p_{i,1}, p_{i,l}), d(t, p_{i,l})) \notin L(t)$.*

Proof. Let $(i, d(p_{i,1}, p_{i,j}), d(s, p_{i,j})) \in L(s)$ and $(i, d(p_{i,1}, p_{i,k}), d(t, p_{i,k})) \in L(t)$ be triples that minimize the candidate distance. Let us choose j, k with $j < k$ such that $k - j$ is as small as possible. Suppose there exists $p_{i,l}$ with $j < l < k$ such that $(i, d(p_{i,1}, p_{i,l}), d(s, p_{i,l})) \in L(s)$. Then the candidate distance computed from triples $(i, d(p_{i,1}, p_{i,l}), d(s, p_{i,l}))$ and $(i, d(p_{i,1}, p_{i,k}), d(t, p_{i,k}))$ must also be the minimum, because

$$\begin{aligned}& d(s,p_{i,l}) + d(p_{i,l},p_{i,k}) + d(p_{i,k},t)\\ =\ & d(s,p_{i,l}) + d(p_{i,1},p_{i,k}) - d(p_{i,1},p_{i,l}) + d(p_{i,k},t)\\ =\ & (d(s,p_{i,j}) + d(p_{i,j},p_{i,l})) + d(p_{i,1},p_{i,k})\\ & -(d(p_{i,1},p_{i,j}) + d(p_{i,j},p_{i,l})) + d(p_{i,k},t)\\ =\ & d(s,p_{i,j}) + d(p_{i,1},p_{i,k}) - d(p_{i,1},p_{i,j}) + d(p_{i,k},t)\\ =\ & d(s,p_{i,j}) + d(p_{i,j},p_{i,k}) + d(p_{i,k},t).\end{aligned}$$

Note that P_i is a shortest path. This is a contradiction. □

[1]The label $L(v)$ does not necessarily contain triples from all indexes i nor contains triples for all the vertices on a path P_i, because in our preprocessing, we will reduce the total size of labels $L(v)$.

Therefore, like previous labeling methods, by sorting triples in labels with indexes and the distances from the starting point of the path in ascending order beforehand, we can answer an s-t query in $O(|L(s)| + |L(t)|)$ time using a merge-sort-like algorithm.

7.2 Pruned Highway Labeling

In this section we propose our label computation algorithm for highway-based labelings. Throughout this section, we assume that a highway decomposition $\mathscr{P}$ is given. In Section 7.3, we will explain how to obtain such a decomposition.

7.2.1 Naive Highway Labeling

Before presenting our efficient preprocessing algorithm, we first give a naive algorithm to compute correct labels for the highway-based labeling framework. We start with empty labels L_0 (i.e., $L_0(v) = \emptyset$ for each vertex v) and then construct new labels L_{i+1} from L_i iteratively. In order to construct the labels L_i, we first copy the labels L_{i-1} to L_i, and then conduct the Dijkstra search from each vertex $p_{i,j}$ on the path P_i and add the distance between vertices v and $p_{i,j}$ to the label $L_i(v)$. That is, $L_i(v) = L_{i-1}(v) \cup (i, d(p_{i,1}, p_{i,j}), d(v, p_{i,j}))$ for all $p_{i,j} \in P_i$. After we conduct all Dijkstra searches, we obtain the labels $L = L_N$. Because $L(v)$ contains the distance from v to all vertices, the labels can answer correct distances between v and any other vertex. Therefore we have the following.

Lemma 7.2. *For any pair of vertices s and t,* $\textsc{Query}(s, t, L) = d(s, t)$.

7.2.2 Pruned Highway Labeling

We now explain our efficient algorithm for computing labels for the highway-based labeling framework named *pruned highway labeling*. This algorithm is based on the pruned landmark labeling.

Similarly to the naive algorithm, we start with empty labels L'_0 and then construct new labels L'_{i+1} from L'_i iteratively. In order to construct the labels L'_i, we conduct the *pruned* Dijkstra search from all the vertices on the path P_i simultaneously as described in Algorithm 7.1. When we visit a vertex v from the vertex $p_{i,j}$ with distance δ, if $\textsc{Query}(v, p_{i,j}, L'_i)$ is less than or equal to δ, we prune the search. Otherwise, we add the triple $(i, d(p_{i,1}, p_{i,j}), \delta)$ to $L'_i(v)$ and check edges from the vertex v. After we conduct the pruned Dijkstra searches from all the paths, we obtain the labels $L' = L'_N$ for the highway-based labeling framework. We prove the correctness of our algorithm in Section 7.2.4, but in the next subsection, we shall give intuition how our algorithm goes.

7.2.3 Example For Pruned Highway Labeling

Figure 7.1 illustrates examples for the pruned highway labeling. For simplicity, we assume that the length of all edges is 1.

First, we decompose the input graph. Suppose that we are given a highway decomposition $\mathscr{P} = \{P_1, P_2, P_3\}$, where $P_1 = \{0, 1, 2, 3, 4\}, P_2 = \{5, 6\}, P_3 = \{7, 8\}$. Then we conduct the pruned Dijkstra search from the path P_1 (Figure 1a). A triple (1, 2, 1) is added to the label $L'_1(5)$. However, a triple (1, 3, 2) is not added to the label $L'_1(5)$ because $\textsc{Query}(5, 3, L'_1) = d(5, 2) + d(2, 3) + d(3, 3) = 1 + 1 + 0 = 2$. Therefore, the search from the vertex 3 is pruned. In the same way, only one triple (1, 3, 1) is added to the label $L'_1(6)$. On the other hand,

Algorithm 7.1 Pruned Dijkstra search from P_i to compute the labels L'_i

procedure PRUNEDDIJKSTRASEARCH(G, P_i, L'_{i-1})
  $Q \leftarrow$ an empty priority queue
  Push $(0, p_{i,j}, p_{i,j})$ onto Q for all $p_{i,j} \in P_i$
  $L'_i(v) \leftarrow L'_{i-1}(v)$ for all $v \in V$
  while Q is not empty **do**
    Pop $(\delta, v, p_{i,j})$ from Q
    if QUERY$(v, p_{i,j}, L'_i) \leq \delta$ **then**
      continue
    $L'_i(v) \leftarrow L'_i(v) \cup (i, d(p_{i,1}, p_{i,j}), \delta)$
    Push $(\delta + w(v, u), u, p_{i,j})$ onto Q for all $(v, u) \in E$
  return L'_i

Algorithm 7.2 Preprocessing by the pruned highway labeling

procedure PREPROCESS(G)
  $L'_0(v) \leftarrow \emptyset$ for all $v \in V$
  $\mathscr{P} \leftarrow$ a highway decomposition of G
  $N \leftarrow$ the size of $\mathscr{P}$
  for $i = 1$ to N **do**
    $L'_i \leftarrow$ PRUNEDDIJKSTRASEARCH(G, P_i, L'_{i-1})
  return L'_N

two triples (1, 0, 1) and (1, 1, 1) are added to the label $L'_1(7)$ because we cannot prune the search.

Next, we conduct the pruned Dijkstra search from the path P_2 (Figure 1b). Because the search is pruned, no triples are added to labels $L'_2(2)$ nor $L'_2(3)$ and no other vertices are visited. Similarly, we conduct the pruned Dijkstra search from the path P_3 (Figure 1c).

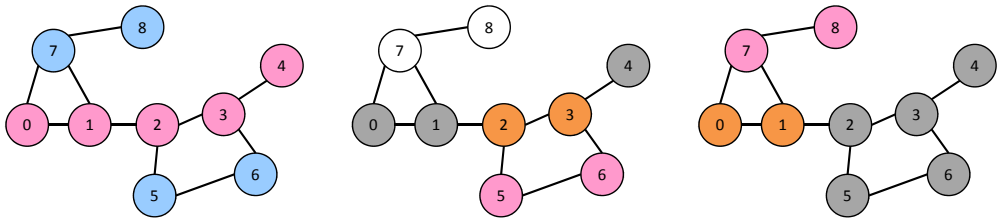
For a query between vertices 6 and 7, we check the two labels $L'(6)$ and $L'(7)$. The triple (1, 3, 1) is contained in $L'(6)$ and the triple (1, 0, 1) is contained in $L'(7)$, so the distance computed by these triples is $d(6,3) + d(3,0) + d(0,7) = 1 + 3 + 1 = 5$. $L'(7)$ also contains the triple (1, 1, 1). We can indeed get 4 as the distance by using this triple. Although both labels $L'(6)$ and $L'(7)$ contain other triples, these triples do not coincide to the index of the path. Therefore, we do not need to check these triples. As a result, QUERY$(6, 7, L') = 4$.

7.2.4 Proof of Correctness

In this subsection, we prove the correctness of the pruning. Specifically, we prove that the distance computed by using labels L' from the pruned highway labeling is equal to the distance computed by using labels L from the naive highway labeling.

Theorem 7.1. *For any pair of vertices s and t,* QUERY(s, t, L') = QUERY(s, t, L).

Proof. For vertices s and t, let i be the index such that QUERY$(s, t, L_{i'}) \neq d(s, t)$ for any $i' < i$ and QUERY$(s, t, L_i) = d(s, t)$. Then there exist vertices $p_{i,j}, p_{i,k} \in P_i$ that satisfy $d(s, t) = d(s, p_{i,j}) + d(p_{i,j}, p_{i,k}) + d(p_{i,k}, t)$. Among others, we choose a pair (j, k) such that no vertices on the shortest path between s and $p_{i,j}$ or between $p_{i,k}$ and t are on the path P_i. Suppose that for some $i' < i$, a vertex $p_{i',j'}$



(a) Pruned Dijkstra search from the path P_1 (b) Pruned Dijkstra search from the path P_2 (c) Pruned Dijkstra search from the path P_3

Figure 7.1: Examples for the pruned highway labeling. Pink vertices are on the starting path P_i, blue vertices are visited and added to some labels, gray vertices are already used as the starting points of the previous searches, orange vertices are visited but pruned, and white vertices are not visited.

on the path $P_{i'}$ is on a shortest path between s and $p_{i,j}$. Then the label $L(s)$ contains the triple $(i', d(p_{i',1}, p_{i',j'}), d(s, p_{i',j'}))$ and the label $L(t)$ contains the triple $(i', d(p_{i',1}, p_{i',j'}), d(t, p_{i',j'}))$, and therefore $\textsc{Query}(s, t, L_{i'}) = d(s, t)$ holds, which is a contradiction to the choice of i. Therefore any path $P_{i'}$ with $i' < i$ contains no vertices on the shortest paths between s and $p_{i,j}$. Thus the search from $p_{i,j}$ to s is not pruned and the triple $(i, d(p_{i,1}, p_{i,j}), d(s, p_{i,j}))$ is added to the label $L'(s)$. In the same way, the triple $(i, d(p_{i,1}, p_{i,k}), d(t, p_{i,k}))$ is added to the label $L'(t)$. As a result, $\textsc{Query}(s, t, L') = \textsc{Query}(s, t, L)$ holds. □

From this theorem, we can answer a query for any pair of vertices correctly by using labels computed by the pruned highway labeling.

Corollary 7.1. *Let L' be the labels computed by the pruned highway labeling. Then for any pair of vertices s and t,*

$$\textsc{Query}(s, t, L') = d(s, t).$$

7.3 Detailed Algorithm Description

7.3.1 Heuristic Highway Decomposition

Until this point, we assumed that a highway decomposition is already given, and then we conduct the first pruned Dijkstra search. We now show how to construct a highway decomposition and labeling simultaneously. Specifically, we choose a path P_i from remaining vertices just before conducting the i-th pruned Dijkstra search. In the pruned landmark labeling algorithm, the order of vertices to start BFSs is crucial to achieve small labels. Similarly, in our method, the order of paths P_i in a highway decomposition is important, and this allows us to achieve small label sizes.

At a high level, we want to choose a path that *hits* many shortest paths at the early stage of Dijkstra search, because this would allow us to prune future Dijkstra searches. In what follows, we propose strategies for selecting such a good path.

First, we focus on the speed of an edge (i.e., the geometrical length divided by the travel time). In the real life, we tend to use fast highways when we travel long distance. Therefore, we may assume that the vertices connected to fast edges tend to be passed by many shortest paths. For this reason, we group vertices into

several levels according to the speed of their connected edges. We assign vertices connected to faster edges to higher levels and then we choose a path consisting of the highest level vertices. When the number of vertices in the highest level is fewer than some threshold, we mix unused vertices in the highest level with vertices in the second highest level to make it possible to select a path of enough length.

We now describe how to choose a path from the highest level vertices. Because the selected path must be a shortest path between two vertices, we first compute the shortest path tree from a randomly selected root vertex, and then pick a path between the root vertex and another vertex in the shortest path tree. The more descendants a vertex has in the shortest path tree, the more shortest paths on the tree hit the vertex. Therefore, in order to select a path that hits many shortest paths, we choose the path by starting from the root vertex and iteratively pick a child with the largest number of descendants.

Finally, we describe our technique to skip unimportant vertices. Suppose that a vertex v on the selected path hits many shortest paths, but most of them also contain another vertex w on the selected path. In this case, even if we skip the vertex v while keeping the vertex w, most of the future searches that will be pruned by v are still pruned by w. Thus we skip such unimportant vertices from the selected path. Suppose that we chose a vertex v on the shortest path tree, and then choose its child w. If the difference between the number of descendants of v and w is small, most of the shortest paths on the tree that contain v also contain w, and therefore we skip v. This does not affect the correctness of the algorithm because this operation corresponds to add shortcut edges across skipped vertices.

In our implementation used in the experiments, we group vertices into four levels and we skip a vertex v when the difference between the number of descendants of vertices v and its child w is smaller than five percents.

