Answering Shortest Path Distance Queries in Large Complex Networks

Muhammad Farhan

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

May 2022

Certificate Of Authorship/Originality

I, Muhammad Farhan, declare that this thesis is submitted in fulfilment of the requirements for the degree of Doctor of Philosophy, in the School of Computing at the Australian National University.

I certify that this thesis is my own original work, except where otherwise indicated. I also certify that this document has not been submitted for obtaining an award at any other academic institution.

Dedicated to my dear family, relatives and friends.

Acknowledgements

I could never be thankful enough to Almighty Allah who has always blessed me with success in life. Now, I would like to express my sincere gratitude to my supervisors, A/Prof. Qing Wang, Dr. Yu Lin and Prof. Brendan McKay for their continuous guidance and support over the past several years. They shared their wisdom and experience with me and encourage me to think critically, comprehensively, and objectively from multiple perspectives about doing research.

I would like to extend my special gratitude to A/Prof. Qing Wang who helped me tremendously in my personal growth throughout my PhD. She consistently supported me with her knowledge and wisdom in times of difficulties and provided me with constructive feedback to improve the quality of my research work during my PhD. Her enthusiasm on research helped me a lot towards becoming an independent researcher. I would say, I am very lucky to have A/Prof. Qing Wang as my supervisor, she is exactly the kind of supervisor I wanted before starting my PhD.

I would like to thank my collaborator, Dr. Henning Koehler, for his excellent support and advice in transforming research ideas to a publication. Moreover, I am fortunate enough to have many nice colleagues and friends with me during my Ph.D. studies at the Australian National University. I would like to thank all of them for their extensive support, quality time, understanding, and accompanies during my time in Canberra. I also would like to acknowledge all kinds of help and support provided by administrative staff including Melissa, Christie, Jasmine and many others at the ANU School of Computing and the HDR Team at the ANU College of Engineering and Computer Science.

I would like to show my sincere gratitude to my family and relatives for their immense amount of support and their remarkable companionship. I am grateful for having great parents, Pervaiz Akhter and Farzana Kausar, who planted the seeds of greatness in me and cultivated my values. I appreciate the understanding and extensive support that I received from my parents and my siblings, Rizwan Ahmed, Muniba Pervaiz, and Ayesha Pervaiz.

Publications

The contributions from the work presented in this thesis have been published across multiple peer-reviewed journals and conferences. A list of publications in reverse chronological order is given below:

[1] Farhan, M., Wang, Q., Koehler, H. "BatchHL: Answering Distance Queries on Batch Dynamic Networks at Scale." *The ACM Special Interest Group on Management of Data (SIGMOD)*, 2022.

The concepts, formulation, development of the batch-dynamic framework and algorithms, theoretical analysis, complexity analysis, algorithm implementation and evaluations of this paper are presented in Chapter 6.

[2] Farhan, M., Wang, Q., Lin, Y., Mckay, B. "Fast Fully Dynamic Labelling for Distance Queries." *The VLDB Journal*, 2021.

The concepts, formulation, development of the fully dynamic framework and algorithms, theoretical analysis, complexity analysis, algorithm implementation and evaluations of this paper are presented in Chapter 5.

[3] Farhan, M., Wang, Q. "Efficient Maintenance of Distance Labelling for Incremental Updates in Large Dynamic Graphs." *The 24th International Conference on Extending Database Technology (EDBT)*, 2021.

The development of the online incremental algorithm, theoretical analysis, complexity analysis, algorithm implementation and evaluations of this paper are presented in Chapter 5.

[4] Farhan, M., Wang, Q., Lin, Y., Mckay, B. "A Highly Scalable Labelling Approach for Exact Distance Queries in Complex Networks." *The 22nd International Conference on Extending Database Technology (EDBT)*, 2019.

The concepts, formulation, development of the highway cover property and labelling construction algorithm, bounded distance querying framework and algorithm, theoretical analysis, complexity analysis, algorithms implementation and evaluations of this paper are presented in Chapter 4.

Contents

Certificate Of Authorship/Originality iii

Acknowledgements vii

Publications ix

Table Of Content xi

List Of Figures xv

List Of Tables xix

Abstract xxi

1 Introduction 1
- 1.1 Background . . . 1
  - 1.1.1 Complex Networks . . . 2
  - 1.1.2 Dynamic Networks . . . 3
- 1.2 Research Objectives . . . 4
- 1.3 Research Challenges . . . 6
- 1.4 Contributions . . . 8
  - 1.4.1 Highway Cover Labelling For Distance Queries . . . 8
  - 1.4.2 Fully Dynamic Labelling For Distance Queries . . . 9
  - 1.4.3 Batch-Dynamic Labelling For Distance Queries . . . 9
- 1.5 Outline . . . 10

2 Preliminaries 11

3 Literature Review 15
- 3.1 Answering Distance Queries on Static Graphs . . . 15
  - 3.1.1 Search-based Methods . . . 15
  - 3.1.2 Labelling-based Methods . . . 15
  - 3.1.3 Hybrid Methods . . . 19
- 3.2 Answering Distance Queries on Dynamic Graphs . . . 20
  - 3.2.1 Labelling-based Methods . . . 20
  - 3.2.2 Hybrid Methods . . . 23

4 HL: Highway Cover Labelling For Distance Queries 25
4.1 Overview 25
4.2 Problem Definition 27
4.3 Highway Cover Labelling 28
4.3.1 Highway and Highway Cover 28
4.3.2 Labelling Construction Algorithm 30
4.3.3 Order Independence 32
4.3.4 Minimality 34
4.4 Bounded Distance Querying Framework 35
4.4.1 Computing Upper Bounds 35
4.4.2 Distance-Bounded Bi-Directional Search 36
4.5 Optimization Techniques 37
4.5.1 Label Construction 37
4.5.2 Label Compression 38
4.5.3 Query Processing 38
4.6 Theoretical Results 39
4.6.1 Proof of Correctness 39
4.6.2 Preservation of Minimality 40
4.6.3 Complexity Analysis 41
4.7 Experimental Setup 41
4.7.1 Datasets 41
4.7.2 Baseline Methods 43
4.7.3 Test Data Generation 44
4.8 Results and Discussion 44
4.8.1 Performance Comparison 45
4.8.2 Performance under Varying Landmarks 47
4.9 Summary 50

5 FulHL: Fully Dynamic Labelling For Distance Queries 51
5.1 Overview 51
5.2 Problem Definition 53
5.3 Fully Dynamic Labelling Framework 53
5.3.1 Jumped-and-Pruned Search 53
5.3.2 Incremental Algorithm 58
5.3.3 Improved Incremental Algorithm 61
5.3.4 Decremental Algorithm 63
5.4 Theoretical Results 66
5.4.1 Proof of Correctness 67
5.4.2 Preservation of Minimality 67
5.4.3 Complexity Analysis 68
5.5 Experimental Setup 68
5.5.1 Datasets 69
5.5.2 Baseline Methods 69
5.5.3 Test Data Generation 70

5.6 Results and Discussion . . . 71
5.6.1 Performance Comparison . . . 71
5.6.2 Performance under Varying Landmarks . . . 77
5.6.3 Analysis of Affected Vertices . . . 78
5.6.4 Scalability of Updates . . . 79
5.7 Summary . . . 80

6 BatchHL: Batch-Dynamic Labelling For Distance Queries 83
6.1 Overview . . . 83
6.2 Problem Definition . . . 85
6.3 Batch-dynamic Labelling Framework . . . 85
6.3.1 Batch Search . . . 86
6.3.2 Batch Repair . . . 90
6.4 Optimization . . . 91
6.4.1 Improved Batch Search . . . 91
6.4.2 Improved Batch Repair . . . 97
6.4.3 Landmark Parallelism . . . 98
6.5 Analysis of BatchHL . . . 99
6.6 Theoretical Results . . . 100
6.6.1 Proof of Correctness . . . 100
6.6.2 Preservation of Minimality . . . 101
6.6.3 Complexity Analysis . . . 101
6.7 Experimental Setup . . . 101
6.7.1 Datasets . . . 102
6.7.2 Baseline Methods . . . 102
6.7.3 Test Data Generation . . . 103
6.8 Results and Discussion . . . 104
6.8.1 Performance Comparison . . . 104
6.8.2 Performance under Varying Landmarks . . . 108
6.8.3 Performance under Varying Batch Size . . . 109
6.8.4 Performance on Directed Graphs . . . 110
6.9 Summary . . . 111

7 Extensions 113
7.1 Directed graphs . . . 113
7.2 Weighted graphs . . . 113

8 Conclusions and Future Work 115

Bibliography 119

List of Figures

1.1 Distribution of affected vertices by a single graph change in various networks, where the results for 1000 randomly selected graph changes, containing 50% edge insertions and 50% edge deletions are sorted in the descending order. 5

4.1 A high-level overview of the state-of-the-art methods and our proposed method (HL) for answering exact distance queries. Note that, Figure (a) is based on networks of size up to 400M edges. 26

4.2 An example graph G where one shortest path between two vertices 2 and 11 is highlighted in red. 27

4.3 An illustration of highway cover distance labelling: (a) an example graph G, (b) a highway H that connects to other vertices, and (c) a distance labelling that fulfills the highway cover property over G. 29

4.4 An illustration of the highway cover labelling algorithm: (a), (b) and (c) describe the pruned BFSs that are rooted at the landmarks 1, 5 and 9, respectively, where yellow vertices denote roots, green vertices denote those being labeled, red vertices denote landmarks, and white vertices are not labelled. LS and ET at the top right corner denote the labelling size and the number of edges traversed during the pruned BFSs, respectively. 32

4.5 An illustration of the pruned landmark labelling algorithm [Akiba et al. 2013]: (a)-(c) show an example of constructing labels through pruned BFSs from three landmarks in the labelling order $\langle 1, 5, 9 \rangle$; (d)-(f) show an example of constructing labels using the same three landmarks but in a different labelling order $\langle 9, 5, 1 \rangle$. Yellow vertices denote landmarks that are the roots of pruned BFSs, green vertices denote those being labeled, grey vertices denote vertices being visited but pruned, and red vertices denote landmarks which have already been visited. 33

4.6 An illustration of the distance-bounded shortest-path search algorithm [Hayashi et al. 2016]: (a) shows the sparsified graph after removing three landmarks $\{1, 5, 9\}$ from the graph in Figure 4.3(a); (b) shows an example of computing the bounded distance between vertices 2 and 11 as highlighted in yellow, and green vertices denote the visited vertices in the forward and reverse searches. 38

4.7 Distance distribution of 100,000 random pairs of vertices on all the datasets. . . . 44

4.8 Construction time using our method HL under 10-50 landmarks on all the datasets. . . . 47

4.9 Labelling size produced by our method HL under 10-50 landmarks and FD under 20 landmarks on all the datasets. . . . 48

4.10 Query time using our method HL under 10-50 landmarks on all the datasets. . . . 48

4.11 Pair coverage ratios using our method HL under 10-50 landmarks and using FD on all the dataset. . . . 49

5.1 An illustration of affected vertices by an edge insertion $(2, 5)$ occurring on the graph in (a), where three landmarks 0, 10 and 4 are highlighted in yellow. In (b)-(d), the affected vertices w.r.t. the landmarks 0, 10 and 4 are highlighted in green, respectively. Note that the affected vertices w.r.t. the landmarks 0, 10 and 4 would remain the same if an edge deletion $(2, 5)$ occurs on the graph in (b). . . . 54

5.2 An illustration of our incremental algorithm IncHL for an edge insertion (2,5): (a), (b) and (c) describe the JP-BFSs that are rooted at the landmarks 0, 10 and 4, respectively, where yellow vertices denote the landmarks (i.e., the roots), green color denotes affected vertices whose labels are updated and red color denotes affected vertices that are pruned during the JP-BFSs. . . . 63

5.3 An illustration of our decremental algorithm DecHL for an edge deletion $(2, 5)$: (a)-(b), (c)-(d) and (e) describe the JP-BFSs that are rooted at the landmarks 0, 10 and 4, respectively, where yellow vertices denote the landmarks (i.e., the roots), green color denotes affected vertices whose labels need to be updated, and red color denotes affected vertices being pruned. . . . 65

5.4 (a)-(b) show the distance distribution of 1000 pairs of vertices in E_I on all the datasets, where the distance for each pair is recorded before insertion, and (c)-(d) show the distance distribution of 1000 pairs of vertices in E_D on all the datasets, where the distance for each pair is recorded after deletion. . . . 69

5.5 Comparison of query time of the proposed method FulHL against online search method BiBFS. BiBFS has no results for Clueweb09 and Clueweb12 because it did not finish within 24 hours. . . . 75

5.6 Labelling size comparison of the proposed method FulHL (in colored bars) and the baseline method FulFD (in colored plus grey bars) under 10-150 landmarks. Note that there are no results of FulFD for Clueweb09 and Clueweb12 as FulFD failed to construct labelling. . . . 75

5.7 Average update time comparison for performing decremental updates between the proposed method FulHL (in colored bars) and the baseline method FulFD (in colored plus grey bars) under 10-150 landmarks. Note that there are no results of FulFD for Clueweb09 and Clueweb12 as FulFD failed to construct labelling. 76
5.8 Average update time comparison for performing incremental updates between the proposed method FulHL (in colored bars) and the baseline method FulFD (in colored plus grey bars) under 10-150 landmarks. Note that there are no results of FulFD for Clueweb09 and Clueweb12 as FulFD failed to construct labelling. 76
5.9 Query time comparison of the proposed method FulHL and the baseline method FulFD under 10-150 landmarks: (a) and (c) illustrate the average query time for FulHL; (b) and (d) illustrate the average query time for FulFD. Note that there are no results of FulFD for Clueweb09 and Clueweb12 as FulFD failed to construct labelling. 79
5.10 1000 edge insertions from E_I: (a)-(b) show the distribution of update times, and (c)-(d) show the distribution of the numbers of affected vertices. 80
5.11 1000 edge deletions from E_D: (a)-(b) show the distribution of update times, and (c)-(d) show the distribution of the numbers of affected vertices. 81
5.12 Comparison of average update time of the proposed method IncHL for performing up to 10,000 updates against the construction time. . . . 82
5.13 Comparison of average update time of the proposed method DecHL for performing up to 10,000 updates against the construction time. . . . 82

6.1 A high-level overview of our batch-dynamic method (BatchHL) which performs a batch update in two phases: 1) Batch Search: find vertices that are affected, and 2) Batch Repair: repair vertices returned by Batch Search. 83
6.2 An illustration for the number of vertices affected by batch updates of varying sizes. BHL and BHL$^+$ are our batch-dynamic algorithms, BHL^s is a variant of BHL which splits edge insertions and deletions into sub-batches and performs them sequentially, and UHL handles updates in the single-update setting. 84
6.3 Example graphs, where r is a landmark, the edges marked by $+$ are inserted, and the edges marked by $-$ are deleted: (a) a batch update with two edge insertions and two edge deletions; (b) a sub-batch update with only two edge insertions; and (c) a sub-batch update with only two edge deletions. 86
6.4 An example graph for composite-path affected vertices where r is a landmark. 89
6.5 An example graph where r_1 and r_2 are landmarks, the edges marked by $+$ are inserted, and the edges marked by $-$ are deleted. 91

6.6 Example graphs where the landmarks are circled. (a) Adding the edge (b, v) does not cause a label change for v; (b) Adding the edge (b, v) causes the label of v to be changed where the r-label of v needs to be deleted; (c) Deleting the edge (b, v) does not cause a change on the label of v; (d) Deleting the edge (b, v) causes a change on the label of v where an r-label needs to be inserted in v. 93
6.7 An example graph where r and b are landmarks and the edge (r, b) is deleted. The r-label of v is changed although the r-label of c does not change. 93
6.8 An example graph where r, a and c are landmarks and the edge (r, a) is deleted. The distances from r to a and b are both changed. 94
6.9 Distance distribution of batch updates. 104
6.10 Update time under 10-50 landmarks. 109
6.11 Query time under 10-50 landmarks. 109
6.12 Comparing the total time of querying and updating by the proposed methods against online search methods. 110

List of Tables

2.1 Summary of notations. 13

4.1 Several important properties related to labelling-based methods. 26
4.2 Datasets, where $size(G)$ denotes the size of a graph G with each edge appearing in the forward and reverse adjacency lists and being represented by 8 bytes. 43
4.3 Comparison between the construction time (CT) and query time (QT) of our methods, i.e., HL-P and HL, and the state-of-the-art methods, where CT is the CPU clock time in seconds, and QT is the average query time in milliseconds. HL-P refers a parallel version of HL and DNF denotes that a method did not finish in one day (24 hours) or ran out of memory (512 GB). 45
4.4 Comparison between the labelling size (LS) of our methods, i.e., HL(8) and HL, and the state-of-the-art methods. HL(8) refers to the compressed version of HL that uses 8-bits representation of vertices and ALS is the average number of entries per label. 46

5.1 Flickr, UK and Clueweb12 are the largest networks evaluated by the methods FulPLL [Akiba et al. 2014; D'angelo et al. 2019], FulFD [Hayashi et al. 2016] and FulHL (this work), respectively, where " $-$ " indicates no result due to scalability issues. 52
5.2 Comparison of the update time and query time of the proposed methods with the baseline methods, where "–" denotes that a method did not finish to construct labelling in one day (24 hours) or ran out of memory (512 GB). 71
5.3 Comparison of the update time for edge insertion and edge deletion of the proposed methods IncHL-m, IncHL, IncHL^b, and DecHL with the baseline methods. 72
5.4 Comparison of the labelling size of the proposed methods with the baseline methods, where "–" denotes that a method did not finish to construct labelling in one day (24 hours) or ran out of memory (512 GB). 73
5.5 Construction time, labelling size and query time of the state-of-the-art methods PSL, PSL$^+$ and PSL* where "–" denotes that a method did not finish to construct labelling in one day (24 hours) or ran out of memory (512 GB). 74

5.6 Comparison of the difference in the labelling sizes of the proposed method IncHL and the baseline method IncPLL after applying 1,000 updates. 74

6.1 An illustration of anchor vertices b, c and e w.r.t. the batch update depicted in Figure 6.3(a). The anchor distance for b and c through pre-anchor vertex a is $d_G(r,a)+1=2$. The anchor distance for e through pre-anchor vertex d is $d_G(r,d)+1=2$. 88
6.2 Datasets, where $size(G)$ denotes the size of a graph G with each edge appearing in the forward and reverse adjacency lists and being represented by 8 bytes. 102
6.3 Comparing update time and query time of our methods BHL^p, BHL$^+$, and BHL with the state-of-the-art dynamic methods, where the batch size is 1,000 and thus the update time reported for every method is for 1,000 updates. 105
6.4 Comparing update time of our methods BHL^p and BHL$^+$ with the state-of-the-art dynamic methods, where the batch size is 1,000 and thus the update time reported for every method is for 1,000 updates. . . 106
6.5 Comparing performance of our method BHL$^+$ with the baseline methods in terms of the construction time and labelling size. Note that when a method did not finish the labelling construction in 24 hours, we denote it as "–". 107
6.6 Comparing the average number of vertices affected by BHL$^+$ and BHL after performing batch updates on all the datasets. 108
6.7 Comparing update time, construction time (CT), query time (QT) and labelling size (LS) on directed graphs. 108

Abstract

The *distance query* problem is to find the shortest-path distance between an arbitrary pair of vertices in a graph. It is considered as a fundamental problem in graph theory. Despite a tremendous amount of research on the subject, there is still no satisfactory solution that can scale to large complex networks which may have billions of vertices and edges. Furthermore, many real-world complex networks such as social networks and web graphs are typically dynamic, undergoing discrete changes such as edge insertion and deletion in their topological structure over time. Thus, there is also a pressing need to address the distance query problem on dynamic networks.

The goal of this thesis is to address the distance query problem on large static and dynamic complex networks. Labelling-based methods are well-known for rendering fast response time to distance queries; however, existing labelling-based methods can only construct distance labelling for moderately large graphs with millions of vertices and edges and cannot scale to large graphs with billions of vertices and edges due to their prohibitively large space requirements and unbearably long pre-processing time. This thesis proposes a scalable approach that enables fast construction of a distance labelling of a limited size, which contains only distance information from all vertices in a graph to some "important" vertices (not all) - called *landmarks*. Such a distance labelling is considered as a *partial distance labelling*, in contrast to a *full distance labelling* that contains distance information for all pairs of vertices in a graph. Then, we combine a partial distance labelling that can be computed in an offline manner with online searching to leverage the advantages from both sides - accelerating query processing through a small sized partial distance labelling that provides a good approximation to bound online searches. The proposed method can efficiently construct a distance labelling for a graph with billions of vertices and edges, and enable fast distance computation, e.g. in the order of milliseconds.

Since graphs in real-world are dynamic that undergo changes such as edge insertion or deletion in their topological structure, existing labelling-based methods still greatly suffer from the drawback of scalability on dynamic graphs and they can hardly update a distance labelling efficiently. In this thesis, we propose a fully dynamic method which can efficiently reflect graph changes (i.e., single edge insertions or deletions) by dynamically maintaining a distance labelling in order to answer distance queries on dynamice graphs. At its core, our proposed method incorporates two building blocks: (i) *incremental algorithm* for handling incremental update operations, i.e. edge insertion, and (ii) *decremental algorithm* for handling decremental update operations, i.e. edge deletion. Moreover, this thesis also introduces a batch-dynamic method which can process batch of updates (i.e., batches of edge insertions and deletions) efficiently to further improve the performance of answering distance

queries on graphs that undergo rapid changes in their topological structure. The proposed batch-dynamic method enables us to unify edge insertions and deletion, helps us to avoid unnecessary and repeated computations, and allows us to exploit the potential of parallelism which as a result is much more efficient than processing graph changes separately one by one.

In this thesis, we have conducted extensive experiments on 15-17 real-world networks from a variety of application domains to test the scalability, efficiency, and robustness of the proposed static and dynamic methods against existing state-of-the-art static and dynamic methods.

Chapter 1

Introduction

A network, which is also referred to as a graph in mathematics, consists of a set of entities, called *vertices*, and the relationships between those entities, called *edges*. Networks are used to model real-world problems in various domains, such as transportation, communication infrastructures, information flow, power grids, and social interactions. One such problem is to find the shortest-path distance between a pair of vertices in a graph. This is considered to be a fundamental problem in graph theory that has been widely studied in the last several decades [Cohen et al. 2002; Madkour et al. 2017; Akiba et al. 2012; Potamias et al. 2009].

1.1 Background

The *distance query* problem is, given any two vertices in a graph, to find the shortest-path distance between these two vertices in the graph. It has a wide range of real-world applications [Ukkonen et al. 2008; Vieira et al. 2007; Yahia et al. 2008; Freeman 1977; Sabidussi 1966; Backstrom et al. 2006; Boccaletti et al. 2006]. In web graphs, giving ranks to web pages based on their distances to a recently visited web page helps in finding the more relevant web pages, which is referred to as context-aware web search [Ukkonen et al. 2008; Potamias et al. 2009]. In social networks, it is used to perform socially-sensitive search, the purpose of which is to find more relevant users or contents on social networking sites [Vieira et al. 2007; Yahia et al. 2008]. Moreover, shortest-path distance is also used as a key measure to solve many complicated problems such as centrality [Freeman 1977; Sabidussi 1966], similarity [Cohen et al. 2013; Liben-Nowell and Kleinberg 2003] and community search [Backstrom et al. 2006], which require distances to be computed for a large number of vertex pairs in order to perform social network analysis.

The classical approach to compute the shortest-path distance between a pair of vertices in a graph is to run an instance of the Dijkstra's algorithm [Tarjan 1983] for non-negative weighted graphs or breadth-first search (BFS) algorithm for unweighted graphs. However, these algorithms are inefficient and cannot scale to graphs with millions of vertices and edges (i.e., million-scale graphs). For such graphs, they may take several seconds or even longer to find the shortest-path distance between a single pair of vertices. This is not acceptable for applications such as context-aware web

search or socially-sensitive search since they involve real-time interactions between users and thus have low latency requirements (i.e., require distances to be provided in the order of microseconds or milliseconds), because they need shortest-path distances between a large fraction of vertex pairs to rank items for each search query. To improve query response time, a well-established approach is to pre-compute and store shortest-path distance information for all pairs of vertices in a data structure, called *distance labelling*.

1.1.1 Complex Networks

There are two major classes of real-world networks that have been attracting the interest of researchers in database community. The first one is road networks and the other is complex networks. Research focusing on road networks has been very successful because they often have a (hierarchical) structure which helps researchers to easily grasp and exploit their topology [Geisberger et al. 2012; Abraham et al. 2011; Abraham et al. 2012; Goldberg and Harrelson 2005]. In [Abraham et al. 2011], a *distance query* on a Western Europe road network with 18 million vertices and 22.5 million road segments takes 276 nanoseconds and on a USA road network with 24 million vertices and 29.1 million road segments can be processed in 266 nanoseconds, on average. In contrast, answering distance queries on complex networks is still a highly challenging problem.

Complex networks are quite prevalent in various disciplines, which are used to model complicated connections between different real-world entities. The examples of complex networks include social networks, World Wide Web networks, and computer networks. In a social network, users are represented as the set of vertices, and different types of relationships between users such as friendship, following, messaging and endorsement are represented as the set of edges of the network. The World Wide Web forms a complex network where web pages are the vertices, and the hyperlinks among web pages form the edges. In a computer network, computers are connected and communicated through routers, subnets, interfaces, and network locations which form a complicated network infrastructure. All of these complex networks have a non-trivial topological structure which is very difficult to grasp and exploit. However, they all share several characteristics. One of the interesting characteristics is the small-world phenomenon [Watts and Strogatz 1998], which states that the longest distance between any pair of vertices in a small-world network is usually a small constant. Furthermore, navigability [Kleinberg 2000] is one of the important characteristics in small-world networks. Complex networks also share other characteristics such as clustering coefficient and power-law degree distribution, which can be useful to understand the non-trivial nature of such networks.

Answering distance queries on complex networks requires researchers to have a good understanding of their topological structure when designing efficient methods. Methods for road networks do not work well for complex networks because they are designed based on different understandings in the topological structure of road networks. Many exact and approximate methods [Abraham et al. 2012; Akiba et al.

2013; Akiba et al. 2012; Wei 2010; Jin et al. 2012; Potamias et al. 2009; Tretyakov et al. 2011; Gubichev et al. 2010] have been proposed for complex networks, but they all suffer from the drawback of scalability. The exact distance querying methods often take tens to thousands of seconds to construct a distance labelling for networks with millions of vertices and edges [Akiba et al. 2013; Wei 2010; Jin et al. 2012; Hayashi et al. 2016; Fu et al. 2013]. The approximate methods normally loose precision for close pairs of vertices thus cannot be considered by applications such as socially-sensitive search or context-aware search since these applications distinguish close objects [Akiba et al. 2012; Qiao et al. 2014]. Some of them also claim better precision at the the cost of longer time to perform a distance query [Qiao et al. 2014; Tretyakov et al. 2011; Gubichev et al. 2010].

1.1.2 Dynamic Networks

Networks in real-world applications are typically dynamic, undergoing discrete changes in their topological structure by either adding or deleting edges and vertices. For example, road networks may be dynamically changing in many real-world scenarios such as changing traffic conditions under congestion or accidents, road construction or signal placements. Complex networks such as social networks are also reported to be highly dynamic [Myers and Leskovec 2014; Kumar et al. 2006; Xu et al. 2013], thereby requiring distance information to be dynamically updated in order to perform social network analysis accurately i.e., to find closeness and similarity between users and contents [Vieira et al. 2007; Yahia et al. 2008]. In social networks, new users may sign up, existing users may leave, and links between users may change dynamically. Similarly, communication networks may have faults being detected and recovered, and web graphs may have invalid links or new links being added as the web evolves. We discuss a few examples of real-world applications/scenarios in detail that require shortest-path distance computation under dynamic changes below.

- In communication networks, links between network devices (e.g. routers) may become slow or broken due to congestion of information flow over a network or a deadly fault in a network device. Efficient maintenance of shortest paths to reflect the underlying changes helps vendors to activate new links and preserve the quality of their service [Boccaletti et al. 2006].

- In social networks, Twitter is highly dynamic [Myers and Leskovec 2014] – about 9% of all connections change in a month. Users having 100 followers on average were found to obtain 10% more new followers but lose about 3% of existing followers in a given month. Distance information is often used to recommend the relevant content or new connections [Yahia et al. 2008; Vieira et al. 2007].

1.2 Research Objectives

This thesis primarily focuses on addressing the distance query problem over networks that are typically large (with billions of vertices and edges), complex (with non-trivial topological structure), and either static (with unchanged topological structure) or dynamic (with changes in their topological structure over time).

Current state-of-the-art research for answering distance queries has directed considerable attention towards the development of distance labelling methods. They capture distance information between *all* pairs of vertices in a graph into a distance labelling and typically called as *full distance labelling*. Then, this full distance labelling can be used to answer a *distance query* (i.e., find the distance between any two vertices) in constant time by just using a pre-computed distance information in the labelling. Although these distance labelling methods [Akiba et al. 2013; Li et al. 2019; Li et al. 2017] often provide promising query response time, they cannot scale to very large networks due to the quadratic growth in labelling sizes, leading to very high space requirements and unbearably long construction time. We will show in Tables 4.3 and 5.3, PLL [Akiba et al. 2013] has failed to construct distance labelling on networks with over hundreds of millions of edges, and the parallel variant of PLL [Li et al. 2019] has failed to construct distance labelling on networks with over 1.2 billions of edges.

Objective I - The first research objective of this thesis is to develop a scalable solution for answering exact distance queries over billion-scale static graphs which has the following desirable characteristics:

(1) *Time efficiency:* Construction time of a distance labelling is scalable with the size of a graph. The query time remains in the order of milliseconds, even in graphs with billions of nodes and edges.

(2) *Space efficiency:* Size of a distance labelling is minimized so as to reduce the space overhead.

(3) *Scalability:* Construction time and labelling size is scalable to graphs with billions of vertices and edges.

When a graph dynamically changes by edge insertions and deletions, its distance labelling needs to be changed accordingly; otherwise, distance queries may yield underestimated or overestimated distances. Thus, after an edge is inserted into or deleted from a graph, one could naturally consider either of the following two choices in order to be able to answer distance queries correctly:

(1) recompute a distance labelling from scratch;

(2) conduct an online search on the changed graph.

However, both of these approaches are very inefficient. Take graphs with a couple of millions of vertices such as Livejournal [Leskovec and Sosič 2016] and Hollywood

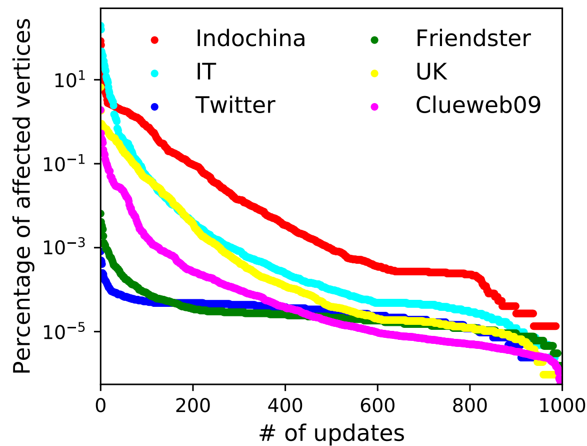


Figure 1.1: Distribution of affected vertices by a single graph change in various networks, where the results for 1000 randomly selected graph changes, containing 50% edge insertions and 50% edge deletions are sorted in the descending order.

[Boldi and Vigna 2004] for example, distance labelling methods may require significant amount of time, typically tens to thousands of seconds, to recompute a distance labelling from scratch as a result of a single change. Furthermore, if only a very small portion of a graph is affected against a change, recomputing a distance labelling from scratch would not only waste computing resources, but also prevent the availability of distance queries during recomputing. As shown in Figure 1.1, the percentage of affected vertices by a single change often ranges from 10^{-5}% to 10% in various real-world complex networks. On the other hand, answering a distance query entirely based on online search is often too slow to be useful in time-sensitive applications; for example, it takes about 30 seconds on average to answer a distance query on a Twitter network with 42 millions of vertices [Boldi and Vigna 2004].

Objective II - The second research objective of this thesis is to develop scalable and efficient solutions that can quickly reflect dynamic changes such as edge insertions and edge deletions in the single-update setting (i.e., one edge insertion or deletion at a time) into graphs. The proposed solutions should be able to answer exact distance queries efficiently and accurately on dynamic graphs and must have the following desirable characteristics:

– *Fully dynamics:* handle both types of changes i.e., edge/node insertion and deletion on a dynamic graph.

– *Time efficiency:* answer exact distance queries and update distance labelling as a result of changes on a dynamic graph efficiently i.e., in the order of milliseconds.

– *Space efficiency:* guarantee the minimum size of a distance labelling under dynamic changes to reduce storage space.

– *Scalability:* scale to very large networks with billions of vertices and edges without compromising query and update performance.

Objective III - The third research objective of this thesis is to develop an efficient yet scalable method to process edge insertions and deletions in the batch-update setting (i.e., multiple edge insertions or deletions together in a batch). The proposed solution must also possess the aforementioned characteristics. We aim to explore the following research questions that are critical in designing such a batch-dynamic method to answer exact distance queries efficiently and accurately:

– Is it possible to design batch-dynamic algorithms for distance queries, which can efficiently reflect batch updates on graphs?