7.3.2 Storing Labels

To make highway-based labelings more practical, we describe the efficient way to store labels in this subsection. The label $L(v)$ is a set of triples $(i, d(p_{i,1}, p_{i,j}), d(v, p_{i,j}))$, where i is an index of a path. The distance information in the label is used in the query only when the indexes of the paths are same in two triples. Therefore, storing the index information and the distance information separately makes the query time faster because we can avoid cache misses. For a single path, multiple triples may be stored to a vertex. Therefore, by storing pairs of an index and the number of triples for the index, we can reduce the space usage. Moreover, this also makes the query time faster because we can reduce unnecessary comparisons when the indexes do not match in two triples. For more efficient implementation, we use pointer arithmetic and align arrays storing labels to cache line.

7.3.3 Contraction Technique

Additionally, to further improve the performance, we introduce a new technique called the *contraction technique*. First, we consider a vertex v of degree one. Any shortest path from v to another vertex passes through its adjacent vertex x. Moreover, the vertex v is never contained in shortest paths between other vertices. Therefore, we can remove the vertex v from the graph in the preprocessing. We can correctly answer the query between the vertex v and another vertex u by referring the label $L(x)$ and adding the length $w(v, x)$, that is, $\textsc{Query}(v, u, L) =$

$\textsc{Query}(x, u, L) + w(v, x)$.

Next, we consider a vertex v of degree larger than one. In this case, the vertex v may be contained in a shortest path between other vertices. Therefore, we need to add some shortcut edges before removing the vertex v to preserve the distances. We can correctly answer the distance between v and another vertex u by evaluating the distances from all the neighbors of v to u, that is, $\textsc{Query}(v, u, L) = \min_{(v,x)\in E(G)}\{\textsc{Query}(x, u, L) + w(v, x)\}$. However, the larger the degree limit is, the slower query time becomes due to the number of neighbors to check. In our experiments, we only set the degree limit to be at most three.

7.4 Experimental Evaluation

7.4.1 Setup

Environment

We conducted experiments on a Linux server with Intel Xeon X5675 processor (3.06 GHz) and 288 GB for main memory. We implemented the proposed method in C++ and compiled it with the GNU C++ compiler 4.4.6 using optimization level 3. We did not parallelize the preprocessing and queries and used one core. We evaluate query time as the average time for 1,000,000 random queries.

Datasets

We used two popular graph instances from 9th DIMACS Implementation Challenge [DGJ09]. One is a road network of USA that has about 24 million vertices and 58 million edges. The other is a road network of Western Europe by PTV AG with about 18 million vertices and 42 million edges. In both instances, we use travel times as the length of edges and treat a graph as an undirected graph.

7.4.2 Performance Comparison

First, we compare the proposed method with several previous methods with regard to preprocessing time, space usage and query time (Table 7.1). The proposed method is set to use the contraction technique for vertices of degree one. We compare the proposed method with the following five previous methods: *contraction hierarchies* (CH) [GSSD08], *transit node routing* (TNR) [BFM$^+$07], combination of TNR and *arc flags* (TNR+AF) [BDS$^+$10], *hub-based labeling* (HL) [ADGW11, ADGW12], and *hub label compression* (HLC) [DGW13]. For HL, we used four variants: HL local [ADGW11], HL global [ADGW11], HL-15 local [ADGW12] and HL-∞ global [ADGW12]. All these methods are also implemented in C++. CH, TNR and TNR+AF were evaluated on a machine with an AMD Opteron 270 processor (2.0 GHz) [BDS$^+$10], and HL and HLC were evaluated on a machine with two Intel Xeon X5680 processors (3.33 GHz) [ADGW11, ADGW12, DGW13]. The preprocessing is parallelized for only HL but not parallelized for other methods including our method. Queries are not parallelized for all methods.

Preprocessing Time

We would like to emphasize our big improvement in preprocessing time. The preprocessing time for the proposed method is much faster than previous labeling methods. Although it seems that HL-15 local is faster than our method, let us

Table 7.1: Comparison of the performance between pruned highway labeling and previous methods. HL is parallelized to use 12 cores in preprocessing and all other methods are not parallelized.

	USA			Europe		
Method	Prep. [h:m]	Space [GB]	Query [ns]	Prep. [h:m]	Space [GB]	Query [ns]
PHL-1 (This work)	0:29	16.4	941	0:34	14.9	1039
CH [BDS+10]	0:27	0.5	130000	0:25	0.4	180000
TNR [BDS+10]	1:30	5.4	3000	1:52	3.7	3400
TNR+AF [BDS+10]	2:37	6.3	1700	3:49	5.7	1900
HL local [ADGW11]	2:24	22.7	627	2:39	20.1	572
HL global [ADGW11]	2:35	25.4	266	2:45	21.3	276
HL-15 local [ADGW12]	-	-	-	0:05	18.8	556
HL-∞ global [ADGW12]	-	-	-	6:12	17.7	254
HLC-15 [DGW13]	0:53	2.9	2486	0:50	1.8	2554

observe that its preprocessing is parallelized to use 12 cores. This improvement shows the efficiency of pruned labeling.

Space Usage

The space consumption for our method is also smaller than HL. While this may be still not very compact, in the situations when space consumption is severe, we can greatly reduce the space usage with help of the contraction technique, at the cost of a little slower query time, as discussed in Section 7.4.3. Therefore, it would be not a big problem.

Query Time

With regard to query time, although our method is a bit slower than HL, it is still sufficiently fast and around 1 μs on average. This shows the efficiency of the highway-based labeling as a labeling framework.

7.4.3 Analysis

Contraction Technique

Table 7.2 lists the performance of our method with different contraction level. The contraction level denotes the limit of the degree of vertices to remove. That is, if the level is 2, we remove vertices of degree no larger than 2. Setting the contraction level zero means we do not use the contraction technique at all.

Even without the contraction technique, our preprocessing is much faster than previous labeling methods. However, by applying the contraction technique, preprocessing time becomes even faster. In particular, it takes only about ten minutes with contraction level three. This is because the input graph size is fairly reduced by this contraction. Moreover, the space usage gets smaller than five gigabytes with contraction level three, while the query time is not very slower. Although the higher the level is, the slower the query time becomes, it is sufficiently fast. Therefore, the contraction technique is highly useful.

Pruning

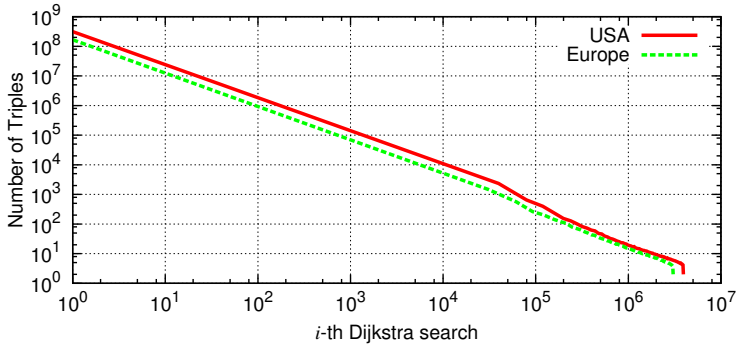
Figure 7.2a illustrates the number of triples added to labels in each pruned Dijkstra search and Figure 7.2b illustrates the cumulative distribution of it. We

Table 7.2: Comparison of the performance by the contraction technique. The contraction level indicates the degree of removed vertices.

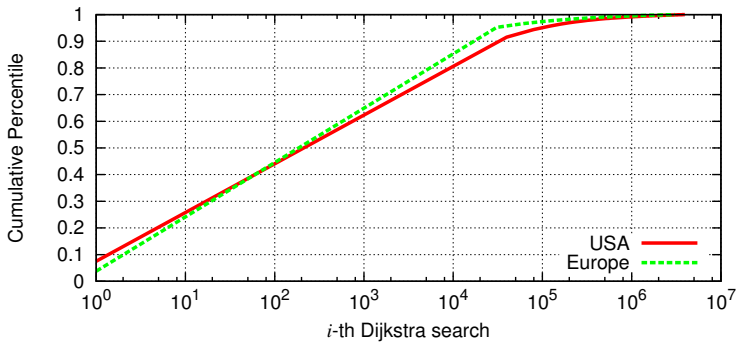
	USA			Europe		
Contraction level	Preprocessing [h:m]	Space [GB]	Query [ns]	Preprocessing [h:m]	Space [GB]	Query [ns]
0	0:38	19.8	906	0:50	20.2	1080
1	0:29	16.4	941	0:34	14.9	1039
2	0:11	6.4	1793	0:22	8.5	2011
3	0:07	4.1	2970	0:11	4.6	3344

can confirm big effect of pruning from these figures. From Figure 7.2a, we observe that the number of triples added to labels in each pruned Dijkstra search decreases dramatically. In consequence, as Figure 7.2b shows, most triples are added at the beginning.

Figure 7.3a reports the average number of common paths in labels of two vertices against Dijkstra rank of them. We used labels constructed without the contraction technique and computed the average on 10,000 pairs of vertices for each rank. We observe that the number of common paths decreases gradually as the two vertices get far. This indicates that our preprocessing algorithm successfully separates input graphs by shortest paths.

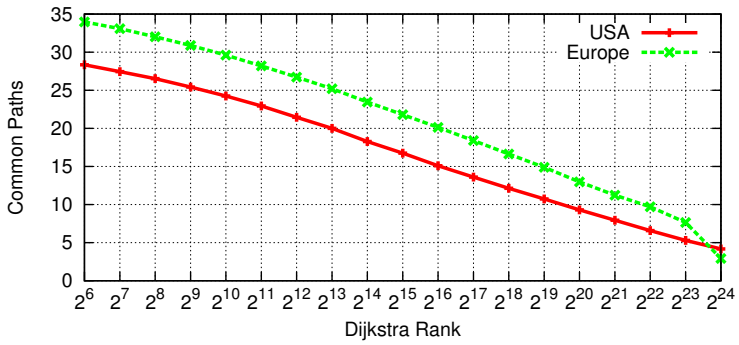


(a) Number of triples added in each pruned Dijkstra search.

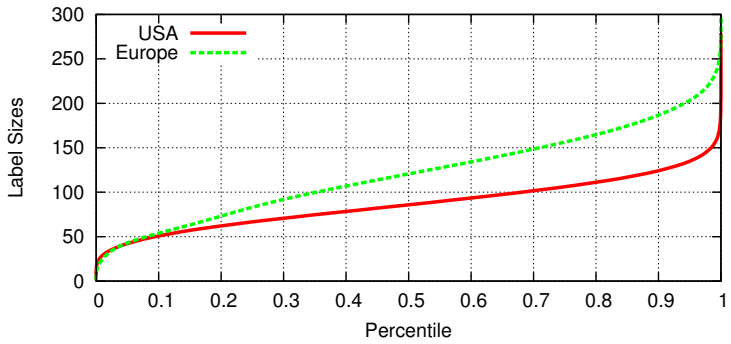


(b) Cumulative distribution of the number of triples added in each pruned Dijkstra search.

Figure 7.2: Effect of pruning.



(a) Number of common paths in labels against Dijkstra Rank



(b) Distribution of the sizes of labels

Figure 7.3: Label properties.

Sizes of Labels

Figure 7.3b shows the distribution of the sizes of labels. We constructed labels without the contraction technique. We can confirm that the size of a label for each vertex does not vary much for different vertices, and few vertices have much larger labels than the average. This indicates that the query time for our method is quite stable.

Chapter 8

Historical Labeling for Evolving Complex Networks

When analyzing historical networks, for which timestamps of vertices and edges are also available, in addition to the latest snapshot, the shortest paths and distances on previous snapshots or transition of them by time are also of interest. In this chapter, we call such queries about previous snapshots *historical queries*. We study indexing methods for such kinds of historical queries (Figure 8.1). In particular, we deal with the following two kinds of queries: a *snapshot query* asks the shortest path or distance on a specified previous snapshot, and a *change-point query* asks all the moments when the distance between two vertices has changed.

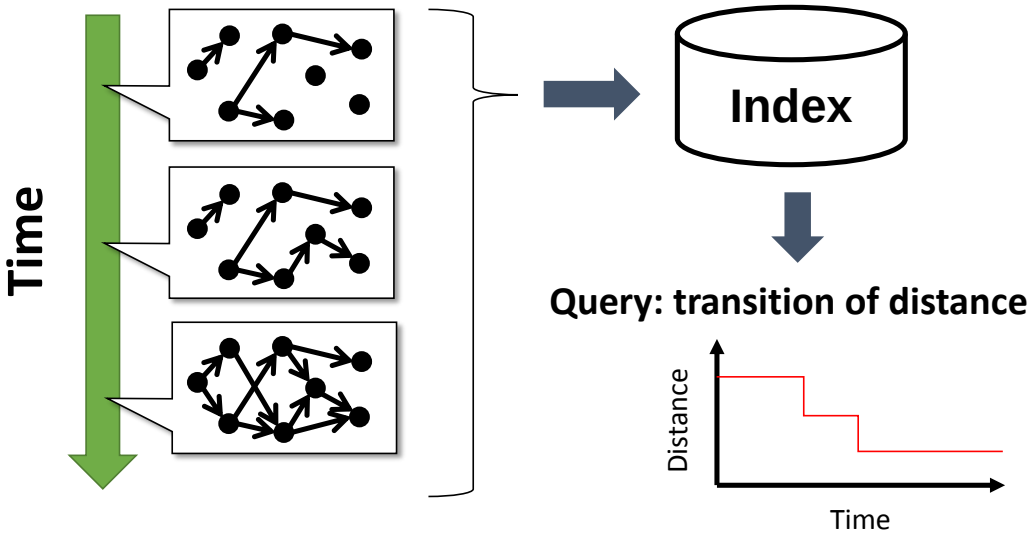


Figure 8.1: An illustration of indexing methods for historical queries.

We first design an indexing method based on pruned landmark labeling for these kinds of queries (Section 8.1). Second, we present the result of experimental evaluation (Section 8.2). Finally, the usefulness of these new kinds of historical queries is demonstrated via application to evolving network analysis (Section 8.3).

Formal Problem Definition

In this chapter, we study indexing methods that, given a time series of a evolving graph (i.e., a graph with edge time stamps), construct an index to quickly answer the following queries.

Problem 8.1 (Historical Snapshot Distance Query)**.**
Given*: Two query vertices s, t and time τ.*

Answer: *Distance* $d_\tau(s,t)$.