– Can such batch-dynamic algorithms offer significant performance gains in comparison with state-of-the-art algorithms in the single-update setting?

– Can we parallelize such batch-dynamic algorithms to further boost performance in a parallel setting, whenever parallel computing resources are available?

1.3 Research Challenges

Recent research [Hayashi et al. 2016; Li et al. 2017; Li et al. 2019] has shown that the labelling based methods are the fastest methods for answering exact distance queries on million-scale graphs. However, they cannot scale to large graphs with billions of vertices and edges (i.e., billion-scale graphs) due to quadratic space requirements and unbearably long labelling construction time. Thus, the question is still open as to how scalable solutions to answer exact distance queries in billion-scale graphs can be developed. Essentially, there are three computational factors to be considered concerning the performance of algorithms for answering distance queries: labelling construction time, labelling size, and querying time. A plethora of existing work [Abraham et al. 2011; Abraham et al. 2012; Akiba et al. 2013; Akiba et al. 2012; Wei 2010; Hayashi et al. 2016; Tretyakov et al. 2011; Potamias et al. 2009; Fu et al. 2013; Jin et al. 2012; Qiao et al. 2014; Gubichev et al. 2010; Li et al. 2017; Chang et al. 2012; Delling et al. 2014] has focused on exploring trade-offs among these computational factors for a distance labelling that can be constructed using a variety of different properties such as 2-hop cover property [Cohen et al. 2002; Akiba et al. 2013], tree-width decomposition [Wei 2010; Akiba et al. 2012], or highway based property [Jin et al. 2012; Akiba et al. 2014]. Despite extensive efforts in addressing the shortest-path distance query problem for many years, there is still a high demand for scalable solutions that can be used to support analysis tasks over large and ever-growing networks.

Previously, the primary focus of these studies is to answer distance queries on static graphs, with limited attention being paid to dynamics on graphs. It has been reported [Akiba et al. 2014; D'angelo et al. 2019] that designing a fully dynamic

method for answering distance queries is very challenging. The difficulty of updating a distance labelling lies in two aspects:

(1) When adding an edge into a graph, outdated and redundant entries of distance labelling may occur. Although outdated and redundant entries do not affect the correctness of distance query answering, they would deteriorate the query performance over time. However, identifying and removing such entries is known to be a complicated task [Akiba et al. 2014].

(2) When deleting an edge from a graph, outdated distance entries have to be removed; otherwise distance queries cannot be correctly answered. Hence, entries of distance labelling being affected must be accurately identified and repaired with new distances. However, finding the new distances between affected vertices is computationally expensive and indeed much more challenging than the case of adding an edge into a graph [Qin et al. 2017; Akiba et al. 2014; D'angelo et al. 2019].

Both aspects, in a nutshell, require us to pinpoint affected vertices so as to update their labels efficiently. Moreover, although query time and update time are both critical for answering distance queries on dynamic graphs, it is not easy, if not impossible, to design a solution that is efficient in both. This requires us to find new insights into dynamic properties of a distance labelling, as well as a good trade-off between query time and update time. Last but not least, scaling distance queries to dynamic graphs with billions of nodes and edges is hard. Previous work [Akiba et al. 2014; Qin et al. 2017; Hayashi et al. 2016; D'angelo et al. 2019] has mostly considered 2-hop distance labelling [Cohen et al. 2002], which has quadratic space requirements and unbearably long labelling construction time; as a result, their query and update performance is dramatically degraded on large-scale dynamic graphs. Ideally, the labelling size of a graph should be much smaller than its original size. This is important for achieving efficiency in updating a distance labelling on large dynamic graphs because of its small size. However, the state-of-the-art distance labelling method, i.e. pruned landmark labelling method (PLL) [Akiba et al. 2013], still yields a distance labelling whose size is several orders of magnitude larger than the original size of a dataset. Furthermore, distance labelling methods for dynamic graphs [Akiba et al. 2014; D'angelo et al. 2019; Qin et al. 2017] cannot perform updates efficiently because they update a distance labelling of very large size and require much more complicated analyses to reflect graph changes into a distance labelling that captures distance information of all pairs of vertices in a graph.

Up to now, several distance labelling methods for distance queries on dynamic graphs have been studied in the single-update setting, which handles one single update (e.g., edge insertion or edge deletion) at a time [Akiba et al. 2014; D'angelo et al. 2019; Qin et al. 2017; Hayashi et al. 2016]. Due to the rapid nature of data acquisition, it is often unrealistic to process graph changes sequentially in the single-update setting. Rather, updates may be aggregated in batches, and reflected into graphs in a batch-update setting [Dhulipala et al. 2020], i.e., process all the updates in a batch together. However, the batch-update setting poses significant challenges on algorithm

design due to the combinatorial explosion of different interactions possibly occurring among updates. Very recently, several batch-dynamic algorithms have been reported, mostly focusing on traditional graph problems such as graph connectivity [Acar et al. 2019], dynamic trees [Acar et al. 2020] and k-clique counting [Dhulipala et al. 2020]. As of yet, batch-dynamic algorithms for shortest-path distance have been left unexplored, despite the fact that computing the distance between an arbitrary pair of vertices (i.e., *distance queries*) is a fundamental problem in many real-world applications.

1.4 Contributions

Traditional distance labelling methods pre-compute a *full distance labelling* and are the fastest known methods for the distance query problem. However, as we have discussed before, they significantly suffer from the drawbacks of: (1) scalability when an input graph is large, and (2) efficiency when the underlying structure of an input graph is dynamic. To address these limitations, this thesis aims to develop distance labelling methods that pre-compute a *partial distance labelling* rather than a *full distance labelling*, which only capture the distance information of some essential pairs of vertices (e.g., the distance between a vertex and a landmark or between two landmarks) in a graph. Such a *partial distance labelling* then can be combined with an online search to answer exact distance queries.

1.4.1 Highway Cover Labelling For Distance Queries

We develop a scalable solution for answering exact distance queries to accomplish the first objective. Our solution is based on two ingredients: (i) a scalable algorithm for constructing a partial distance labelling, and (ii) a querying framework that supports fast distance-bounded shortest-path search on a sparsified graph. More specifically, we first develop a novel labelling construction algorithm that can scale to graphs at the billion-scale. We observed that, for a given number of landmarks, the distance entries from these landmarks to other vertices in a graph can be further minimized if the definition of 2-hop cover distance labelling is relaxed. Thus, we formulate a relaxed notion for labelling, called the *highway cover distance labelling*, and develop a simple yet scalable labelling algorithm that adds a significantly small number of distance entries into the label of each vertex. We prove that a distance labelling constructed by our labelling algorithm is minimal, and also experimentally verify that the construction process is scalable in Chapter 4.

Then, we formalize a querying framework for exact distance queries, which combines our proposed highway cover distance labelling with distance-bounded shortest-path searches to enable fast distance computation. This querying framework is capable of balancing the trade-off between construction time, index size and query time through an offline component (i.e. the proposed highway cover distance labelling) and an online component (i.e. distance-bounded searches). The basic idea is to select a small number of highly central landmarks that allow us to efficiently

compute the upper bounds of distances between all pairs of vertices using an offline distance labelling, and then conduct a distance-bounded search over a sparsified graph to find distances efficiently. We present experiments in Chapter 4 which show that the query time of distance queries within this framework is still in milliseconds for large graphs with billions of vertices and edges.

1.4.2 Fully Dynamic Labelling For Distance Queries

We choose to design our dynamic algorithms based on the highway cover distance labelling approach that is introduced in Chapter 4, rather than a full distance labelling [Akiba et al. 2013]. Previous distance labelling methods such as pruned landmark labelling (PLL) [Akiba et al. 2013] can efficiently answer distance queries using a *full distance labelling*; however, their labelling size grows quadratically with the size of a graph and the computational cost of updating such a labelling to reflect rapid changes is often unbearably high. Hence, we choose to combine offline labelling and online searching so as to leverage the advantages from both sides - accelerating query processing through a *partial distance labelling* that is of limited size but provides a good approximation to bound online searches. This brings two significant computational benefits: (i) labelling construction can scale to very large graphs; (ii) labelling maintenance can be efficiently handled on dynamic graphs.

To deal with the second objective, we develop an efficient method that consider the *single-update setting* for updates occurring on dynamic graphs. That is, it processes each update either an edge insertion or deletion individually i.e., one by one. Our proposed fully dynamic method efficiently updates a partial distance labelling to answer distance queries over large dynamic graphs. At its core, our proposed method incorporates two building blocks: (i) *incremental algorithm* for handling incremental update operations, i.e. edge insertion, and (ii) *decremental algorithm* for handling decremental update operations, i.e. edge deletion. These two building blocks are built in the highly scalable framework of distance query answering introduced in Chapter 4. To the best of our knowledge, our method is the first fully dynamic method that can scale to graphs with billions of vertices and edges, without compromising performance on query time and labelling size.

1.4.3 Batch-Dynamic Labelling For Distance Queries

Similarly, we choose to develop our batch-dynamic method based on the highway cover labelling approach presented in Chapter 4 due to the same reason as mentioned in the second contribution. To deal with the third objective, we propose a robust method that processes multiple updates in the batch-update setting to achieve better efficiency for performing both updates and distance queries in a way that reflects batch updates on a graph. Our propose batch-dynamic method, called BatchHL, dynamizes a highway cover distance labelling efficiently in order to reflect large batches of updates on a graph. BatchHL consists of two phases: (1) *Batch search* which finds vertices whose labels are affected by batch updates; (2) *Batch repair* which repairs the

labels of affected vertices to ensure correctness and minimality of labelling. To the best of our knowledge, this is the first study to develop a batch-dynamic solution for answering distance queries on large-scale graphs which has the following benefits:

- *Unifying edge insertion and deletion:* We explore the core properties shared by edge insertion and edge deletion in a batch. Based on this, we unearth an elegant pattern that unifies these two fundamental kinds of graph updates.
- *Avoiding unnecessary and repeated computation:* We analyse how updates interact with each other, and based on that, design pruning rules to reduce search and repair spaces so as to leverage the computational efficiency of batch updates.
- *Exploiting the potential of parallelism:* We parallelize batch search and batch repair in a simple but easy-to-implement way to speedup the performance.

1.5 Outline

The rest of this thesis is organised as follows. In Chapter 2, we present the notations and definitions used in this thesis. In Chapter 3, we provide a comprehensive literature review of the related work studied in this thesis by outlining the strengths and weaknesses of the existing approaches. In Chapter 4, we study the problem of answering exact distance queries on static networks. In Chapter 5, we study the problem of answering exact distance queries on dynamic networks in the single-update setting. In Chapter 6, we study the problem of answering distance queries on dynamic networks in the batch-update setting. In Chapter 7, we discuss extensions. We finally conclude the thesis and discuss future research opportunities in Chapter 8.

For each of Chapters 4, 5 and 6, we start with an overview of the specific problem studied in the chapter. We then propose the models and the algorithms for solving the problem. We finish each chapter by presenting our experimental results and a summary of the chapter.

Chapter 2

Preliminaries

This chapter presents the basic definitions and notations to be used throughout the thesis. Table 2 summarises the frequently used notations.

In this thesis, we model a network as a graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges representing relationships between vertices. We have $n = |V|$ and $m = |E|$. Without loss of generality, we assume that the graph G is an undirected and unweighted. We denote by $N(v)$ the set of neighbors of vertex $v \in V$, i.e. $N(v) = \{u \in V \mid (u, v) \in E\}$. We use $P_G(u, v)$ to denote the set of all shortest paths between u and v in G. Given a subset of vertices $V' \subseteq V$, the induced graph of V', denoted by $G[V']$, is a subgraph of G whose vertex set is V' and whose edge set consists of all of the edges in E that have both endpoints in V'.

The *distance* between two vertices s and t in G, denoted as $d_G(s, t)$, is the length of a shortest-path from s to t. We consider $d_G(s, t) = \infty$, if there does not exist a path from s to t.

Fact 1. *For any three vertices $s, u, t \in V$, the following* triangle inequalities *are satisfied:*

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

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

If $u \in P_G(s, t)$, then $d_G(s, t) = d_G(s, u) + d_G(u, t)$ holds.

Given a special subset of vertices $R \subseteq V$ of G, so-called *landmarks*, a *label* $L(v)$ for each vertex $v \in V$ is a set of *distance entries* $\{(r_i, \delta_L(r_i, v))\}_{i=1}^{|R|}$ where $r_i \in R$, $\delta_L(r_i, v) = d_G(r_i, v)$. We call $(r_i, \delta_L(r_i, v))$ the r_i*-label* of vertex v. The set of labels for all vertices in V, i.e., $\{L(v)\}_{v \in V}$, form a *distance labelling* over G. The *size* of a distance labelling is defined as $\sum_{v \in V} |L(v)|$.

In the literature, a distance labelling is often constructed following the 2-hop cover property [Cohen et al. 2002] which requires at least one vertex $(w, \delta_L(r, w)) \in L(u) \cap L(v)$ to be on a shortest-path between u and v.

Definition 1 (2-hop Cover Labelling)**.** *A distance labelling L over a graph $G = (V, E)$ is*

a 2-hop cover labelling *if the following holds for any two vertices* $u, v \in V$*:*

$$d_G(u,v) = min\{\delta_L(r_i,u) + \delta_L(r_j,v) \mid (r_i, \delta_L(r_i,u)) \in L(u), (r_j, \delta_L(r_j,v)) \in L(v)\} \tag{2.3}$$

Thus, for any two vertices $u, v \in V$, an exact distance query can be answered by only looking up the labels of u and v in a 2-hop cover labelling. We define $Q(s,t,L) = \infty$, if $L(s)$ and $L(t)$ do not share any landmark. If $Q(s,t,L) = d_G(s,t)$ holds for any two vertices s and t of G, L is called a *2-hop cover distance labelling* over G [Cohen et al. 2002; Abraham et al. 2012]. Given a graph G, the complexity of finding a minimal 2-hop cover labelling of G is known to be NP-hard [Cohen et al. 2002].

We consider two fundamental types of updates on graphs, *edge insertion* and *edge deletion*. Given a graph $G = (V, E)$, an *edge insertion* is to add an edge (a, b) into G where $\{a, b\} \subseteq V$ and $(a, b) \notin E$. Conversely, an *edge deletion* is to delete an edge (a, b) from G where $(a, b) \in E$. We define two settings to reflect such updates on graphs, *unit update setting* and *batch update setting*.

- In the *unit update setting*, a sequence of edge insertions and deletions are processed one by one. In this setting, we consider vertex insertion and vertex deletion as a sequence of edge insertions and edge deletions, respectively. When inserting a new vertex, we first create an isolated vertex and then insert edges incident to it, and vice versa for deleting a vertex.
- In the *batch update setting* a sequence of edge insertions and deletions are processed altogether. In this setting, we consider node insertion or deletion as a batch update containing only edge insertions or only edge deletions, respectively.

In the case that the same edge is being inserted and deleted within one batch update, we simply eliminate both of them. An update is *valid* if it makes a change on a graph, i.e., inserting an edge (a, b) into G when $(a, b) \notin E$, and deleting an edge (a, b) from G when $(a, b) \in E$. We ignore invalid updates. We use $G \hookrightarrow G'$ to explicitly indicate that a graph G is changed to a graph G' by a unit update or a batch update.

The following facts are important for designing dynamic algorithms. They state that an edge insertion may decrease distances between vertices, and conversely an edge deletion may increase distances between vertices.

Fact 2. *Let* $G' = (V, E \cup \{(u, v)\})$ *be the graph after inserting an edge* (u, v) *into* $G = (V, E)$*. Then for any two vertices* $s, t \in V$, $d_G(s, t) \geq d_{G'}(s, t)$*.*

Fact 3. *Let* $G' = (V, E \cup \{(u, v)\})$ *be the graph after deleting an edge* (u, v) *from* $G = (V, E)$*. Then for any two vertices* $s, t \in V$, $d_G(s, t) \leq d_{G'}(s, t)$*.*

Table 2.1: Summary of notations.

Notation	Description
$G = (V, E)$	A graph G with the set of vertices V and edges E
n	Number of vertices in a graph $G = (V, E)$
m	Number of edges in a graph $G = (V, E)$
R	A subset of vertices, also called landmarks
L	A distance labelling
$L(v)$	Label of a vertex v
H	A highway
$\Gamma = (H, L)$	A highway cover labelling
P_{st}	Set of vertices in a shortest-path between s and t
$N_G(v)$	Set of vertices adjacent to v in a graph G
$d_G(u, v)$	Length of a shortest-path between u and v in a graph G
$d^*_{G'}(r, v)$	Contingent distance i.e., the minimum distance through unaffected neighbors of v to landmark r
$P_G(u, v)$	Set of all shortest paths between u and v in a graph G
$\delta_H(u, v)$	Highway distance between u and v
$\delta_L(u, v)$	Labelling distance between u and v
$\delta_{BFS}(u, v)$	BFS distance between u and v
$G \hookrightarrow G'$	A graph G is changed to a graph G' by an edge insertion or an edge deletion
$G[V']$	An induced subgraph with vertex set V' and edge set with end-points in V'
$G[V \backslash R]$	A sparsified graph after removing R from G
$d^{\top}_{uv}$	An upper distance bound between s and t
$Q(u, v, \Gamma)$	An exact query between two vertices u and v using highway cover labelling Γ
Λ_r	Set of affected vertices w.r.t. a landmark $r \in R$
V_{AFF}	Set of composite-path affected (CP-affected) vertices w.r.t. a landmark $r \in R$
$V_{\text{AFF+}}$	Set of landmark-distance affected (LD-affected) vertices w.r.t. a landmark $r \in R$
B	A batch update
$\mathbb{N}$	Length of a path p
$\mathbb{B}$	Deletion flag, True iff a path p passes through a deleted edge, otherwise False
$\oplus$	Append operator to update the landmark length of a path
$d_{\text{BOU}}(v, S)$	Distance bound of a vertex v w.r.t. a set S
$d^L_{\text{BOU}}(v, S)$	Landmark distance bound of a vertex v w.r.t. a set S
$\|S\|$	Number of elements in a set S

Chapter 3

Literature Review

This chapter provides the background and related work on the *distance query* problem. Specifically, this chapter discusses different types of methods, i.e., search-based methods, labelling-based methods, and hybrid methods, which have been developed in the literature for distance computation over static and dynamic complex networks.

3.1 Answering Distance Queries on Static Graphs

We first review the work related to static graphs whose topological structure remains unchanged over time.

3.1.1 Search-based Methods

A traditional approach for exact shortest-path distance computation is to run the Dijkstra's search for non-negative weighted graphs in $O(|E| + |V|log|V|)$ time or breadth-first search (BFS) for unweighted graphs in $O(E)$, from a source vertex to a destination vertex [Tarjan 1983]. To improve search efficiency, a bidirectional scheme can be used to run two such searches: one from the source vertex and the other from the destination vertex [Pohl 1969]. However, these traditional approaches fail to achieve desired response time performance required by many real-world applications that operate on increasingly large graphs.

Generally, two types of methods have been proposed in the literature which can improve the distance query response time tremendously i.e., labelling-based methods and hybrid methods. In the following we discuss them in detail.

3.1.2 Labelling-based Methods

Labelling-based methods have emerged as an attractive way of accelerating response time to distance queries [Cohen et al. 2002; Chang et al. 2012; Abraham et al. 2012; Jin et al. 2012; Fu et al. 2013; Akiba et al. 2013; Wei 2010; Farhan et al. 2019; Chang et al. 2012; Abraham et al. 2011]. Most of these methods constructed a labelling based on the 2-hop cover property [Cohen et al. 2002]. The 2-hop cover framework was proposed by Cohen et al. [Cohen et al. 2002] for the purpose of computing a distance labelling over a graph in order to answer distance queries efficiently. For each vertex

v, it stores a list of intermediate vertices u in $L_{out}(v)$ that can be reached by v along with their shortest-path distances, and a list of vertices u in $L_{in}(v)$ that can reach v along with their distances. The size of labelling is determined by the total number of intermediate vertices stored in the label(s) of each vertex. Then to answer a distance query for a pair of vertices s and t, we only need to check vertices $u \in L_{out}(s) \cap L_{in}(t)$ and pick one minimizing $d_G(s,u) + d_G(u,t)$ because the labels obey the *2-hop cover property*. This property requires that for any two vertices s and t, there must exist at least one vertex in $L_{out}(s) \cap L_{in}(t)$ on one shortest-path from s to t. It has also been shown that computing a minimal 2-hop cover labelling is NP-hard [Abraham et al. 2012; Cohen et al. 2002].

Generally, the labelling-based methods can be categorized into two classes based on different types of networks such as complex networks and road networks which we discuss in detail in the following.

A tremendous amount of research focusing on labelling-based methods [Cohen et al. 2002; Chang et al. 2012; Abraham et al. 2012; Fu et al. 2013; Akiba et al. 2013; Wei 2010; Chang et al. 2012; Potamias et al. 2009; Tretyakov et al. 2011; Gubichev et al. 2010; Li et al. 2019] has been conducted for shortest-path distance computation on complex networks. In [Cohen et al. 2002], Cohen et al. proposed an algorithm which can compute a 2-hop cover labelling in $O(n^4)$ with size no larger than the minimum possible size of a 2-hop cover labelling and whose average label size is within a factor of $O(\log n)$. However, the labelling construction time of this algorithm is very high in practice thus making it limited to small-size networks with only thousands of vertices and edges. Cheng and Yu [Cheng and Yu 2009] proposed a heuristic-based algorithm to construct a 2-hop distance labelling on directed graphs. Their method used the property of strongly connected components to exploit graph partitioning techniques. However, such a graph partitioning process introduces high computational time cost because it has to find vertex separators recursively. Furthermore, their method is limited to handle only directed graphs.

Tree decomposition based approaches [Wei 2010; Akiba et al. 2012] have also been studied for answering distance queries on graphs. Wei [Wei 2010] proposed an index for shortest-path query answering, called TEDI, which heuristically decomposes a graph into a tree through tree decomposition. Given a graph G, a tree decomposition of G yields a tree T in which each vertex is associated with a set of vertices in the graph G (also called a *bag* [Robertson and Seymour 1984]). The shortest-path distances between all pairs of vertices in the same bag are pre-computed and stored in the corresponding bags. Then, given a distance query, a bottom-up operation along the tree T can be carried out to answer the distance query. Further, Akiba et al. [Akiba et al. 2012] proposed an improved TEDI index that exploits a core-fringe structure in a graph. However, due to the presence of core-fringe structure in complex networks [Callaway et al. 2000; Newman et al. 2001], these methods may produce bags of large sizes in a decomposed tree and computing pairwise distances of vertices in these large bags can require long pre-processing time and huge storage space, making it impractical on large graphs. Moreover, the decomposition time of a large graph is very costly and only small-sized graphs can be processed within a

reasonable amount of time.

Hierarchical hub-labelling (HHL) was proposed by Abraham et al. [Abraham et al. 2012], which is based on a partial order of vertices. In their method, they use a top-down approach to compute a partial order of vertices that can produce a small sized HHL. However, their method is not scalable to handle large graphs due to very high storage and computation requirements for finding a partial order of vertices. Another method called Highway Centric labelling (HCL) was proposed by Jin et al. [Jin et al. 2012], which exploits highway structure of a graph by finding a spanning tree that can be used as a highway to efficiently compute 2-hop distance labelling for fast distance computation.

Fu et al. proposed IS-Label [Fu et al. 2013] which gained a significant scalability in precomputing a 2-hop cover distance labelling for large graphs in a memory-constrained environment. IS-Label uses the notion of an independent set of vertices in a graph. Their method preprocesses a graph to efficiently compute a distance labelling. It recursively computes independent sets of vertices and augments edges by removing these sets to preserve the distance information. Akiba et al. proposed the pruned landmark labelling (PLL) [Akiba et al. 2013] to pre-compute a 2-hop distance labelling by performing a pruned breadth-first search from every vertex. The idea is to prune vertices whose distance information can be obtained using the partially available 2-hop distance labelling constructed via previous breadth-first searches. This work helps to achieve low construction cost and small labelling size on million-scale networks and serves as the state-of-the-art labelling-based method for distance queries. Recently, Li et al. [Li et al. 2019] developed a parallel method called parallel shortest distance labelling (PSL) for constructing PLL in parallel to increase scalability for answering distance queries on graphs with billions of edges. Apart from these 2-hop distance labelling techniques, a multi-hop distance labelling approach has also been studied in [Chang et al. 2012], which reduces the size of labelling at the cost of increased response time.

Furthermore, many labelling-based approximation methods [Potamias et al. 2009; Tretyakov et al. 2011; Qiao et al. 2014; Gubichev et al. 2010; Das Sarma et al. 2010] have also been studied to enhance scalability at the cost of loosing accuracy. The general idea is to select a small set of landmarks R and pre-compute the shortest-path distances from each landmark $r \in R$ to all other vertices $v \in V$ in a graph. Then to answer the distance between a pair of vertices (s, t), they return $\mathtt{min}_{r \in R}\{\delta(s,r) + \delta(r,t)\}$ as the distance estimation. The quality of a *distance query* depends upon the presence of landmarks on the shortest-paths between query pairs. In [Potamias et al. 2009], Potamias et al. attempted to find a minimum number of landmarks such that for any pair of vertices (s, t) in G, there exists at least one landmark on a shortest-path from s to t. They call it the LANDMARK-COVER problem which guarantees the exact distance against a query search. Further, they show that this problem is NP-hard and can be solved by a set-cover framework. Motivated by this, they then studied different heuristics in order to select highly central landmarks and verified that these heuristics provide better estimation accuracy in comparison with basic landmark selection proxies such as selecting landmarks randomly and based on highest degree.

These methods can answer distance queries efficiently but it is reported that they do not have a high precision for close pairs of vertices [Tretyakov et al. 2011; Qiao et al. 2014] which might be critical for applications such as socially sensitive search or context-aware web search since these applications distinguish close objects using distance queries.

Other labelling-based approximation methods include distance-sketch [Das Sarma et al. 2010], a distance oracle which can provide fairly accurate estimation on real-world web graphs. Gubichev et al. [Gubichev et al. 2010] proposed a path-sketch, a distance oracle which not only discovers distances but also discovers shortest paths in large graphs [Gubichev et al. 2010]. They observed that the average path lengths in complex (small-world) networks are usually small enough to be considered as almost constant and therefore store shortest path information in addition to a distance labelling. In their method, they introduced several techniques, such as *cycle elimination* and *tree-based search*, in order to improve the distance query accuracy. Later on, many researchers focused on improving the accuracy of distance query estimation [Tretyakov et al. 2011; Qiao et al. 2014]. Tretyakov et al. [Tretyakov et al. 2011], by following the same line of thoughts further attempted to improve the accuracy and proposed a method which by using highly central landmarks computes a path-sketch for large networks. In order to approximate the distance between any two vertices, this method extracts all the shortest-paths from a path-sketch, i.e., from query vertices to all the landmarks and improves the answer by finding loops and shortcuts during BFS traversal on an extracted subgraph. Although these methods significantly improve the accuracy but they are slower up to three orders of magnitude than other methods [Potamias et al. 2009; Vieira et al. 2007].

Labelling based methods for road networks have also been studied with great success. Abraham et al. [Abraham et al. 2010] has discovered that several of the fastest distance computation algorithms for road networks [Geisberger et al. 2008; Sanders and Schultes 2005; Bast et al. 2007; Bauer et al. 2010] perform well on graphs with small *highway dimension*. A graph has small *highway dimension* if for every $r > 0$, there is a sparse set of vertices S_r such that every shortest-path of length greater than r includes a vertex from S_r [Abraham et al. 2010]. A set is sparse if every ball of radius $O(r)$ contains a small number of elements of S_r. Further, they demonstrated that the method with the best time bound for distance computation on road networks is a 2-hop cover based labelling algorithm. In [Abraham et al. 2011], they proposed a hub-based labelling algorithm which heuristically constructs a distance labelling on large road networks by processing contraction hierarchies. Contraction hierarchy algorithm proposed in [Geisberger et al. 2008] defines a total order among vertices as a vertex hierarchy by assigning each vertex v a rank $r(v)$ using some predefined criteria and then applies a shortcut operation to each vertex in this order. When preprocessing a vertex v, the shortcut operation temporarily removes v from the graph and adds shortcut edges between its neighbors to preserve the shortest-path distances. The output of contraction hierarchy algorithm is a graph $G^+ = (V, E \cup E^+)$ (where E^+ is the set of shortcut edges added) and the order in which vertices were preprocessed. During a label generation process, for a given vertex v, hub-based

labelling algorithm [Abraham et al. 2011] runs a forward contraction hierarchy search and a backward contraction hierarchy search on G^+ from v and adds visited vertices in $L_f(v)$ and $L_b(v)$, respectively. Then, given any two vertices s and t, the intersection of $L_f(s)$ and $L_b(t)$ contains the maximum-rank vertex u on a shortest-path between s and t, and $\delta(s,u)+\delta(u,t)$ will be the shortest-path distance from s to t. Further, it reduces the size of labels by removing vertices w from $L_f(v)$ and $L_b(v)$ if $\delta(v,w) > d_G(v,w)$. More precisely, if a vertex $u \in L_f(s) \cap L_b(t)$ with $\delta(s,u) = d_G(s,u)$ and $\delta(u,t) = d_G(u,t)$, u will not be removed. The quality of the labelling depends on the vertex order for contraction hierarchy.

Another method called *pruned highway labelling* was proposed by Akiba et al. [Akiba et al. 2014]. This method aims to encode more information in each label, i.e., store distances to paths instead of hubs in the labels for each vertex. It first decomposes a road network into disjoint shortest paths and then computes a label for each vertex v, which contains the distance from v to vertices in a small subset of the computed paths. Using these labels, a distance query $d_G(s,t)$ can be answered by hopping from s to a path starting from a vertex $u \in L(s) \cap L(t)$ and then hopping from the path to t. Very recently, Dian et al. [Ouyang et al. 2018] proposed *Hierarchical 2-Hop Index* (H2H-Index) which has shown to outperform previous state-of-the-art methods, hub labelling method and pruned highway labelling method. The general idea of this method is to design a label for each vertex as well as a hierarchy among all vertices in a road network. Then, it answers a shortest distance query $d_G(s,t)$ without exploiting all vertices in the labels of s and t, and only visit a subset of vertices in the labels of s and t attentively with the help of the vertex hierarchy. Intuitively, when the distance between s and t is small which can be achieved with the assistance of a vertex hierarchy, only a small subset of vertices in the labels of s and t need to be visited and thus resulting in faster query processing time.

3.1.3 Hybrid Methods

Goldberg et al. [Goldberg and Harrelson 2005] combined the bidirectional A* search algorithm with labelling techniques to improve the search performance. In their method, they pre-compute labelling based on landmarks to estimate the lower bounds, and used that estimate with a bidirectional A* search for efficient computation of shortest-path distances. However, this method is known to work only for road networks and does not scale well on complex networks.

IS-Label (IS-L) method discussed in Section 3.1.2 is generally regarded as a hybrid method which computes a partial distance labelling rather than a full distance labelling to have a good trade-off between three main computational factors discussed in Chapter 1. It combines a partial distance labelling with graph traversal to answer distance queries. Following the same line of thought, very recently, Hayashi et al. [Hayashi et al. 2016] proposed a fully dynamic (FD) method to accelerate shortest-path distance computation on million-scale complex networks. This work is most closely related to our work that is presented in Chapter 4. The key idea of the method in [Hayashi et al. 2016] is to select a small set of landmarks R and then

pre-computes shortest-path trees (SPTs) rooted at each $r \in R$. Given any two vertices s and t, it first computes the upper bound by taking the minimum length among the paths that pass through R. Then a bidirectional BFS between s and t is conducted on a graph $G[V \backslash R]$ to compute the shortest-path distances that do not pass through any landmark in R and take the minimum of these two results as the answer to an exact distance query between s and t. The experiments in [Hayashi et al. 2016] showed that this method can scale to graphs with millions of vertices and billions of edges, and outperforms the state-of-the-art exact methods PLL [Akiba et al. 2013], HDB [Jiang et al. 2014], RXL and CRXL [Delling et al. 2014] with significantly reduced construction time and labelling size, while the query times are higher but still remain acceptable.

Although the method proposed in [Hayashi et al. 2016] has been tested on a large network with millions of vertices and billions of edges, it still fails to construct labelling for billion-scale networks in general, particularly with billions of vertices. In contrast, the method we propose in Chapter 4 cannot only construct a distance labelling linearly with a set of landmarks in graphs with billions of vertices and edges, but also enable the size of a distance labelling to be significantly smaller than the original size of a graph. In addition to these, the deterministic nature of the distance labelling presented in Chapter 4 allows us to achieve further gains in computational efficiency using parallel BFSs w.r.t. multiple landmarks, which is highly scalable for handling billion-scale networks.

3.2 Answering Distance Queries on Dynamic Graphs

Now we review the work related to dynamic graphs that undergo discrete changes such as edge insertion and edge deletion in their topological structure over time.

3.2.1 Labelling-based Methods

A few attempts have been made to study distance queries over dynamic complex networks [Akiba et al. 2014; Qin et al. 2017; Hayashi et al. 2016; D'angelo et al. 2019; Ouyang et al. 2020], which are mostly based on the idea of dynamically maintaining a 2-hop cover labelling or its variants. To reflect graph changes, Akiba et al. [Akiba et al. 2014] studied the problem of updating pruned landmark labelling for incremental updates (i.e. vertex additions and edge additions). This work however does not remove outdated and redundant distance entries from the labels of affected vertices because the authors considered that detecting such entries is too costly. This inevitably breaks the minimality of pruned landmark labelling, leading to an ever increase in the size of a distance labelling, and deteriorates query performance over time. Qin et al. [Qin et al. 2017] and D'angelo et al. [D'angelo et al. 2019] studied the problem of updating a pruned landmark labelling for decremental updates (i.e. edge deletions). These methods can only scale to graphs with a few millions of nodes due to their high time complexities. Their experiments [Qin et al. 2017; D'angelo et al. 2019] showed that the average update time of an edge deletion on a graph with 19

millions of edges is 135 seconds in [Qin et al. 2017] and on a graph with 16 millions of edges is 19 seconds in [D'angelo et al. 2019], which are significantly inefficient. In the work by D'angelo et al. [D'angelo et al. 2019], they combined the algorithm for incremental updates proposed in [Akiba et al. 2014] with their method for decremental updates to form a fully dynamic algorithm. Nevertheless, this fully dynamic algorithm can only be applied to networks with around 20 millions of edges. A recent method by D'Emidio et al. [D'Emidio 2020] claims an improvement over the method proposed in [D'angelo et al. 2019] for decremental updates. However, this method is limited to graphs with few millions of nodes and updates labelling in the order of seconds. Nonetheless, all of these methods only considered the unit-update setting i.e., to perform updates one at a time. Very recently, Zhang et al. [Zhang et al. 2021] has also attempted to extend a recent parallel method (PSL) for constructing PLL [Li et al. 2019] to dynamic graphs for fast distance computation which unfortunately can only accommodate million-scale graphs.

Alternatively, methods for maintaining all-pair shortest paths (APSP) have been studied to allow direct look-up of the shortest-path distance at the cost of quadratic space and update time. Theoretically, the update time and space complexities of maintaining all-pair shortest paths (APSP) data structure are prohibitively very high and cannot scale to large graphs, e.g., the dynamic algorithm proposed in [Demetrescu and Italiano 2004] takes $\tilde{O}(n^2)$ amortized time per update operation and $O(n^3)$ space. A recent method [Gutenberg and Wulff-Nilsen 2020] proposed an improved bound in the form of a deterministic algorithm with $\tilde{O}(n^{2+2/3})$ update time and a Las-Vegas algorithm with $\tilde{O}(n^{2+1/2})$ update time for unweighted graphs having $\tilde{O}(n^2)$ space requirements. With these quadratic time and space requirements, they are not practical to be applied to large graphs having millions of vertices.

A labelling-based approximation algorithm proposed in [Tretyakov et al. 2011] can also reflect dynamic graph changes into distance estimation. They proposed algorithms to update the SPTs i.e., the distances of vertices from each landmark as a result of an edge insertion or edge deletion. Approximate methods for dynamically maintaining APSP have also been studied in order to improve the update time of maintaining APSP data structure. The work in [Roditty and Zwick 2004] maintains dynamic all-pair $(1+\epsilon)$ approximate shortest paths in $\tilde{O}(mn/t)$ update time and $\tilde{O}(n^2)$ space, where t is a parameter that describes the trade-off between the update time and query time. Another $(2+\epsilon)$ approximate algorithm by Bernstein et al. [Bernstein 2009] claimed a faster update time of $o(mn^{\epsilon} logR/\epsilon)$ for any fixed ϵ, where R is the ratio between the largest and the smallest edge weights. Unfortunately, their update times suffer the drawback of a super-polynomial dependence on ϵ.

Another line of research related to our work is streaming graph algorithms. In the streaming setting, a rapidly changing graph is often modeled using certain compressed data structures due to space constraints. Updates are received as a stream, but may be accumulated into batches through a sliding window and applied to the underlying graph. In this setting, a number of methods [McGregor 2014; Feigenbaum et al. 2005; Pacaci et al. 2020] have been proposed to address distance queries which operate under certain constraints, e.g., limited amount of memory and accuracy of

graph structure. Different from these streaming graph methods, our work considers applications which operate on batch-dynamic graphs that are explicitly stored and can be processed in the main memory of a single machine. Nevertheless, the ideas of our algorithm can be easily extended to deal with batch updates in the streaming setting.

In the past years, a good number of methods [Delling et al. 2017; Ouyang et al. 2020; Geisberger et al. 2012; Batz et al. 2009; Batz et al. 2010; Kontogiannis et al. 2016; Zhang et al. 2021] have studied the distance query problem on dynamic road networks. In real-world applications, road networks often have stable topological structure but their edge weights are dynamically updated due to changing traffic conditions. In [Geisberger et al. 2012], Geisberger et al. has proposed a method to maintain the index structure of the contraction hierarchy algorithm described in Section 3.1.2 for dynamic road networks. Their method first identifies affected vertices in the breadth-first search manner through the shortcuts generated by the contraction hierarchy algorithm during the pre-processing stage. After identifying all the affected vertices, their method then updates the weight of the affected edge and applies the vertex contraction operation on all the affected vertices following a total order of vertices. The resulting shortcut index is an updated index for the changed road network. A recent method [Ouyang et al. 2020] has been proposed as an improvement over the method developed in [Geisberger et al. 2012]. Instead of identifying affected vertices, the improved method aimed to identify affected shortcuts based on the observation that the weights of only very few shortcuts need to be changed when the weight of an edge is updated in a road network.

Previously, the shortest-path query problem, which is closely related to the distance query problem, has been studied in time-dependent road networks. In a time-dependent road network $G = (V, E, t)$, each edge (u, v) is assigned with a function $t : E \rightarrow \mathbb{R}^+$ specifying the time t to reach vertex v from vertex u. Then, the shortest-path query problem can be modeled on time-dependent road networks as follows. Given a time-dependent road network G, a source vertex s, and a target vertex t, *a time-dependent shortest-path query* returns a shortest path from s to t with the fastest travel time in G, and *a shortest travel time profile query* on G returns the departing time at which it takes the fastest travel time from s to t along with the corresponding shortest path. The method proposed in [Batz et al. 2009] pre-computes an index named *time-dependent contraction hierarchy* (TCH) which is a generalization of contraction hierarchy originally for static road networks to support shortest travel time profile queries in time-dependent road networks. After that, the method proposed in [Batz et al. 2010] has attempted to further improve scalability by reducing space requirements of TCH. Their method adopts an approximation based shortcut technique which leverages the idea of selecting a small set of landmarks and then computing the travel-time summaries from the landmarks towards all reachable vertices. Later on, several other indexing based methods [Kontogiannis et al. 2016; Kontogiannis et al. 2015; Kontogiannis et al. 2015] have been proposed to speed up shortest-path queries in time-dependent road networks.

3.2.2 Hybrid Methods

The work [Hayashi et al. 2016] discussed in Section 3.1.3 has also introduced a fully dynamic method for addressing the distance query problem which updates a partial distance labelling to reflect dynamic changes such as edge insertion and edge deletion into a graph. Specifically, this work proposes two algorithms separately: one for edge insertion and the other for edge deletion, which can be combined to process both edge insertions and deletions in the fully dynamic setting. This fully dynamic method, however, suffers from the drawbacks of: (1) scalability i.e., it fails to construct partial distance labellings for billion-scale graphs, and (2) update efficiency, i.e., it is slow to update a partial distance labelling in order to reflect changes on a graph, particularly for changes that are edge deletions.

In this thesis, we propose a fully dynamic method that can leverage the advantages of the highway cover distance labelling method proposed in Chapter 4 and also overcome the limitations of the previous methods for distance queries on large dynamic graphs. Our proposed fully dynamic method provides a novel, fast and scalable solution for answering distance queries on large and dynamic graphs. We further consider the batch-update setting for addressing the distance query problem on dynamic graphs. We propose a batch-dynamic method which is also based on the highway cover distance labelling method proposed in Chapter 4. Designing dynamic algorithms is usually quite complicated and difficult, as reported by the previous methods in the single-update setting, and arguably even more so in the batch-dynamic setting or parallel setting.

Chapter 4

HL: Highway Cover Labelling For Distance Queries

4.1 Overview

In this chapter, we study the problem of answering exact shortest-path distance queries in large static networks. We develop a scalable solution which has two main components: (i) a partial distance labelling, and (ii) a fast distance-bounded shortest-path search on a sparsified graph. In the first component, we propose a relaxed notion of 2-hop cover distance labelling, called the *highway cover distance labelling* and develop a novel algorithm that can efficiently construct a partial highway cover distance labelling for graphs with billions of vertices and edges. In the second component, we formalize a querying framework for processing exact distance queries, which combines a partial distance labelling with a distance-bounded shortest-path search to enable fast distance computation. These two components enable us to achieve a good trade-off between construction time, labelling size and query time.

Figure 4.1 summarizes the performance of our proposed method and the state-of-the-art methods for answering exact distance queries [Akiba et al. 2013; Fu et al. 2013; Abraham et al. 2012; Hayashi et al. 2016; Jiang et al. 2014; Tarjan 1983; Pohl 1969; Chang et al. 2012]. We can see in Figure 4.1(a) that offline labelling-based methods pruned landmark labelling (PLL) [Akiba et al. 2013], hop-doubling (HDB) [Jiang et al. 2014], and hierarchical hub labelling (HHL) [Abraham et al. 2012] can answer distance queries efficiently. However, they have huge space requirements and very long labelling construction time. On the contrary, traditional online search-based methods such as Dijkstra's [Tarjan 1983] and bidirectional breadth-first search (Bi-BFS) [Pohl 1969] have a very high response time thus are not applicable to large-scale networks where distances are required in the order of milliseconds. Unlike other labelling-based and online search-based methods, the hybrid-based methods fully dynamic (FD) [Fu et al. 2013], independent set based labelling (IS-L) [Hayashi et al. 2016] and highway labelling (HL) (our method) combine an offline distance labelling with an online graph traversal technique to provide a good trade-off between query response time and labelling size. Moreover, in Figure 4.1(b), we can see that only our proposed method HL can handle networks of size 8B and is thus scalable to perform

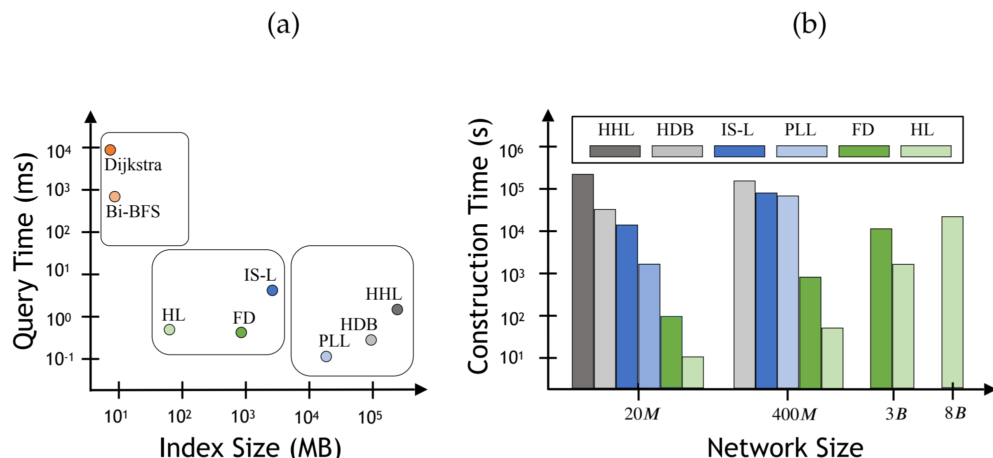


Figure 4.1: A high-level overview of the state-of-the-art methods and our proposed method (HL) for answering exact distance queries. Note that, Figure (a) is based on networks of size up to 400M edges.

exact distance queries on networks with billions of vertices and edges.

Table 4.1 presents several important properties of labelling-based methods. The column ORDERING DEPENDENT refers to whether a distance labelling depends on the ordering of landmarks when being constructed by a method. Only our method HL and FD are not ordering-dependent. The columns 2HC-MINIMAL and HWC-MINIMAL refer to whether a distance labelling constructed by a method is minimal in terms of the 2-hop cover (2HC) and highway cover (HWC) properties (described in detail in Section of Chapter 4), respectively. PLL is 2HC-minimal, but not HWC-minimal. Our method HL is the only method that is HWC-minimal. The column PARALLEL refers to what kind of parallelism a method can support. FD and PLL support bit-parallelism (mentioned in Section of Chapter 4) for up to 64 neighbors of a landmark. Our method HL supports parallel computation for multiple landmarks, depending on the number of available processors. Other methods did not mention any parallelism.

Table 4.1: Several important properties related to labelling-based methods.

Method	ORDERING-DEPENDENT?	2HC-MINIMAL?	HWC-MINIMAL?	PARALLEL?
HL (ours)	no	n/a	yes	landmarks
FD [Hayashi et al. 2016]	no	no	no	neighbors
IS-L [Fu et al. 2013]	yes	no	no	no
PLL [Akiba et al. 2013]	yes	yes	no	neighbors
HDB [Jiang et al. 2014]	yes	no	no	no
HHL [Abraham et al. 2012]	yes	no	no	no

The main contributions of this chapter are as follows,

- We introduce a new labelling property, namely *highway cover labelling*, which re-

laxes the notion of 2-hop cover labelling. Based on this new labelling property, we propose a highly scalable labelling algorithm that can scale to construct a distance labelling for graphs with billions of vertices and edges.

- We prove that the proposed algorithm can construct HWC-minimal labelling, which is independent of the ordering of landmarks. Then, due to this deterministic nature of labelling, we further develop a parallel algorithm which can run parallel BFSs from multiple landmarks to speed up labelling construction.

- We combine our novel labelling algorithm with an online bounded-distance graph traversal to efficiently answer exact distance queries. This querying framework enables us to achieve a good trade-off between construction time, labelling size and query time.

- We experimentally verify the performance of our proposed methods on 15 large real-world complex networks. The results show that our methods can not only handle networks with billions of vertices, but also be up to 70 times faster in constructing a distance labelling and save up to 90% of labelling space.

The rest of this chapter is organized as follows. Section 4.2 introduces the problem definition. Section 4.3 formulates the highway cover labelling problem and present a novel algorithm to efficiently compute a highway cover distance labelling over a graph G. Section 4.4 formulates the querying framework. Section 4.6 shows that the proposed algorithms are correct and preserve the property of minimality. Section 4.5 introduces several optimization techniques. Section 4.7 discusses the experimental results, which compare the performance of our proposed algorithms against the baseline algorithms. Section 4.9 summarises the chapter.

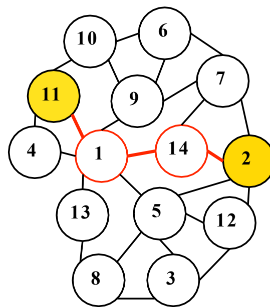


Figure 4.2: An example graph G where one shortest path between two vertices 2 and 11 is highlighted in red.

4.2 Problem Definition

Generally speaking, given any two vertices u and v in a graph $G = (V, E)$, a *distance query* for u and v is to find the minimum length of paths between these two vertices in the graph G.

Example 1. *In Figure 4.2, the distance (i.e., the minimum length of paths) between vertices* 2 *and* 11 *is* 3 *which can be obtained through the paths* $\langle 2, 5, 1, 11\rangle$ *and* $\langle 2, 14, 1, 11\rangle$.

In this thesis, we aim to construct a distance labelling of a limited size, which contains only distance information from all vertices in the graph to some "important" vertices (not all) - landmarks. Such a distance labelling is considered as a *partial distance labelling*. Then, we combine offline partial distance labelling with online searching to leverage the advantages from both sides - accelerating query processing through a small sized partial distance labelling that provides a good approximation to bound online searches. More formally, we define the *distance query* problem studied in this chapter as:

Definition 2 (Distance Query Problem). *Let* $G = (V, E)$ *be a graph and* $R \subseteq V$ *be a set of landmarks in the graph G. Then, the* distance query problem *is to efficiently compute the shortest-path distance* $d_G(s, t)$ *between any two vertices s and t in G, using a partial distance labelling* Γ *over G and online searching over* $G[V \backslash R]$.

4.3 Highway Cover Labelling

In this section, we formulate the highway cover labelling property and propose a novel algorithm to efficiently construct a partial distance labelling, which is a highway cover distance labelling over a graph G.

4.3.1 Highway and Highway Cover

We begin with the definition of highway, and then present the definition of highway cover.

Definition 3 (Highway). *A highway H is a pair* (R, δ_H), *where R is a set of landmarks and* δ_H *is a distance decoding function, i.e.* $\delta_H : R \times R \rightarrow \mathbb{N}^+$, *such that for any* $\{r_1, r_2\} \subseteq R$ *we have* $\delta_H(r_1, r_2) = d_G(r_1, r_2)$.

Given a landmark $r \in R$ and two vertices $s, t \in V \backslash R$ (i.e. $V \backslash R = V - R$), a *r-constrained shortest-path* between s and t is a path between s and t satisfying two conditions:

(1) It goes through the landmark r;

(2) It has the minimum length among all paths between s and t that go through r.

We use P_{st} to denote the set of vertices in a shortest-path between s and t, and P^r_{st} to denote the set of vertices in a r-constrained shortest-path between s and t.

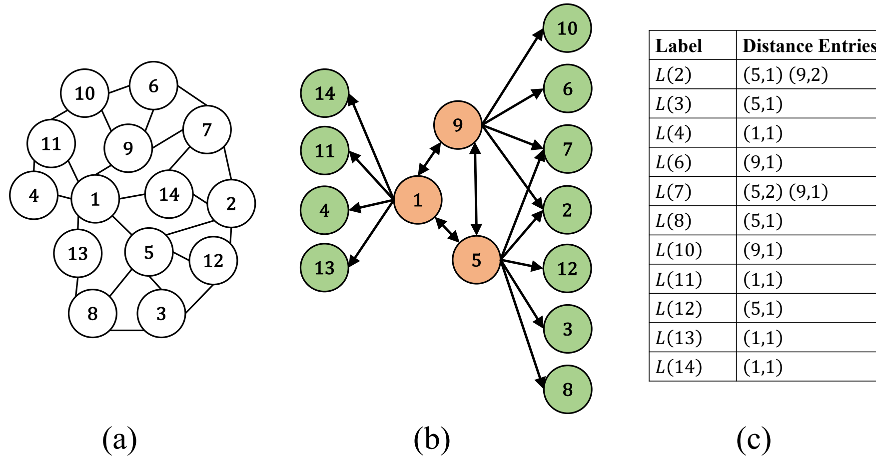
Definition 4 (Highway Cover). *Let* $G = (V, E)$ *be a graph,* $H = (R, \delta_H)$ *a highway and L a distance labelling over* (G, H). *Then for any two vertices* $s, t \in V \backslash R$ *and for any landmark* $r \in R$, *there exist* $(r_i, \delta_L(r_i, s)) \in L(s)$ *and* $(r_j, \delta_L(r_j, t)) \in L(t)$ *such that* $r_i \in P_{rs}$ *and* $r_j \in P_{rt}$, *where* r_i *and* r_j *may equal to r.*

If the label of a vertex v contains a distance entry $(r, \delta_L(r, v))$, we also say that the vertex v is *covered* by the landmark r in the distance labelling. Intuitively, the highway cover property guarantees that, given a highway H with a set of landmarks R and $r \in R$, any r-constrained shortest-path distance between two vertices s and t can be found using only the labels of these two vertices and the given highway. A distance labelling is called a *highway cover distance labelling* if it satisfies the highway cover property. More formally, a *highway cover distance labelling* is a pair $\Gamma = (H, L)$ where H is a highway and L is a distance labelling satisfying that, for any vertex $v \in V \backslash R$ and $r \in R$, we have:

$$d_G(r, v) = \mathtt{min}\{\delta_L(r_i, v) + \delta_H(r, r_i) \mid (r_i, \delta_L(r_i, v)) \in L(v)\} \tag{4.1}$$

Example 2. *Consider the graph G depicted in Figure 4.3(a), the highway H has three landmarks $\{1, 5, 9\}$ as highlighted in red in Figure 4.3(b). Based on the graph in Figure 4.3(a) and the highway in Figure 4.3(b), we have $\langle 11, 1, 4\rangle$ which is a shortest-path between the vertices 11 and 4 constrained by the landmark 1, i.e. 1-constrained shortest-path between 11 and 4. In contrast, neither of the paths $\langle 11, 10, 9, 1, 4\rangle$ and $\langle 11, 4\rangle$ is a 1-constrained shortest-path between 11 and 4.*

In Figure 4.3(b), the outgoing arrows from each landmark point to vertices in G that are covered by this landmark in the highway. The distance labelling in Figure 4.3(c) satisfies the highway cover property because for any two vertices that are not landmarks and any landmark $r \in \{1, 5, 9\}$, we can find the r-constrained shortest-path distance between these two vertices using their labels and the highway.



Label	Distance Entries
$L(2)$	(5,1) (9,2)
$L(3)$	(5,1)
$L(4)$	(1,1)
$L(6)$	(9,1)
$L(7)$	(5,2) (9,1)
$L(8)$	(5,1)
$L(10)$	(9,1)
$L(11)$	(1,1)
$L(12)$	(5,1)
$L(13)$	(1,1)
$L(14)$	(1,1)

Figure 4.3: An illustration of highway cover distance labelling: (a) an example graph G, (b) a highway H that connects to other vertices, and (c) a distance labelling that fulfills the highway cover property over G.

Definition 5 (Highway Cover Labelling Problem)**.** *Given a graph G and a highway H over G, the* highway cover labelling problem *is to efficiently construct a highway cover*

distance labelling $\Gamma = (H, L)$.

Several choices naturally come up for how to construct a distance labelling: (1) One is to add a distance entry for each landmark into the label of every vertex in $V \backslash R$, as the approach proposed in [Hayashi et al. 2016]; (2) Another is to use the pruned landmark labelling approach [Akiba et al. 2013] to add distance entries w.r.t. a landmark r into the labels of vertices in $V \backslash R$ if they can not be pruned during a BFS rooted at r; (3) We can also extend the pruned landmark labelling approach to construct highway cover labelling by replacing the 2-hop cover pruning condition with the highway cover condition as defined in Definition 4.

In all these cases, the labelling construction process would not guarantee scalability for large-scale complex networks with billions of vertices and edges. Moreover, these approaches would potentially lead to the construction of a distance labelling with different sizes w.r.t. different ordering of vertices. A question arising is: *how to construct a minimal highway cover distance labelling without redundant label entries?* In a nutshell, it is a challenging task to construct a highway cover distance labelling that can scale to very large networks, ideally in linear time, but also with the minimal labelling size.

4.3.2 Labelling Construction Algorithm

We propose a novel algorithm for solving the highway cover labelling problem, which can construct a highway cover distance labelling in linear time.

The key idea of our algorithm is to construct a *label* $L(v)$ for every vertex $v \in V \backslash R$ such that a distance entry $(r_i, \delta_L(r_i, v))$ of each landmark $r_i \in R$ is only added into the label $L(v)$ iff there does not exist any other landmark that appears in a shortest-path between r_i and v, i.e. $P_{r_i v} \cap R = \{r_i\}$. In other words, if there exists another landmark $r \in R$ and r_i is in the shortest-path between r and v, then $(r_i, \delta_L(r_i, v))$ is added into $L(v)$ iff r_i is the "closest" landmark from r to v. To compute such labels efficiently, we conduct a breadth-first search from every landmark $r_i \in R$ and add distance entries into labels of vertices that do not have any other landmark in their shortest paths from r_i.

Example 3. *Consider vertex* 7 *in Figure 4.3(c), the label* $L(7)$ *contains the distance entries of landmarks* $\{5, 9\}$*, but no distance entry of landmark* 1*. This is because* 5 *and* 9 *are the closest landmarks to vertex 7 in the shortest paths* $\langle 5, 7 \rangle$ *and* $\langle 9, 7 \rangle$*, respectively. However, for each of the two shortest paths* $\langle 1, 9, 7 \rangle$ *and* $\langle 1, 5, 7 \rangle$ *between* 1 *and* 7*, there is another landmark (i.e.* 5 *or* 9*) that is closer to* 7 *compared with* 1 *in these shortest paths. Thus the distance entry of landmark 1 is not added into* $L(7)$*.*

Our highway cover labelling approach is described in Algorithm 1. Given a graph G and a highway H over G, we start with an empty *highway cover distance labelling* L, where $L(v) = \emptyset$ for every $v \in V \backslash R$. Then, for each landmark $r_i \in R$, we compute the corresponding distance entries as follows. We use two queues $\mathcal{Q}_{label}$ and $\mathcal{Q}_{prune}$ to process vertices to be labeled or pruned at each level of a breadth-first search (BFS) tree, respectively. We start by processing vertices in $\mathcal{Q}_{label}$. For each vertex $u \in \mathcal{Q}_{label}$

at depth n, we examine the children of u at depth $n+1$ that are unvisited. For each unvisited child vertex $v \in N_G(u)$ at depth $n+1$, if $v \in R$ then we prune v, i.e., we do not add a distance entry of the current landmark r_i into $L(v)$ and we also enqueue v to the pruned queue $\mathcal{Q}_{prune}$ (Line 11). Otherwise, we add $(r_i, \delta_{BFS}(r_i, v))$ to the *label* of v, i.e., we add it into $L(v)$ and we also enqueue v to the labeled queue $\mathcal{Q}_{label}$ (Lines 13-14). Here, $\delta_{BFS}(r_i, v)$ refers to BFS decoded distance from root r_i to v. Then we process the pruned vertices in $\mathcal{Q}_{prune}$. These vertices are either landmarks or have landmarks in their shortest paths from r_i, and thus do not need to be labeled. Therefore, for each vertex $v \in \mathcal{Q}_{prune}$ at depth n, we enqueue all unvisited children of v at depth $n+1$ to the pruned queue $\mathcal{Q}_{prune}$. We keep processing these two queues, one after the other, until $\mathcal{Q}_{label}$ is empty.

Algorithm 1: Constructing the highway cover labelling Γ

```
   Input: G = (V.E), H = (R, δ_H)
   Output: L
 1 L(v) ← ∅, ∀v ∈ V\R
 2 foreach r_i ∈ R do
 3 |   Q_label ← ∅
 4 |   Q_prune ← ∅
 5 |   π ← 0
 6 |   Enqueue r_i to Q_label and set r_i as the root of BFS
 7 |   while Q_label is not empty do
 8 |   |   foreach u ∈ Q_label at depth π do
 9 |   |   |   foreach unvisited child v of u at depth π + 1 do
10 |   |   |   |   if v is a landmark then
11 |   |   |   |   |   Enqueue v to Q_prune
12 |   |   |   |   else
13 |   |   |   |   |   Enqueue v to Q_label
14 |   |   |   |   |   Add {(r_i, δ_BFS(r_i, v))} to L(v)
15 |   |   |   |   end
16 |   |   |   end
17 |   |   end
18 |   |   π ← π + 1
19 |   |   foreach v ∈ Q_prune at depth π do
20 |   |   |   Enqueue unvisited children of v at depth π + 1 to Q_prune
21 |   |   end
22 |   end
23 end
24 return L
```

Example 4. *We illustrate how our algorithm conducts pruned BFSs in Figure 4.4. The pruned BFS from landmark* 1 *is depicted in Figure 4.4(a), which labels only four vertices* $\{4, 11, 13, 14\}$ *because the other vertices are either landmarks or contain other landmarks in their shortest paths to landmark* 1. *Similarly, in the pruned BFS from landmark* 5 *depicted*

in Figure 4.4(b), only vertices $\{7, 2, 12, 3, 8\}$ *are labelled, and none of the vertices* 4*,* 11*,* 13 *and* 14 *is labelled because of the presence of landmark* 1 *in their shortest paths to landmark* 5*. Indeed, we can get the distance between landmark* 5 *to these vertices by using the highway, i.e.* $\delta_H(5, 1)$*, and distance entries in their labels to landmark* 1*. The pruned BFS from landmark 9 is depicted in Figure 4.4(c), which works in a similar fashion.*

Note that, although a highway H is given as input in Algorithm 1, we can indeed compute the distances δ_H for a given set of landmarks R along with Algorithm 1.

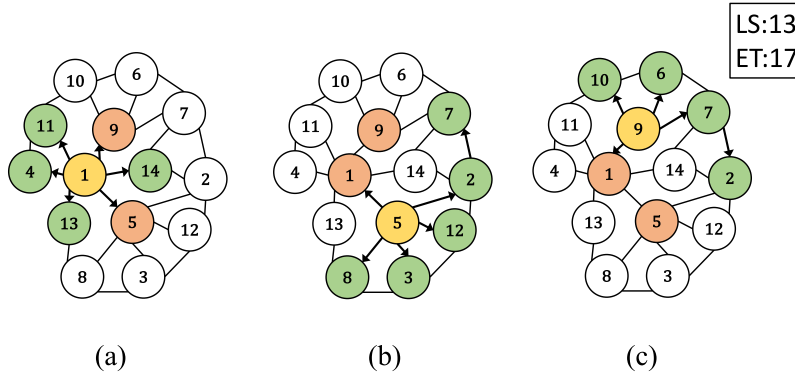


Figure 4.4: An illustration of the highway cover labelling algorithm: (a), (b) and (c) describe the pruned BFSs that are rooted at the landmarks 1, 5 and 9, respectively, where yellow vertices denote roots, green vertices denote those being labeled, red vertices denote landmarks, and white vertices are not labelled. LS and ET at the top right corner denote the labelling size and the number of edges traversed during the pruned BFSs, respectively.

4.3.3 Order Independence

In previous studies [Abraham et al. 2011; Akiba et al. 2013; Abraham et al. 2012; Cohen et al. 2002], given a graph G, a distance labelling algorithm builds a unique canonical distance labelling subject to a labelling order i.e., the order of landmarks used for constructing a distance labelling. It has been well known that such a labelling order is decisive in determining the size of the constructed distance labelling [Qin et al. 2017]. For the same set of landmarks, when using different labelling orders, the sizes of the constructed distance labelling may vary significantly.

The following example shows how different labelling orders in the pruned landmark labelling approach [Akiba et al. 2013] can lead to a distance labelling of different sizes.

Example 5. *In Figure 4.5, the size of the distance labelling constructed using the labelling order* $\langle 1, 5, 9 \rangle$ *in Figure 4.5(a)-4.5(c) is different from the size of the distance labelling constructed using the labelling order* $\langle 9, 5, 1 \rangle$ *in Figure 4.5(d)-4.5(f). In both cases, the first BFS adds a distance entry of the current landmark into the label of each vertex in the graph. Then, the following BFSs check each visited vertex whether the shortest-path distance between the current landmark and the visited vertex can be computed via the 2-hop cover property based*

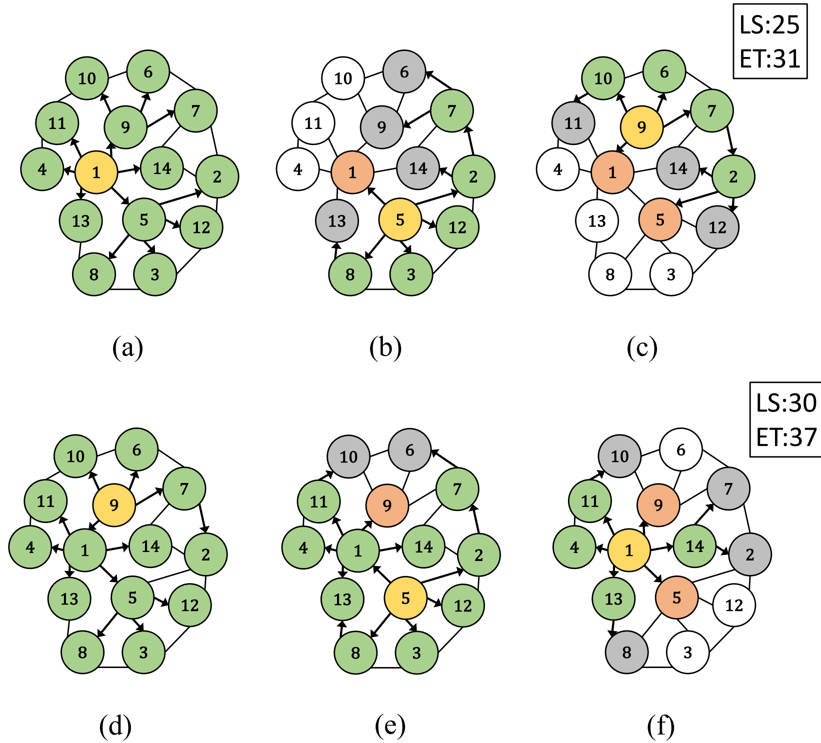


Figure 4.5: An illustration of the pruned landmark labelling algorithm [Akiba et al. 2013]: (a)-(c) show an example of constructing labels through pruned BFSs from three landmarks in the labelling order $\langle 1,5,9 \rangle$; (d)-(f) show an example of constructing labels using the same three landmarks but in a different labelling order $\langle 9,5,1 \rangle$. Yellow vertices denote landmarks that are the roots of pruned BFSs, green vertices denote those being labeled, grey vertices denote vertices being visited but pruned, and red vertices denote landmarks which have already been visited.

on their labels added by the previous BFSs. A distance entry is only added into the label of a vertex if the shortest-path distance cannot be computed by applying the 2-hop cover over the existing labels. Thus, the choice of the labelling order could affect the size of labels significantly. Take the vertex 11 *for example, its label contains only one distance entry* $(1,1)$ *using the labelling order depicted in Figure 4.5(a)-4.5(c), but contains three distance entries* (1.1), $(5,2)$, *and* $(9,2)$ *when the labelling order depicted in Figure 4.5(d)-4.5(f) is used.*

Unlike all previous approaches taken with distance labelling, our highway cover labelling algorithm is order-invariant. That is, distance labellings constructed by our algorithm using different labelling orders over the same set of landmarks always have the same size. In fact, we can show that our algorithm has the following stronger property: a distance labelling constructed using our algorithm is deterministic (i.e., it produces the same label for each vertex) for any given set of landmarks.