Problem 8.2 (Historical Distance Change-point Query)**.**
Given: *Two query vertices* s,t.
Answer: *Set* $C(s,t) = \{(\tau_1,\delta_1),(\tau_2,\delta_2),\ldots\}$ *where* $(\tau_i,\delta_i) \in C(s,t)$ *if and only if* $\delta_i = d_{\tau_i}(s,t) \neq d_{\tau_i-1}(s,t)$.

8.1 Historical Pruned Landmark Labeling

In this section, we propose a new indexing scheme referred to as *historical pruned landmark labeling* to efficiently process historical queries defined above. Unlike our first method for contemporary queries, as there is no previous work on these queries, we start from designing a new index framework (i.e., data structure and query algorithms), named *historical 2-hop cover*, in Section 8.1.1. Then, we propose an offline indexing algorithm that constructs an index from a stored historical graph in Section 8.1.2. Since it is more involved than that for standard contemporary queries, we explain our indexing algorithm with three steps: we start from an algorithm based on dynamic programming, next we turn the algorithm into a BFS-like algorithm, and then introduce pruning to the algorithm. Finally, we present an online incremental update algorithm for online graph changes in Section 8.1.3.

8.1.1 Historical 2-Hop Cover Framework

First, we propose a new indexing framework (i.e., data structure and query algorithms) referred to as the *historical 2-hop cover framework*. Since there is no previous work on these queries, it is the first framework for historical distance queries. The main technical challenge here is to design (almost) linear-time query algorithms for both kinds of historical queries.

Data Structure

For each vertex v, we store a label $L(v)$. Label $L(v)$ is a set of triples (u,τ,δ_{uv}), where u is a vertex, τ describes time, and $\delta_{uv} = d_\tau(u,v)$. Due to Lemma 4.2, $(u,\tau,\delta_{uv}) \in L(v)$ also indicates $d_{\tau'}(u,v) \leq \delta_{uv}$ for $\tau' \geq \tau$.

As with the normal 2-hop cover framework, we store triples in a label in the ascending order of the IDs of destination vertices. In addition, we sort triples that share the same destination vertex in ascending order of distance (i.e. descending order of time).

Answering Snapshot Queries

A snapshot query between a pair of vertices s and t at time τ can be answered in $O(|L(s)|+|L(t)|)$ time. Though we basically conduct a merge-sort-like algorithm as with normal 2-hop cover, there are several differences. First, we need to ignore label entries with time later than τ. In addition, to handle triples in a label that share the same destination vertex, among them we only see the newest label entry with time earlier than or equal to τ. That is, if (u_i,τ_i,δ_i) and $(u_{i+1},\tau_{i+1},\delta_{i+1})$ are consecutive labels in $L(s)$ where $u_i = u_{i+1}$ and $\tau \geq \tau_i > \tau_{i+1}$, then we ignore the second label since $\delta_i \leq \delta_{i+1}$ from Lemma 4.2.

Answering Change-point Queries

Answering a change-point query between vertices s and t is a little more involved, but can be done in $O(l \log l)$ time, where $l = |L(s)| + |L(t)|$. First, we conduct a merge-sort-like algorithm to enumerate candidates of distance change-points. From pairs of triples $(u, \tau_s, \delta_s) \in L(s)$ and $(u, \tau_t, \delta_t) \in L(t)$, we enumerate pairs $(\tau, \delta) = (\max\{\tau_s, \tau_t\}, \delta_s + \delta_t)$, which indicates $d_\tau(s, t) \leq \delta$. Then, we sort these pairs by time τ. Finally, we remove unnecessary pairs. That is, if (τ_i, δ_i) and $(\tau_{i+1}, \delta_{i+1})$ are consecutive pairs, where $\tau_i \leq \tau_{i+1}$ and $\delta_i \leq \delta_{i+1}$, then we remove the second pair.

Again, the remaining issue is to handle triples in a label that share the same destination vertex. If we check every pair of these triples, in the worst case, it would take quadratic time. However, for these triples, we can also apply a merge-sort-like scan algorithm by considering time of these triples. The algorithm is described in Algorithm 8.1, where $v(t) = v, \tau(t) = \tau$ and $\delta(t) = \delta$ for triple $t = (v, \tau, \delta)$. In total, the first step and the final step can be done in linear time, and the time complexity is dominated by sorting.

Algorithm 8.1 Answer change-point query (s, t)

```
1:  procedure QueryChangePoints(s, t, L)
2:      i_s, i_t ← 0
3:      C ← an empty array
4:      while i_s < |L[s]| and i_t < |L[t]| do
5:          if v(L[s][i_s]) < v(L[t][i_t]) then
6:              i_s ← i_s + 1
7:          else if v(L[s][i_s]) > v(L[t][i_t]) then
8:              i_t ← i_t + 1
9:          else
10:             τ ← max{τ(L[s][i_s]), τ(L[t][i_t])}
11:             δ ← δ(L[s][i_s]) + δ(L[t][i_t])
12:             Push (τ, δ) to C
13:             if v(L[t][i_t + 1]) ≠ v(L[t][i_t]) then
14:                 i_s ← i_s + 1
15:             else if v(L[s][i_s + 1]) ≠ v(L[s][i_s]) then
16:                 i_t ← i_t + 1
17:             else if τ(L[s][i_s] > τ(L[t][i_t])) then
18:                 i_s ← i_s + 1
19:             else
20:                 i_t ← i_t + 1
21:     Sort pairs in C by time.
22:     Filter out unnecessary pairs from C.
23:     return C
```

8.1.2 Offline Indexing Algorithm

For presenting the indexing algorithm for contemporary queries, we first described a naive labeling algorithm without pruning, then we introduced pruning to present the indexing algorithm. However, designing an algorithm for historical queries is more challenging since, while the naive labeling algorithm was obvious for contemporary queries, this time, even the naive labeling algorithm without pruning is not trivial for historical 2-hop cover. Therefore, we explain our index-

ing algorithm with three steps: we start from an algorithm based on dynamic programming, next we turn the algorithm into a BFS-like algorithm, and finally introduce pruning to the algorithm to obtain our indexing algorithm.

Dynamic Programming

As usual, we start from an empty index L_0 and construct index L_k from L_{k-1} by adding triples whose destination vertex is v_k. Let D be the maximum distance to a connected vertex from v_k regarding all the snapshots. Let T be a $(D+1) \times |V|$ table. We conduct dynamic programming on the table T so that each cell $T[\delta][u]$ denotes the earliest time τ with $d_\tau(v_k, u) \leq \delta$. First, we fill the cells with distance zero as $T[0][v_k] = 0$ and $T[0][u] = \infty$ for any $u \neq v_k$. Then, we compute the values of cells with distance $\delta > 0$ from smaller δ by the following recurrence relation:

$$T[\delta][u] = \min_{w \in N(u)} \{\max\{T[\delta-1][w], t(w,u)\}\},$$

for any $u \neq v_k$ and $T[\delta][v_k] = 0$.

Lemma 8.1. *Each cell $T[\delta][u]$ denotes the earliest time τ with $d_\tau(v_k, u) \leq \delta$.*

This lemma can be proved by mathematical induction on δ. After computing the table, we add triple (v_k, δ, τ) to label $L_k(u)$ where $\tau = T[\delta][u]$ if $T[\delta][u] \neq \infty$ and $\delta = 0$ or $T[\delta][u] < T[\delta-1][u]$.

Historical Naive Landmark Labeling

While the algorithm above computes the correct index, it takes $\Theta(D\,|E|)$ time and $\Theta(D\,|V|)$ space. In this subsection, we reduce the time and space complexity by skipping unnecessary computations. Again, we suppose we are to construct index L_k from L_{k-1} by adding triples whose destination vertex is v_k.

The key insight here is the following simple fact. For simplicity, we define $T[-1][u] = \infty$ for any vertex u in the following. For any vertex u and $\delta \geq 0$, if $T[\delta-1][w] = T[\delta][w]$ for all $w \in N(u)$, then $T[\delta+1][u] = T[\delta][u]$. Therefore, we avoid vainly computing values of such cells as follows. For each distance $\delta \geq 0$ and vertex u, we initially set $T[\delta+1][u] = T[\delta][u]$. Then, we only check edges (w, u) incident to vertex w with $T[\delta-1][w] \neq T[\delta][w]$, and update $T[\delta+1][u]$ by $\max\{T[\delta][w], t(w,u)\}$ if it is smaller than the current value.

This can be efficiently achieved by managing vertices with queues. We prepare two queues, Q and Q', where initially Q contains v_k and Q' is empty. For each distance $\delta \geq 0$, supposing that Q contains vertices w with $T[\delta-1][w] \neq T[\delta][w]$, we check edges incident to each vertex $w \in Q$ and create Q' for the next distance $\delta+1$. If we obtain $T[\delta+1][u] \neq T[\delta][u]$ and $u \notin Q'$, we push u to Q'. Finally, before incrementing δ, we swap Q and Q', and clear Q'. Note that, the algorithm now behaves quite similarly to BFSs, although it may visit a vertex more than once. We also add triple $(v_k, T[\delta][u], \delta)$ to label $L(u)$ when we draw vertex u from Q.

However, even using queues, it still takes $\Omega(D\,|V|)$ time and $\Theta(D\,|V|)$ space due to the two-dimensional table. Thus, instead of straightforwardly using a two-dimensional table, we use two one-dimensional arrays with length $O(|V|)$, and avoid full initialization for each distance δ. Consequently, conducting queue-based dynamic programming and avoiding $\Theta(|V|)$ time initialization for each step, the total time complexity becomes $O(m')$, where m' is the number of traversed edges including duplications. Also note that, by using queues, we do not need to obtain the maximum distance D beforehand, as it suffices to stop when the queues get empty.

Algorithm 8.2 Pruned BFS from $v_k \in V$ to create index L'_k for historical queries.

```
1: procedure PrunedBFS(G, v_k, L'_{k-1})
2:     L'_k[v] ← L'_{k-1}[v] for all v ∈ V(G).
3:     Q ← a queue with only one element v_k.
4:     T[v_k] ← 0 and T[v] ← ∞ for all v ∈ V(G) \ {v_k}.
5:     T'[v] ← ∞ for all v ∈ V(G).
6:     for all δ = 0, 1, ... until Q gets empty do
7:         Q' ← an empty queue.
8:         for all u ∈ Q do
9:             if QuerySnapshot(v_k, u, T[u], L'_{k-1}) ≤ δ then
10:                continue
11:            L'_k[u] ← L'_k[u] ∪ {(v_k, T[u], δ)}
12:            for all w ∈ N_G(v) do
13:                τ' = max{T[u], t(u, w)}
14:                if τ' < T'[w] and τ' < T[w] then
15:                    if T'[w] = ∞ then
16:                        Enqueue w onto Q'.
17:                    T'[w] ← τ'.
18:        T[u] ← T'[u], T'[u] ← ∞ for all u ∈ Q'.
19:        Q ← Q'.
20:    return L'_k
```

Historical Pruned Landmark Labeling

We finally introduce pruning to the previous algorithm to obtain our indexing algorithm. Suppose we have started with an empty index L'_0 and we are constructing index L'_k from L'_{k-1} and the result of the k-th pruned BFS from v_k. Along with the labeling algorithm for contemporary queries, after drawing vertex u from the queue Q, we issue a query between v_k and u, and if the distance is at most δ, we prune vertex u. The difference from the algorithm for contemporary queries here is that we issue a snapshot query with regard to time $T_0[u]$. The total algorithm is described as Algorithm 8.2.

The correctness of this algorithm is not obvious, but can be proved as the exactly same way as the correctness of the pruned landmark labeling algorithm for contemporary queries.

Theorem 8.1. *For any pair of vertices s, t and $i, \tau \geq 0$,* $\textsc{QuerySnapshot}(s, t, \tau, L_i) = \textsc{QuerySnapshot}(s, t, \tau, L'_i)$.

Corollary 8.1. *For any pair of vertices s and t and $\tau \geq 0$,* $\textsc{QuerySnapshot}(s, t, \tau, L'_n) = d_\tau(s, t)$.

Corollary 8.2. *For any pair of vertices s and t, $(\tau, \delta) \in \textsc{QueryChangePoints}(s, t, L'_n)$ if and only if $d_{\tau_i - 1}(s, t) \neq d_{\tau_i}(s, t) = \delta$.*

By the same discussion as the standard pruned labeling algorithm, the time complexity is roughly estimated as $O(ml + n^2 l)$ time. As with the standard algorithms for contemporary queries, to exploit structures of real networks, we adopt the same vertex ordering strategies (i.e., we order vertices from those with higher degree in the final snapshot).

Note on weighted graphs

For handling weighted graphs, the algorithm can be applied by simply using a priority queue instead of a normal queue. We push triple (v, δ, τ) to the priority queue if v is reachable by distance δ at time τ. We pop the triple with the smallest distance to compute labels and traverse edges.

8.1.3 Online Incremental Update Algorithm

Incrementally updating the index to reflect graph changes can be done in the almost same way as Section 4.5. For a newly added vertex, we just prepare a new empty label, and for a newly added edge, we resume pruned BFSs from the endpoints. The time complexity is also the same.

The only difference here is that, as opposed to the contemporary query scenario for which we overwrite existing label entries when the label has an entry whose destination vertex matches the pair to add, we cannot overwrite existing entries since they would be used to answer the distance for the past. Instead, we add the new entry in such a case.

8.2 Experiments

8.2.1 Setup

We conducted experiments on a Linux server with Intel Xeon X5670 and 48GB of main memory. The proposed methods were implemented in C++. Only indexing was parallelized to use the six cores, and all the other timing results are sequential. We use 64 bits for each triple in that for historical queries (8 bits for distance, 24 bits for vertex IDs, and 32 bits represent time).