Lemma 1. *Let* $G = (V, E)$ *be a graph and* $H = (R, \delta_H)$ *a highway over* G*. For any two different labelling orders over* R*, the highway cover distance labellings* $\Gamma_1 = (H, L_1)$ *and*

$\Gamma = (H, L_2)$ over G constructed in accordance with these two different labelling orders using Algorithm 1 satisfy $L_1(v) = L_2(v)$ for every $v \in V \backslash R$.

Proof. Let O_{L_1} and O_{L_2} be two different labelling orders over R. For any landmark r in O_{L_1} and O_{L_2}, Algorithm 1 generates exactly the same pruned BFS tree. This implies that, for each vertex $v \in V \backslash R$, either the same distance entry $(r, \delta_{BFS}(r, v))$ is added into $L_1(v)$ and $L_2(v)$, or no distance entry is added to $L_1(v)$ and $L_2(v)$. Thus, Algorithm 1 satisfy $L_1(v) = L_2(v)$ for every $v \in V \backslash R$. □

Example 6. *Figure 4.5 shows the labelling size (LS) of the pruned landmark labelling at the top right corner, which is constructed using two different orderings. The first ordering $\langle 1, 5, 9 \rangle$ labels 25 vertices whereas the second ordering $\langle 9, 5, 1 \rangle$ labels 30 vertices. On the other hand, the LS of the highway cover distance labelling is 13 as shown in Figure 4.4. Note that the LS of the highway cover distance labelling does not change, irrespective of ordering.*

4.3.4 Minimality

We discuss the question of minimality, i.e., whether a highway cover distance labelling constructed by our algorithm is always minimal in terms of the labelling size. The proof of minimality if provided in Section 4.6.

The state-of-the-art approaches for distance labelling is primarily based on the idea of 2-hop cover [Akiba et al. 2013; Fu et al. 2013; Abraham et al. 2011]. One may ask the question: how is the highway cover labelling different from the 2-hop cover labelling, such as the pruned landmark labelling [Akiba et al. 2013]? It is easy to verify the following lemma that each pruned landmark labelling satisfies the highway cover property for the same set of landmarks.

Lemma 2. *Let L be a pruned landmark labelling over a graph G constructed using a set of landmarks R. Then L also satisfies the highway cover property over G with a highway $H = (R, \delta_H)$.*

As the pruned landmark labelling algorithm [Akiba et al. 2013] prunes labels based on the 2-hop cover property, but our highway cover labelling algorithm prunes labels based on the property described in Definition 4. We have the following corollary, stating that, for the same set of landmarks, the size of the highway cover labelling is no more than the size of any pruned landmark labelling.

Corollary 1. *For a highway cover distance labelling $\Gamma = (H, L_1)$ produced by Algorithm 1 over G, where $H = (R, \delta_H)$, and a pruned landmark labelling L_2 over G using any labelling order over R, we always have $|L_1| \leq |L_2|$.*

Example 7. *In Figure 4.4, we can see that the highway cover distance labelling constructed by our algorithm is always minimal. The LS of the highway cover distance labelling in Figure 4.4 is much smaller than the LS of either of a pruned landmark labelling constructed using two different orderings in Figure 4.5.*

4.4 Bounded Distance Querying Framework

In this section, we formulate a bounded distance querying framework that allows us to efficiently compute the exact shortest-path distance between two arbitrary vertices in a massive network.

Given a graph G and a highway $H = (R, \delta_H)$ over G, we can precompute a highway cover distance labelling Γ using the landmarks in R, which enables us to efficiently compute the length of any r-constrained shortest-path between any two vertices in $V \backslash R$. The length of such a r-constrained shortest-path must be greater than or equal to the exact shortest-path distance between these two vertices and can thus serve as an upper bound. On the other hand, since the length of such a r-constrained shortest-path between two vertices in $V \backslash R$ can always be efficiently computed by the highway cover distance labelling Γ, the distance-bounded shortest-path search only needs to be conducted over a sparsified graph by removing all landmarks in R from G, i.e. $G[V \backslash R]$.

To compute the shortest-path distance between two vertices s and t in graph G, our querying framework proceeds in two steps:

(1) an upper bound of the shortest-path distance between s and t is computed using the highway cover distance labelling

(2) the exact shortest-path distance between s and t is computed using a distance-bounded shortest-path search over a sparsified graph.

More precisely, we define the bounded distance querying problem as follows.

Definition 6 (Bounded Distance Querying Problem)**.** *Given a sparsified graph $G[V \backslash R]$, a pair of vertices $\{s, t\} \in V$, and an upper (distance) bound $d^{\top}_{st}$, the* bounded distance querying problem *is to efficiently compute the exact shortest-path distance $Q(s, t, \Gamma)$ between s and t over $G[V \backslash R]$ under the upper bound $d^{\top}_{st}$ such that,*

$$Q(s,t,\Gamma) = \begin{cases} d_{G[V\backslash R]}(s,t), & \text{if } d_{G[V\backslash R]}(s,t) \leq d^{\top}_{st} \\ d^{\top}_{st}, & \text{otherwise} \end{cases}$$

In the following, we discuss the two steps of this framework in detail.

4.4.1 Computing Upper Bounds

Given any two vertices s and t, we use a *highway cover distance labelling* Γ to compute an upper bound $d^{\top}_{st}$ for the shortest-path distance between s and t as follows,

$$\begin{aligned} d^{\top}_{st} = \texttt{min}\{\delta_L(r_i, s) + \delta_H(r_i, r_j) + \delta_L(r_j, t) \mid \\ (r_i, \delta_L(r_i, s)) \in L(s), \\ (r_j, \delta_L(r_j, t)) \in L(t)\} \end{aligned} \tag{4.2}$$

This corresponds to the length of a shortest-path from s to t passing through landmarks r_i and r_j, where $\delta_L(r_i, s)$ is the shortest-path distance from r_i to s in $L(s)$, $\delta_H(r_i, r_j)$ is the shortest-path distance from r_i to r_j through highway H, and $\delta_L(r_j, t)$ is the shortest-path distance from r_j to t in $L(t)$.

Example 8. *Consider the graph in Figure 4.3(a), we may use the labels $L(2)$ and $L(11)$ to compute the upper bound for the shortest-path distance between two vertices 2 and 11. There are two cases: (1) for the path $\langle 2, 5, 1, 11 \rangle$ that goes through landmarks 5 and 1, we have $\delta_L(5, 2) + \delta_H(5, 1) + \delta_L(1, 11) = 1 + 1 + 1 = 3$, and (2) for the path $\langle 2, 9, 1, 11 \rangle$ that goes through landmarks 9 and 1, we have $\delta_L(9, 2) + \delta_H(9, 1) + \delta_L(1, 11) = 2 + 1 + 1 = 4$. Thus, we take the minimum of these two distances as the upper bound, which is 3 in this case.*

4.4.2 Distance-Bounded Bi-Directional Search

We conduct a bidirectional search on the sparsified graph $G[V \backslash R]$ which is bounded by the upper bound $d_{st}^{\top}$ obtained from the highway cover distance labelling Γ. For a pair of vertices $\{s, t\} \subseteq V \backslash R$, we run the breadth-first search algorithm from s and t, alternatively [Hayashi et al. 2016]. Algorithm 2 shows the pseudo-code of our distance-bounded bi-directional search algorithm. We use two sets of vertices $\mathcal{P}_s$ and $\mathcal{P}_t$ to keep track of visited vertices from s and t. We use two queues $\mathcal{Q}_s$ and $\mathcal{Q}_t$ to conduct both a forward search from s and a reverse search from t. Furthermore, we use two integers d_s and d_t to maintain the current distances from s and t, respectively.

During initialization, we set $\mathcal{P}_s$ and $\mathcal{P}_t$ to $\{s\}$ and $\{t\}$, and enqueue s and t into $\mathcal{Q}_s$ and $\mathcal{Q}_t$, respectively. In each iteration, we increment d_s or d_t and expand $\mathcal{P}_s$ or $\mathcal{P}_t$ by running either a forward search (FS) or a reverse search (RS) as long as $\mathcal{P}_s$ and $\mathcal{P}_t$ have no any common vertex or $d_s + d_t$ is equal to the upper bound $d_{st}^{\top}$, and $\mathcal{Q}_s$ and $\mathcal{Q}_t$ are not empty. In the forward search from s, we examine the neighbors $N_{G[V \backslash R]}(v)$ of each vertex $v \in \mathcal{Q}_s$. Suppose that we are visiting a vertex $w \in N_{G[V \backslash R]}(v)$, if w is included in the vertex set $\mathcal{P}_t$, then it means that we find a shortest-path to vertex t of length $d_s + 1 + d_t$, because the reverse search from t had already visited w with distance d_t. At this stage, we return $d_s + 1 + d_t$ as the answer since we already know $d_s + d_t + 1 \leq d_G(s, t) \leq d_{st}^{\top}$. Otherwise, we add vertex w to $\mathcal{P}_s$ and enqueue w into a new queue $\mathcal{Q}_{new}$. When we can not find the shortest distance in the this iteration, we replace $\mathcal{Q}_s$ with $\mathcal{Q}_{new}$ and increase d_s by 1, and check if $d_s + d_t = d_{st}^{\top}$. If it holds, then we return $d_{st}^{\top}$ since $d_{st}^{\top} \leq d_G(s, t) \leq d_s + d_t + 1$.

Example 9. *In Figure 4.3(a), the upper distance bound between vertices 2 and 11 is 3, as computed in Example 8. In Figure 4.6(a), we run BFSs from vertices 2 and 11 respectively. First, a forward search from 2 enqueues its neighbors 7, 12 and 14 into $\mathcal{Q}_2$ and increases d_2 by 1. Then a reverse search from 11 enqueues 4 and 10 into $\mathcal{Q}_{11}$ and also sets d_{11} to 1. At this stage, although we have not found any common vertex between $\mathcal{Q}_2$ and $\mathcal{Q}_{11}$, however $d_2 + d_{11} + 1 = 3$ which is equal to the upper bound 3. Thus, we terminate our search and return the upper bound as the query distance.*

Algorithm 2: Distance-Bounded Bi-Directional Search

```
    Input: G[V\R], s, t, d_st^⊤
    Output: d_{G[V\R]}(s, t)
 1  P_s ← {s}, P_t ← {t}, d_s ← 0, d_t ← 0
 2  Enqueue s to Q_s, t to Q_t
 3  while Q_s and Q_t are not empty do
 4  |   if |P_s| ≤ |P_t| then
 5  |   |   found ← FS(Q_s)
 6  |   else
 7  |   |   found ← RS(Q_t)
 8  |   end
 9  |   if found = true then
10  |   |   return d_s + 1 + d_t
11  |   else if d_s + d_t = d_st^⊤ then
12  |   |   return d_st^⊤
13  |   end
14  end
15  return ∞
```

4.5 Optimization Techniques

In this section, we present optimization techniques for label construction, label compression, and query processing.

4.5.1 Label Construction

A technique called Bit-Parallelism (BP) has been previously used in several methods [Akiba et al. 2013; Hayashi et al. 2016] to speed up the label construction process. The key idea of BP is to perform BFSs from a given landmark r and up to 64 of its neighbors simultaneously, and encode the relative distances (-1, 0 or 1) of these neighbors w.r.t. the shortest paths between r and each vertex v into a 64-bit unsigned integer. In the work [Akiba et al. 2013], BP was applied to construct bit-parallel labels from initial vertices without pruning, which aimed to leverage the information from these bit-parallel labels to cover more shortest paths between vertices. Then, both bit-parallel labels and normal labels were constructed in the pruned BFSs. The work in [Hayashi et al. 2016] also used BP to construct thousands of bit-parallel shortest-path trees (SPTs) because it is very costly to construct thousands of normal SPTs in memory owing to their prohibitively large space requirements and very long construction time.

In our work, we develop a simple yet rigorous parallel algorithm (HL-P) which can run parallel BFSs from multiple landmarks (depending on the number of processors) to construct labelling in an extremely efficient way for massive networks, with much less time as will be demonstrated in our experiments in Section 4.7.

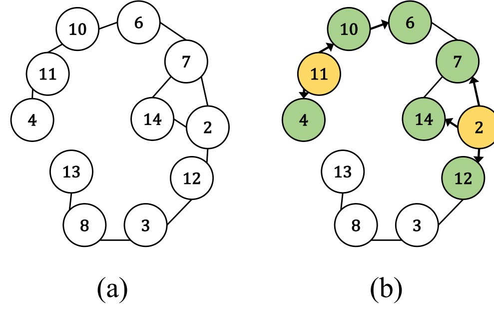


Figure 4.6: An illustration of the distance-bounded shortest-path search algorithm [Hayashi et al. 2016]: (a) shows the sparsified graph after removing three landmarks $\{1, 5, 9\}$ from the graph in Figure 4.3(a); (b) shows an example of computing the bounded distance between vertices 2 and 11 as highlighted in yellow, and green vertices denote the visited vertices in the forward and reverse searches.

4.5.2 Label Compression

The choice of the data structure for labels may significantly affect the performance of labelling size and memory usage. As noted in [Li et al. 2017], some works [Abraham et al. 2012; Delling et al. 2014] did not elaborate on what data structure they have used for representing labels. Nonetheless, for the works that are most relevant to ours, such as FD [Hayashi et al. 2016] and PLL [Akiba et al. 2013], they used 32-bit integers to represent vertices and 8-bit integers to represent distances for normal labels. In addition to this, they also used 64-bits to encode the distances from a landmark to up to 64 of its neighbors in their shortest paths to other vertices. Since our approach only selects a very small number of landmarks to construct the highway cover labelling (usually no more than 100 landmarks), we may use 8 bits to represent landmarks and another 8 bits to store distances for labels. In order to fairly compare methods from different aspects, we have implemented our methods using both 32 bits and 8 bits for representing vertices in labels. However, different from the BP technique that uses 64-bits to encode the distance information of up to 64 neighbors of a landmark, our parallel algorithm (HL-P) does not use a different data structure for labels constructed in parallel BFSs.

4.5.3 Query Processing

We show that computing the upper bound $d^{\top}_{st}$ can be optimized based on the observation, captured by the following lemma.

Lemma 3. *For a highway cover distance labelling $\Gamma = (H, L)$ over G, and any $\{s, t\} \subseteq V$, if a landmark r appears in both $L(s)$ and $L(t)$, then $\delta_L(r, s) + \delta_L(r, t) \leq \delta_L(r, s) + \delta_H(r, r') + \delta_L(r', t)$ holds for any other $r' \in R$.*

Proof. By the definition of the highway cover property, we know that r is not in any shortest-path between r' and t. Then by the triangle inequality in Equation 2.1, this lemma can be proven. □

Thus, in order to efficiently compute the upper bound $d^{\top}_{st}$, for any landmarks that appear in both $L(s)$ and $L(t)$, we compute the r-constrained shortest-path distance between s and t using Equation 2.3, while for a landmark r' that only appear in one of $L(s)$ and $L(t)$, we use Equation 4.2 to calculate the r'-constrained shortest-path distance between s and t. This would lead to more efficient computations for queries when the landmarks appear in both labels of two vertices.

4.6 Theoretical Results

In this section, we prove the correctness of our labelling construction algorithm, i.e., it constructs a distance labelling that satisfies the highway cover property. We also prove the minimality property of our labelling construction algorithm, i.e., it constructs a minimal highway cover labelling, a desirable property that has a direct impact on both query time and space efficiency. Then, we prove the correctness of our querying framework. Finally, we briefly analyse the complexity of our proposed algorithms.

4.6.1 Proof of Correctness

First, we prove the correctness of our labelling construction algorithm which is descried in Algorithm 1.

Lemma 4. *In Algorithm 1, for each pruned BFS rooted at $r_i \in R$, $(r_i, \delta_L(r_i, v))$ is added into the label of a vertex $v \in V \backslash R$ iff there is no any other landmark appearing in the shortest-path between r_i and v, i.e., $P_{r_i v} \cap R = \{r_i\}$.*

Proof. Suppose that Algorithm 1 is conducting a pruned BFS rooted at r_i and v is an unvisited child of another vertex u in $\mathcal{Q}_{label}$ (start from $\mathcal{Q}_{label} = \{r_i\}$) (Lines 6-9). If $v \in R$ (Line 10), then we have $(P_{r_i w} \cap R) \supseteq \{r_i, v\}$ (Lines 11, 19-21),and $(r_i, \delta_L(r_i, w))$ can not be added into the label of any child w of v, i.e., put w into $\mathcal{Q}_{prune}$. Otherwise, by $v \notin R$ and v is an unvisited child of a vertex u in $\mathcal{Q}_{label}$ (Lines 8-9), we know that $P_{r_i v} \cap R = \{r_i\}$ and thus $(r_i, \delta_L(r_i, v))$ is added into $L(v)$ (lines 12-14). □

Then, by Lemma 4, we have the following corollary.

Corollary 2. *Let $r \in R$ be a landmark, $v \in V \backslash R$ a vertex, and $\Gamma = (H, L)$ a distance labelling constructed by Algorithm 1. If $(r, \delta_L(r, v)) \notin L(v)$, then there must exist a landmark r_j such that $(r_j, \delta_L(r_j, v)) \in L(v)$ and $d_G(r, v) = \delta_L(r_j, v) + \delta_H(r, r_j)$.*

Theorem 1. *The highway cover distance labelling $\Gamma = (H, L)$ over G constructed using Algorithm 1 satisfies the highway cover property.*

Proof. To prove that, for any two vertices $s, t \in V \backslash R$ and for any $r \in R$, there exist $(r_i, \delta_L(r_i, s)) \in L(s)$ and $(r_j, \delta_L(r_j, t)) \in L(t)$ such that $r_i \in P_{rs}$ and $r_j \in P_{rt}$, we consider the following four cases: (1) If $(r, \delta_L(r, s)) \in L(s)$ and $(r, \delta_L(r, t)) \in L(t)$, then $r = r_i = r_j$. (2) If $(r, \delta_L(r, s)) \in L(s)$ and $(r, \delta_L(r, t)) \notin L(t)$, then $r_i = r$ and by Lemma

2, there exists another landmark r_j such that r_j is in the shortest-path between t and r and $(r_j, \delta_L(r_j, t)) \in L(t)$. (3) If $(r, \delta_L(r, s)) \notin L(s)$ and $(r, \delta_L(r, t)) \in L(t)$, then similarly we have $r_j = r$, and by Lemma 2, there exists another landmark r_i such that r_i is in the shortest-path between s and r and $(r_i, \delta_L(r_i, s)) \in L(s)$. (4) If $(r, \delta_L(r, s)) \notin L(s)$ and $(r, \delta_L(r, t)) \notin L(t)$, then by Lemma 2 there exist another two landmarks r_i and r_j such that r_i is in a shortest-path between s and r and $(r_i, \delta_L(r_i, s)) \in L(s)$, and r_j is in a shortest-path between t and r and $(r_j, \delta_L(r_j, t)) \in L(t)$. The proof is done. □

Now, we prove the correctness of our querying framework based on the following two lemmas. More specifically, Lemma 5 can be derived from the highway cover property and the definition of $d^{\top}_{st}$. Lemma 6 can be proven by the property of shortest-path and the definition of the sparsified graph $G[V \backslash R]$.

Lemma 5. *For a highway cover distance labelling $\Gamma = (H, L)$ over G), we have $d^{\top}_{st} \geq d_G(s, t)$ for any two vertices s and t of G, where $d^{\top}_{st}$ is computed using L and H.*

Lemma 6. *For any two vertices $\{s, t\} \subseteq V \backslash R$, if there is a shortest-path between s and t in G that does not include any vertex in R, then $d_G(s, t) = d_{G[V \backslash R]}(s, t)$ holds.*

Thus, the following theorem holds:

Theorem 2. *Let $G = (V, E)$ be a graph and $\Gamma = (H, L)$ a highway cover distance labelling over G. Then, for any two vertices $\{s, t\} \subseteq V$, the querying framework yields $Query(s, t, \Gamma) = d_G(s, t)$.*

Proof. We consider two cases: (1) P_{st} contains at least one landmark. In this case, By Lemma 5 and the definition of the highway cover property, we have $d^{\top}_{st} = d_G(s, t)$. (2) P_{st} does not contain any landmark. By Lemma 6, we have $d_{G[V \backslash R]}(s, t) = d_G(s, t)$. □

4.6.2 Preservation of Minimality

We prove the following theorem to show that the highway cover labelling constructed by Algorithm 1 is minimal.

Theorem 3. *The highway cover distance labelling $\Gamma = (H, L)$ over G constructed using Algorithm 1 is minimal, i.e., for any highway cover distance labelling $\Gamma' = (L', H')$ over G, $size(\Gamma') \geq size(\Gamma)$ must hold.*

Proof. We prove this by contradiction. Let us assume that there is a highway cover distance labelling L' with $size(L') < size(L)$. Then, this would imply that there must exist a vertex $v \in V \backslash R$ and a landmark $r \in R$ such that $(r, \delta_L(r, v)) \in L(v)$ and $(r, \delta_L(r, v)) \notin L'(v)$. By Lemma 4 and $(r, \delta_L(r, v)) \in L(v)$, we know that there is no any other landmark in R that is in any shortest-path between r and v. However, by the definition of the highway cover property (i.e. Definition 4) and $(r, \delta_L(r, v)) \notin L'(v)$, we also know that there must exist another landmark $(r_i, \delta_L(r_i, v)) \in L(v)$ and $r_i \in P_{rv}$, which contradicts with the previous conclusion that there is no any other landmark in any shortest-path between r and v. Thus, $size(L') \geq size(L)$ must hold for any highway cover distance labelling L'. □

4.6.3 Complexity Analysis

Let l be the maximum size of labels (i.e., $l = |R|$) and d be the maximum degree. We visit $O(n \cdot l)$ vertices in total, where we traverse $O(d)$ edges. Therefore, the time complexity of Algorithm 1 is roughly estimated as $O(n \cdot l \cdot d)$.

4.7 Experimental Setup

In our experiments, all algorithms are implemented in C++11 using STL libraries and compiled with g++ 5.5.0 using the -O3 option. All the experiments are performed on a Linux server Intel Xeon W-2175 (2.50GHz CPU) with 28 cores and 512GB of main memory.

In the following, we first present datasets and the baseline methods and then discuss test data generation.

4.7.1 Datasets

We use 15 real-world large complex networks from a variety of domains in our experiments, in order to empirically verify the efficiency, scalability and robustness of our algorithms. Among them, the largest two networks are Clueweb09 [Rossi and Ahmed 2015] and Clueweb12 [Boldi and Vigna 2004; Boldi et al. 2011] which have 2 billions and 1 billion of vertices, and 8 billions and 43 billions of edges, respectively. We include these networks in our experiments for the purpose of evaluating the robustness and scalability of the proposed methods. In previous works [Hayashi et al. 2016; Li et al. 2019], the largest dataset that has been reported is UK which has only around 100 millions of vertices and 3.7 billions of edges. These networks are accessible at Stanford Network Analysis Project [Leskovec and Sosič 2016], Laboratory for web Algorithmics [Boldi and Vigna 2004; Boldi et al. 2011], and Network Repository [Rossi and Ahmed 2015]. We treat these networks as undirected and unweighted graphs. The types of networks, number of vertices and edges, and statistical information of these network datasets are summarized in Table 4.2, while a brief description of each dataset is given below:

– **Youtube:** This is a social network among users of YouTube (`www.youtube.com`), in which nodes are users that form friendships (edges) with each other [Akiba et al. 2014].

– **Skitter:** This is an internet topology network, obtained by running daily traceroute in 2005 [Leskovec et al. 2005], in which nodes represent routers and edges represent communication links.

– **Flickr:** This is a social network of users and their connections in a photo sharing website Flickr (`www.flickr.com`) [Mislove et al. 2007].

– **Wikitalk:** This is a social network containing information about communication among editors of Wikipedia (`www.wikipedia.org`) on editing on-talk pages

till January 2008 [Akiba et al. 2013], where nodes represent Wikipedia editors and a directed edge from node i to node j represents that user i at least once edited an on-talk page of user j.

- **Hollywood:** This is a social network of movie actors, where vertices represent actors and two actors are joined by an edge whenever they appeared in a movie together in 2009 [Boldi and Vigna 2004; Boldi et al. 2011].

- **Orkut:** This is a social network of users and their connections in a social networking website, Orkut (`www.orkut.com`) [Mislove et al. 2007].

- **Enwiki:** This is a network of hyperlinks from a snapshot of English Wikipedia, obtained in 2013, where vertices represent pages and edges indicate hyperlinks between pages [Boldi and Vigna 2004; Boldi et al. 2011].

- **Livejournal:** This is a social network which allows its members to manage their journals and blogs, and to declare which other members and their friends they belong to, in an online social website (`www.livejournal.com`) [Backstrom et al. 2006; Leskovec et al. 2009].

- **Indochina:** This is a web graph of web pages, obtained by performing a large crawl of the country domains of Indochina in 2004 for the Nagaoka University of Technology [Boldi and Vigna 2004; Boldi et al. 2011].

- **IT:** This is a web graph, obtained by performing a fairly large crawl of the .it domain in 2004 [Boldi and Vigna 2004; Boldi et al. 2011].

- **Twitter:** This is a social network with information about who follows whom on Twitter, where vertices represent users and edges represent follow relationships between users, in an online social website (`www.twitter.com`) [Boldi and Vigna 2004; Boldi et al. 2011].

- **Friendster:** This is a social gaming network, where users are connected with friendship relationships, in an online website (`www.friendster.com`) [Yang and Leskovec 2015].

- **UK:** This is a web graph which is part of a time-aware network, obtained by collecting monthly snapshots of the .uk domain for twelve months in 2006 and 2007 [Boldi et al. 2008].

- **Clueweb09:** This is a web graph of web pages in ten languages collected in January and February 2009, where nodes represent unique URLs (pages) and edges represent links between pages [Rossi and Ahmed 2015].

- **Clueweb12:** This dataset is a successor to the Clueweb09 dataset. It was obtained by crawling the web for about 1 billion English web pages between February 10, 2012 and May 10, 2012 [Boldi and Vigna 2004; Boldi et al. 2011].

Table 4.2: Datasets, where $size(G)$ denotes the size of a graph G with each edge appearing in the forward and reverse adjacency lists and being represented by 8 bytes.

Dataset	Network	n	m	m/n	avg. deg.	max. deg.	avg. dist.	$size(G)$
Youtube	social (u)	1.1M	3M	2.63	5.265	28754	5.3	23 MB
Skitter	comp (u)	1.7M	11M	6.54	13.08	35455	5.0	85 MB
Flickr	social (u)	1.7M	16M	9.07	18.13	27224	5.3	119 MB
Wikitalk	comm (d)	2.4M	5M	1.95	3.890	100029	3.9	36 MB
Hollywood	social (u)	1.1M	114M	49.5	98.91	11467	3.9	430 MB
Orkut	social (u)	3.1M	117M	38.1	76.28	33313	4.2	894 MB
Enwiki	social (d)	4.2M	101M	21.9	43.75	432260	3.4	701 MB
Livejournal	social (d)	4.8M	69M	8.84	17.68	20333	5.6	327 MB
Indochina	web (d)	7.4M	194M	20.4	40.73	256425	7.7	1.1 GB
IT	web (d)	41M	1.2B	24.9	49.77	1326744	7.0	7.7 GB
Twitter	social (d)	42M	1.5B	28.9	57.74	2997487	3.6	9.0 GB
Friendster	social (u)	66M	1.8B	27.4	55.06	5214	5.0	13 GB
UK	web (d)	106M	3.7B	31.4	62.77	979738	6.9	25 GB
Clueweb09	web (d)	1.7B	7.8B	4.64	9.27	6444720	7.4	58.2 GB
Clueweb12	web (d)	$\sim$1B	43B	39.1	78.25	75611696	5.2	279 GB

4.7.2 Baseline Methods

We compared our proposed method with the following baseline methods:

(1) A fully dynamic (FD) method [Hayashi et al. 2016] that combines a distance labelling with a graph traversal algorithm to answer distance queries.

(2) An independent set based labelling (IS-L) method [Fu et al. 2013] that combines a distance labelling with a graph traversal algorithm to answer distance queries.

(3) The pruned landmark labelling (PLL) method [Akiba et al. 2013] which is completely based on a distance labelling to answer distance queries.

(4) The optimized online bidirectional BFS method which expands search from the direction with less vertices alternatively to answer distance queries [Hayashi et al. 2016], and we name this algorithm as BiBFS in our experiments.

Besides these, there are a number of other methods for answering distance queries, such as HDB [Jiang et al. 2014], RXL and CRXL [Delling et al. 2014], HCL [Jin et al. 2012], HHL [Abraham et al. 2012] and TEDI [Wei 2010]. However, since the experimental results of the previous works [Hayashi et al. 2016; Akiba et al. 2013] have shown that FD outperforms HDB, RXL and CRXL, and PLL outperforms HCL, HHL and TEDI, we omit the comparison with these methods.

In our experiments, the implementations of the baseline methods FD, IS-L and PLL were provided by their authors, which were all implemented in C++. We used the same parametric settings for running these methods as suggested by their authors. For instance, the number of landmarks is chosen to 20 for FD [Hayashi et al. 2016], the number of bit-parallel BFSs is set to 50 for PLL [Akiba et al. 2013], and

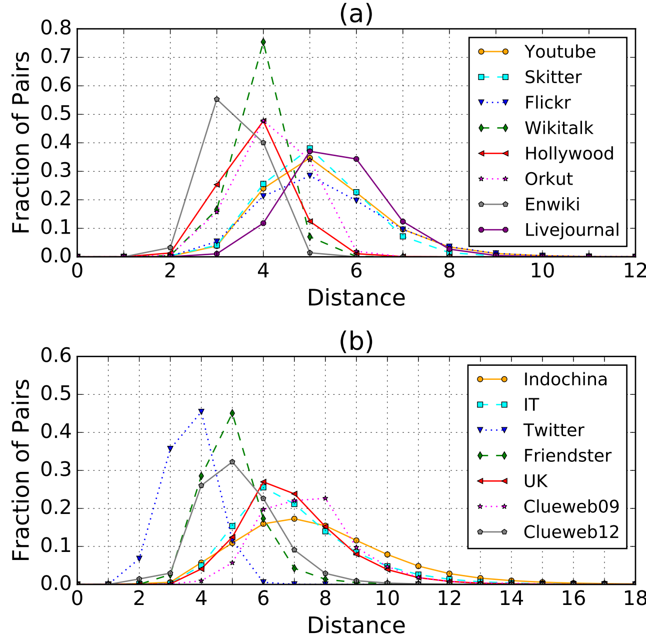


Figure 4.7: Distance distribution of 100,000 random pairs of vertices on all the datasets.

k is 6 for graphs larger than 1 million vertices for IS-L [Fu et al. 2013]. To make a consistent comparison with the baseline methods [Hayashi et al. 2016; Akiba et al. 2013; Fu et al. 2013], we chose top 20 vertices as landmarks except for the largest two datasets i.e., Clueweb09 and Clueweb12 after sorting based on decreasing order of their degrees. We set the number of landmarks to 150 for Clueweb09 and Clueweb12 because a small number of landmarks on such large networks do not help much in pruning the search space. We also used 32-bit integers to represent vertices and 8-bit integers to represent distances.

4.7.3 Test Data Generation

We randomly sampled 100,000 pairs of vertices from all pairs of vertices in each network, i.e., $V \times V$, to evaluate the query performance. The distance distribution of these 100,000 randomly sampled pairs of vertices is shown in Figure 5.4(a)-5.4(b), from which we can confirm that most of pairs of vertices in these networks have a small distance ranging from 2 to 8. Table 4.2 also lists the average distance in each network using these 100,000 randomly sampled pairs of vertices. We can see that most of these networks have a small average distance between 3 to 8.

4.8 Results and Discussion

In this section, we discuss the experimental results of our proposed algorithms, and compare the performance of our algorithms with the baseline algorithms.

Table 4.3: Comparison between the construction time (CT) and query time (QT) of our methods, i.e., HL-P and HL, and the state-of-the-art methods, where CT is the CPU clock time in seconds, and QT is the average query time in milliseconds. HL-P refers a parallel version of HL and DNF denotes that a method did not finish in one day (24 hours) or ran out of memory (512 GB).

Dataset	Construction Time (CT) [s]					Query Time (QT) [ms]				
	HL-P	HL	FD	PLL	IS-L	HL	FD	PLL	IS-L	BiBFS
Youtube	0.15	1.46	3.56	83.8	351	0.005	0.007	0.001	1.913	56.76
Skitter	0.28	2.68	8.31	389	1,042	0.027	0.018	0.003	3.556	129.2
Flickr	0.30	3.17	10.8	756	8,359	0.007	0.011	0.003	33.76	162.6
Wikitalk	0.22	1.93	4.68	39.5	225	0.005	0.006	0.001	1.301	48.63
Hollywood	0.56	6.32	24.7	12,679	DNF	0.025	0.033	0.015	–	328.9
Orkut	2.55	24.6	90.3	DNF	DNF	0.102	0.097	–	–	1,139
Enwiki	2.79	24.4	91.1	11,530	DNF	0.052	0.031	0.005	–	1,042
Livejournal	1.94	20.3	48.3	DNF	20,583	0.043	0.043	–	56.85	900.1
Indochina	0.92	9.06	30.1	4,183	DNF	0.712	0.741	0.003	–	405.2
IT	7.41	76.4	231	DNF	DNF	1.069	0.919	–	–	3,245
Twitter	57.1	540	2,010	DNF	DNF	0.863	0.168	–	–	27,148
Friendster	113	1,202	3,476	DNF	DNF	0.816	0.814	–	–	53,979
UK	22.6	176	625	DNF	DNF	3.443	5.234	–	–	8,189
Clueweb09	DNF	46,366	DNF	DNF	DNF	16.94	–	–	–	–
Clueweb12	DNF	22,370	DNF	DNF	DNF	9.375	–	–	–	–

4.8.1 Performance Comparison

To evaluate the performance of our proposed methods, we compared our approach with the baseline methods in terms of the construction time of labelling (CT), the size of labelling (LS), and querying time to perform distance queries (QT). The experimental results are presented in Tables 4.3 and 4.4, where DNF denotes that a method did not finish in one day or ran out of memory.

Construction Time

As shown in Table 4.3, our proposed method (HL) has successfully constructed the distance labelling on all the datasets for a significantly less amount of time than the state-of-the-art methods. As compared to FD, our method is on average 5 times faster and have results on all the datasets. In contrast to this, FD fails to construct labelling for the largest two dataset Clueweb09 and Clueweb12. PLL fails for 7 out of 12 datasets, including the datasets Orkut and Livejournal which have less than 120 millions of edges, due to its prohibitively high preprocessing time and memory requirements for building labelling. IS-L fails to construct labelling for all the datasets that have edges more than 100 million due to its very high cost for computing independent sets on massive networks, i.e. it fails for 9 out of 12 datasets. We can also see from Table 4.3 that the parallel version of our method (HL-P) is much faster than the sequential version (HL). Compared with FD, HL-P is more than 50-70 times faster for the two large datasets Friendster and UK. This confirms that our method

Table 4.4: Comparison between the labelling size (LS) of our methods, i.e., HL(8) and HL, and the state-of-the-art methods. HL(8) refers to the compressed version of HL that uses 8-bits representation of vertices and ALS is the average number of entries per label.

Dataset	Labelling Size (LS)					Average Label Size (ALS)			
	HL(8)	HL	FD	PLL	IS-L	HL	FD	PLL	IS-L
Youtube	20 MB	48 MB	83 MB	1.25 GB	289 MB	9	20+64	65+50	24
Skitter	42 MB	102 MB	153 MB	2.45 GB	507 MB	12	20+64	138+50	51
Flickr	34 MB	81 MB	152 MB	3.69 GB	679 MB	10	20+64	290+50	50
Wikitalk	41 MB	100 MB	74 MB	2.05 GB	201 MB	9	20+64	12+50	21
Hollywood	27 MB	67 MB	263 MB	12.6 GB	–	12	20+64	2206+50	–
Orkut	70 MB	170 MB	711 MB	–	–	11	20+64	–	–
Enwiki	82 MB	200 MB	608 MB	12.6 GB	-	10	20+64	471+50	–
Livejournal	122 MB	299 MB	663 MB	–	3.8 GB	13	20+64	–	69
Indochina	81 MB	191 MB	840 MB	18.7 GB	–	5	20+64	441+50	–
IT	855 MB	2.03 GB	4.73 GB	–	–	10	20+64	–	–
Twitter	1.14 GB	2.78 GB	3.83 GB	–	–	14	20+64	–	–
Friendster	2.43 GB	5.97 GB	9.14 GB	–	–	19	20+64	–	–
UK	1.78 GB	4.29 GB	11.8 GB	–	–	8	20+64	–	–
Clueweb09	163 GB	404 GB	–	–	–	51	–	–	–
Clueweb12	48.9 GB	121 GB	–	–	–	27	–	–	–

can construct labelling very efficiently and is scalable on large networks with billions of vertices and edges.

Labelling Size

As we can see from Table 4.4 that the labelling sizes of all the datasets constructed by the proposed method are significantly smaller than the labelling sizes of FD and much smaller than PLL and IS-Label. Specifically, our labelling sizes using 32-bits representation of vertices (HL) are 2-5 times smaller than FD except for Clueweb09 and Clueweb12 (as discussed before, FD fails to construct labelling for these two datasets), 7 times smaller than IS-Label on Skitter, Flickr and Livejournal and more than 60 times smaller than PLL for Skitter, Flickr, Hollywood, Enwiki and Indochina. The compressed version of our method that uses 8-bits representation of vertices (i.e. HL(8)) produces further smaller labelling size as compared to uncompressed version (HL). Here, It is important to note that the labelling sizes of almost all the datasets are also significantly smaller than the original sizes of the datasets shown in Table 4.2. This also shows that our method is highly scalable on large networks in terms of the labellng sizes.

Query Time

The average query time of our method (HL) is comparable with FD and PLL and faster than IS-L. Particularly, the average query time of our method on Flickr and Hollywood is is very close to PLL and faster than FD. This is due to a very small average labelling size (i.e., 12) as compared with FD and PLL (i.e., 20+64 and 2206+50,

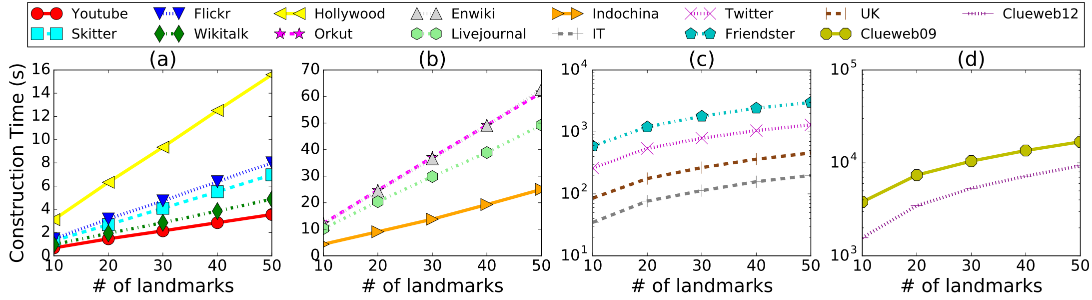


Figure 4.8: Construction time using our method HL under 10-50 landmarks on all the datasets.

respectively) and a very small average distance. The average query time of HL on Twitter is 3 times slower than FD. This may be due to a large portion of covered pairs by FD as shown in Figure 4.11 which contributes towards an effective bounded traversal on the sparsified network since the landmarks of Twitter have very high degrees and the average distance is also very small. Moreover, the average query times of HL and FD on Indochina, IT, Friendster and UK are more than 1ms due to comparatively large average distances than other datasets as shown in Figure 4.7(b). Note that all the baseline methods are not scalable enough to have results for Clueweb09 and Clueweb12, and the average query time of our method HL on these two largest datasets is small because of a very large portion of covered pairs and a small average label size. We also report the average query time for online bidirectional BFS algorithm (BiBFS) using randomly selected 1000 pairs of vertices in Table 4.3. As we can see that BiBFS has considerably long query times, which are not practicable in applications for performing distance queries in real time.

4.8.2 Performance under Varying Landmarks

We have also evaluated the performance of our method (HL) by varying the number of landmarks between 10 and 50, which are again selected based on highest degrees.

Construction Time

The construction time of our method HL against different numbers of landmarks (from 10 to 50) are shown in Figure 4.8. We can see that the construction time is linear in terms of the number of landmarks, which confirms the scalability of our method. In Figure 4.8(a)-4.8(b), our method is able to construct labelling for 7 datasets under 50 landmarks from 20 seconds to 2 minutes, which is not possible with any state-of-the-art methods. In Figure 4.8(c), the construction time using 50 landmarks of Friendster is 3 times faster and the construction time of UK is 4 times faster than FD using only 20 landmarks as shown in Table 4.3. Figure 4.8(d) shows the construction time for Clueweb09 and Clueweb12 which has billions of vertices and edges. The significant improvement in construction time allows us to compute labelling for a

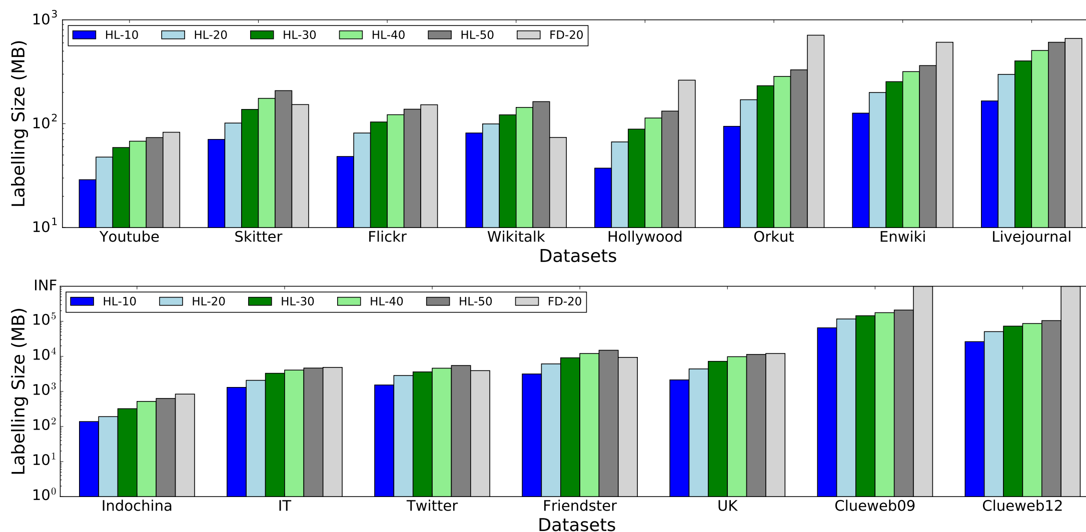


Figure 4.9: Labelling size produced by our method HL under 10-50 landmarks and FD under 20 landmarks on all the datasets.

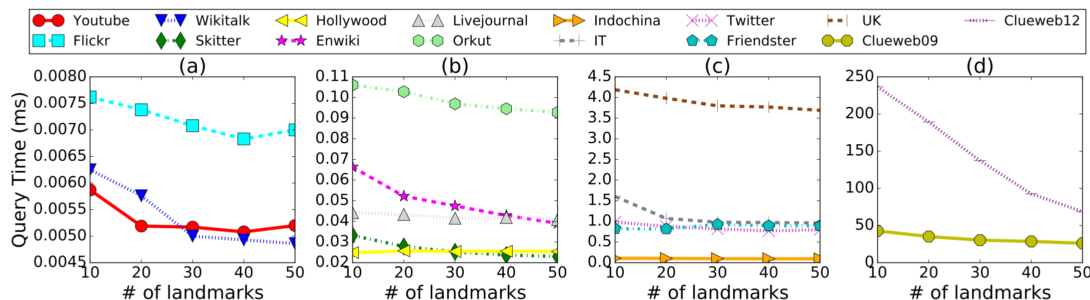


Figure 4.10: Query time using our method HL under 10-50 landmarks on all the datasets.

large number of landmarks, leading to better pair coverage ratios to tighten upper distance bounds (will be further discussed in Section 4.8.2).

Labelling Size

Figure 4.9 shows the labelling sizes of HL using 10, 20, 30, 40 and 50 landmarks on all the dataset, and of FD using only 20 landmarks on all the datasets except for Clueweb09 and Clueweb12 (as discussed before, FD fails to construct labelling for these two largest datasets). It can be seen that the labelling size of HL increase linearly with the increased number of landmarks, and even the labelling sizes of HL using 50 landmarks are almost always smaller than the labelling sizes constructed by FD using only 20 landmarks. This reduction in labelling sizes enables us to save space and memory, thus makes our method scalable on large networks.

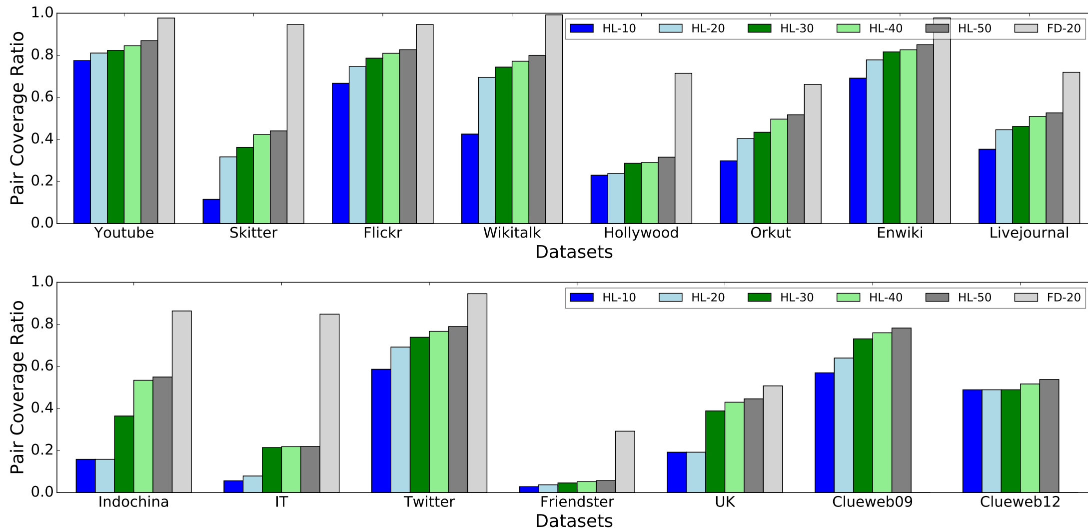


Figure 4.11: Pair coverage ratios using our method HL under 10-50 landmarks and using FD on all the dataset.

Query Time

Figure 4.10 shows the impact of using different numbers of landmarks between 10 and 50 on average query time of our method. The average query time either decrease or remain the same when the number of landmarks increases, except for Orkut when using 30 landmarks and for Friendster when using landmarks greater than 20. In particular, on Friendster, labelling sizes are very large as shown in Figure 4.9 and the fraction of covered pairs (i.e., pair coverage ratio) is very small as shown in Figure 4.11, which may have slowed down our query processing due to a longer time for computing upper distance bounds and ineffective use of bounded-distance traversal.

Pair Coverage

Figure 4.11 presents the ratios of pairs of vertices covered by at least one landmark (i.e., pair coverage ratios) in HL using 10-50 landmarks and in FD using 20 landmarks. As we can observe that the pair coverage ratios for HL increase when the number of landmarks increases and 40 turns out to be the better choice on the number of landmarks for most of the datasets. Specifically, pair coverage ratios on Orkut, Enwiki, Indochina and UK with 40 landmarks are good, resulting in better query times than using 20 landmarks, as shown in Figure 4.10. On datasets such as Hollywood and it2004, 30 landmarks are a better option than 40 landmarks because they only slightly differ in the pair coverage ratios and query times w.r.t. using 40 landmarks, but with reduced labelling sizes. The pair coverage ratios by FD are greater than HL on all the datasets except for Clueweb09 and Clueweb12, which may be the reason behind its better query times for some datasets as shown in Table 4.3.

4.9 Summary

In this chapter, we have studied the distance query problem on static graphs. To address this problem, we developed a method that can scale to answer exact shortest-path distance queries over billion-scale networks. The key ideas of this method are a highway cover distance labelling and a bounded distance querying framework.

More specifically, we formulated a novel distance labelling property called highway cover distance labelling and proposed an efficient labelling construction algorithm that can efficiently construct a highway cover distance labelling. Then, we formulated a bounded distance querying framework that combines a highway cover distance labelling with distance-bounded shortest-path search to enable fast distance computation. We proved that our proposed labelling construction algorithm can construct a unique highway cover distance labelling that is independent of the order of landmarks and is also highway cover minimal. We further developed a parallel algorithm to speed up the labelling construction process by conducting BFSs simultaneously w.r.t. multiple landmarks. We showed in our experiments that our proposed method significantly outperforms the existing state-of-the-art methods.

Chapter 5

FulHL: Fully Dynamic Labelling For Distance Queries

5.1 Overview

In this chapter, we study the problem of answering exact shortest-path distance queries in large dynamic networks whose topological structure evolves over time in the single-update setting. We propose a fully dynamic method to efficiently answer distance queries over large graphs that are changed by edge insertion and deletion. At its core, our proposed method incorporates two building blocks: (i) *incremental algorithm* for handling incremental update operations, i.e. edge insertion, and (ii) *decremental algorithm* for handling decremental update operations, i.e. edge deletion. These two building blocks are built in a highly scalable framework of distance query answering proposed in Chapter 4. Our fully dynamic method enables fast processing of graph changes and can scale to graphs with billions of vertices and edges, without compromising performance on query time and labelling size.

Table 5.1 summarizes the performance of our fully dynamic method FulHL against with the two state-of-the-art methods FulPLL [Akiba et al. 2014; D'angelo et al. 2019] and FulFD [Hayashi et al. 2016]. We present the results for the largest network that was previously evaluated by each of these methods. We can see that FulHL significantly outperforms FulPLL and FulFD in all three dimensions, i.e., update time, query time and labelling size, and can scale to billion-scale networks. On the other hand, FulPLL and FulFD fail to scale to networks of size over 16 millions and 3.7 billions of edges, respectively.

The main contributions of this chapter are as follows,

- Our incremental algorithm overcomes the challenge of eliminating outdated and redundant distance entries in order to preserve the minimality of labelling. None of the previous studies have addressed this challenge because detecting outdated and redundant distance entries is computationally expensive [Akiba et al. 2014]. When an edge is inserted, the previous studies only add new distance entries or modify existing distance entries without removing outdated and redundant ones. This however leads to an ever increasing size of labelling. Then, both query performance and space efficiency deteriorate over time.

Table 5.1: Flickr, UK and Clueweb12 are the largest networks evaluated by the methods FulPLL [Akiba et al. 2014; D'angelo et al. 2019], FulFD [Hayashi et al. 2016] and FulHL (this work), respectively, where " – " indicates no result due to scalability issues.

Network	Network Size		Update Time			Query Time			Labelling Size		
	$\|V\|$	$\|E\|$	FulPLL	FulFD	FulHL	FulPLL	FulFD	FulHL	FulPLL	FulFD	FulHL
Flickr	1.7M	16M	6810 ms	7.655 ms	**0.053 ms**	0.009 ms	0.012 ms	**0.007 ms**	12.7 GB	152 MB	**34 MB**
UK	106M	3.7B	–	337.6 ms	**1.075 ms**	–	5.858 ms	**3.488 ms**	–	11.8 GB	**1.78 GB**
Clueweb12	∼1B	43B	–	–	**1796 ms**	–	–	**9.375 ms**	–	–	**49.1 GB**

- Our decremental algorithm can efficiently identify affected vertices and update their labels without compromising on query time and labelling size. We achieve this based on two observations. The first is to characterize a special kind of vertices, called *anchor vertices*, which are critical for updating labelling. The second is to prune unnecessary searches by characterizing *prunable vertices*, thereby improving update efficiency. Previous work [D'angelo et al. 2019] has reported that edge deletion requires much longer update time than edge insertion, but no interpretation was provided. We fill in this gap by analyzing the fundamental differences between edge insertion and edge deletion on dynamic graphs.

- We theoretically prove the correctness of our fully dynamic method and show that it preserves the minimality of labelling under update operations, including edge insertion and edge deletion. Note that, by leveraging the property of highway cover [Farhan et al. 2019], the minimal size of a distance labelling in this work is much smaller than the size of a 2-hop cover labelling in the previous work [Akiba et al. 2013; Hayashi et al. 2016]. We also provide a complexity analysis for our fully dynamic method.

- To empirically verify the efficiency and scalability of our fully dynamic method, we conduct experiments using 15 real-world large networks across different domains. In particular, our methods can perform updates within a couple of seconds even on networks with billions of vertices and edges, while still answering distance queries efficiently in the order of milliseconds and maintaining very small labelling sizes.

The rest of this chapter is organized as follows. Section 5.2 introduces the problem definition. Section 5.3 formulates the fully dynamic framework and presents two novel algorithms for reflecting incremental and decremental changes on graphs. Section 5.4 shows that the proposed algorithms are correct and preserve the property of minimality. Section 5.5 discusses the experimental results, which compare the performance of our proposed algorithms against the baseline algorithms. Section 5.7 summarises the chapter.

5.2 Problem Definition

We define the *distance query* problem on dynamic graphs as the fully dynamic labelling problem. Given a graph that is dynamically changed by edge insertions or deletions over time, the fully dynamic labelling problem is concerned about efficiently updating a distance labelling to ensure that distance queries can be correctly answered on the changed graph. Below, we first define the incremental and decremental update problems in terms of edge insertions and edge deletions on a graph, respectively. Then, we define the fully dynamic labelling problem for addressing the distance query problem on dynamic graphs.

Definition 7 (Incremental Update Problem)**.** *Let $G = (V, E)$ and $G' = (V, E')$ be two graphs, and G be changed to G' by edge insertions. The* incremental update *problem is, given a highway cover distance labelling Γ over G such that $Q(u, v, \Gamma) = d_G(u, v)$ for any two vertices u and v in G, to compute a distance labelling Γ' over G' such that $Q(u, v, \Gamma') = d_{G'}(u, v)$ for any two vertices u and v in G'.*

Definition 8 (Decremental Update Problem)**.** *Let $G = (V, E)$ and $G' = (V, E')$ be two graphs, and G be changed to G' by edge deletions. The* decremental update *problem is, given a highway cover distance labelling Γ over G such that $Q(u, v, \Gamma) = d_G(u, v)$ for any two vertices u and v in G, to compute a distance labelling Γ' over G' such that $Q(u, v, \Gamma') = d_{G'}(u, v)$ for any two vertices u and v in G'.*

Definition 9 (Fully Dynamic Labelling Problem)**.** *Let $G = (V, E)$ and $G' = (V, E')$ be two graphs, and G be changed to G' by edge insertions or deletions. The* fully dynamic labelling *problem is, given a highway cover distance labelling Γ over G such that $Q(u, v, \Gamma) = d_G(u, v)$ for any two vertices u and v in G, to compute a distance labelling Γ' over G' such that $Q(u, v, \Gamma') = d_{G'}(u, v)$ for any two vertices u and v in G'.*

5.3 Fully Dynamic Labelling Framework

In this section, we formulate a fully dynamic framework for distance queries on large graphs. This framework consists of two novel dynamic algorithms: *incremental algorithm* and *decremental algorithm*, which efficiently update a highway cover labelling after edge insertions or edge deletions, respectively. We first introduce a key search strategy, i.e., *jumped-and-pruned search*, used by both incremental algorithm and decremental algorithm. Then, we present the algorithmic details of these two algorithms.

5.3.1 Jumped-and-Pruned Search

To efficiently reflect changes on graphs, we develop a jumped-and-pruned search strategy for updating highway cover distance labelling. This strategy requires us to identify two special types of vertices in a fully dynamic graph: *affected vertices* and *anchor vertices*.

Affected Vertices

When an update operation occurs on a graph $G = (V, E)$, no matter whether it is an edge insertion or an edge deletion, there always exists a subset of "affected" vertices in V whose labels need to be updated as a consequence of this update operation on the graph. But, *can we identify such vertices efficiently?* To answer this question, we define the notion of affected vertices and analyse their properties.

Definition 10 (Affected Vertex)**.** *Let $G = (V, E)$ and $R \subseteq V$ be a set of landmarks on G. A vertex $v \in V$ is* affected *by $G \hookrightarrow G'$ if $P_G(v, r) \neq P_{G'}(v, r)$ holds for at least one $r \in R$, and* unaffected *otherwise.*

For simplicity, we use $\Lambda_r = \{v \in V \mid P_G(v, r) \neq P_{G'}(v, r)\}$ to denote the set of all affected vertices w.r.t. a landmark r and $\Lambda = \bigcup_{r \in R} \Lambda_r$ refers to the set of all affected vertices. Note that, vertices affected by $G \hookrightarrow G'$ are the same as vertices affected by $G' \hookrightarrow G$, i.e., the same set of vertices is affected when inserting an edge (a, b) into a graph G or deleting an edge (a, b) from a graph G'.

Example 10. *Consider Figure 5.1(a) in which* 0, 10 *and* 4 *are three landmarks. After inserting an edge* $(2, 5)$ *in Figure 5.1(b)-(d), we have* $\Lambda = \{0, 1, 2, 5, 8, 9, 10, 13, 14\}$ *because* $\Lambda_0 = \{5, 8, 9, 10, 13, 14\}$, $\Lambda_{10} = \{0, 1, 2\}$ *and* $\Lambda_4 = \{2\}$.

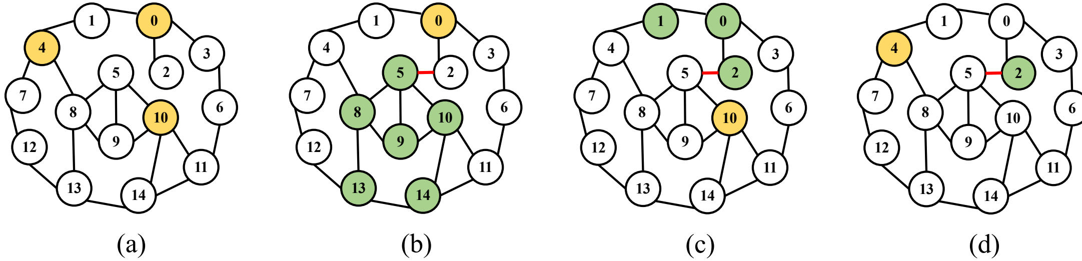


Figure 5.1: An illustration of affected vertices by an edge insertion $(2, 5)$ occurring on the graph in (a), where three landmarks 0, 10 and 4 are highlighted in yellow. In (b)-(d), the affected vertices w.r.t. the landmarks 0, 10 and 4 are highlighted in green, respectively. Note that the affected vertices w.r.t. the landmarks 0, 10 and 4 would remain the same if an edge deletion $(2, 5)$ occurs on the graph in (b).

The following lemma states how affected vertices relate to an edge being inserted or deleted.

Lemma 7. *When $G \hookrightarrow G'$ for an edge insertion (a, b), a vertex $v \in \Lambda_r$ iff there exists a shortest-path between v and r in G' passing through (a, b). Similarly, when $G \hookrightarrow G'$ for an edge deletion (a, b), a vertex $v \in \Lambda_r$ iff there exists a shortest-path between v and r in G passing through (a, b).*

Proof. For any vertex $v \in V$, if $d_{G'}(r, v) = d_{G'}(r, a) + d_{G'}(a, b) + d_{G'}(b, v)$ after inserting an edge (a, b) and $d_G(r, v) = d_G(r, a) + d_G(a, b) + d_G(b, v)$ after deleting an edge (a, b), then by Definition 10, $P_G(v, r) \neq P_{G'}(v, r)$. Thus, v is an affected vertex. $\square$

A naive way of finding affected vertices would be to apply Definition 10 directly, by computing the set of all shortest paths from a landmark to each vertex on G and G', respectively, and comparing them. However, the computational cost of this would be prohibitive, even for small graphs. The more practical approach is to explore only affected shortest paths rather than all to efficiently reflect changes in a graph.

Following Lemma 7, we have the following corollary.

Corollary 3. *When $G \hookrightarrow G'$ with an inserted or deleted edge (a, b), if $d_G(r, a) = d_G(r, b)$ holds for a landmark $r \in R$, then we have $\Lambda_r = \emptyset$.*

This corollary allows us to reduce the search space of affected vertices by eliminating landmarks r with $d_G(r, a) = d_G(r, b)$. Without loss of generality, we assume that $d_G(r, b) > d_G(r, a)$ w.r.t. a landmark r in the rest of this section.

The following lemma enables us to further reduce the search space of affected vertices by "jumping" from the root of a BFS to the vertex b.

Lemma 8. *When $G \hookrightarrow G'$ with an inserted or deleted edge (a, b), $d_G(r, v) \geq d_G(r, a) + 1$ hold for any affected vertex $v \in \Lambda_r$.*

Proof. By Lemma 7, there exists a shortest-path from any affected vertex v to r going through inserted or deleted edge (a, b) and thus through a. Since a is unaffected and the distance from a to v is equal to or greater than 1, $d_G(r, v) \geq d_G(r, a) + 1$ must hold. □

Anchor Vertices

Although efficiently identifying affected vertices is critical for dynamic algorithms, it is equally important to efficiently update the labels of affected vertices against changes on a graph. A naive approach is to run a full BFS from each landmark r on the changed graph in order to decide the new distances of affected vertices w.r.t. a landmark r. However, this is inefficient, particularly if only a very small portion of vertices in a graph is affected by an update operation. *Can we pinpoint the differences between the old labels of affected vertices in an original graph and their new labels in the changed graph, so as to change a distance labelling in an efficient way?* To answer this question, we need to identify a special kind of affected vertices, called *anchor vertices*, which have the smallest distance to a landmark r on the changed graph.

Definition 11 (Anchor Vertex)**.** *When $G \hookrightarrow G'$, a vertex $v \in V$ is an* anchor vertex *w.r.t. a landmark r in G' if $v \in \Lambda_r$ and $d_{G'}(r, v) \leq d_{G'}(r, u)$ for any vertex $u \in \Lambda_r$.*

The following lemma states that the exact distances of anchor vertices can be inferred from their unaffected neighbors. Note that this does not generally hold for every affected vertex. Let $d^*_{G'}(r, v)$ refer to a *contingent distance* between a landmark r and a vertex $v \in \Lambda_r$ in G', which is the minimum length of paths between v and r going through only unaffected vertices in G'. If a vertex $v \in \Lambda_r$ in G' has no any unaffected neighbors, we consider $d^*_{G'}(r, v) = \infty$.

Lemma 9. *When $G \hookrightarrow G'$, if a vertex $v \in \Lambda_r$ has the smallest contingent distance to a landmark r among all vertices in Λ_r, then v is an anchor vertex w.r.t. landmark r and $d_{G'}(r,v) = d^*_{G'}(r,v)$ holds.*

Proof. We prove this by contradiction. Assume that $d_{G'}(r,v) \neq d^*_{G'}(r,v)$ for such a vertex v. Then $d_{G'}(r,v) < d^*_{G'}(r,v)$ must hold because $d_{G'}(r,v)$ is the shortest-path distance. Since $d^*_{G'}(v,r)$ is the minimum length of all paths between v and r that go through only unaffected vertices, one shortest-path between v and r must go through at least one affected vertex $v' \in \Lambda_r$ and $d^*_{G'}(v,r) > d^*_{G'}(v',r)$ must hold. This contradicts with the assumption that v has the minimum contingent distance to r in Λ_r. Hence, $d_{G'}(r,v) = d^*_{G'}(r,v)$. Accordingly, v must be an anchor vertex w.r.t. r. □

The observation here is that, once anchor vertices are identified, we can *locally* infer their new distances from their unaffected neighbors. Then, new distances of other affected vertices can be inferred *inductively* by a level-by-level propagation in a BFS tree from r through unaffected neighbors and affected neighbors whose new distances have already been inferred.

Example 11. *Firstly, we consider Figure 5.2(a) in which the edge $(2,5)$ is inserted. This causes the vertices $\{5,8,9,10,13,14\}$ to be affected w.r.t. the landmark 0. Among these vertices, the vertex 5 has the smallest contingent distance (i.e., the distance through unaffected vertices) and thus is an anchor vertex. Now we consider Figure 5.3(a) in which the edge $(2,5)$ is deleted. This causes the same set of vertices to be affected w.r.t. the landmark 0. However, in this case, the vertex 5 has the contingent distance ∞ because there is no path between vertex 5 and landmark 0 passing through only unaffected vertices. Instead, the vertices $\{8,10\}$ have the smallest contingent distances and thus are anchor vertices in this case.*

Jumped-and-Pruned BFS

Our dynamic algorithms, including both incremental and decremental algorithms, use a *jumped-and-pruned* search strategy to efficiently update a distance labelling. The key idea is that, instead of conducting a full BFS from a landmark to all vertices, we conduct a partial BFS (named as JP-BFS) that jumps from the root of a BFS directly to affected vertices, thereby skipping unaffected vertices. Further, a JP-BFS exploits the property of highway cover labelling (i.e., an distance entry of a vertex v w.r.t. a landmark r can be pruned if there is another landmark lying in a shortest-path between v and r) to prune affected vertices as many as possible after its jump.

Definition 12 (Prunable Vertex)**.** *When $G \hookrightarrow G'$, a vertex v is* prunable *w.r.t. a landmark r iff there exists a landmark $r' \in R - \{r\}$ such that all of the following conditions hold:*

(1) $d_G(r,v) = d_G(r,r') + d_G(r',v)$;

(2) $d_{G'}(r,v) = d_{G'}(r,r') + d_{G'}(r',v)$;

(3) $d_G(r',v) = d_{G'}(r',v)$.