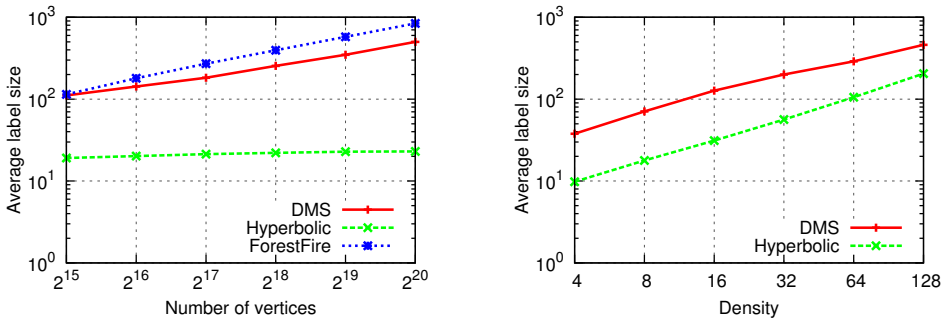
We used the same set of dynamic graph datasets used in Chapter 5. See Section 5.2 for the information and description.

8.2.2 Indexing Time, Index Size, and Label Size

For each dataset, we constructed an index from a historical graph data with all the snapshots except insertions of last 10,000 edges by the offline indexing algorithm. These last 10,000 edges are used for measuring average update time afterward. Indexing time, index size and average label size are shown in Table 8.1.

Although indexing time and index size are a little larger than those for contemporary queries, they are still acceptable even for large dynamic networks. For example, it took only two hours for constructing an index from the Wikipedia dataset. Also note that, since our method can incrementally update an index, we do not need to reconstruct an index frequently. The index size was 13 GB, which adequately fits in the main memory of commodity computers of the day. However, even if only computers with smaller main memory are available, we can first construct an index on crowd computing services such as Amazon EC2 using an instance with larger main memory, then we can conduct disk-based query answering on local computers by storing the index on a local disk.

Average label size is further investigated in Figure 8.2, which presents the average label size on synthetic networks with different graph size or density. We find that the average label size does not grow rapidly for both graph size and density. The indexing time, index size and query time follow similar trend as the label size.



(a) Varying graph size (b) Varying density

Figure 8.2: Label size for historical queries on synthetic networks with different size and density.

8.2.3 Query Time

Average query time reported in Table 8.1 is measured by 1,000,000 random queries after reflecting all the dynamic updates. As a baseline for snapshot queries, we also report the average time to evaluate a snapshot query by a BFS for 1,000 random queries. As a baseline for change-point queries, we suppose a naive method that conducts a BFS for each snapshot. Since this baseline method takes too long query time, we estimated the average query time by the product of the average snapshot-query time and the number of snapshots. For both snapshot queries and change-point queries, the query time of our method for historical queries is also generally microseconds and orders of magnitude faster than the baselines.

8.2.4 Update Time and Label Increase

Average update time listed in Table 8.1 is measured by inserting the last 10,000 edges of each dataset. As with the method for contemporary queries, this method for historical queries also handles each update in milliseconds. Moreover, as average label increase in Table 8.1 is small, we can confirm that the label size grows slowly.

Table 8.1: Experimental results of our method for historical queries against real-world and synthetic networks.

Dataset	Historical Pruned Landmark Labeling (This work)							BFS	
	Indexing time	Index size	Snapshot query time	Change-point query time	Update time	Label size	Label increase	Snapshot query time	Change-point query time
`Epinions`	23.1 s	236 MB	2.7 μs	4.6 μs	0.3 ms	234.2	2.6×10^{-4}	6.2 ms	2593 s
`Enron`	5.4 s	86 MB	1.6 μs	3.0 μs	0.2 ms	128.5	4.0×10^{-4}	5.3 ms	3041 s
`P2P`	2596.4 s	9.7 GB	12.4 μs	22.0 μs	10.7 ms	1227.9	2.7×10^{-4}	79.5 ms	$>$ 1day
`YouTube`	1281.8 s	9.1 GB	4.5 μs	8.1 μs	2.1 ms	374.7	4.0×10^{-5}	177.5 ms	$>$ 1day
`Wikipedia`	5165.9 s	13.0 GB	9.8 μs	12.7 μs	8.3 ms	919.5	3.0×10^{-5}	413.3 ms	$>$ 1day
`DMS`	920.0 s	3.8 GB	4.0 μs	5.6 μs	1.0 ms	481.1	8.6×10^{-5}	126.4 ms	$>$ 1day
`Hyperbolic`	24.7 s	195 MB	0.5 μs	0.7 μs	0.1 ms	23.9	2.9×10^{-6}	86.2 ms	$>$ 1day
`Forestfire`	1056.3 s	6.5 GB	8.2 μs	12.8 μs	3.9 ms	835.9	3.6×10^{-4}	91.4 ms	$>$ 1day

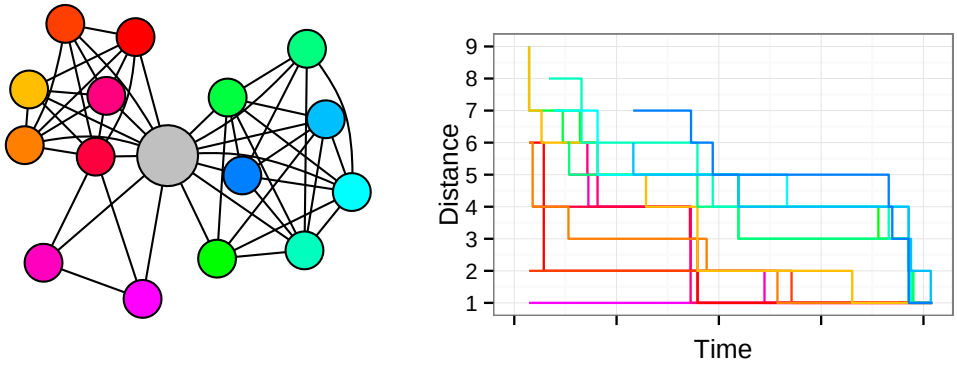
8.3 Application to Evolving Network Analysis

In this section, we demonstrate the usefulness of our historical indexing method for evolving network analysis. The proposed method enables quick and fine-grind temporal analysis of large-scale dynamic networks. For example, with our index, users can instantly and interactively check the transition of various features related to distances, which has never been available at all without our index.

We see by the case study in Section 8.3.1 that transitions of distance and shortest-path themselves are useful and of interest. Furthermore, based on our method, we see that we can also efficiently compute the transition of the following kinds of network features in the following sections.

8.3.1 Ego Network Analysis

Figure 8.3 illustrates an example of analysis on a real-world Facebook subgraph [VMCG09] based on historical change-point queries. Figure 8.3a depicts an ego network, i.e., the induced subgraph of a *center vertex* and its neighbors, where the center vertex is the gray one. From the figure, we can observe that there are two clusters on the left and right of the center vertex. Figure 8.3b shows the transition of the distances between the center vertex and its neighbors, where the colors of the lines correspond to those of the vertices in Figure 8.3a. We can confirm that the time periods of the appearance of the friendship links are different between the two clusters. Moreover, we find that the two clusters happened quite differently. That is, while the left cluster gradually approached the center vertex, the right cluster became neighbors almost instantly.

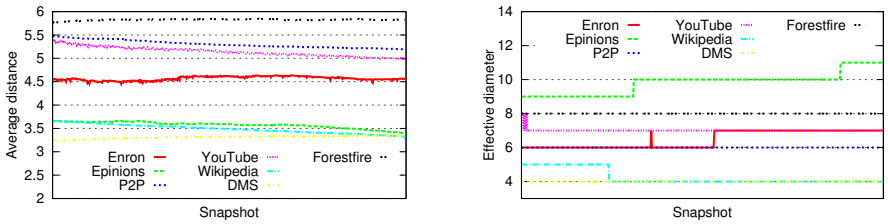


(a) An ego network (b) Distances to the neighbors

Figure 8.3: An example of social network analysis on a dynamic Facebook subgraph [VMCG09] using our method for historical shortest-path distance queries.

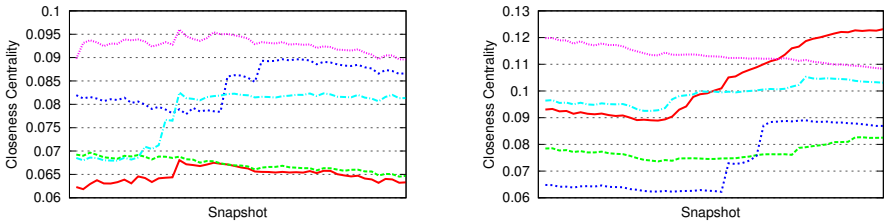
8.3.2 Average Distance and Effective Diameter

Distance distribution is one of the most important features of networks, and the transition of distance distribution of dynamic networks is of strong interest to the data mining and social network analysis community [LKF07]. Though calculating distance distribution of one graph is already too costly, to obtain the transition of it, we need to do so for many snapshots of graphs, which would be impossible for large historical networks. Using our historical indexing method, however, we can estimate the transition just by evaluating random change-point queries with regard to a set of randomly sampled pairs of vertices. To demonstrate the effectiveness of our method, we computed the transition of the average distance



(a) Average distance (b) Effective diameter

Figure 8.4: Transition of average distance and effective diameter.



(a) Enron (b) Epinions

Figure 8.5: Transition of the closeness centrality of some popular vertices.

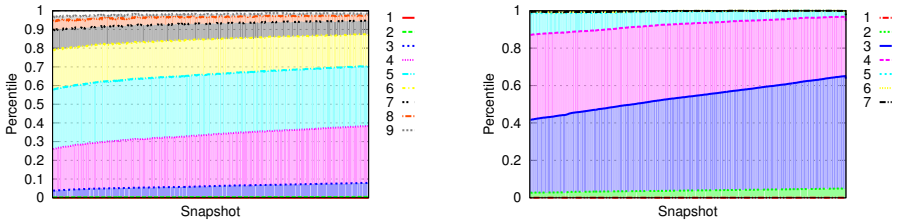
between pairs and the *effective diameter* (the 90th percentile distance) of various networks (Figure 8.4). We can observe that average distance decreases over time, which confirms the claim of [LKF07], but the effective diameter sometimes increases.

8.3.3 Closeness Centrality

Closeness centrality is one of the most popular network centralities defined on vertices. There are several different definitions, but all of them are based on distances to other vertices, and thus they can be efficiently estimated by random change-point queries. Here, we adopt the definition that defines the closeness centrality of vertex v as $\frac{1}{|V|}\sum_{u \in V} 2^{-d(v,u)}$. We picked up several vertices of high closeness centralities from `Enron` and `Epinions` and computed the transition of their closeness centralities by the proposed method (Figure 8.5). We can see that the closeness centrality sometimes drastically increases as some moment.

8.3.4 Temporal Hop Plot

To study distance distribution in depth, the *(temporal) hop plot*, which is the transition of the fraction of pairs within a fixed distance, is also used. We can approximate the hop plot by evaluating change-point queries with regard to a set of randomly sampled pairs of vertices. In Figure 8.6, we illustrate the hop plot of `YouTube` and `Wikipedia` for various distances. We observe that it tends to increase over time, as expected from the fact that average distance decreases over time.



(a) YouTube (b) Wikipedia

Figure 8.6: Temporal hop plot.

Chapter 9

Top-k Distance Queries on Complex Networks

In this chapter, we study top-k shortest-path distance queries. While many efficient methods for answering standard (top-1) distance queries have been developed, none of these methods are directly extensible to top-k distance queries. We develop a new framework for top-k distance queries based on *2-hop cover* and then present an efficient indexing algorithm based on our *pruned landmark labeling* scheme.

First we propse the indexing method for distance queries in Section 9.1, and then we present the experimental results in Section 9.2. Moreover, in Section 9.3, we discuss the usefulness of top-k distance queries in real applications.

Motivation

As discussed before, the shortest-path distance is widely applied as a proximity measure between vertex pairs. However, on complex unweighted graphs such as social and web graphs, there is a fundamental drawback of basing relevance on distance alone. Specifically, distances should be integers and the diameters of real-world networks are typically small [Mil67, TM69, WS98, BBR+12, BV12]. Such small diameter greatly reduce the number of possible distances and preclude the full use of the underlying structure.

This problem is clearly depicted in Figure 9.1. In each graph in the figure, the distance between the pair of black vertices is four. Hence, based on distance alone, the black pairs in all three graphs have the same similarity. However, the pair in graph (c) seems more tightly connected than the pairs in graphs (a) and (b), since this pair is connected by a greater number of shortest paths.

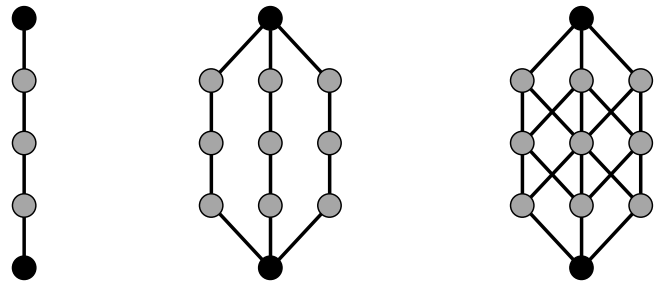


Figure 9.1: Examples of connection between two vertices.

This intuitive concept can be naturally implemented by adopting the *top-k distances*. Table 9.1 presents the top-k distances between the pair of black vertices in each graph of Figure 9.1. Although the pairs in each graph are separated by the same distance, their top-k distances markedly vary, providing a potential means of distinguishing these three graph structures.

Table 9.1: Distances and top-k distances between the two black vertices in the examples above.

Graph	Distance	Top-k Distances
(a)	4	[4, 6, 6, 6, 6, 8, 8, ...]
(b)	4	[4, 4, 4, 6, 6, 6, 6, ...]
(c)	4	[4, 4, 4, 4, 4, 4, 4, ...]

Formal Problem Definition

Let $\mathcal{P}$ be a set of paths. The *i-th shortest path in $\mathcal{P}$* refers to the i-th path in P, ordered by length, where ties are broken arbitrarily. For a pair of vertices (s, t), let $\mathcal{P}_{st}$ be the set of all (unnecessarily simple) paths between s and t. Then for two vertices s and t, the *i-th shortest path between s and t* is the i-th shortest path in $\mathcal{P}_{st}$. Let $d_{i\text{-th}}(s, t)$ denote the length of the i-th shortest path in $\mathcal{P}_{st}$. If the size of the corresponding set is less than i, then we set them to ∞. We study the following top-k distance query problem.

Problem 9.1 (Top-k Distance Query)**.**
Given: *A pair of vertices (s, t).*
Answer: *An array $(d_{1\text{st}}(s, t), d_{2\text{nd}}(s, t), \ldots, d_{k\text{-th}}(s, t))$.*

An *internal vertex* of a path refers to a vertex in the path that is not an endpoint of it. For a vertex v, let $\mathcal{P}_{st}^{>v}$ be the set of paths in $\mathcal{P}_{st}$ whose internal vertices are all larger than v. Similarly, let $\mathcal{P}_{st}^{\not>v}$ be the set of paths in $\mathcal{P}_{st}$ such that at least one internal vertex is smaller than or equal to v. Let $d_{i\text{-th}}^{>v}(s, t)$ and $d_{i\text{-th}}^{\not>v}(s, t)$ denote the length of the i-th shortest path in $\mathcal{P}_{st}^{>v}$ and $\mathcal{P}_{st}^{\not>v}$, respectively. We define $d_{i\text{-th}}^{\geq v}(s, t)$ and $d_{i\text{-th}}^{\not\geq v}(s, t)$ similarly.

9.1 Top-k Pruned Landmark Labeling

This section describes our proposed method and show its correctness. We also suggest several important techniques for practical performance enhancement.

9.1.1 Index Data Structure

The data structure and query algorithm of the proposed method are based on the general framework of *2-hop cover*, which is designed for standard (top-1) distance queries. However, as normal distance queries do not consider the number of paths, the main challenge in processing top-k distance queries is preventing multiple counts of the same path. To this end, we require a more involved framework.

For each vertex v, our method precomputes and stores the following two labels:

- *Distance label* $L(v)$, comprising a set of pairs (u, δ) of a vertex and a path length. If we gather lengths in $L(v)$ associated with a vertex u, they should form the sequence $(d_{1\text{st}}^{>v}(v, u), d_{2\text{nd}}^{>v}(v, u), \ldots, d_{\ell\text{-th}}^{>v}(v, u))$ for some $1 \leq \ell \leq k$.

- *Loop label* $C(v)$, constituting a sequence of k integers $(\delta_1, \delta_2, \ldots, \delta_k)$. This sequence should equal $(d_{1\text{st}}^{\geq v}(v,v), d_{2\text{nd}}^{\geq v}(v,v), \ldots, d_{k\text{-th}}^{\geq v}(v,v))$.

An *index* is a pair $I = (L, C)$, where L and C are the sets of distance labels $\{L(v)\}_{v \in V}$ and loop labels $\{C(v)\}_{v \in V}$, respectively.

9.1.2 Query Algorithm

Given an index $I = (L, C)$ and a pair of vertices (s, t), we compute the top-k distances between s and t as follows. First, we compute the following multiset.

$$\Delta(I, s, t) = \{\delta_{sv} + \delta_{vv} + \delta_{vt} \mid (v, \delta_{sv}) \in L(s), \\ \delta_{vv} \in C(v), (v, \delta_{vt}) \in L(t)\}.$$

Intuitively, we first move from s to v, then loop back to v several steps later, and finally move from v to t. Note that from the definition of distance labels and loop labels, every internal vertex in the path from s to t (except v itself) is larger than v.

Let $\textsc{Query}(I, s, t)$ denote the smallest k elements in the multiset $\Delta(I, s, t)$. If $|\delta(I, s, t)| < k$, the remaining entries are filled with ∞. This case occurs only when s and t are disconnected or when $s = t$ and isolated from other vertices. Our answer to the query (s, t) is $\textsc{Query}(I, s, t)$.

9.1.3 Indexing Algorithm

Our index constructing algorithm is summarized in Algorithm 9.1. We first compute the loop label $C(v)$ for every vertex v. We then construct the distance labels L by conducting a *pruned BFS* from each vertex.

Algorithm 9.1 Indexing Algorithm

```
1: procedure ConstructIndex(G)
2:     for i = 1 to n do Compute C(v_i) using the modified BFS.
3:     L(v) ← ∅ for all v ∈ V.
4:     for i = 1 to n do PrunedBFS(G, v_i).
5:     return (C, L).
```

Algorithm for Computing Loop Labels

We construct the loop labels as follows. For each vertex v, using vertices larger than or equal to v, we perform a modified version of breadth first search (BFS). In the BFS, each vertex may be visited up to k times. The first k visits to the vertex v gives the distance sequence $d_{1\text{st}}^{\geq v}(v,v), d_{2\text{nd}}^{\geq v}(v,v), \ldots, d_{k\text{-th}}^{\geq v}(v,v)$.

The modified BFS returns to the starting vertex long before all vertices in the graph have been visited. Consequently, the running time is very small in practice and empirically estimated as $O(nk)$ time in total from our experiments.

Algorithm for Computing Distance Labels

We assume that vertices in V are ordered as $v_1, v_2, \ldots, v_n$. Then for each $1 \leq i \leq n$, we perform a *pruned BFS* from v_i (Algorithm 9.2). The pruned BFS is essentially a modified version of the BFS from v that visits the same vertex at most k times. The crucial difference is the non-trivial pruning; that is, when visiting a vertex u at distance δ, the process is discontinued if δ is larger than

or equal to the k-th shortest distance computable by the current index (L, C) (Line 5).

Algorithm 9.2 Pruned Top-k BFS from $v \in V$.

```
1: procedure PrunedBFS(G, v)
2:     Q ← a queue with only one element (v, 0).
3:     while Q is not empty do
4:         Dequeue (u, δ) from Q.
5:         if δ < max(Query((L, C), v, u)) then
6:             Add (v, δ) to L(u).
7:             for all w ∈ V such that (u, w) ∈ E, w > v do
8:                 Enqueue (w, δ + 1) onto Q.
```

We now estimate the time complexity of this algorithm. Let l be the average size of labels. We visit $O(nl)$ vertices in total, traversing $O(\frac{m}{n})$ edges on average and evaluating a query in $O(l)$ time (by using the fast pruning technique introduced later). Thus, the total time complexity of this part is $O(ml + nl^2)$. In our experiments, l was a few hundred.

9.1.4 Proof of Correctness

The correctness of our method is shown as follows. Let L_i denote the set of distance labels L after the i-th pruned BFS from v_i. We define $L_0(v) = \emptyset$ for any v. Let I_i denote pair (L_i, C) of the partially constructed set of distance labels and the set of loop labels. We prove the following lemma.

Lemma 9.1. *For every integer i where $0 \leq i \leq n$, and every pair of vertices (s, t), $\textsc{Query}(I_i, s, t) = (d_{\text{1st}}^{\not> v_i}(s, t), d_{\text{2nd}}^{\not> v_i}(s, t), \ldots, d_{k\text{-}th}^{\not> v_i}(s, t))$ holds.*

Proof. We prove the claim by induction on i. When $i = 0$, we have $\textsc{Query}(I_i, s, t) = (\infty, \infty, \ldots, \infty)$ and the claim clearly holds. Suppose that the claim holds for every $i' < i$. For a fixed pair of vertices (s, t) where $s \neq t$, we validate the claim for i and the pair (s, t).

Note that we can already compute $\textsc{Query}(I_{i-1}, s, t) = (d_{\text{1st}}^{\not> v_{i-1}}(s, t), d_{\text{2nd}}^{\not> v_{i-1}}(s, t), \ldots, d_{k\text{-th}}^{\not> v_{i-1}}(s, t))$. Let $\mathcal{P}$ denote the set of paths P such that (i) P is in $\mathcal{P}_{st}^{> v_{i-1}}$, (ii) P passes through v_i, and (iii) the length of P is smaller than $d_{k\text{-th}}^{\not> v_{i-1}}(s, t)$. Let $\mathcal{P}'$ be the first k elements in $\mathcal{P}$. It suffices to show that, after the i-th pruned BFS, we can also compute the distances of paths in $\mathcal{P}'$.

Let $P \in \mathcal{P}'$. We can split P into three parts P_{sv_i}, $P_{v_i v_i}$, and $P_{v_i t}$. Here, P_{sv_i} denotes the subsequence of P from s to the first appearance of v_i in P, $P_{v_i v_i}$ denotes the subsequence of P from the first appearance of v_i to the final appearance of v_i in P, and $P_{v_i t}$ denotes the subsequence of P from the last appearance of v_i in P to t. Note that $P_{v_i v_i}$ must be among the first k elements in $\mathcal{P}_{v_i v_i}^{> v_i}$; otherwise shorter k paths are possible and $P \in \mathcal{P}'$ is contradicted. Hence, $C(v_i)$ must include the length of $P_{v_i v_i}$.

Now we observe that the BFS from v_i along path $P_{v_i t}$ is not pruned in the i-th pruned BFS (and similarly for P_{sv_i}). To illustrate by contradiction, suppose that the BFS is pruned at some vertex u on path $P_{v_i t}$. In this case, there exist at least k paths in $\mathcal{P}_{v_i u}^{\not> v_{i-1}}$ shorter than δ, where δ is the distance from v_i to u in the BFS. For each of these k paths, we concatenate P_{sv_i}, $P_{v_i v_i}$, and the suffix of $P_{v_i t}$ from u to t. Then, we obtain k paths in $\mathcal{P}_{st}^{\not> v_{i-1}}$ that are shorter than P,

and therefore shorter than $d_{k\text{-th}}^{\not\ni v_{i-1}}(s,t)$ from condition (iii). Hence, we reach a contradiction. □

Corollary 9.1. *At the end of Algorithm 4.2, we can correctly answer top-k distance queries using the constructed index.*

9.1.5 Techniques for Efficient Implementation

We introduce several key techniques for practical performance improvement.

Vertex Ordering Strategy

By properly selecting the order of vertices from which we conduct pruned BFSs, our pruning can drastically reduce the search space and label sizes by exploiting the structure of real-world networks, greatly enhancing the efficiency of the proposed method. This is possible because the real networks contain highly centralized vertices (sometimes called *hubs*). As a heuristic vertex ordering strategy, vertices are selected in order of decreasing degrees. Further discussion is provided in [AIY13].

Fast Pruning

When constructing distance labels, many queries are evaluated for pruning. However, when conducting a pruned BFS from a vertex v, queries are limited to "Are there more than k paths of length less than δ between v and u?" Given this restriction, we can reduce the query time. For each vertex w in the distance label of v, we can precompute the number $c_{w,\delta'}$ of paths between v and w of length not exceeding δ' using the loop label $C(w)$. Suppose that we have reached vertex u in the pruned BFS conducted from v. We can then compute the number of paths between v and u of length less than δ as $\sum_{(w,\delta',c)\in L(u)} c \cdot c_{w,\delta-\delta'}$.

Merged Queue Entries

When a (pruned) BFS is performed from a vertex v, rather than pair (u,δ), which denotes the existence of a path of length δ between v and u, triplets (u,δ,c) are pushed onto the queue. These triples specify that c paths of length δ exist between v and u, This technique enables the simultaneous handling of many paths, and significantly reduces the number of pushes onto the queue. Hence, it significantly reduces the running time.

Merged Label Entries

Related to the above technique, instead of pairs (u,δ), which denotes that there is a path of length δ between v and u, triplets (u,δ,c) are stored in distance labels. These triplets indicate that c paths of length δ exist between v and u. A similar technique is applicable to loop labels.

9.1.6 Extensions

Directed graphs

If the input graph is a directed graph, we compute and store two distance labels $L_{\mathrm{IN}}(v)$ and $L_{\mathrm{OUT}}(v)$ for each vertex v, where $L_{\mathrm{IN}}(v)$ and $L_{\mathrm{OUT}}(v)$ contain the distances from and to v, respectively.

Weighted graphs

For weighted graphs, we can replace the pruned BFS by pruned Dijkstra's algorithm. In this scheme, the queue used in Algorithm 4.1 is replaced by a priority queue. The time complexity becomes $O(ml + nl(\log n + l))$.

9.2 Experiments

In this section, we show the scalability, efficiency and robustness of the proposed method by experimental results using real-world networks.

9.2.1 Setup

Environment

All experiments were conducted on a Linux server with Intel Xeon X5670 (2.93 GHz) and 48 GB of main memory. The proposed method was implemented in C++.

Datasets

The target applications of the proposed method are graph mining tasks such as network-aware searching and link prediction. Therefore, our experiments were conducted on publicly available real-world social and web graphs[1][2][3][4][5]. The sizes and types of these graphs are listed in Table 9.2. We treated all the graphs as unweighted undirected graphs.

Algorithms

As there are no previous indexing methods for top-k distances, the proposed method was evaluated against the following two algorithms without precomputation.

- The first is the BFS-based naive approach, which uses a FIFO queue in the graph search, but which allows at most k visits to each vertex. This algorithm was also implemented in C++ by the authors.
- The second is Eppstein's algorithm [Epp98], which theoretically attains near-optimal time complexity. We adopted the C++ implementation of Jon Graehl[6].

In what follows, we denote the proposed method by *Top-k PLL* and these two previous methods by *BFS* and *Eppstein*.

[1] http://lovro.lpt.fri.uni-lj.si/support.jsp
[2] http://grouplens.org/datasets/hetrec-2011/
[3] http://snap.stanford.edu/
[4] http://socialnetworks.mpi-sws.org/datasets.html
[5] http://law.di.unimi.it/datasets.php [BV04]
[6] http://www.ics.uci.edu/ eppstein/pubs/p-kpath.html

Table 9.2: Dataset information and performance of the proposed and existing methods on real-world datasets ($k = 8$).