A vertex v is weakly prunable *iff it only satisfies the conditions (2) and (3).*

Intuitively, the conditions in the above definition state that we can prune a vertex v only if there is another landmark r' lying in a shortest-path between this vertex and the landmark r in both G and G', i.e. (1) and (2), and the distance from this vertex to r' also remains the same in both G and G', i.e. (3). A vertex v satisfying these three conditions implies that its label w.r.t. landmark r remains the same in G and G', and a JP-BFS can thus prune v as well as the children of v from search. When a vertex v is weakly prunable, it means that the label of v may contain outdated or redundant entries w.r.t. landmark r, which would affect the correctness of a highway cover distance labelling in the case of edge deletion but not edge insertion. In fact, the case of weakly prunable vertices can only occur during edge insertion due to newly added shortest path(s).

The following example illustrates the notions of prunable vertex and weakly prunable vertex. We will discuss further how vertices are pruned during a JP-BFS in our incremental and decremental algorithms in Sections 5.3.2, 5.3.3 and 5.3.4, respectively.

Example 12. *Let us consider the vertex* 8 *in Figure 5.2(a) which is pruned because it satisfies all three conditions in Definition 12. We can see that the path* $\langle 0, 1, 4, 8\rangle$ *exists before and after adding the edge* $(2, 5)$ *and passes through the landmark* 4 *satisfying all the conditions (1), (2) and (3). For the vertex* 14 *in Figure 5.2(a), it is weakly pruned due to the newly added path* $\langle 0, 2, 5, 10, 14\rangle$ *through landmark* 10 *that did not exist before adding the edge* $(2, 5)$ *thus satisfying only the conditions (2) and (3). Now we consider the pruned vertices highlighted in Figure 5.3(a). All of these vertices satisfy the three conditions in Definition 12, e.g., we prune from 10 because a path* $\langle 0, 3, 6, 10\rangle$ *exists between* 0 *and* 10 *before and after deleting the edge* $(2, 5)$.

In a nutshell, a JP-BFS has the following two features: (1) *Jumping* from the root (i.e., a landmark) to affected vertices so as to traverse locally, rather than globally; (2) *Pruning* affected vertices that are prunable or weakly prunable whenever possible.

Algorithmic Design

Before introducing our incremental and decremental algorithms in detail (as will be shown in Sections 5.3.2, 5.3.3 and 5.3.4), we briefly discuss how this jumped-and-pruned search strategy is applied in these algorithms.

In the most general case, two kinds of JP-BFS are needed. One kind of JP-BFS is to *identify affected vertices w.r.t. a landmark* r. By Lemmata 7 and 8, such a JP-BFS jumps from the root r to the vertex b, and starts to identify affected vertices iteratively through checking neighbors and their old distances. The other kind of JP-BFS is to *update affected vertices w.r.t. a landmark* r. By Lemma 9, such a JP-BFS jumps from the root r to anchor vertices, and starts to update the labels of affected vertices through a level-by-level propagation in order to infer the new distances of these affected vertices.

Nonetheless, we notice the following:

- In the case of edge insertion, there exists exactly one anchor vertex for an inserted edge; further, such an anchor vertex can be easily identified according to the inserted edge. This enables an efficient design for our incremental algorithm which can not only identify affected vertices, but also simultaneously update the labels of affected vertices through a carefully designed propagation on new distances. Hence, instead of conducting two separate JP-BFSs, our incremental algorithm merges these two JP-BFSs into one JP-BFS for improving efficiency. In Section 5.3.2, we will first present an incremental algorithm which uses two JP-BFSs to perform incremental updates (edge insertions), i.e., first to find affected vertices and then to update their labels. Then, we will discuss in Section 5.3.3 how a merged JP-BFS is designed by combining these two JP-BFSs in our improved incremental algorithm to efficiently perform incremental updates (i.e. Algorithm 6).
- In the case of edge deletion, finding anchor vertices turns out to be challenging. For a deleted edge, there may exist multiple anchor vertices; further, these anchor vertices can be far away from the deleted edge and cannot be identified without knowing a full picture on how vertices are affected by the deleted edge. Hence, our decremental algorithm must first find affected vertices in the first JP-BFS, which leads to identifying anchor vertices, and then update the labels of affected vertices in the second JP-BFS based on the information about anchor vertices and affected vertices obtained from the first JP-BFS. Section 5.3.4 will discuss further on how these two separate JP-BFS are designed in our decremental algorithm (i.e. Algorithm 8).

5.3.2 Incremental Algorithm

In this section, we propose an incremental algorithm, called IncHL^b, which performs incremental updates (edge insertions) in two separate stages i.e., first find affected vertices and then update their labels. More specifically, this algorithm performs two JP-BFSs, one for identifying affected vertices and the other for repairing the labels of affected vertices in order to reflect graph changes into a highway cover labelling. Algorithm 3 describes the main steps of IncHL^b. Below, we discuss them in detail.

Finding Affected Vertices

In the first step, we perform a JP-BFS which starts from the first affected vertex b and identifies the other affected vertices through local neighborhoods iteratively. Algorithm 4 describes the process for finding affected vertices. Given a graph G with an inserted edge (a, b) and a highway cover labelling $\Gamma = (H, L)$ over G, we conduct a *jumped* BFS w.r.t. a landmark r starting from the vertex b with its new depth $\pi = Q(r, a, \Gamma) + 1$ (Lines 3-4). For every $(v, \pi) \in \mathcal{Q}$, we enqueue all the neighbors of v that are affected into $\mathcal{Q}$ with new distances $\pi + 1$ (Lines 7-8) and add v to Λ_r as affected vertex (Line 9). This process continues until $\mathcal{Q}$ is empty.

Algorithm 3: Incremental algorithm (IncHL^b).

```
   Input: G = (V, E), G' = (V, E ∪ {(a, b)}), (a, b) ∉ E, Γ = (H, L) over G
   Output: Γ' = (H', L') over G
1  foreach r ∈ R do
2  |  Λ_r ← FindAffected(G, (a, b), r, Γ)
3  └  RepairAffected(G', (a, b), Λ_r, r, Γ)
```

Algorithm 4: Finding affected vertices for IncHL^b.

```
1  Function FindAffected(G, (a, b), r, Γ)
2  |  Q ← ∅, Λ_r ← ∅
3  |  π ← Q(r, a, Γ) + 1
4  |  Enqueue (b, π) to Q
5  |  while Q is not empty do
6  |  |  Dequeue (v, π) from Q
7  |  |  foreach w ∈ N(v) s.t. Q(r, w, Γ) ≥ π + 1 do
8  |  |  └  Enqueue (w, π + 1) to Q
9  |  └  Λ_r = Λ_r ∪ {v}
10 └  return Λ_r
```

Example 13. *Figure 5.1 illustrates how our algorithm finds affected vertices as a result of inserting an edge* $(2,5)$*. The BFS rooted at landmark* 0 *is depicted in Figure 5.2(b), which jumps to vertex* 5 *and finds six affected vertices* $\{5,8,9,10,13,14\}$*. The BFS rooted at landmark* 10 *is depicted in Figure 5.2(c), which jumps to vertex* 2 *and finds three affected vertices* $\{0,1,2\}$*. Similary, the BFS rooted at landmark* 4 *is depicted in Figure 5.2(d), which jumps to vertex* 2 *and finds only one affected vertex* $\{2\}$*.*

Repairing Affected Vertices

Now, to repair the labels of all affected vertices, we perform the second JP-BFS which again starts from the first affected vertex b and is performed only on affected vertices. Further, to avoid unnecessary computations, we distinguish two kinds of affected vertices: (1) affected vertices that are *prunable* (Definition 12) and can thus be easily repaired by removing an entry from their labels; (2) affected vertices whose labels need to be repaired with accurately calculated distances on a changed graph.

Algorithm 12 describes our algorithm for repairing affected vertices. Given a graph G with an inserted edge (a,b) and a set of affected vertices Λ_r, we conduct a JP-BFS w.r.t. a landmark r starting from the vertex b with its new distance $\pi = d_G(r,a)+1$ (Lines 3-4). We use two queues $\mathcal{Q}_{label}$ and $\mathcal{Q}_{prune}$ to process vertices to be labeled or pruned, respectively. If b is pruned, we enqueue (b,π) to $\mathcal{Q}_{prune}$ and remove the entry of r from the labels of affected vertices (Line 25). Otherwise, we enqueue (b,π) to $\mathcal{Q}_{label}$ and start processing vertices in $\mathcal{Q}_{label}$ (Line 5). For each

Algorithm 5: Repairing affected vertices for IncHL^b.

```
1  Function RepairAffected(G', (a,b), Λ_r, r, Γ)
2      Q_label ← ∅, Q_prune ← ∅
3      π ← d_G(r,a) + 1
4      Enqueue (b,π) to Q_prune if prunable; otherwise to Q_label
5      while Q_label is not empty do
6          while (v,π) ∈ Q_label at depth π do
7              forall w ∈ N(v) s.t. w ∈ Λ_r at depth π+1 do
8                  if w is prunable then
9                      if w is a landmark then
10                         δ_H(r,w) ← π+1
11                     else
12                         Remove r from L(w)
13                     Enqueue (w, π+1) to Q_prune
14                 else
15                     Add/Modify {(r, π+1)} in L(w)
16                     Enqueue (w, π+1) to Q_label
17                 Remove w from Λ_r
18             Dequeue (v,π) from Q_label
19         while (v,π) ∈ Q_prune at depth π do
20             forall w ∈ N(v) s.t. w ∈ Λ_r at depth π+1 do
21                 Remove r from L(w)
22                 Remove w from Λ_r
23                 Enqueue (w, π+1) to Q_prune
24             Dequeue (v,π) from Q_prune
25     Remove entry r from remaining vertices in Q_prune
```

vertex $v \in \mathcal{Q}_{label}$ at depth π, we examine its affected neighbors w at depth $\pi+1$. If w is pruned, then if w is a landmark, we update the highway (Line 10); otherwise we remove the entry of r from $L(w)$ (Line 12) because there must exist another landmark in the shortest path from w to r and add $(w, \pi+1)$ to $\mathcal{Q}_{prune}$ (Line 13). Otherwise, we add/modify the entry of r with the new distance $\pi+1$ in $L(w)$ and enqueue w to $\mathcal{Q}_{label}$ (Lines 15-16). After that, we remove w from Λ_r (line 17). Then, for each $(v, \pi) \in \mathcal{Q}_{prune}$, we remove r from the labels of affected neighbors of v, remove these affected vertices from Λ_r and enqueue them to $\mathcal{Q}_{prune}$ (Lines 19-24). We process these two queues, one after the other, until $\mathcal{Q}_{label}$ is empty. Finally, we remove the entry of r from the labels of the remaining vertices in $\mathcal{Q}_{prune}$ (Line 25).

Example 14. *Figure 5.2 illustrates how our algorithm repairs labels as a result of inserting an edge* $(2,5)$. *The JP-BFS for landmark* 0 *is depicted in Figure 5.2(a), which jumps to vertex* 5 *and repairs three affected vertices* $\{5,9,10\}$. *The vertices* $\{8,13,14\}$ *are pruned by landmarks* 4 *and* 10. *The JP-BFS for landmark* 10 *is depicted in Figure 5.2(b), in which*

vertices $\{0, 2\}$ are repaired and vertex 1 is pruned by landmarks 0 and 4. Similarly, the JP-BFS for landmark 4 is depicted in Figure 5.2(c), in which none of the vertices is repaired and vertex 2 is pruned by landmarks 0.

5.3.3 Improved Incremental Algorithm

Now we present our improved incremental algorithm, called IncHL. Unlike the previous incremental algorithm IncHL^b, this algorithm unifies two separate processes into a single process, i.e., find affected vertices and update their labels *simultaneously*. More specifically, this algorithm combines two JP-BFSs into one JP-BFS in order to efficiently update a highway cover labelling to reflect graph changes (i.e., edge insertions).

Algorithm 6: Improved incremental algorithm IncHL.

```
   Input: G = (V, E), G' = (V, E ∪ {(a, b)}), (a, b) ∉ E, Γ = (H, L) over G
   Output: Γ' = (H', L') over G'
 1 foreach r ∈ R with d_G(r, b) > d_G(r, a) do
 2     Q ← ∅, V_r^infer ← ∅, π ← d_G(r, a) + 1
 3     Enqueue (b, π) to Q
 4     while Q is not empty do
 5         Dequeue (v, π) from Q
 6         if v is prunable then
 7             if v is a landmark then
 8                 δ_H(r, v) ← π (Updating H)
 9             end
10         else
11             Update(r, v, π, V_r^infer, ∅ )
12             foreach w ∈ N(v) and d_G(r, w) > π do
13                 Enqueue (w, π + 1) to Q
14             end
15         end
16         Add (v, π) to V_r^infer
17     end
18 end
```

We start with the following lemma that characterises anchor vertices in the case of edge insertion.

Lemma 10. *For $G \hookrightarrow G'$ with an edge insertion (a, b), if $d_G(r, a) < d_G(r, b)$ holds for a landmark $r \in R$, then b must be the only anchor vertex w.r.t. the landmark r.*

By the above lemma, since b is the only anchor vertex in the case of edge insertion, our incremental algorithm IncHL is carefully designed to merge two JP-BFSs (i.e., one for identifying affected vertices and the other for updating the labels of

Algorithm 7: Updating the label $L(v)$.

```
1 Function Update(r, v, π, V_r^infer, V_r^uninfer)
2     V_parent(v) = {w | w ∈ N(v) and ((w, π) ∈ V_r^infer or ((w, π') ∉ V_r^uninfer and
        Q(r, w, Γ) = π − 1))}
3     if ∀w ∈ V_parent(v) s.t. r appears in L(w) then
4         Add/Modify (r, π) to L(v)
5     else
6         Remove r from L(v) (if exists)
7     end
8 end
```

affected vertices) into one JP-BFS to identify affected vertices and updates their labels simultaneously. This significantly improves update efficiency for edge insertion. Another insight we obtain is that a large amount of weakly prunable vertices can be pruned away during this merged JP-BFS to further considerably improve update efficiency.

Algorithm 6 describes the detailed steps of our incremental algorithm. Given a graph G with an inserted edge (a, b) and a highway cover labelling $\Gamma = (H, L)$ over G, we conduct one JP-BFS for each landmark $r \in R$ starting from the vertex b with its new distance $\pi = Q(r, a, \Gamma) + 1$, and enqueue (b, π) into $\mathcal{Q}$ (Lines 1-3, Algorithm 6). To identify all affected vertices and update their labels, this JP-BFS works as follows. For every $(v, \pi) \in \mathcal{Q}$, if v is prunable, then we stop the search from v by simply updating the highway H (Lines 6-9, Algorithm 6). This also eliminates weakly prunable vertices and we will prove this in Section 5.4. Otherwise, we update the label of v in Function `Update` using the neighbors of v that appear in at least one shortest-path between v and r in the changed graph G', i.e., $V_{parent}(v)$ (Line 2, `Update`). Based on $V_{parent}(v)$, we update $L(v)$ as follows. If there exists at least one vertex $w \in V_{parent}(v)$ that does not contain r in its label, then there must exist another landmark in a shortest-path between v and r, and we thus remove r from $L(v)$ if exists (Line 6, `Update`); otherwise we add/modify (r, π) in the label of v (Line 4, `Update`). After updating the label of v, we enqueue all affected neighbors of v into $\mathcal{Q}$ with new distances $\pi + 1$ (Lines 12-14, Algorithm 6) and add (v, π) to V_r^{infer}, where V_r^{infer} contains the set of affected vertices w.r.t. the landmark r whose new distances have been inferred (Line 16, Algorithm 6). This process of identifying affected vertices and updating their labels continues until $\mathcal{Q}$ is empty.

Example 15. *Figure 5.2 illustrates how our incremental algorithm updates affected labels as a result of inserting an edge* $(2, 5)$*. The JP-BFS starting from the anchor vertex* 5 *w.r.t. the landmark* 0 *is depicted in Figure 5.2(a). The labels of vertices* 5 *and* 9 *are updated using the information in the labels of their parents i.e.,* $V_{parent}(5) = \{2\}$ *and* $V_{parent}(9) = \{5\}$*. This JP-BFS is pruned from vertices* 8 *and* 10 *because the landmark* 4 *lies in the shortest-path from vertex* 8 *to landmark* 0*, and vertex* 10 *is a landmark itself. Accordingly, the highway is updated. Similarly, the JP-BFS w.r.t. the landmark* 10 *is depicted in Figure 5.2(b). This*

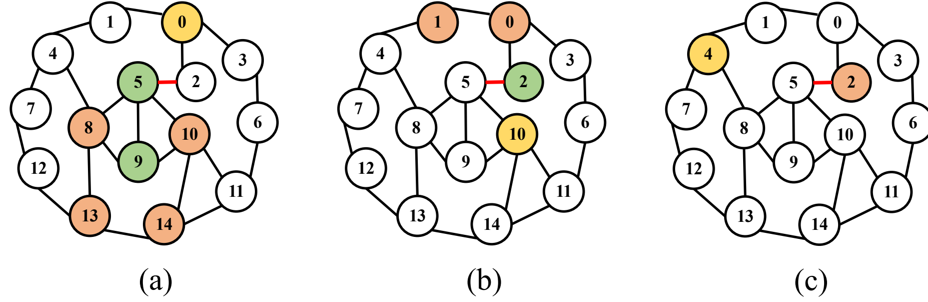


Figure 5.2: An illustration of our incremental algorithm IncHL for an edge insertion (2,5): (a), (b) and (c) describe the JP-BFSs that are rooted at the landmarks 0, 10 and 4, respectively, where yellow vertices denote the landmarks (i.e., the roots), green color denotes affected vertices whose labels are updated and red color denotes affected vertices that are pruned during the JP-BFSs.

JP-BFS starts from vertex 2 *which updates the label of* 2 *with* $V_{parent}(2) = \{5\}$ *and prunes from the landmark* 0 *after updating the highway. The JP-BFS w.r.t. the landmark* 4 *is depicted in Figure 5.2(c), which works in a similar fashion.*

Unlike IncHL^b, this algorithm allows outdated and redundant entries in a distance labelling due to weakly pruned vertices in the case of edge insertion. For clarity, we formally define the notions of outdated and redundant entries for the caser of edge insertion below.

Definition 13 (Outdated Entry)**.** *An entry* $(r, \delta_L(r,v)) \in L(v)$ *is* outdated *on a graph G iff* $\delta_L(r,v) \neq d_G(r,v)$.

Definition 14 (Redundant Entry)**.** *An entry* $(r, \delta_L(r,v)) \in L(v)$ *is* redundant *on a graph G iff* $\delta_L(r,v) = d_G(r,v)$ *and* $Q(u,v,\Gamma) = Q(u,v,\Gamma')$ *hold for any vertex u in G, where* Γ' *is obtained from* Γ *by only removing* $(r, \delta_L(r,v))$ *from* $L(v)$.

These entries do not affect the correctness of answering distance queries when a graph is changed only by edge insertions [D'angelo et al. 2019]. However, they may deteriorate query and update performance over time. Thus, to ensure that neither outdated nor redundant entries exist, we can revise the design of a merged JP-BFS in IncHL by removing its pruning step (Line 6, Algorithm 6), which leads to a distance labelling without any outdated and redundant entries. This variant of IncHL is called IncHL-m. The proof for the preservation of minimality by IncHL-m is provided in Section 5.4.

5.3.4 Decremental Algorithm

We also propose a decremental algorithm, called DecHL, which can efficiently update a highway cover labelling to reflect changes caused by an edge deletion.

Different from edge insertion, by Fact 3, distances between vertices may increase in the case of edge deletion. This thus poses the following new challenges. First, outdated and redundant entries do affect the correctness of answering distance queries

Algorithm 8: Decremental algorithm DecHL.

```
   Input: G = (V,E), G' = (V,E\{(a,b)}), (a,b) ∈ E, Γ = (H,L) over G
   Output: Γ' = (H',L') over G'
1  foreach r ∈ R with d_G(r,b) > d_G(r,a) do
2      V_r^infer ← ∅, V_r^uninfer ← FindAffected(r,b)
3      foreach (v,π) ∈ V_r^uninfer with min. π do
4          if v is already pruned then
5              if v is a landmark then
6                  δ_H(r,v) ← π (Updating H)
7              end
8          else
9              Update(r, v, π, V_r^infer, V_r^uninfer)
10         end
11         foreach w ∈ N(v) and (w,π') ∈ V_r^uninfer and π' < π+1 do
12             Modify (w,π') with (w,π+1) in V_r^uninfer
13         end
14         Remove (v,π) from V_r^uninfer to V_r^infer
15     end
16 end
```

Algorithm 9: Finding affected vertices for DecHL.

```
1  Function FindAffected(r, b)
2      Q ← ∅, V_aff ← ∅, π' ← Q(r,b,Γ)
3      Enqueue (b,π') to Q
4      while Q is not empty do
5          Dequeue (v,π') from Q
6          π ← ∞
7          if ∃w s.t. w ∈ N(v) and d_G(r,w) ≤ π' and w ∉ V_aff then
8              π ← min_w{Q(r,w,Γ)} + 1
9          end
10         Add (v, π) to V_aff
11         if v is prunable then
12             continue
13         else
14             foreach w ∈ N(v) and d_G(r,w) > π' do
15                 Enqueue (w,π'+1) to Q
16             end
17         end
18     end
19     return V_aff
20 end
```

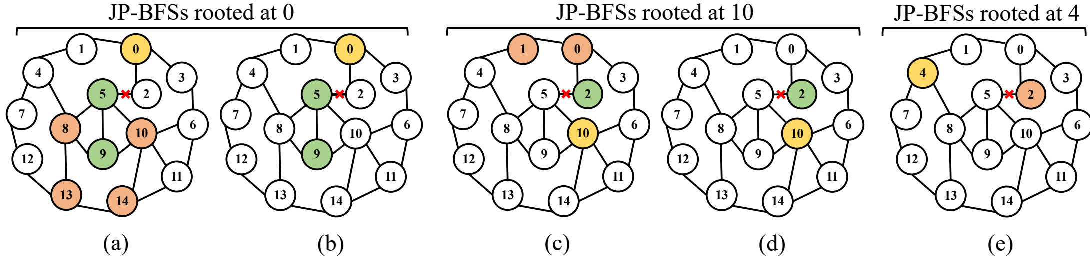


Figure 5.3: An illustration of our decremental algorithm DecHL for an edge deletion $(2,5)$: (a)-(b), (c)-(d) and (e) describe the JP-BFSs that are rooted at the landmarks 0, 10 and 4, respectively, where yellow vertices denote the landmarks (i.e., the roots), green color denotes affected vertices whose labels need to be updated, and red color denotes affected vertices being pruned.

in the case of edge deletion. Second, identifying anchor vertices becomes much harder for edge deletion due to two reasons:

(1) More than one anchor vertex may exist w.r.t. a landmark for edge deletion, in contrast to edge insertion which has exactly one anchor vertex w.r.t. a landmark;

(2) Anchor vertices can be far away from a deleted edge and are thus difficult to identify, whereas for an inserted edge there exists exactly one anchor vertex that must be incident to the inserted edge, as stated in Lemma 10.

Example 16. *Consider Figure 5.3(a), after deleting the edge* $(2,5)$*, the set of affected vertices w.r.t. the landmark* 0 *is* $\{5,8,9,10,13,14\}$*. Among these vertices, the vertices* $\{8,10,13,14\}$ *are pruned. In Figure 5.3(b), there are two anchor vertices* $\{8,10\}$ *w.r.t. the landmark* 0*. None of these vertices* 8 *and* 10 *are incident to the deleted edge* $(2,5)$*.*

Since anchor vertices for a deleted edge (a,b) may be different from the vertex b (recall that $d_G(r,b) > d_G(r,a)$ is assumed), we need to conduct two JP-BFSs w.r.t. a landmark r in the case of edge deletion. The first JP-BFS starts from b to identify affected vertices and their contingent distances through local neighborhoods iteratively. The second JP-BFS starts from anchor vertices to infer new distances of affected vertices and update their labels via a level-by-level propagation. Prunable vertices are identified and pruned away in the first JP-BFS, which helps improve update efficiency in the second JP-BFS significantly.

Algorithm 8 describes the detailed steps of our decremental algorithm. Given a graph G with a deleted edge (a,b) and a highway cover labelling $\Gamma = (H, L)$ over G. For each landmark $r \in R$, the first JP-BFS occurs in Function `FindAffected` starting from vertex b with $\pi' = Q(r,b,\Gamma)$, and enqueues (b,π') to $\mathcal{Q}$ (Line 3, `FindAffected`). Then, for every $(v,\pi') \in \mathcal{Q}$, if a neighbor w of v is unaffected and has a depth less than or equal to π', we compute the contingent distance π of v w.r.t. r based on the distance between w and r (Lines 7-9, `FindAffected`), and add (v,π) into V_{aff}

(Line 10, `FindAffected`). If v is prunable, we stop the search from v; otherwise, we continue to traverse all the children of v and enqueue them to $\mathcal{Q}$ as affected vertices (Lines 11-16, `FindAffected`). This process continues iteratively until $\mathcal{Q}$ is empty. Thus, the first JP-BFS identifies affected vertices to be updated, as well as their contingent distances to r in V_{aff}. If contingent distances of all vertices in V_{aff} are ∞, we remove the distance entry of r from the labels of these vertices because the deleted edge must cut all these vertices off from other vertices as being disconnected. Otherwise, we return V_{aff} and perform the second JP-BFS.

The second JP-BFS starts from the anchor vertices which are the vertices in V_{aff} with the minimum contingent distance (Line 3, Algorithm 8) and infers new distances of affected vertices iteratively. At each iteration, for each $(v, \pi) \in V_r^{uninfer}$ with the minimum contingent distance π, if v is already pruned, then if it is a landmark, we update the highway H; otherwise we update the label of v using Function `Update` (Lines 4-10, Algorithm 8). After that, we update the contingent distances of the affected neighbors of v (Lines 11-13, Algorithm 8). Next, we remove v from $V_r^{uninfer}$ to V_r^{infer} meaning that the new distance of v w.r.t. r has been inferred so that $V_r^{uninfer}$ contains only affected vertices whose new distances have not been inferred. This process continues until the new distances of all affected vertices in V_{aff} are inferred and updated.

Example 17. *Figure 5.3 illustrates how our decremental algorithm updates affected labels as a result of deleting an edge* $(2, 5)$. *Figure 5.3(a) depicts the first JP-BFS w.r.t. the landmark* 0 *for finding affected vertices. This JP-BFS identifies affected vertices* $\{5, 8, 9, 10, 13, 14\}$, *among which vertices* $\{8, 10, 13, 14\}$ *are pruned. Then, the second JP-BFS w.r.t. the landmark* 0 *for updating the labels is depicted in Figure 5.3(b). This JP-BFS starts from anchor vertices* $\{8, 10\}$ *with the minimum contingent distances. Then it moves to the affected neighbors* $\{5, 9\}$ *and updates their labels using the information in the labels of their parents i.e.,* $V_{parent}(5) = \{8, 10\}$ *and* $V_{parent}(9) = \{8, 10\}$. *Similarly, the first JP-BFS w.r.t. the landmark* 10 *is depicted in Figure 5.3(c) which identifies affected vertices* $\{0, 1, 2\}$, *among which* $\{0, 1\}$ *are pruned. The second JP-BFS w.r.t. the landmark* 10 *is depicted in Figure 5.3(d) which starts from anchor vertex* $\{0\}$ *with the minimum contingent distance and moves to the affected neighbor* $\{2\}$ *and update its label using the information in* $V_{parent}(2) = \{0\}$. *Next, the JP-BFSs w.r.t. the landmarks* 4 *are depicted in Figure 5.2(e), respectively, which work in the same manner.*

5.4 Theoretical Results

In this section, we prove the proposed fully dynamic method is correct, i.e., after each update operation, queries on the updated labelling return exact distances; and can preserve minimality of the labelling, a desirable property that has an impact on both query time and space efficiency. Then, we briefly analyse the complexity of the proposed algorithms.

5.4.1 Proof of Correctness

Let $G_1 \hookrightarrow G_2 \ldots, \hookrightarrow G_n$ by a sequence of update operations including both edge insertions or edge deletions. Our fully dynamic method, denoted as FulHL, is to update a highway cover labelling Γ_1 over G_1 into a highway cover labelling Γ_n over G_n such that IncHL and DecHL are applied for edge insertions and edge deletions, respectively. We consider FulHL to be *correct* iff, whenever $Q(u, v, \Gamma_1) = d_{G_1}(u, v)$ holds for any two vertices u and v in G_1, $Q(u, v, \Gamma_n) = d_{G_n}(u, v)$ also holds for any two vertices u and v in G_n.

Below, we first prove that Lemmata 11- 13 hold for both algorithms IncHL and DecHL. For simplicity, let G' refer to the changed graph after applying an edge insertion or deletion on a graph G in the input of IncHL and DecHL.

Lemma 11. *A pair (v, π) appears in $\mathcal{Q}$ iff $v \in \Lambda_r$.*

Proof. IncHL (Lines 12-14, Algorithm 6) guarantees that an inserted edge (a, b) is in one shortest-path between any vertex added to $\mathcal{Q}$ and a landmark r in G'. Similarly, DecHL (Lines 14-16, `FindAffected`) guarantees that a deleted edge (a, b) is in one shortest-path between any vertex added to $\mathcal{Q}$ and a landmark r in G. From Lemma 7, we thus have that a vertex is added to $\mathcal{Q}$ iff $v \in \Lambda_r$. □

Lemma 12. *A pair (v, π) is in V_r^{infer} iff $\pi = d_{G'}(r, v)$.*

Proof. In both IncHL (Lines 5 and 16, Algorithm 6) and DecHL (Lines 3 and 14, Algorithm 8), (v, π) is added into V_r^{infer} iff v has its new distance being inferred from unaffected and affected vertices whose new distances have already been inferred w.r.t. a landmark r in G'. Following the proof for Lemma 9, we can thus prove $\pi = d_{G'}(r, v)$. □

Lemma 13. *In Function `Update`, V_{parent} is sufficient and necessary to update $L(v)$.*

Proof. In both IncHL and DecHL, V_r^{infer} contains all inferred vertices which lie in the shortest path(s) between r and v, and whose new distances to r have been correctly inferred in G' (according to Lemma 12). Therefore, Line 2 in Function `Update` ensures that V_{parent} consists of both unaffected and affected vertices that are parents of v w.r.t. r in G', which is sufficient and necessary to update $L(v)$. □

Based on Lemmata 11-13, the definitions of prunable and weakly prunable vertices, and the fact that a highway H is updated by Algorithm 6 (Line 8) and Algorithm 8 (Line 6), respectively, the following theorem can be proven.

Theorem 4. FulHL *is correct.*

5.4.2 Preservation of Minimality

It has been reported in [Farhan et al. 2019] that, given a graph G, a highway cover labelling $\Gamma = (H, L)$ over G can be constructed using an algorithm proposed in their

work and such a highway cover labelling Γ is also guaranteed to be minimal in terms of the labelling size, i.e., $size(L') \geq size(L)$ holds for any $\Gamma' = (H, L')$ over G. Following Lemmata 11-13 and Fact 3, we can prove the following theorem.

Theorem 5. *When $G_1 \hookrightarrow G_2 \ldots, \hookrightarrow G_n$, let* IncHL-m *and* DecHL *update a highway cover labelling Γ_1 over G_1 into a highway cover labelling Γ_n over G_n for edge insertions and edge deletions, respectively. If Γ_1 is minimal over G_1, then Γ_n is also minimal over G_n.*

We use FulHL-m to refer to our fully dynamic method that preserves the minimality of labelling. More specifically, for $G_1 \hookrightarrow G_2 \ldots, \hookrightarrow G_n$ by a sequence of update operations (edge insertions or edge deletions), FulHL-m updates a highway cover labelling Γ_1 over G_1 into a highway cover labelling Γ_n over G_n such that IncHL-m and DecHL are applied for edge insertions and edge deletions, respectively.

5.4.3 Complexity Analysis

Let a be the total number of affected vertices, s be the maximum size of labels (i.e., $s = |R|$) and d be the maximum degree. For each landmark, IncHL-m and IncHL visit $O(a)$ affected vertices in the worse case, update each affected label by checking d neighbors in $O(d \cdot s)$ time. Thus, the time complexity of IncHL-m and IncHL is $O(|R| \cdot a \cdot d \cdot s)$. On the other hand, DecHL takes $O(a \cdot d \cdot s)$ time to find all affected vertices with their contingent distances (in Function `FindAffected`) and takes $O(a \cdot d)$ to fix the labels of all affected vertices. We omit s from $O(a \cdot d)$ for Algorithm 8 because distances for all unaffected neighbors of affected vertices can be stored during the first JP-BFS to avoid query cost while updating the labels during the second JP-BFS. Thus, the time complexity of DecHL is $O(|R| \times a \cdot d(s+1))$. In our experiments, we notice that a is usually orders of magnitudes smaller than the total number of vertices n while s is significantly smaller than $|R|$.

5.5 Experimental Setup

In this section, we empirically evaluate our incremental, decremental and fully dynamic algorithms. The purpose of these experiments is to answer the following questions:

- Q1 How efficiently can our incremental, decremental and fully dynamic algorithms deal with updates in very large dynamic networks, in comparison with the state-of-the-art methods?
- Q2 How does the number of landmarks affect the performance of our incremental, decremental and fully dynamic algorithms?
- Q3 How do affected vertices correlate to update efficiency in our incremental, decremental and fully dynamic algorithms?
- Q4 How well can our dynamic algorithms scale to deal with updates in very large dynamic networks?

We first introduce the datasets and the baseline methods considered in our experiments and then discuss how test data is generated to evaluate our methods.

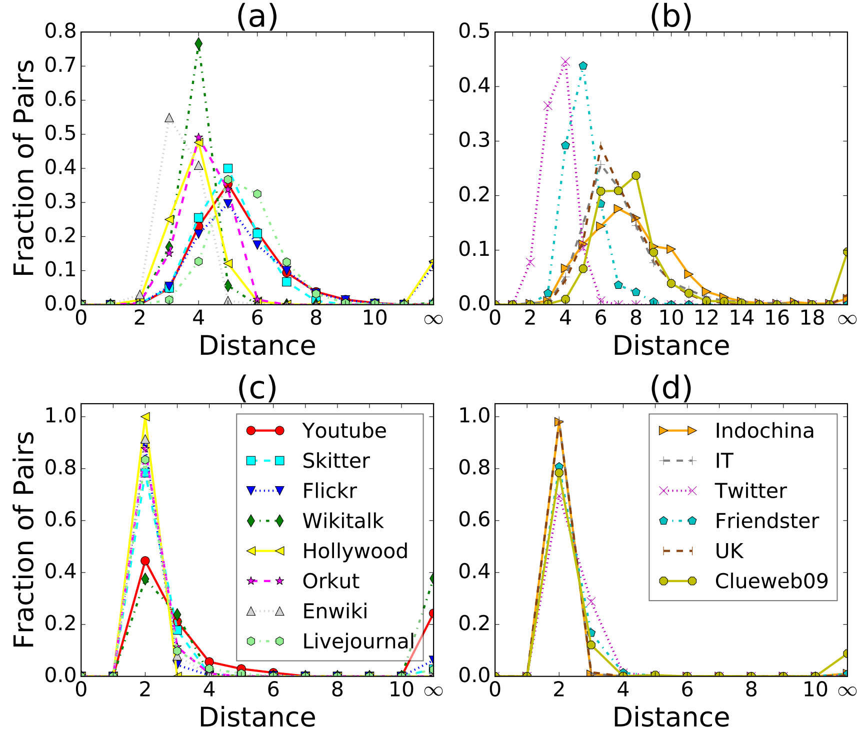


Figure 5.4: (a)-(b) show the distance distribution of 1000 pairs of vertices in E_I on all the datasets, where the distance for each pair is recorded before insertion, and (c)-(d) show the distance distribution of 1000 pairs of vertices in E_D on all the datasets, where the distance for each pair is recorded after deletion.

5.5.1 Datasets

We have used 15 real-world large networks in our experiments to answer the aforementioned questions. The detailed description and summary about these dataset is provided in Section 4.7.1 and Table 4.2.

5.5.2 Baseline Methods

We compare our fully dynamic methods (i.e. FulHL-m and FulHL), as well as the incremental and decremental algorithms (IncHL, IncHL-m, IncHL^b and DecHL), with the following state-of-the-art methods:

(1) The dynamic algorithms IncFD, DecFD and FulFD proposed in [Hayashi et al. 2016], which combine a distance labelling with a graph traversal algorithm for answering distance queries;

(2) The dynamic algorithms IncPLL [Akiba et al. 2014], DecPLL [D'angelo et al. 2019] and FulPLL proposed in [D'angelo et al. 2019; Akiba et al. 2014], which are based on the pruned landmark labelling (PLL) [Akiba et al. 2013] to answer distance queries;

(4) The parallel pruned landmark labelling methods PSL, PSL$^+$ and PSL* for static graphs proposed in [Li et al. 2019], which are also based on the pruned landmark labelling (PLL) [Akiba et al. 2013] to answer distance queries;

(5) The optimized online bidirectional BFS algorithm which answers distance queries by applying an optimized strategy to expand search from the direction with less vertices [Hayashi et al. 2016], and we name this algorithm BiBFS in our experiments.

The implementation of these baseline methods were provided by their authors and are in C++. We used the same parameter settings for these methods as suggested by their authors unless otherwise stated. The initial distance labellings were constructed using their original static methods, i.e., HL for FulHL [Farhan et al. 2019], FD for FulFD [Hayashi et al. 2016], and PLL for FulPLL [Akiba et al. 2013; D'angelo et al. 2019; Akiba et al. 2014]. For a fair comparison, we set the number of landmarks to 20 for our methods, following the same setting of FulFD [Hayashi et al. 2016] except for the largest datasets i.e., Clueweb09 and Clueweb12. We set the number of landmarks to 150 for Clueweb09 and Clueweb12 because a small number of landmarks on such large networks do not help much in pruning the search space. For parallel PLL methods PSL, PSL$^+$ and PSL* [Li et al. 2019], we set the number of threads to the total number of available cores in our server, i.e., 28.

5.5.3 Test Data Generation

For each network $G = (V, E)$, we randomly sampled 1,000 pairs of vertices as edge insertions, denoted as E_I, where $E_I \cap E = \emptyset$, and 1,000 pairs of vertices as edge deletions, denoted as E_D, where $E_D \subseteq E$. We use E_I and E_D to evaluate incremental and decremental algorithms, respectively. Then, we randomly selected 1,000 pairs of vertices $E_F \subseteq E_I \cup E_D$ with 50% from E_I (edge insertions) and 50% from E_D (edge deletions) to evaluate fully dynamic methods.

The distance distribution before applying updates in E_I is shown in Figure 5.4(a)-(b), and the distance distribution after applying the updates in E_D is shown in Figure 5.4(c)-(d). We can see that most of the pairs have a small distance ranging from 1 to 10 in E_I and from 1 to 4 in E_D for most of the datasets and only a few of them are disconnected (i.e., have distance ∞).

We use the same sampling method as described in Section 4.7.3 to generate queries for evaluating the query performance on graphs that are changed by updates in E_F. We report the labelling size produced by fully dynamic methods after performing updates in E_F.

Table 5.2: Comparison of the update time and query time of the proposed methods with the baseline methods, where "–" denotes that a method did not finish to construct labelling in one day (24 hours) or ran out of memory (512 GB).

Dataset	Update Time (UT) [ms]				Query Time (QT) [ms]		
	FulHL-m	FulHL	FulFD	FulPLL	FulHL	FulFD	FulPLL
Youtube	0.027	0.028	2.089	9040	0.005	0.010	0.006
Skitter	1.096	1.019	10.67	20400	0.027	0.019	0.006
Flickr	0.055	0.053	7.655	6810	0.007	0.012	0.009
Wikitalk	0.127	0.128	17.17	4950	0.006	0.008	0.005
Hollywood	0.223	0.212	10.54	–	0.027	0.037	–
Orkut	1.234	1.075	40.12	–	0.101	0.103	–
Enwiki	1.488	1.459	88.54	–	0.054	0.035	–
Livejournal	0.275	0.179	2.564	–	0.044	0.046	–
Indochina	1.414	0.598	107.2	–	0.737	0.839	–
IT	22.96	10.62	160.3	–	1.069	1.013	–
Twitter	73.37	72.76	2512	–	0.863	0.177	–
Friendster	2.131	2.097	21.64	–	0.814	0.904	–
UK	2.755	1.075	337.6	–	3.443	5.858	–
Clueweb09	103.1	56.25	–	–	16.93	–	–
Clueweb12	15950	1796	–	–	9.375	–	–

5.6 Results and Discussion

In this section, we discuss the results of our experiments to answer the aforementioned questions.

5.6.1 Performance Comparison

We compare the performance of our methods against the labelling-based methods and online search methods.

Labelling-based Dynamic Methods

We first compare our methods with labelling-based dynamic methods in terms of update time, labelling size and query time.

Update Time. Table 4.3 shows that the average update times taken by our methods FulHL-m and FulHL are significantly less than the average update times taken by the baseline methods FulFD and FulPLL. As we can see, only our methods can scale to very large networks with billions of vertices and edges. Specifically, FulFD failed to have results for Clueweb09 and Clueweb12, and FulPLL failed for 11 out of 13 networks. There are several reasons why FulPLL cannot scale to large networks. Firstly, FulPLL is based on the pruned landmark labelling algorithm [Akiba et al. 2013] which has very high space requirements and construction time of labelling on large networks. Secondly, FulPLL has a very high updating cost in restoring the

Table 5.3: Comparison of the update time for edge insertion and edge deletion of the proposed methods IncHL-m, IncHL, IncHL^b, and DecHL with the baseline methods.

Dataset	Incremental Algorithms					Decremental Algorithms		
	IncHL-m (ms)	IncHL (ms)	IncHL^b (ms)	IncFD (ms)	IncPLL (ms)	DecHL (ms)	DecFD (ms)	DecPLL (sec.)
Youtube	0.004	0.006	0.007	0.043	0.208	0.059	6.213	15.2
Skitter	0.133	0.075	0.194	0.447	2.189	1.443	19.48	21.3
Flickr	0.005	0.005	0.006	0.046	1.869	0.152	17.71	11.7
Wikitalk	0.002	0.003	0.004	0.022	0.073	0.231	30.08	11.0
Hollywood	0.027	0.026	0.031	0.078	48.97	0.265	21.03	–
Orkut	1.687	1.423	2.026	2.039	–	0.418	48.12	–
Enwiki	0.119	0.105	0.134	0.129	6.596	2.969	163.8	–
Livejournal	0.201	0.122	0.245	0.225	–	0.300	7.406	–
Indochina	2.587	1.187	5.443	167.7	2021	0.233	60.60	–
IT	49.77	21.34	95.92	241.8	–	5.843	210.5	–
Twitter	0.017	0.015	0.027	0.106	–	192.6	5126	–
Friendster	0.119	0.119	0.159	0.396	–	2.409	42.92	–
UK	4.071	2.132	11.49	397.7	–	0.267	151.5	–
Clueweb09	27.04	9.205	40.68	–	–	131.8	–	–
Clueweb12	26365	2061	61661	–	–	2129	–	–

2-hop cover property [D'angelo et al. 2019] for the decremental case. Overall, our methods are more than 30 times faster as compared to FulFD and several orders of magnitude faster than FulPLL.

In Table 5.3, the average update times taken by incremental and decremental algorithms are compared separately. For incremental algorithms, our methods IncHL-m and IncHL significantly outperform the baseline methods IncHL$^+$, IncFD and IncPLL on all the datasets. Further, IncHL is faster than IncHL-m which strictly preserves the minimality of labelling. This performance difference between IncHL and IncHL-m provides us good insights on the additional cost required by guaranteeing the minimality property of labelling on dynamic graphs. For the decremental algorithms, we can also see that the average update time taken by DecHL is significantly less than DecFD and DecPLL on all the datasets. DecPLL took time in seconds to update the labellings and failed to update the labelling for Hollywood, Enwiki and Indochina due to very high update time complexity, which is cubic in the worst case in terms of the number of vertices [D'angelo et al. 2019].

Labelling Size. Table 5.4 shows that the labelling sizes of our method FulHL after applying updates in E_F are significantly (ranges from 30% to 90%) smaller than the labelling sizes of FulFD and FulPLL. When updates occur on a graph, the labelling sizes of FulFD and FulHL remain stable because their average label size is bounded by the constant (i.e. the size of landmarks set $|R|$). Specifically, FulFD stores complete shortest-path trees w.r.t. the landmarks; while FulHL stores pruned shortest-path trees thereby leading to a labelling of much smaller sizes than FulFD. However, the labelling sizes of FulPLL increase because its incremental algorithm does not remove outdated and redundant entries. In Table 5.6, we present the difference Δ_{IncPLL} and

Table 5.4: Comparison of the labelling size of the proposed methods with the baseline methods, where "–" denotes that a method did not finish to construct labelling in one day (24 hours) or ran out of memory (512 GB).

Dataset	Labelling Size (LS)		
	FulHL	FulFD	FulPLL
Youtube	20 MB	83 MB	2.83 GB
Skitter	42 MB	153 MB	11.6 GB
Flickr	34 MB	152 MB	12.7 GB
Wikitalk	41 MB	74 MB	4.30 GB
Hollywood	27 MB	263 MB	–
Orkut	70 MB	711 MB	–
Enwiki	82 MB	608 MB	–
Livejournal	122 MB	663 MB	–
Indochina	87 MB	840 MB	–
IT	862 MB	4.73 GB	–
Twitter	1.14 GB	3.83 GB	–
Friendster	2.43 GB	9.14 GB	–
UK	1.78 GB	11.8 GB	–
Clueweb09	163 GB	–	–
Clueweb12	49 GB	–	–

Δ_{IncPLL} in the labelling sizes before and after updating the labelling by our method IncHL and the baseline method IncPLL, respectively. This also shows how much the labelling sizes increase after applying the updates in E_I in several datasets. It confirms that the increase in the labelling size by our method is negligibly small while IncPLL has considerably large increase in the labelling sizes. Particularly, IncPLL has a huge increase (several gigabytes) in the labelling size of Indochina for just 1000 updates. Thus, IncPLL may cause FulPLL to produce an ever increasing labelling sizes, particularly when graphs are updated frequently.

Query Time. Table 5.2 shows that the average query times after applying updates in E_F. FulHL performs comparably with FulFD and FulPLL. More specifically, as compared to FulFD, the query time of FulHL outperforms on 7 out of 11 datasets where FulFD can construct labelling. For the two largest datasets, FulFD fails to construct labelling and thus cannot answer queries. Notice that FulHL considerably underperforms FulFD on Twitter when ignoring updates on graphs. This is because the maximum degree of Twitter is very high (Table 4.2, i.e., 2997487) and FulFD maintains shortest-path trees for landmarks along with their neighbors which may cause a large fraction of pairs to be covered by very high degree landmarks. However, if we consider the overall query time on dynamic graphs as the sum of the total update time plus the query time after the update operation, then our method FulHL would indeed significantly outperform FulFD on Twitter. It has been reported in [D'angelo et al. 2019] that the average query time is dependent on the labelling size. As discussed in Section 6.8.1, the update operations do not considerably affect the labelling sizes of FulFD and FulHL; thus, their query times remain stable. The query

Table 5.5: Construction time, labelling size and query time of the state-of-the-art methods PSL, PSL$^+$ and PSL* where "–" denotes that a method did not finish to construct labelling in one day (24 hours) or ran out of memory (512 GB).

Dataset	Construction Time (sec.)			Labelling Size (GB)			Query Time (ms)		
	PSL	PSL$^+$	PSL*	PSL	PSL$^+$	PSL*	PSL	PSL$^+$	PSL*
Youtube	6	5	3	0.72	0.48	0.32	0.002	0.002	0.002
Skitter	28	24	17	2.16	1.72	1.01	0.003	0.005	0.007
Flickr	36	26	16	2.80	1.78	0.98	0.004	0.004	0.005
Wikitalk	6	4	4	0.91	0.22	0.16	0.001	0.001	0.001
Hollywood	577	325	261	11.2	6.17	4.15	0.020	0.020	0.146
Orkut	22755	22983	18971	147	146	121	0.086	0.086	0.192
Enwiki	363	368	302	10.0	9.84	7.04	0.005	0.005	0.021
Livejournal	6149	5754	3179	80.7	73.8	40.4	0.035	0.035	0.047
Indochina	336	79	71	17.3	5.05	3.39	0.004	0.003	0.007
IT	–	15599	10377	–	227	130	–	0.016	0.059
Twitter	–	–	–	–	–	–	–	–	–
Friendster	–	–		–	–	–	–	–	–
UK	–	–	–	–	–	–	–	–	–
Clueweb09	–	–	–	–	–	–	–	–	–
Clueweb12	–	–	–	–	–	–	–	–	–

time of FulPLL increases because it allows the existence of outdated and redundant entries in the labels of affected vertices which deteriorates query performance over time, particularly when graphs are updated frequently.

Table 5.6: Comparison of the difference in the labelling sizes of the proposed method IncHL and the baseline method IncPLL after applying 1,000 updates.

$size(L_{\text{After}}) - size(L_{\text{Before}})$	Datasets				
	Skitter	Flickr	Hollywood	Enwiki	Indochina
Δ_{IncPLL}	5 MB	9 MB	22 MB	2 MB	7,334 MB
Δ_{IncHL}	7 KB	5 KB	1 KB	0 KB	1,833 KB

Labelling-based Static Methods

To understand how our dynamic algorithms perform against the state-of-the-art methods for static graphs, we compare the performance of our proposed methods against the parallelised pruned landmark labelling methods PSL, PSL$^+$ and PSL* which have been shown to achieve the state-of-the-art performance for answering distance queries on static graphs [Li et al. 2019]. The results for PSL, PSL$^+$ and PSL* are presented in Table 5.5. It is worth to note that, PSL, PSL$^+$ and PSL* are not dynamic methods, thereby requiring us to reconstruct labelling from scratch after each update in a graph. As we can see in Table 5.5 that the construction time of distance labelling is by far greater than the update time of our methods FulHL-m and FulHL

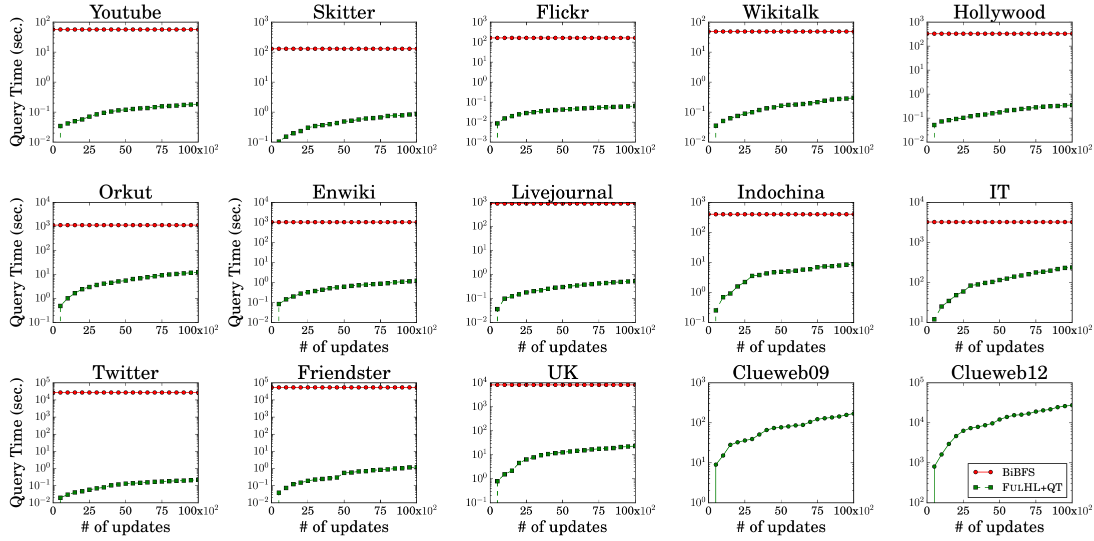


Figure 5.5: Comparison of query time of the proposed method FulHL against online search method BiBFS. BiBFS has no results for Clueweb09 and Clueweb12 because it did not finish within 24 hours.

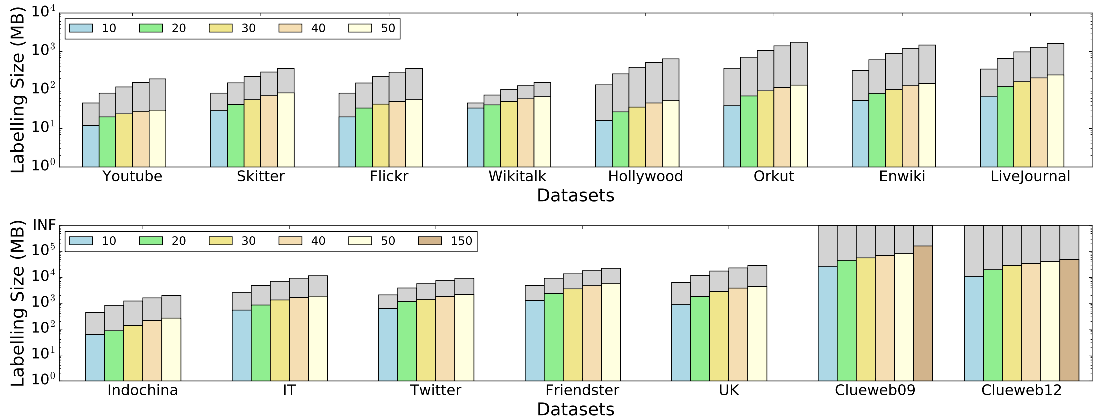


Figure 5.6: Labelling size comparison of the proposed method FulHL (in colored bars) and the baseline method FulFD (in colored plus grey bars) under 10-150 landmarks. Note that there are no results of FulFD for Clueweb09 and Clueweb12 as FulFD failed to construct labelling.

in Table 5.2. For example, our methods FulHL-m and FulHL takes 1 millisecond on average to update labelling on Skitter, whereas PSL takes 28 seconds on average to construct labelling. Furthermore, PSL, PSL$^+$ and PSL* all failed to scale to large graphs. Specifically, PSL failed for 6 out of 13 datasets, and PSL$^+$ and PSL* failed for 5 out of 13 datasets and have a very high construction cost on datasets with large average degrees such as Orkut and Livejournal. Although the construction times of PSL$^+$ and PSL* are reduced using index reduction techniques, these index reduction techniques affect the query performance. For query performance, PSL* is comparable

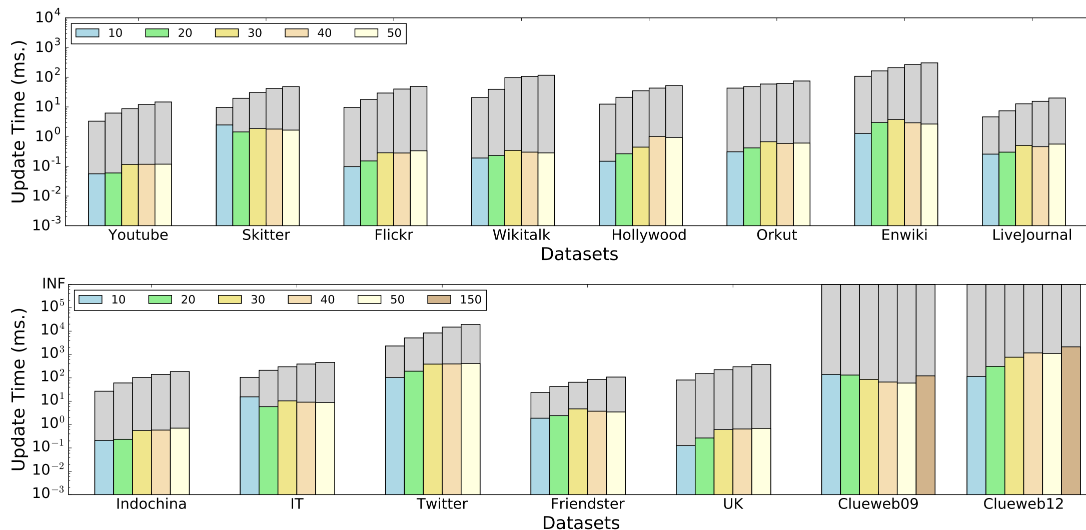


Figure 5.7: Average update time comparison for performing decremental updates between the proposed method FulHL (in colored bars) and the baseline method FulFD (in colored plus grey bars) under 10-150 landmarks. Note that there are no results of FulFD for Clueweb09 and Clueweb12 as FulFD failed to construct labelling.

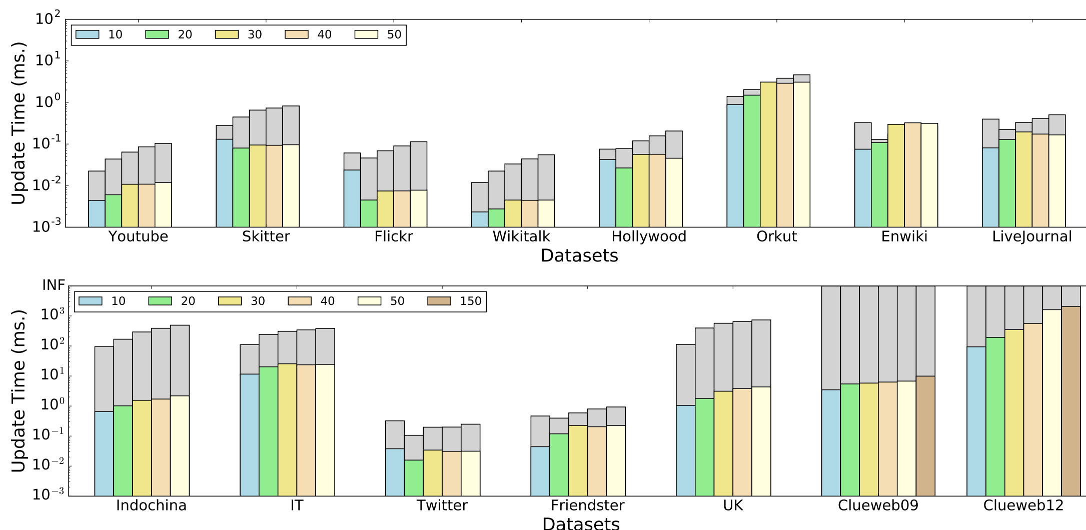


Figure 5.8: Average update time comparison for performing incremental updates between the proposed method FulHL (in colored bars) and the baseline method FulFD (in colored plus grey bars) under 10-150 landmarks. Note that there are no results of FulFD for Clueweb09 and Clueweb12 as FulFD failed to construct labelling.

to our method FulHL on Flickr, Hollywood, Orkut, Enwiki and Livejournal.

We also notice that the labelling sizes of PSL, PSL$^+$ and PSL* (as presented in Table 5.5) are much larger than FulHL (as presented in Table 5.4). As we can see in Table 5.5, PSL* produces the labelling of size almost 99% larger than the labelling of

FulHL for Orkut and IT. The query times of PSL are the fastest among all baseline methods, but unfortunately, unbearably long construction times and large labelling sizes make these methods hardly scale to very large graphs. This situation becomes even worse when the underlying graphs are dynamic. Considering the overall performance w.r.t. three main factors i.e., query time, labelling size and construction time, FulHL stands out in claiming the best trade-offs between query time, labelling size and construction time among all other baseline methods for large and dynamic graphs.

Online Search Methods

To understand the impact of labelling on distance queries, we also compare the query performance of our fully dynamic method FulHL with an online search method BiBFS. The results are shown in Figure 5.5. To make a fair comparison, we consider the overall query time of our method as the sum of the total update time on labelling for randomly sampled updates of varying sizes (i.e., 1 to 10,000) plus the query time of 1,000 queries after applying the updates, denoted as FulHL+QT. For the baseline method BiBFS, we take only the query time of 1,000 queries after applying updates. We see that, the overall performance of our methods is significantly better than BiBFS on all the datasets. In particular, our methods show promising performance on large networks even when the number of updates is 10,000. Only FulHL is able to have results within 24 hours for the two largest datsets Clueweb09 and Clueweb12. These confirm that our method is efficient and can scale to very large networks.

5.6.2 Performance under Varying Landmarks

We also evaluate the performance of our method FulHL under different numbers of landmarks. The results are presented in Figures 5.6, 5.8, 5.7 and 5.9. For the largest two datasets Clueweb09 and Clueweb12, the baseline method FulFD failed to construct labelling. Thus, Figures 5.6, 5.8, 5.7 and 5.9 does not include any results for FulFD on these two datasets.

Labelling Size

Figure 5.6 shows the labelling sizes produced by FulHL and FulFD after applying the updates in E_F under different numbers of landmarks on all the datasets. As we can see, when the number of landmarks increases, the labelling sizes of FulHL and FulFD also increase. The labelling sizes of our method FulHL increase sublinearly when increasing the number of landmarks. This is due to the pruning during JP-BFSs. In contrast, the labelling sizes of FulFD increase linearly with increasing the number of landmarks since it does not have the minimality of labelling. Thus, the labelling sizes of FulHL are always by far smaller than the labelling sizes of FulFD.

Update Time

Figures 5.8 and 5.7 show the average update times of our methods IncHL and DecHL against the baseline methods IncFD and DecFD after applying the updates in E_I and E_D, respectively. As we can see in Figure 5.8, IncHL outperforms IncFD on all the datasets against every selection of landmarks, except Orkut and Enwiki for which IncHL and IncFD have comparable results. This is because the average distances on these networks are small, and only a small fraction of vertices are affected to update their labels. In such cases, the performance of our method is comparable with IncFD. When a large fraction of vertices is affected against a graph change, our method better leverages the pruning power to perform than IncFD. For instance, our method significantly outperforms IncFD on the datasets Indochina and UK because a large fraction of affected vertices are caused by graph changes, as can be seen from Figure 5.10(d).

From Figure 5.7, we can also confirm that DecHL outperforms DecFD on all the datasets under every selection of landmarks. Further, we observe that the average update times of our methods IncHL and DecHL either remain low or increase very slowly when we increase the number of landmarks. This is because a larger number of landmarks can contribute more to leverage the pruning power of our methods, thereby performing much better than the baseline methods.

Query Time

In Figure 5.9, we also show the trend in the average query times of our method FulHL in comparison with the baseline method FulFD under varying landmarks $\{10, 20, 30, 40, 50\}$ for all datasets and $\{150\}$ for the two largest datasets after applying the updates in E_F. As we can see in Figure 5.9(a), generally the trend in the average query times of FulHL is decreasing or remains the same, whereas the trend in Figure 5.9(b) is increasing for FulFD with the increased number of landmarks. Furthermore, we notice that the trend in the average query times of Indochina, IT and UK in Figure 5.9(a) and 5.9(b) are all decreasing. This is because they have large average distances (in Table 4.2), due to which an increased number of landmarks might cover a large fraction of shortest paths and yield the tighter upper-distance bounds to help efficient querying. Overall, the increased number of landmarks help improve query time performance.

5.6.3 Analysis of Affected Vertices

To understand how affected vertices correlate with update times against different types of updates: edge insertion and deletion, we analyze the distributions of the numbers of affected vertices and their update times. The results are presented in Figures 5.10 and 5.11, in which 1000 edge insertions and edge deletions are taken from E_I and E_D, respectively, and their numbers of affected vertices and update times are sorted in ascending order.

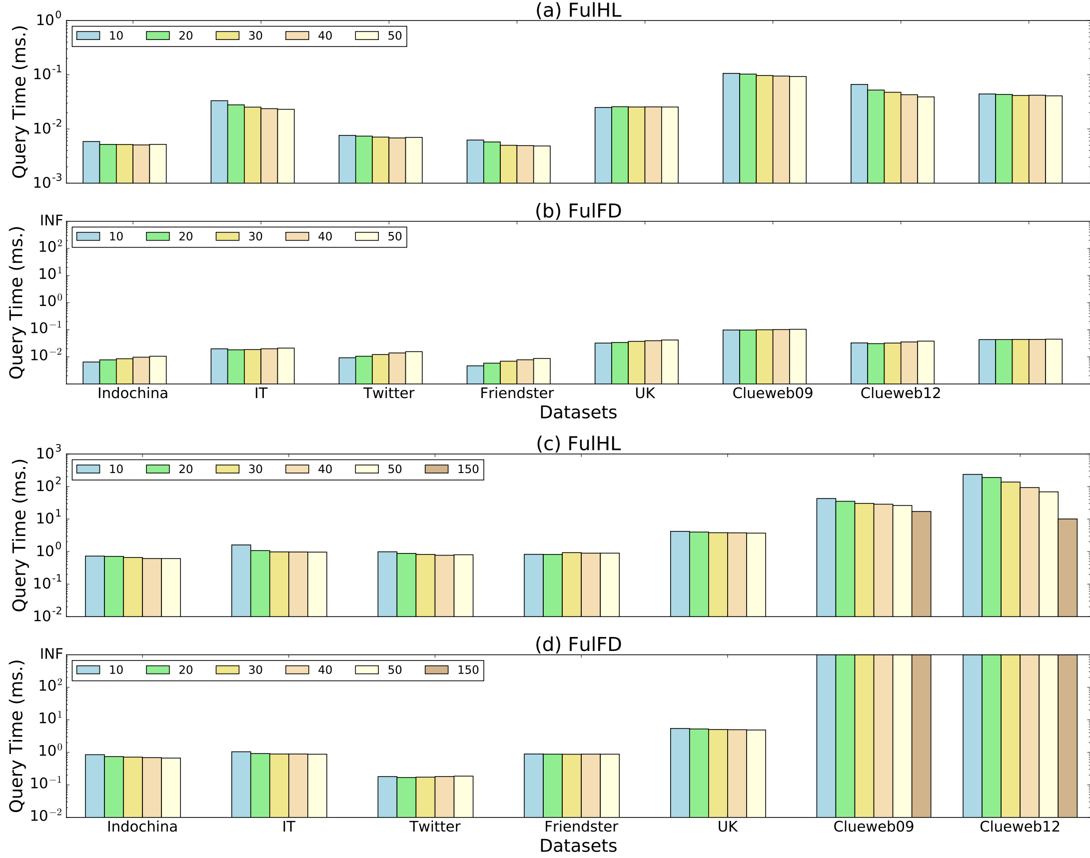


Figure 5.9: Query time comparison of the proposed method FulHL and the baseline method FulFD under 10-150 landmarks: (a) and (c) illustrate the average query time for FulHL; (b) and (d) illustrate the average query time for FulFD. Note that there are no results of FulFD for Clueweb09 and Clueweb12 as FulFD failed to construct labelling.