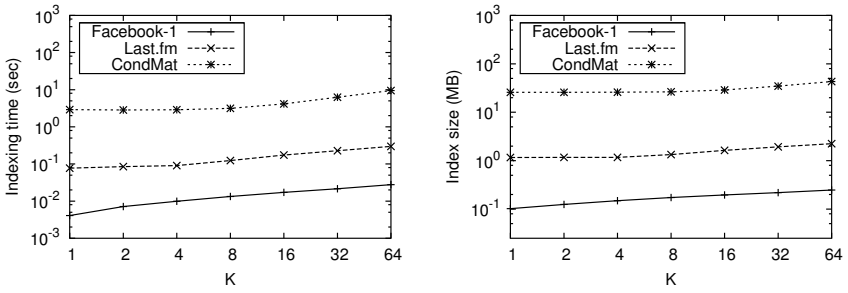
Dataset				Top-k PLL (This work)			BFS	Eppstein
Name	Type	$\|V\|$	$\|E\|$	Indexing time	Index size	Query time		
Facebook-1	Social	334	2,218	13.7 ms	178.6 KB	1.9 μs	227.1 μs	378.4 μs
Last.fm	Social	1,892	12,717	125.3 ms	1.3 MB	1.7 μs	1.6 ms	7.5 ms
GrQc	Social	5,242	14,496	152.9 ms	2.7 MB	1.6 μs	2.2 ms	7.3 ms
HepTh	Social	9,877	25,998	631.2 ms	7.8 MB	2.2 μs	5.5 ms	16.5 ms
CondMat	Social	23,133	186,936	3.2 s	26.4 MB	3.1 μs	15.2 ms	158.8 ms
Facebook-2	Social	63,732	1,545,686	239.0 s	716.8 MB	15.2 μs	117.6 ms	2.7 s
YouTube-1	Social	1,157,828	4,945,382	624.3 s	2.3 GB	5.1 μs	1.5 s	7.0 s
YouTube-2	Social	3,238,848	18,512,606	1627.1 s	9.6 GB	3.9 μs	5.0 s	41.1 s
NotreDame	Web	325,729	1,497,134	52.3 s	617.7 MB	2.9 μs	249.8 ms	1.7 s
Stanford	Web	281,903	2,312,497	42.5 s	230.0 MB	1.7 μs	454.9 ms	2.9 s
BerkStan	Web	685,230	7,600,595	108.7 s	1.0 GB	1.9 μs	643.3 ms	10.8 s
Indo	Web	1,382,906	16,539,644	2695.3 s	6.0 GB	12.1 μs	1.4 s	25.4 s

9.2.2 Indexing Time and Index Size

The high scalability of the proposed method is evident from the index construction time and constructed index size reported in Table 9.2. Indices were constructed from large social and web graphs comprising tens of millions of edges (YouTube-2 and Indo) in one hour. The index sizes are below 10 GB, easily accommodated by the main memories of modern commodity computers.

While the index construction of all datasets was consistently efficient, we observe that the indexing time does not depend on graph size alone. The efficiency of the proposed method relies on the efficiency of pruning, and is thus related to network properties such as degree distribution and clustering coefficient. However, because the graphs of real-world social, web, computer and biological networks exhibit similar qualitative properties, the proposed method is robust and consistently efficient. The same argument is valid for index size.

Figure 9.2a and 9.2b illustrate the effect of k on the indexing time and index size in the proposed method. Both are relatively insensitive to the value of k.



(a) Indexing time (b) Index size

Figure 9.2: Effect of k on indexing time and index size.

9.2.3 Query Time

The proposed method generally answers queries within microseconds, very much faster than the other algorithms (Table 9.2). Indeed for the largest dataset, YouTube-2, the query time was six orders of magnitude faster than those of the BFS-based and Eppstein algorithms. This query time enables top-k distances to be used in real-time interactive applications such as network-aware search for the first time. In our experiments the BFS-based method was faster than Eppstein's algorithm. This is due to the big constant factor hidden in the O-notation of the time complexity of Eppstein's algorithm, as it involves complex data structure manipulation.

Figure 9.3 plots the query time as a function of k. Although the query time increased with k, it remained sufficiently fast at high k.

9.3 Application to Graph Data Mining

In [AHN+15], the usefulness of our indexing method for top-k distance queries are demonstrated by applying it to the link prediction problem [LNK03]. In particular, we confirm that top-k distances can contribute to prediction precision improvement. Note that our indexing method enables the first use of the top-k distances for such tasks, because top-k distances must be computed for many pairs of vertices during training and evaluation.

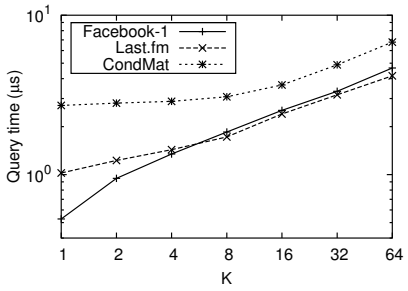


Figure 9.3: Effect of k on query time.

We selected link prediction as it is one of the most fundamental and popular problems on graphs in the AI and Web communities. However, the results suggest the applicability of top-k distances to other graph tasks such as network-aware searching.

it is empirically shown that the support vector machine (SVM) with the top-k distances as its feature outperforms a number of baseline methods including singular value decomposition and random walk with restart. We emphasize that our indexing method makes it possible for the first time to use the top-k distances for such tasks.

Chapter 10

Treewidth and Empirical Graph Tractability

In this chapter, we tackle the long-standing question in this field: what is the key factor in addition to network size that has a large effect on the size of constructed indices for graph path queries? As we have discussed several times in this thesis, state-of-the-art graph indexing algorithms, including ours, exploit many common structural properties of real-world graphs. Therefore, the performance (i.e., indexing time, index size and query time) of such algorithms depends not only the sizes of networks, and indeed vary largely even between networks with almost the same size (we confirm this point in Section 10.2.1).

Therefore, our goal here is to empirically estimate the *tractability* (or *difficulty*) of networks. To that end, we propose to use upper bound of treewidth obtained by heuristic tree decomposition algorithms. Here we focus on 2-hop-based indexing methods for distance queries, but we believe that similar results also hold for other problems where state-of-the-art methods are designed to exploit the structures of real networks.

First, as previous decomposition algorithms suffer from the drawback of scalability, we present a faster algorithm based on the new notion of *star-based representation* in Section 10.1. Then, we discuss the experimental results in Section 10.2. In particular, we confirm that the width of a tree decomposition obtained by our algorithm is indeed informative.

10.1 Tree Decomposition Algorithm

In this section, we give a detailed description of our algorithm for constructing a tree decomposition. A naive algorithm for the decomposition can be obtained by extending the well-known *min-degree heuristic* [BHS03]; however, its scalability is highly limited because of its costly clique materialization. To construct the decompositions for large networks, we propose a new algorithm with several orders of magnitude better scalability using the new idea of a *star-based representation*. Our method is also based on the min-degree heuristic; thus we first explain this heuristic (Section 10.1.1), and then present our new algorithm (Section 10.1.2).

10.1.1 Min-degree Heuristic Algorithm

Our algorithm is based on the *min-degree heuristic* [BHS03], which is a standard tree decomposition algorithm in practice [XJB05, Wei10, ASK12].

At a high level, the algorithm first generates a list of bags, and then constructs a tree of these bags (Algorithm 10.1). To generate a list of bags, the algorithm repeatedly *reduces* a vertex with the smallest degree. The reduction of vertex v includes of three steps. First, we create a new bag V_v including v and all its neighbors. Second, we change the graph G by removing node v. Third, we create

a clique among those vertices in $V_v \setminus \{v\}$. Then, we construct the tree of the tree decomposition from the list of bags. We set the parent of bag V_v as bag V_p, where $(V_v \setminus \{v\}) \subseteq V_p$. We can always find the valid parent because all neighbors of a reduced vertex are contained in a clique. Figure 10.1 illustrates a running example of the min-degree heuristic algorithm.

Algorithm 10.1 Min-degree heuristic.

1: Repeatedly reduce a vertex with minimum degree to generate a list of bags.
2: Add a bag with all the remaining vertices to the list as the root bag.
3: Construct a tree of these bags.

Drawback of Scalability Even if we assume that the adjacency lists are managed in hash tables and operations on edges can be performed in $O(1)$ time, reducing vertex v takes $\Theta(|V_v|^2)$ time. Thus, in total, the algorithm takes $\Theta(\sum_{t \in T \setminus \{r\}} |V_t|^2)$ time. Furthermore, we need to materialize edges of cliques; hence space consumption is also too large. Therefore, even if we use a relatively small parameter d (e.g., 100), it becomes impractical to apply the described algorithm to large-scale networks.

10.1.2 Proposed Tree Decomposition Algorithm

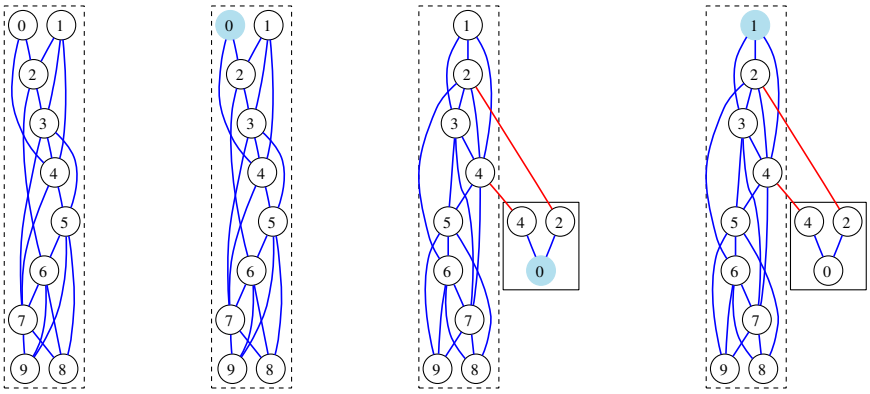
Overview

The idea behind our method is to *virtually* conduct the min-degree heuristic algorithm to avoid costly clique materialization. Rather than naively inserting all the edges of the cliques, we introduce *star-based representation* to maintain clique edges efficiently. In this representation, all operations on graphs used in the min-degree heuristic correspond to simple operations, such as modification of roles of vertices and *contraction* of edges, which leads to improved scalability of several orders of magnitude.

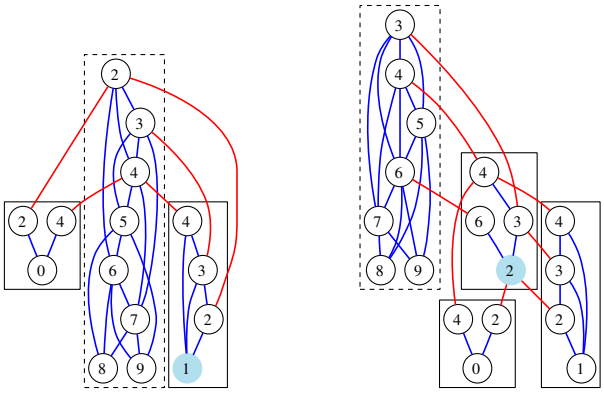
Star-based Representation Here we deal with two kinds of graphs: A *(star-based) representation graph* is what we store and maintain in the memory, and a *represented graph* corresponds to a virtual snapshot of a graph represented by a representation graph that would be maintained by the naive min-degree heuristic algorithm.

In the star-based representation, each vertex belongs to one of the following two types: *normal vertices* or *hub vertices*. Two hub vertices are never connected, i.e., edges connect either two normal vertices or a normal vertex and a hub vertex. The represented graph can be obtained by a representation graph by *(1)* adding edges to make its neighbors a clique for all hub vertices and *(2)* removing hub vertices. For example, the representation graphs shown in Figures 10.2a and 10.2c represents the graphs shown in Figures 10.2b and 10.2d, respectively.

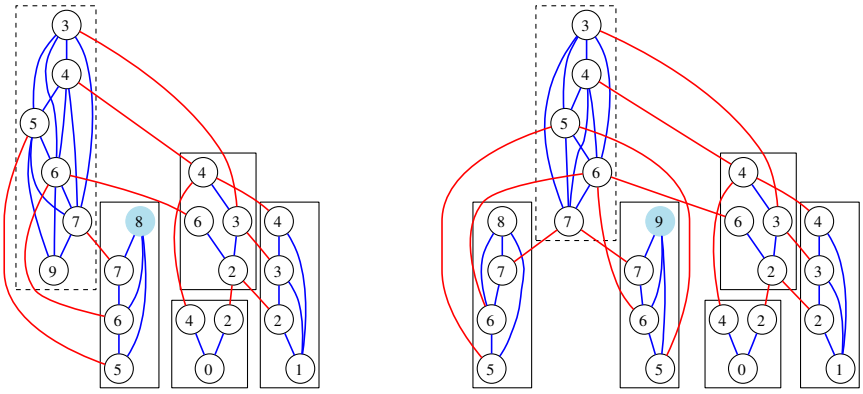
Overall Algorithm At a high level, our algorithm conducts the min-degree heuristic (Algorithm 10.1) on virtually represented graphs, i.e., it repeatedly reduces vertices in the represented graph to generate a list of bags. Then our algorithm constructs the tree of the bags. To construct the tree, we use the same tree construction algorithm. Therefore, as noted before, the main difference here is that, during the reduction phase, we do not maintain the represented graph



(a) The original graph. (b) Chose vertex 0. (c) Reduced vertex 0. (d) Chose vertex 1.



(e) Reduced vertex 1. (f) Reduced vertex 2.



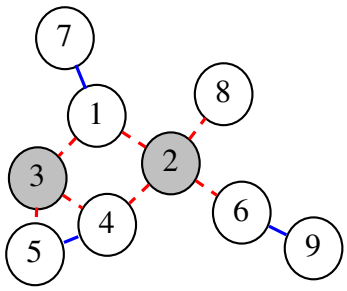
(g) Reduced vertex 8. (h) Reduced vertex 9.

Figure 10.1: An example of computation process of a tree decomposition.

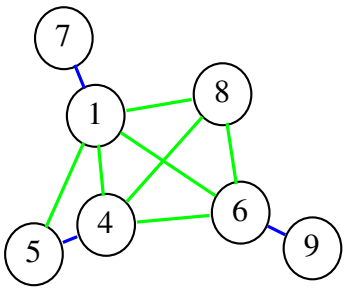
itself. We manage the star-based representation graph instead. Thus, we explain how to reduce vertices using star-based representation for enumerating bags.

Reducing a Vertex First, for an easier case, we consider a situation in which we reduce vertex v, whose neighbors are all normal vertices. To remove v and make its neighbors a clique in the represented graph, we must alter the vertex type of v from normal to hub.

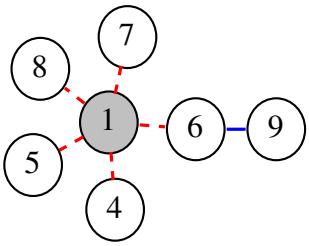
Let us now consider a general situation, where some of v's neighbors in the representation graph are hub vertices. One of the challenges here is the fact



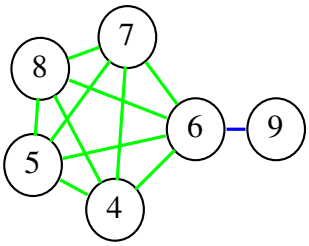
(a) A star-based representation of a graph.



(b) The represented graph.



(c) The star-based representation after reducing vertex 1.



(d) The represented graph after reducing vertex 1.

Figure 10.2: Star-based representation and reduction. The white vertices are normal vertices, and gray vertices are hub vertices.

that no direct edges can exist in the representation graph between some of v's neighbors due to these neighbor hub vertices. To make v's neighbors a clique in the represented graph, we must create a new hub vertex that is connected to all these neighbors.

Rather than creating such a new hub from scratch, the new hub can be efficiently composed by contracting v and all the neighboring hub vertices. Contraction of two vertices means removing the edge between them and merging two endpoints. For example, reducing the vertex 1 in the represented graph shown in Figure 10.2b corresponds to contracting vertices 1, 2, and 3 in the representation graph depicted in Figure 10.2a, thus yielding the representation graph in Figure 10.2c. By doing so, as we will discuss in Section 10.1.2, the time complexity becomes almost linear to the output size using proper data structures. Moreover, we never add any new edge to the representation graph. Therefore, the number of edges in the representation graph never increases; thus, space consumption is also kept in linear.

Details

Finding a Vertex to Reduce Precisely finding a vertex with minimum degree is too costly because we must track the degree of all vertices. Therefore, we approximately find a vertex with minimum degree by using a priority queue as follows. First, we insert all vertices into the priority queue, and use their current degree as keys. To find a vertex to reduce, we pop a vertex with the smallest key from the priority queue. If its current degree is the same as the key, then we reduce the vertex. Otherwise, we reinsert the vertex to the priority queue with its new degree.

Data Structures We manage adjacency lists in hash tables to operate edges in constant time. To efficiently contract edges, we manage groups of vertices and merge adjacency lists, similar to the weighted quick-find algorithm [Yao76].

Time Complexity We roughly estimate the time complexity. As we contract vertices similar to the weighted quick-find algorithm, the expected total time consumed for edge contractions is $O(m)$ time [KS78]. Reducing a vertex v takes approximately $O(d')$, where d' is the degree of v. Therefore, we expect that the proposed algorithm computes a tree decomposition in $O(\sum_{i<d} id_i + m)$ time, where d_i is the number of vertices of degree i.

10.2 Results and Discussion

In Table 10.1, the information of small datasets and results of our tree decomposition are given. Note that our tree decompositions are not necessarily optimal, and hence the width given in Table 10.1 is just an upper bound of treewidth. However, the aim of this section is to show how informative the width of a tree decomposition obtained by our algorithm is.

In the following section, we focus on two state-of-the-art indexing methods: *pruned landmark labeling* and *IS-label* [FWCW13]. Both are based on the 2-hop cover framework. While both pruned landmark labeling and IS-label use further sophisticated frameworks based on 2-hop cover, they can be used to construct standard 2-hop indices. Therefore, for simplicity, in our experiments we constructed standard 2-hop indices by these methods, i.e., pruned landmark labeling was not combined with the bit-parallel labeling technique and IS-label constructed complete vertex hierarchy.

Table 10.1: Information of small datasets, results of our full tree decomposition, and sizes of 2-hop indices constructed by state-of-the-art indexing methods for shortest-path distance queries.

Dataset Information				Tree decomposition			Distance Indices (MB)	
Name	$\|V\|$	$\|E\|$	Type	Time (s)	Width d	$d/\|V\|$	PLL	ISL [FWCW13]
ca-grqc	5,242	28,980	social (u)	0.02	253	0.048	1.4	3.9
ca-hepth	9,877	51,971	social (u)	0.16	798	0.081	4.4	24.7
wiki-vote	7,115	103,689	social (d)	0.59	1,332	0.187	2.4	23.8
ca-condmat	23,133	186,936	social (u)	1.5	2,160	0.093	13.4	156.8
ca-hepph	12,008	237,010	social (u)	0.52	1,406	0.117	8.8	65.5
email-enron	36,692	367,662	social (d)	1.43	2,178	0.059	8.4	136.1
ca-astroph	18,772	396,160	social (u)	3.34	3,497	0.186	19.5	233.5
email-euall	265,214	420,045	social (d)	1.67	1,033	0.004	84.1	453.0
soc-epinions1	75,879	508,837	social (d)	11.43	5,504	0.073	45.5	1,033.3
soc-slashdot0811	77,360	905,468	social (d)	28.31	8,555	0.111	77.1	1,772.3
soc-slashdot0902	82,168	948,464	social (d)	29.41	9,181	0.112	85.3	2,028.5
web-notredame	325,729	1,497,134	web (d)	0.99	2,938	0.009	95.4	2,835.4
web-stanford	281,903	2,312,497	web (d)	2.82	1,611	0.006	64.3	2,086.7
web-google	875,713	5,105,039	web (d)	80.28	18,229	0.021	712.3	64,540.1
web-berkstan	685,230	7,600,595	web (d)	10.81	3,272	0.005	195.5	10,482.0
p2p-gnutella08	6,301	20,777	p2p (u)	0.12	1,263	0.200	5.0	26.3
p2p-gnutella09	8,114	26,013	p2p (u)	0.19	1,618	0.199	8.0	43.5
p2p-gnutella06	8,717	31,525	p2p (u)	0.66	2,199	0.252	10.2	65.2
p2p-gnutella05	8,846	31,839	p2p (u)	0.35	2,215	0.250	10.5	65.5
p2p-gnutella04	10,876	39,994	p2p (u)	0.61	2,789	0.256	15.8	102.6
p2p-gnutella25	22,687	54,705	p2p (u)	0.9	3,618	0.159	45.8	284.1
p2p-gnutella24	26,518	65,369	p2p (u)	1.59	4,320	0.163	50.0	389.6
p2p-gnutella30	36,682	88,328	p2p (u)	2.04	5,596	0.153	100.9	707.6
p2p-gnutella31	62,586	147,892	p2p (u)	5.78	9,385	0.150	233.6	2,050.2
cit-hepth	27,770	352,807	citation (d)	11.61	8,515	0.307	158.4	687.0
cit-hepph	34,546	421,578	citation (d)	10.21	10,718	0.310	178.4	1,155.1

10.2.1 Non-Trivial Factors for Index Size

We first confirm the existence of non-trivial factors for the index size. Table 10.1 lists the sizes of 2-hop indices constructed by the two methods, pruned landmark labeling and IS-label [FWCW13] for our datasets. The results indicate that, even if two graphs are of similar size, indices constructed from these graphs by the same algorithm may be of quite different sizes.

For example, datasets `email-enron`, `ca-astroph`, `cit-hepth` and `cit-hepph` have similar sizes in terms of the number of vertices and edges. However, sizes of indices for these datasets vary largely and, surprisingly, indices for `cit-hepph` are approximately ten times larger than those for `email-enron` for both indexing algorithms. Similar differences can be observed from datasets `ca-condmat` and `p2p-gnutella30`.

This is because these state-of-the-art indexing methods heuristically exploit the structures of real networks explicitly or implicitly, and thus they depend on the properties of each network. Indeed, it is proved that 2-hop indices may have $\Theta(n^2)$ space for general graphs [GPPR04]. Therefore, it is impossible to construct small 2-hop indices without exploiting the network structures.

However, the long-standing question among researchers in this field is that, having understood this, what is the key factor besides network size that has a large effect on the size of constructed shortest-path distance indices? In the following section, we show that obtaining the width of a tree decomposition can take us closer to the answer.

10.2.2 Qualitative Empirical Analysis

Widths of tree decompositions obtained using our algorithm are also listed in Table 10.1. We can observe that the difference in index sizes seems to be highly related to widths. For example, widths of the tree decompositions for datasets `email-enron` and `cit-hepph` are 10,718 and 2,178, respectively. Similarly, tree decompositions for `ca-condmat` and `p2p-gnutella30` have widths of 1,406 and 5,596, respectively.

Interestingly, from our results, index sizes for graphs with larger widths are almost always larger than those for graphs with smaller widths. Therefore, treewidth could be the key factor that has a large effect on the sizes of constructed shortest-path distance indices.

10.2.3 Quantitative Empirical Analysis

To further analyze the relation between widths of tree decompositions and sizes of distance indices, let us recall the theoretical results discussed in Section 4.3.4. Theorem 4.4 tells that, while there are no non-trivial theoretical bounds on the size of 2-hop indices that are better than $O(n^2)$ for general graphs [GPPR04], we can prove that there are small 2-hop indices for graphs of small treewidth. Specifically, it says that there is a 2-hop index with size $O(nw \log n)$ for a graph with treewidth w.

Although the aforementioned methods do not necessarily yield 2-hop indices of that size (i.e., $O(nw \log n)$), we show that this theoretical bound works quite well as an estimation. Figure 10.3 illustrates the relation between the estimated value $nw \log n$ and actual sizes of constructed indices showing that these values correlate well.

Moreover, in Table 10.2, we see that Spearman's correlation coefficient between estimation $nd \log n$ and actual index sizes is significantly higher than other

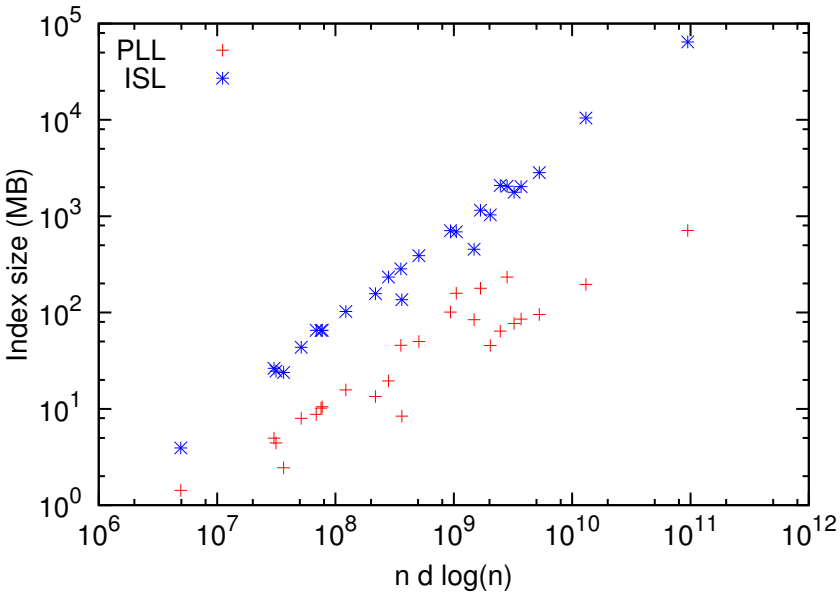


Figure 10.3: Actual index sizes and estimation using the widths of tree decompositions.

estimations such as n and m. This indicates that the width d of our tree decomposition is indeed informative. Note that Spearman's correlation coefficient uses only ranks and thus, for example, the score for estimation of n^2 would be exactly the same as that of n.

Table 10.2: Spearman's correlation between actual index sizes and estimation with and without width d.

Methods	n	m	$n+m$	d	$nd \log n$
PLL	0.819	0.774	0.798	0.795	**0.899**
IS-Label [FWCW13]	0.940	0.792	0.875	0.719	**0.983**

Chapter 11

Conclusions

In this thesis, we studied graph indexing methods for path-related queries. Previously, state-of-the-art methods for different queries or different families of graphs are developed almost independently, and their approaches were also quite different. Even limiting to labeling-based methods, their labeling algorithms were totally different. By contrast, the methods presented in this thesis are based on the same notion of *pruned labeling*, which were demonstrated to be competitive or comparable with state-of-the-art methods for each kind of query. Specifically, we designed indexing methods for reachability queries on directed acycilic graphs, shortest-path queries on road networks, historical shortest-path queries on evolving networks, and top-k shortest-path queries on complex networks. We demonstrated that each of these methods is also comparable with state-of-the-art methods for each kind of query, thus showing exceptional generality of our unified approach.

Lastly, we discuss the follow-up work of our pruned labeling algorithms by other groups and future work in this field.

Follow-up Work

The emergence of pruned labeling algorithms have had large impact to research communities. As evidence, there have already been several papers presenting follow-up work of them by other groups. We introduce a few notable results.

More robust vertex ordering strategies [DGPW14] As we observed, on many network instances with skewed degree distribution, simple vertex ordering strategies such as the Degree strategy work surprisingly well. However, Delling et al. show that, by approximately solving optimization problems, better vertex orders can be obtained for pruned labeling, which lead to further smaller label sizes. Moreover, they also showed that their ordering strategies are much more robust, in the sense that they can find good vertex ordering on various kinds of graph families.

Fully dynamic indexing for reachability queries [ZLWX14] We have presented incremental index update algorithm based on pruned labeling against edge insertion. Zhu et al. have also independently proposed index update algorithm based on pruned labeling for labeling-based reachability indices. In contrast to our discussion, they focus on vertex insertion and vertex removal. Vertex removal is much slower than vertex insertion, though it is faster than full reconstruction.

External memory labeling algorithm [JFWX14] Our indexing algorithm assumed that given graphs fit in main memory, and does not work well when they are in external memory due to frequent random access. Jiang et al. devised a labeling algorithm for the 2-hop cover framework, which can efficiently process graphs in external memory. It is also built on the notion of pruned labeling.

Future Work

More scalability While our methods improved the scalability of exact graph querying (e.g., distance queries on complex networks) by orders of magnitude, they are still not sufficiently scalable when considering huge social and web graphs, which may consist of billions of vertices and edges. Therefore, we still need more scalable indexing methods.

Complex queries Some of real applications requires path-related queries that have more involved constraints. For example, in graph database systems, answering path queries with label constraints is a fundamental building block [LB13], which cannot be efficiently processed by our current indexing techniques.

Stronger connection to theory Unfortunately, besides the seminal work mentioned in Chapter 3, theoretical research has had little impact to practical algorithms. This is because the interest of the theory community (e.g., worst-case asymptotic complexity on general graphs) is in a quite different direction from reality (e.g., real running time on real instances). Such discrepancy is quite common in the field of algorithmics. This work sometimes borrowed ideas from graph theory and theoretical graph algorithms, such as tree decomposition. There might be more theoretical ideas that are actually useful in practice. Contrary to this, the author also hopes to contribute to the *calibration* of current theoretical fields from the practice side.

List of Figures

List of Tables

References

[ABJ89] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.

[ADGW11] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, pages 230–241, 2011.

[ADGW12] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35. 2012.

[AFGW10] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA*, pages 782–793, 2010.

[AG06] I. Abraham and C. Gavoille. Object location using path separators. In *PODC*, pages 188–197, 2006.

[AHN+15] T. Akiba, T. Hayashi, N. Nori, Y. Iwata, and Y. Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In *AAAI*, 2015. to appear.

[AIKK14] T. Akiba, Y. Iwata, K. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *ALENEX*, pages 147–154, 2014.

[AIY13] 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.

[AIY14] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*, pages 237–248, 2014.

[AJB99] R. Albert, H. Jeong, and A. L. Barabasi. The diameter of the world wide web. *Nature*, 401:130–131, 1999.

[ALM09] F. Aidouni, M. Latapy, and C. Magnien. Ten weeks in the life of an edonkey server. In *IPDPS*, pages 1–5, 2009.

[AMK14] T. Akiba, T. Maehara, and K. Kawarabayashi. Network structural analysis via core-tree-decomposition. Manuscript, 2014.

[AP89] S. Arnborg and A. Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Appl. Math.*, 2:11–24, 1989.

[APPB10] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SC*, pages 1–11, 2010.

[ASK12] T. Akiba, C. Sommer, and K. Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *EDBT*, pages 144–155, 2012.

[BA99] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[Bar05] A.-L. Barabasi. The origin of bursts and heavy tails in human dynamics. *Nature*, 435:207–211, 2005.

[BBR+12] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. In *WebSci*, pages 33–42, 2012.

[BCSV04] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software Pract. Ex.*, 34(8):711–726, 2004.

[BDG+14] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. Werneck. Route planning in transportation networks. Technical report, MSR-TR-2014-4, 2014.

[BDS+10] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *J. Exp. Algorithmics*, 15(2.3):1–31, 2010.

[Bel58] R. E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BFM+07] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *ALENEX*, pages 46–59, 2007.

[BGJ+12] A. Bhattacharyya, E. Grigorescu, K. Jung, S. Raskhodnikova, and D. P. Woodruff. Transitive-closure spanners. *SIAM J. Comput.*, 41(6):1380–1425, 2012.

[BHKL06] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

[BHS03] A. Berry, P. Heggernes, and G. Simonet. The minimum degree heuristic and the minimal triangulation process. In *WG*, volume 2880 of *LNCS*, pages 58–70. 2003.

[BKM+00] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 33(1-6):309–320, June 2000.

[BLM+06] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4-5):175–308, 2006.

[BMSV14] P. Boldi, A. Marino, M. Santini, and S. Vigna. Bubing: Massive crawling for the masses. In *WWW Companion*, pages 227–228, 2014.

[BRSV11] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 587–596, 2011.

[BV04] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW*, pages 595–602, 2004.

[BV12] P. Boldi and S. Vigna. Four degrees of separation, really. In *ASONAM*, pages 1222–1227, 2012.

[Car71] B. A. Carré. An algebra for network routing problems. *IMA J. Appl. Math.*, 7(3):273–294, 1971.

[CGK05] L. Chen, A. Gupta, and M. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, pages 493–504, 2005.

[Cha03] D. Chamberlin. XQuery: a query language for XML. In *SIGMOD*, pages 682–682, 2003.

[CHKZ03] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[CNSW00] D. S. Callaway, M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Network robustness and fragility: Percolation on random graphs. *Phys. Rev. Lett.*, 85:5468–5471, 2000.

[CSTW12] W. Chen, C. Sommer, S.-H. Teng, and Y. Wang. A compact routing scheme and approximate distance oracle for power-law graphs. *TALG*, 9(1):4:1–26, 2012.

[CY09] J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT*, pages 481–492, 2009.

[CYL+06] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[DGJ09] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. AMS, 2009.

[DGPW14] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *ESA*, pages 321–333, 2014.

[DGSF14] D. Delling, A. V. Goldberg, R. Savchenko, and R. F.Werneck. Hub labels: Theory and practice. In *SEA*, pages 259–270, 2014.

[DGW13] D. Delling, A. V. Goldberg, and R. F. Werneck. Hub label compression. In *SEA*, pages 18–29, 2013.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DMS00] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin. Structure of growing networks with preferential linking. *Phys. Rev. Lett.*, 85:4633–4636, 2000.

[Epp98] D. Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.

[FFF99] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *SIGCOMM*, pages 251–262, 1999.

[Flo62] R. W. Floyd. Algorithm 97: Shortest path. *Comm. ACM*, 5(6):345, 1962.

[FLWW12] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[For56] L. R. Ford. Network flow theory. Report P-923, The Rand Corporation, 1956.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[FWCW13] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *PVLDB*, 6(6):457–468, 2013.

[GBSW10] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*, pages 499–508, 2010.

[GH05] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA*, 2005.

[GPPR04] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. *J. Algorithms*, 53(1):85 – 112, 2004.

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[Gut04] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *ALENEX*, 2004.

[Hal76] R. Halin. s-function for graphs. *J. Geometry*, 8:171–186, 1976.

[HL71] P. W. Holland and S. Leinhardt. Transitivity in structural models of small groups. *Small Group Research*, 2(2):107–124, 1971.

[HWYY07] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[IHI+94] T. Ikeda, M. Y. Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by ai search techniques. In *VNSI*, pages 291–296, 1994.

[JFWX14] M. Jiang, A. W. Fu, R. C. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB*, 7(12):1203–1214, 2014.

[Jor69] C. Jordan. Sur les assemblages de lignes. *J. Reine Angew Math*, 70:185–190, 1869.

[JRDX12] R. Jin, N. Ruan, S. Dey, and J. Xu. SCARAB: scaling reachability computation on large graphs. In *SIGMOD*, pages 169–180, 2012.

[JRXL12] R. Jin, N. Ruan, Y. Xiang, and V. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD*, pages 445–456, 2012.

[JXRF09] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.

[JXRW08] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, pages 595–608, 2008.

[Kar29] F. Karinthy. *Lancszemek*. 1929.

[KKT03] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.

[KMS06] E. Kohler, R. H. Mohring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, pages 41–72, 2006.

[KS78] D. E. Knuth and A. Schönhage. The expected linearity of a simple equivalence algorithm. *Theor. Comput. Sci.*, 6:281–315, 1978.

[Kur30] K. Kuratowski. Sur le Problème des Courbes Gauches en Topologie. *Fundamenta Mathematicae*, 15:271–283, 1930.

[KY04] B. Klimt and Y. Yang. The enron corpus: A new dataset for email classification research. In *ECML*, volume 3201 of *LNCS*, pages 217–226. 2004.

[LB13] A. Likhyani and S. Bedathur. Label constrained shortest path estimation. In *CIKM*, pages 1177–1180, 2013.

[LG14] F. Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303, 2014.

[LHK10a] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *WWW*, pages 641–650, 2010.

[LHK10b] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *CHI*, pages 1361–1370, 2010.

[LKF05] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[LKF07] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1), 2007.

[LLDM09] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[LNK03] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *CIKM*, pages 556–559, 2003.

[MA05] P. Massa and P. Avesani. Controversial users demand local trust metrics: an experimental study on epinions.com community. In *AAAI*, pages 121–126, 2005.

[MAIK14] T. Maehara, T. Akiba, Y. Iwata, and K. Kawarabayashi. Computing personalized pagerank quickly by exploiting graph structures. *PVLDB*, 7(12):1023–1034, 2014.

[Mil67] S. Milgram. The small world problem. *Psychology Today*, 1:61–67, 1967.

[Mis09] A. Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, 2009.

[MLH09] C. Magnien, M. Latapy, and M. Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *J. Exp. Algorithmics*, 13:10:1.10–10:1.9, February 2009.

[MMG+07] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, pages 29–42, 2007.

[MVLB14] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. Graph structure in the web — revisited: A trick of the heavy tail. In *WWW Companion*, pages 427–432, 2014.

[NSW01] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E*, 64(2):026118 1–17, 2001.

[Nuu95] E. Nuutila. *Effcient transitive closure computation in large digraphs*. PhD thesis, Finnish Academy of Technology, 1995.

[PAG09] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *Transactions on Database Systems*, 34(3):16:1–16:45, September 2009.

[PBCG09] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876, 2009.

[PBMW99] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[PSV04] R. Pastor-Satorras and A. Vespignani. *Evolution and structure of the Internet: A statistical physics approach*. Cambridge University Press, 2004.

[QCCY12] M. Qiao, H. Cheng, L. Chang, and J. X. Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. In *ICDE*, pages 462–473, 2012.

[RAD03] M. Richardson, R. Agrawal, and P. Domingos. Trust management for the semantic web. In *ISWC*, volume 2870, pages 351–368. 2003.

[RAS+05] S. A. Rahman, P. Advani, R. Schunk, R. Schrader, and D. Schomburg. Metabolic pathway analysis web service (pathway hunter tool at cubic). *Bioinformatics*, 21(7):1189–1193, 2005.

[Rep97] T. Reps. Program analysis via graph reachability. In *ILPS*, pages 5–19, 1997.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[RIF02] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, January 2002.

[RPFM14] M. P. Rombach, M. A. Porter, J. H. Fowler, and P. J. Mucha. Core-Periphery Structure in Networks. *SIAM J. Appl. Math.*, 74(1):167–190, February 2014.

[RS84] N. Robertson and P. D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[RS04] N. Robertson and P. Seymour. Graph minors. XX. wagner's conjecture. *J. Comb. Theory, Ser. B*, 92(2):325 – 357, 2004. Special Issue Dedicated to Professor W.T. Tutte.

[RS06] S. A. Rahman and D. Schomburg. Observing local and global properties of metabolic pathways: 'load points' and 'choke points' in the metabolic networks. *Bioinformatics*, 22(14):1767–1774, 2006.

[Sim88] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1):325–346, 1988.

[Som14] C. Sommer. Shortest-path queries in static networks. *ACM Comput. Surv.ag*, 46:45:1–31, 2014.

[STW04] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An efficient connection index for complex XML document collections. In *EDBT*, pages 237–255, 2004.

[TACGBn+11] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *CIKM*, pages 1785–1794, 2011.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[TC03] L. Tang and M. Crovella. Virtual landmarks for the internet. In *SIGCOMM*, pages 143–152, 2003.

[Tho99] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[Tho04] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004.

[TM69] J. Travers and S. Milgram. An experimental study of the small world problem. *Sociometry*, 32:425–443, 1969.

[TWRC09] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE*, pages 405–416, 2009.

[TZ05] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, January 2005.

[UCDG08] A. Ukkonen, C. Castillo, D. Donato, and A. Gionis. Searching the wikipedia with contextual information. In *CIKM*, pages 1351–1352, 2008.

[VFD+07] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. d. C. Reis, and B. Ribeiro-Neto. Efficient search ranking in social networks. In *CIKM*, pages 563–572, 2007.

[vHNM+00] J. van Helden, A. Naim, R. Mancuso, M. Eldridge, L. Wernisch, D. Gilbert, and S. J. Wodak. Representing and analysing molecular and cellular function using the computer. *Biol. Chem.*, 381(9-10):921–935, 2000.

[VMCG09] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *WOSN*, pages 37–42, 2009.

[vSdM11] S. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, pages 913–924, 2011.

[Wag37] K. Wagner. Über eine Eigenschaft der ebenen Komplexe. *Mathematische Annalen*, 114(1):570–590, 1937.

[War62] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[WED+08] Z. Wu, G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, and J. Srinivasan. Implementing an inference engine for RDFS/OWL constructs and user-defined rules in oracle. In *ICDE*, pages 1239–1248, 2008.

[Wei10] F. Wei. Tedi: efficient shortest path query answering on graphs. In *SIGMOD*, pages 99–110, 2010.

[WS98] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, pages 440–442, 1998.

[WXD+12] 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.

[XJB05] J. Xu, F. Jiao, and B. Berger. A tree-decomposition approach to protein structure prediction. In *CSB*, pages 247–256, 2005.

[YAIY13] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pages 1601–1606, 2013.

[Yao76] A. C.-C. Yao. On the average behavior of set merging algorithms. In *STOC*, pages 192–195, 1976.

[YBLS08] S. A. Yahia, M. Benedikt, L. V. S. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *PVLDB*, 1(1):710–721, 2008.

[YCZ12] H. Yildirim, V. Chaoji, and M. Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *The VLDB Journal*, 21(4):509–534, 2012.

[ZLWX14] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: A total order approach. In *SIGMOD*, pages 1323–1334, 2014.

[ZYQ+12] Z. Zhang, J. Yu, L. Qin, Q. Zhu, and X. Zhou. I/O cost minimization: reachability queries processing over massive graphs. In *EDBT*, pages 468–479, 2012.