From the figures, we can see that there is a correlation between the number of affected vertices and update times of these updates. We observe that the difference in the number of affected vertices is not significant among these updates and only a few updates correspond to a large number of affected vertices for which the update times are also high in most of the datasets.

5.6.4 Scalability of Updates

We analyse the performance of our methods with increasing the number of updates. We start with 500 updates, and then iteratively increase 500 updates until 10,000 updates. Figures 5.12-5.13 show the average update times after constructing the labelling from scratch, and updating the labelling using our incremental and decremental algorithms after each increase.

We observe from Figure 5.12 that the update time of IncHL on all the datasets is almost always below the construction time of labelling. On IT and Twitter, the

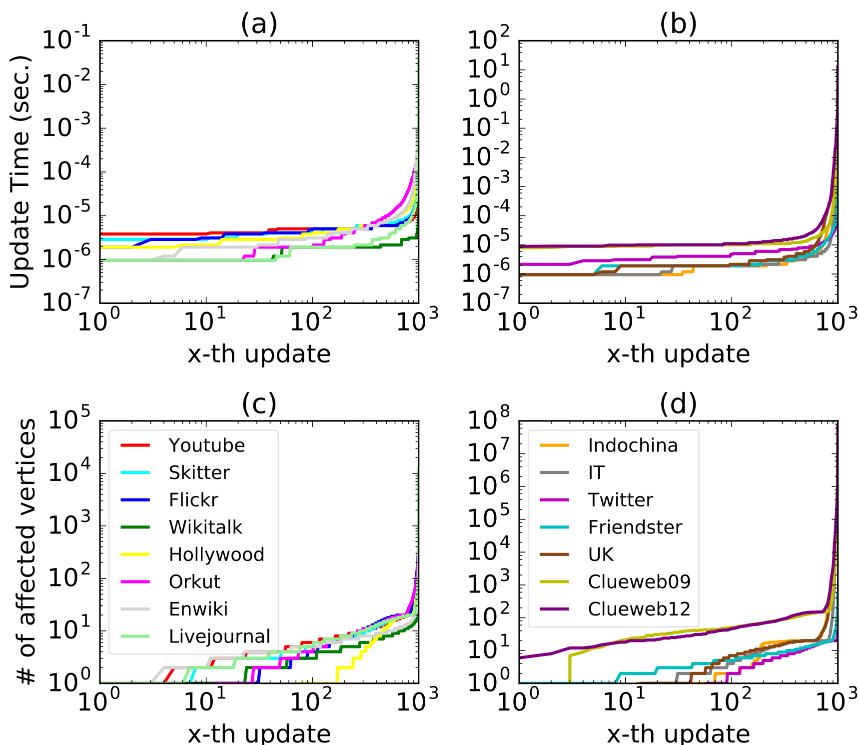


Figure 5.10: 1000 edge insertions from E_I: (a)-(b) show the distribution of update times, and (c)-(d) show the distribution of the numbers of affected vertices.

update time reaches the construction time after performing 5,000 updates. This is because the average distance of IT is large as depicted in Table 4.2, which may lead to high percentages of affected vertices to be updated; although the average distance of Twitter is small, the density of Twitter is high and fewer updates can still cause a large fraction of vertices to be affected as can be observed from Figure 5.10(d).

In Figure 5.13, we can also see that DecHL generally performs well on all the datasets. Compared to the other datasets, DecHL performs relatively worse on Skitter, Enwiki, Twitter and Clueweb12. This is because the updates in these networks can have larger distances after removal, as can be observed from the distance distribution in Figure 5.4, which may cause more vertices to be affected and require more update time as depicted in Figure 5.11. Overall, the performance of the proposed algorithms is dependent on the fraction of affected vertices and our methods can scale to perform large batches of updates efficiently.

5.7 Summary

In this chapter we have studied the distance query problem on dynamic graphs in the single-update setting. We considered two fundamental changes in dynamic graphs, i.e., edge insertions and edge deletions. We proposed incremental, decremental and

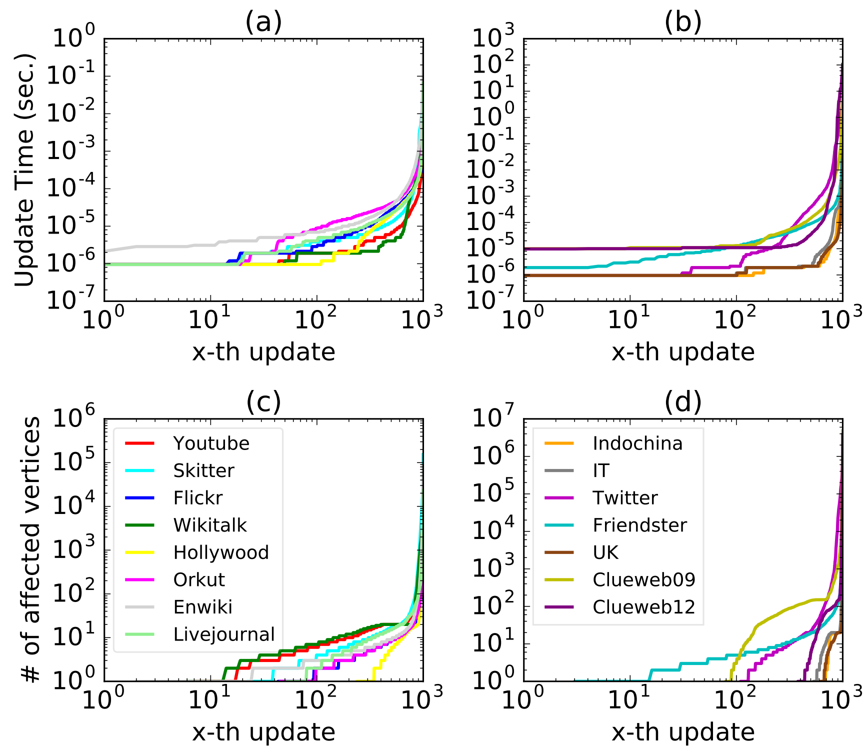


Figure 5.11: 1000 edge deletions from E_D: (a)-(b) show the distribution of update times, and (c)-(d) show the distribution of the numbers of affected vertices.

fully-dynamic algorithms that exploit the properties of the highway cover distance labelling presented in Chapter 4 in order to efficiently maintain a highway cover distance labelling for dynamic graphs undergoing changes such as edge insertions and deletions. We theoretically proved that the proposed fully dynamic method is correct and can preserve the minimality property of highway cover distance labelling after updating a highway cover distance labelling to reflect changes into a graph. We conducted extensive experiments to empirically verify the efficiency and scalability of the proposed algorithms. The results showed that the proposed algorithms significantly outperform the state-of-the-art methods.

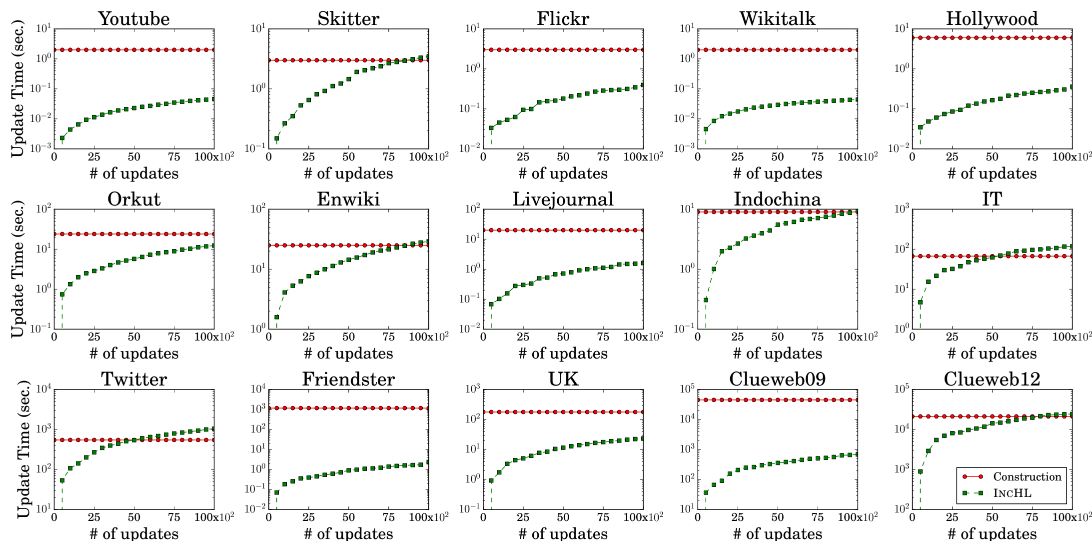


Figure 5.12: Comparison of average update time of the proposed method IncHL for performing up to 10,000 updates against the construction time.

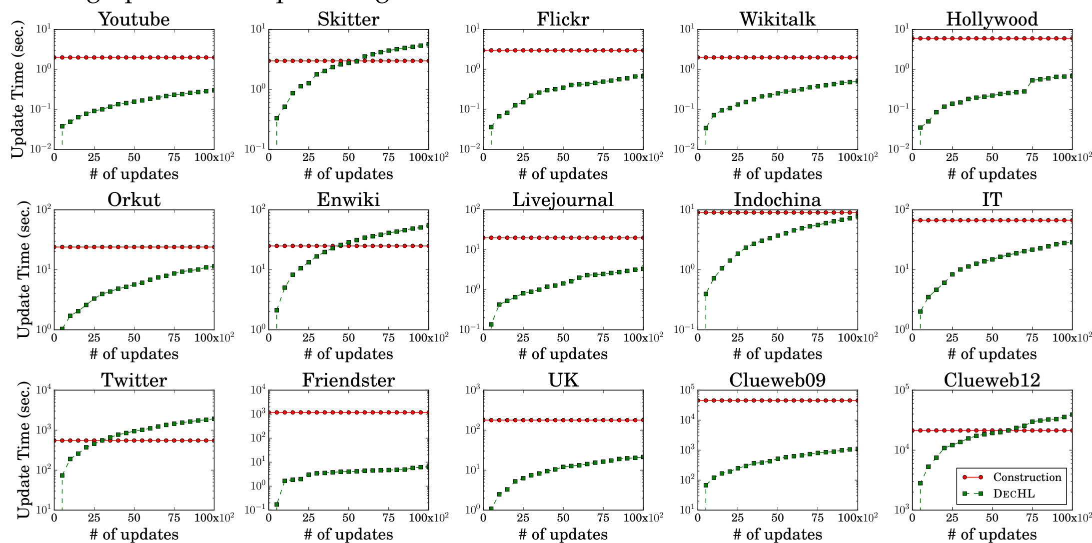


Figure 5.13: Comparison of average update time of the proposed method DecHL for performing up to 10,000 updates against the construction time.

Chapter 6

BatchHL: Batch-Dynamic Labelling For Distance Queries

6.1 Overview

In this chapter, we study the problem of answering exact shortest-path distance queries in batch-dynamic networks that undergo rapid changes in their topological structure over time. In real-world applications, it is often unrealistic to process changes such as edge insertion and deletion one by one in a sequential manner on graphs. Rather, changes may be aggregated to reflect into graphs in large batches efficiently.

We propose a batch-dynamic method, called BatchHL, to dynamize a distance labelling in order to reflect large batches of updates on a graph. BatchHL consists of two phases:

(1) *Batch search* which finds vertices whose labels are affected by batch updates.

(2) *Batch repair* which updates the labels of affected vertices to ensure correctness of distance queries and minimality of distance labelling.

Figure 6.1 presents a high-level overview of BatchHL which performs batch search and then batch repair.

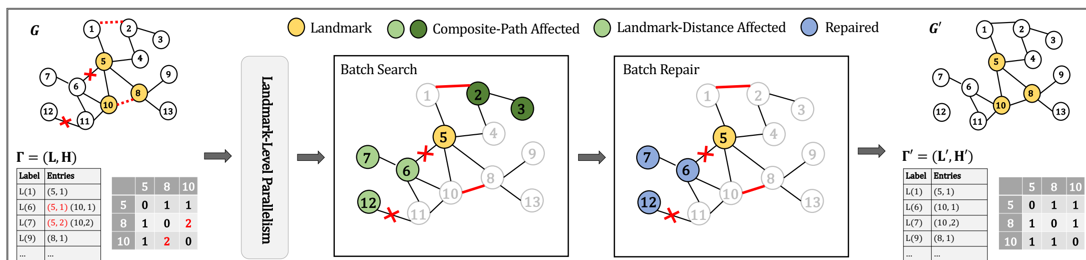


Figure 6.1: A high-level overview of our batch-dynamic method (BatchHL) which performs a batch update in two phases: 1) Batch Search: find vertices that are affected, and 2) Batch Repair: repair vertices returned by Batch Search.

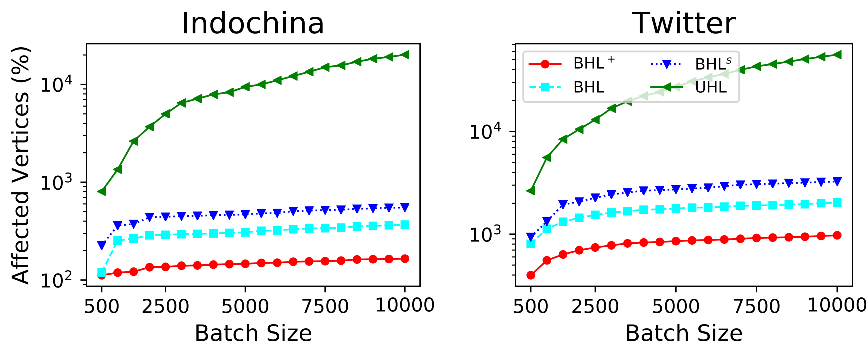


Figure 6.2: An illustration for the number of vertices affected by batch updates of varying sizes. BHL and BHL$^+$ are our batch-dynamic algorithms, BHL^s is a variant of BHL which splits edge insertions and deletions into sub-batches and performs them sequentially, and UHL handles updates in the single-update setting.

Further, we discuss the limitations of performing updates in the single-update setting, compared to the batch-update setting. Figure 6.2 illustrates the gap in the number of affected vertices by batch updates and unit updates. As we can see in Figure 6.2, the gaps in the number of vertices affected by batch updates when different variants of our method are used in the batch-update setting, in comparison with the single-update setting, the number of affected vertices in the single-update setting (i.e., UHL) is much higher than the ones in the batch-update setting (i.e., BHL^s, BHL and BHL$^+$). This is because one vertex may be affected by multiple updates in a batch, which would unavoidably lead to repeated and unnecessary computations in the single-update setting.

The main contributions of this chapter are as follows:

- We propose a batch-dynamic method which can handle batch updates efficiently and uniformly so as to reflect them on graphs by updating a highway cover labelling. Previous studies [Akiba et al. 2014; D'angelo et al. 2019] reported that handling edge deletions on a graph has been recognized as being computational expensive and difficult, even in the single-update setting. Our method alleviates this challenge and can handle both edge insertions and deletions in batches efficiently.

- We develop efficient pruning strategies in our method, i.e., in both batch search and batch repair, to eliminate repeated and unnecessary computations on graphs. As a result, when dealing with batch updates, we traverse much smaller numbers of vertices than in the single-update setting where each update is handled independently. We also design an inference mechanism to compute new distances based on boundary vertices and incorporate this into batch repair in our method.

- We prove that our proposed method can preserve the minimality of labelling on batch-dynamic graphs. Notice that, maintaining the minimality is a difficult

but highly desirable property to have for designing a distance labelling over dynamic graphs. Otherwise, a distance labelling may have increasingly unnecessary entries left in its labels and query performance would deteriorate over time.

- Our proposed method can scale to very large dynamic graphs. This is due to several reasons: the design choices on combining offline labelling and online searching, the properties of highway cover labelling, the pruning strategies in batch search and batch repair, and landmark-based parallelism. We will discuss these in detail in Section 6.7.

The rest of this chapter is organized as follows. Section 6.2 introduces the problem definition. Section 6.3 formulates the proposed batch-dynamic framework. Section 6.4 introduces several optimization techniques. Section 6.5 provides a detailed analysis of the proposed algorithm. Section 6.6 shows that the proposed algorithms are correct and can preserve the property of minimality. Section 6.7 discusses the experimental results, which compare the performance of our proposed algorithms against the baseline algorithms. Section 6.9 summarises the chapter.

6.2 Problem Definition

In this chapter, we study the *distance query* problem on batch-dynamic graphs. We formulate our problem as the batch-dynamic labelling problem for distance queries. Given a graph that undergoes rapid changes such as edge insertions or deletions over time, the batch-dynamic labelling problem is concerned about updating a highway cover distance labelling to ensure that distance queries can be correctly answered on the batch-dynamic graph. Formally, we define this problem below.

Definition 15 (Batch-Dynamic Labelling Problem)**.** *Let $G = (V, E)$ and $G' = (V, E')$ be two graphs, and G be changed to G' by a batch B of edge insertions and deletions. The* batch-dynamic labelling *problem is, given a highway cover distance labelling Γ over G such that $Q(u, v, \Gamma) = d_G(u, v)$ for any two vertices u and v in G, to compute a highway cover distance labelling Γ' over G' such that $Q(u, v, \Gamma') = d_{G'}(u, v)$ for any two vertices u and v in G'.*

6.3 Batch-dynamic Labelling Framework

In this section, we present how to answer distance queries for any two vertices in a batch-dynamic graph by combining the highway cover labelling with online searching. The key idea is to dynamically maintain a highway cover labelling on a batch-dynamic graph, and then use such a highway cover labelling to bound online searches on a sparsified search space in order to accelerate query processing.

The major challenge is how to design an algorithm that can efficiently maintain a highway cover labelling for answering distance queries on graphs that undergo batch updates, particularly when graphs are very large?

Algorithm 10: BatchHL (BHL)

Input: G, G', B, R, Γ over G
Output: Γ' over G'
1 $\Gamma' \leftarrow \Gamma$
2 **foreach** $r \in R$ **do**
3 $\quad V_{\text{AFF}} \leftarrow \texttt{BatchSearch}(G', B, r, \Gamma)$
4 $\quad \texttt{BatchRepair}(G', V_{\text{AFF}}, r, \Gamma, \Gamma')$
5 **end**

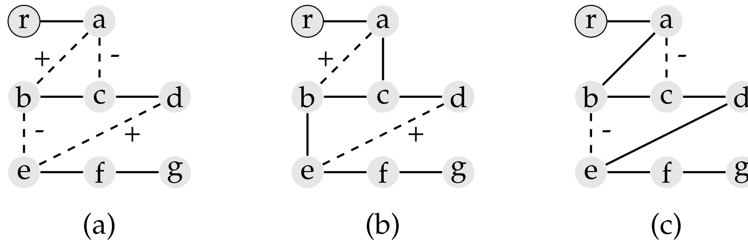


Figure 6.3: Example graphs, where r is a landmark, the edges marked by $+$ are inserted, and the edges marked by $-$ are deleted: (a) a batch update with two edge insertions and two edge deletions; (b) a sub-batch update with only two edge insertions; and (c) a sub-batch update with only two edge deletions.

6.3.1 Batch Search

In the following let $G = (V, E)$ be a graph, $R \subseteq V$ a set of landmarks and B a batch update resulting in the updated graph $G' = (V', E')$. We denote the highway cover labelling on G and G' by Γ and Γ', respectively. Our first aim is to identify vertices for which the set of shortest paths to a given landmark changes.

Definition 16 (Affected Vertex)**.** *A vertex $v \in V$ is* affected *by a batch update B w.r.t. a landmark $r \in R$ iff $P_G(r, v) \neq P_{G'}(r, v)$.*

We use $V_{\text{AFF}}(r, B) = \{v \in V \mid P_G(v, r) \neq P_{G'}(v, r)\}$ to denote the set of all affected vertices by a batch update B w.r.t. a landmark r. It is worth noting that Lemma 7 presented in Chapter 4 states how affected vertices relate to a single update (either edge insertion or edge deletion).

An edge insertion or deletion (a, b) can create or eliminate shortest paths starting from a landmark r and passing through (a, b). By Corollary 3 of chapter 4, we know that any update on an edge (a, b) with $d_G(r, a) = d_G(r, b)$ is *trivial* w.r.t. a landmark r, since such an update does not affect any vertices w.r.t. the landmark r.

Until now, the standard way of handling graph changes is to treat edge insertion and edge deletion separately, since they have opposite effects on a graph. A natural extension on batch updates would then be to devise an incremental algorithm for batch edge insertions and a decremental algorithm for batch edge deletions. However, for a batch update that contains both edge insertions and edge deletions, we

Algorithm 11: Batch Search

```
   Input: G′, B, r, Γ
   Output: V_AFF+
 1 foreach (a, b) ∈ B do
 2     if d_G(r, a) < d_G(r, b) then
 3         add (d_G(r, a) + 1, b) to Q
 4     end
 5     else if d_G(r, a) > d_G(r, b) then
 6         add (d_G(r, b) + 1, a) to Q
 7     end
 8 end
 9 while Q is not empty do
10     remove minimal (d, v) from Q
11     if v ∉ V_AFF+ then
12         add v to V_AFF+
13         foreach w ∈ N_G′(v) do
14             if d + 1 ≤ d_G(r, w) then
15                 add (d + 1, w) to Q
16             end
17         end
18     end
19 end
```

would then need to split it into two sub-batches - one for edge insertions and the other for edge deletions, and apply incremental and decremental algorithms, respectively. Thus, repeated computations across edge insertions and deletions cannot be eliminated because no interaction between edge insertion and deletion can be captured.

Example 18. *Consider Figure 6.3(a) with four updates. If handling edge insertions and deletions separately in two sub-batches as shown in Figure 6.3(b)-6.3(c), insertions of (a, b) and (d, e) lead to affected vertices $\{b, e, f, g\}$, while deletions of (a, c) and (b, e) lead to affected vertices $\{c, d, e, f, g\}$. The traversal on edges (e, f) and (f, g) is repeated.*

To overcome the aforementioned shortcomings, we propose an efficient algorithm that unifies edge insertions and deletions. The key idea is based on our observation of a "*shared pattern*" that characterises affected vertices w.r.t. a landmark in a unified way for both edge insertions and edge deletions.

Let $r \in R$ and $(a, b) \in B$. Here, (a, b) is any update. That is, (a, b) is either an inserted edge or a deleted edge.

Definition 17 (Anchor Vertex and Pre-anchor Vertex)**.** *The* anchor vertex *of an update (a, b) in a batch B is either a or b, whichever is further away from r, and the* pre-anchor vertex *of (a, b) is a vertex in $\{a, b\}$ that is not the anchor.*

Table 6.1: An illustration of anchor vertices b, c and e w.r.t. the batch update depicted in Figure 6.3(a). The anchor distance for b and c through pre-anchor vertex a is $d_G(r,a)+1=2$. The anchor distance for e through pre-anchor vertex d is $d_G(r,d)+1=2$.

	v	=	a	b	c	d	e	f	g
	$d_G(r,v)$	=	1	3	2	3	4	5	6
Anchor vertex	$d_{G'}(b,v)$	=	1	0	1	2	3	4	5
b	Eq. 6.1	=	False	True	False	False	False	False	False
Anchor vertex	$d_{G'}(c,v)$	=	2	1	0	1	2	3	4
c	Eq. 6.1	=	False	True	True	True	True	True	True
Anchor vertex	$d_{G'}(e,v)$	=	4	3	2	1	0	1	2
e	Eq. 6.1	=	False	False	False	False	True	True	True

Definition 18 (Anchor Distance)**.** *The* anchor distance *of an update (a,b) in a batch B is defined as $d_G(r,u')+1$ where u' is the pre-anchor vertex in $\{a,b\}$. Note that when $d_G(r,a) = d_G(r,b)$, there is no anchor vertex nor pre-anchor vertex corresponding to the update (a,b).*

For a batch B of updates, there exists a set of anchor vertices corresponding to the updates in B.

Lemma 14. *An affected vertex v in G w.r.t. r by a batch B of updates can be found if the following condition is satisfied by at least one anchor vertex u from B:*

$$d_G(r,v) \geq (d_G(r,u')+1) + d_{G'}(u,v). \tag{6.1}$$

Proof. By Lemma 7, if v is affected w.r.t. r, there must exist a shortest path from vertex v to r passing through (u',u). The length of such a path is $d_G(r,u')+1+d_G(u,v)$ which should always be smaller than or equal to the original length $d_G(r,v)$. Thus $d_G(r,v) \geq (d_G(r,u')+1) + d_G(u,v)$ holds. □

Example 19. *Consider Figure 6.3.a again, which has three anchor vertices b, c and e corresponding to the four updates. By applying Eq. 6.1, we can identify affected vertices $\{b,c,d,e,f,g\}$ as shown in Table 6.1.*

This striking pattern enables us to design a simple yet efficient algorithm for finding affected vertices which only needs to traverse local neighbors v of each anchor vertex u on G' recursively, i.e., computing $d_{G'}(u,v)$, regardless whether updates are edge insertions or deletions. The anchor distance $d_G(r,u')+1$ and the distance $d_G(r,v)$ on G can be efficiently computed from the highway cover labelling Γ. The searches by different updates can be combined into a single search in the order of the anchor distances plus their distances to the anchor vertices to avoid unnecessary computation.

We note that due to this unified handling of insertions and deletions, optimization that apply to only one of these operations cannot simply be applied to the combined algorithm. However, we show how one such optimization can still be leveraged in Section 6.4.1.

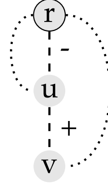


Figure 6.4: An example graph for composite-path affected vertices where r is a landmark.

Armed with these ideas, Algorithm 11 eliminates unnecessary searches on unaffected vertices v with $d_G(r,v) < d_G(r,u') + 1$ and also avoids traversing vertices affected by multiple updates more than once. However, Algorithm 11 does not precisely compute the set of all affected vertices, but a superset of it. The following example illustrates why this happens, and why it is difficult to avoid.

Example 20. *Consider the graph in Figure 6.4 with two updates. The dotted edge between r and u indicates a long path between them, and the dotted edge between r and v indicates an even longer path. When both edge deletion (r,u) and edge insertion (u,v) occur, the distance between r and u in G is used to compute the anchor distance of v for the update (u,v), ignoring that the distance between r and u has changed. It is difficult to identify whether v is affected – it hinges on whether the long path between r and v is longer than the long path between r and u plus 1, which cannot be ascertained by Γ.*

We now characterize the set of vertices returned by Algorithm 11.

Definition 19 (Composite Path)**.** *A path from r to v in $G \cup G'$ is a* composite path *iff it consists of two parts: a part that lies in G followed by a part in G'.*

A composite path is *significant* iff it passes through at least one deleted and at least one inserted edge. In Figure 6.3.a, $r-a-b-c$ and $r-a-c-d$ are insignificant composite paths, $r-a-c-d-e$ is a significant composite path, and $r-a-b-e$ is not a composite path as a deleted edge comes after an inserted edge.

Definition 20 (Composite-Path Affected)**.** *A vertex $v \in V$ is* composite-path-affected (CP-affected) by a batch update B w.r.t. a landmark $r \in R$ *iff*

(i) v is affected w.r.t. r, or

(ii) there exists a significant composite path from r to v of length $d_G(r,v)$ or less.

We will show that Algorithm 11 returns the set of all composite-path-affected vertices. Clearly this includes all affected vertices. Additional vertices due to condition (ii) are undesirable but hard to avoid, as illustrated in Example 20. From an algorithmic perspective, it happens because our starting distance is calculated w.r.t. G, so we are effectively considering paths for which the first part (from r to an anchor) lies in G, and the rest in G'.

6.3.2 Batch Repair

In the following, we develop an efficient algorithm to repair labels. At its core is an inference mechanism for the distances of affected vertices, which allows us to update their labels. We start with *boundary vertices* that lie on the boundary of affected and unaffected vertices, and for which the distance to a landmark r can be computed from neighboring vertices whose distance did not change. Importantly, even though a vertex may be affected by multiple edge updates in a batch, its r-label only needs to be updated once.

Definition 21 (Boundary Vertex)**.** *A vertex v is a* boundary vertex *w.r.t. a landmark r if v is an affected vertex but has an unaffected neighbor w, i.e. $w \notin V_{\text{AFF}}(r, B) \wedge w \in N(v)$.*

Let $v \in V_{\text{AFF}}$. For every neighbor w of v in G', $d_{G'}(r, v)$ must be upper-bounded by $d_{G'}(r, w) + 1$. If such a neighbor lies outside of V_{AFF}, the value of $d_{G'}(r, w) = d_G(r, w)$ can easily be obtained. By taking the minimum of such known upper bounds, we get a readily available distance bound for v.

Definition 22 (Distance Bound)**.** *Let $S \subset V \setminus \{r\}$ be a set of vertices. The* distance bound *of v w.r.t. S is:*

$$d_{\text{BOU}}(v, S) := \min\{d_{G'}(r, w) + 1 \mid w \in N_{G'}(v) \setminus S\}.$$

The following lemma allows us to compute the distance of vertices in V_{AFF} from r in G' using their distance bounds.

Lemma 15. *Let $S \subset V \setminus \{r\}$ and $v \in S$ with minimal distance bound. Then $d_{G'}(r, v) = d_{\text{BOU}}(v, S)$.*

Proof. For $d_{G'}(r, v) = \infty$ this is trivial. Otherwise let p be a shortest-path from r to v in G', v' is the first vertex in p that lies in S, and w its predecessor in p. Since $w \notin S$ we have $d_{\text{BOU}}(v', S) \leq d_{G'}(r, w) + 1 = d_{G'}(r, v')$. If $v' \neq v$ then $d_{G'}(r, v') < d_{G'}(r, v) \leq d_{\text{BOU}}(v, S)$, and therefore $d_{\text{BOU}}(v', S) < d_{\text{BOU}}(v, S)$. This contradicts the minimality of $d_{\text{BOU}}(v, S)$, so $v' = v$. It follows that $d_{\text{BOU}}(v, S) = d_{G'}(r, v)$. □

Note that $d_{G'}(r, v) = d_{\text{BOU}}(r, S)$ does not generally hold for every boundary vertex v. This can, for example, be seen in the graph of Example 6.5 - when computing distances to r_1, e must be repaired before f, as the new shortest-path between r_1 and f passes through e. The situation is reversed when computing distance to r_2. Based on Lemma 15, we repair labels by starting with boundary vertices that have the smallest distance bound. After each iteration, we treat affected vertices with repaired labels as being unaffected and find boundary vertices that have the smallest distance bound again. This process terminates only when the labels of all affected vertices are repaired.

To further improve efficiency, we develop a landmark pruning strategy based on the following lemma.

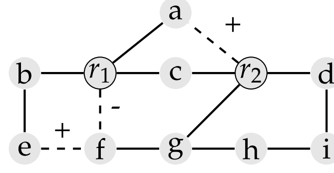


Figure 6.5: An example graph where r_1 and r_2 are landmarks, the edges marked by $+$ are inserted, and the edges marked by $-$ are deleted.

Lemma 16. *A vertex $v \in V_{\text{AFF}}(r, B)$ can be pruned from repairing (i.e., do not repair its label) if there exists a landmark $r' \in R \backslash \{r\}$ lying on one shortest path in $P_{G'}(v, r)$.*

Proof. By Definition 4, if there exists such a landmark r', $d_{G'}(r, v)$ can be computed using Equation 4.1 based on the highway and label of v w.r.t. the landmark r'. Thus, the entry in $L(v)$ w.r.t. the landmark r is not needed. □

By Lemma 16, we thus notice that, if a vertex $v \in V_{\text{AFF}}(r, B)$ can be pruned, then any vertex $v' \in \text{AFF}(r, B)$ satisfying $d_{G'}(r, v') = d_{G'}(r, v) + d_{G'}(v, v')$ can also be pruned. Putting all together, we incorporate our landmark pruning strategy into our algorithm for batch repair.

Algorithm 12 shows the pseudo-code of our batch repair algorithm. Given a graph G' and a set of all affected vertices V_{AFF}, we first compute the distance bounds of vertices in V_{AFF} using their unaffected neighbors. We then find vertices in V_{AFF} with the minimal distance bounds and remove them from V_{AFF}. By Lemma 15 their distance to r in G' equals their distance bounds. For each $v \in V_{\min}$, we check if v is prunable by Lemma 16. Then, if v is a landmark, we update the highway; otherwise we remove r-label of v and prune the search from v (Lines 8-13). If v is not prunable, we set r-label of v to $(r_i, D_{\text{BOU}}[v])$ (Line 15) and assign new distances to the affected children of v in V_{AFF} (Lines 14-15). We continue this process until V_{AFF} is empty.

6.4 Optimization

In this section, we propose an optimized batch search method which aims to precisely compute the set of affected vertices. We also present a landmark-level parallel approach to parallelize our batch-dynamic method.

6.4.1 Improved Batch Search

So far we aimed at computing affected vertices. However, changes to shortest paths between r and v do not always cause a change in distance. Thus we shall differentiate between new and eliminated paths, and strengthen the pruning condition $d + 1 \leq d_G(r, w)$ in Line 14 of Algorithm 11 to $d + 1 < d_G(r, w)$ for new paths.

Things get a little trickier though, as we may need to eliminate redundant labels, or restore previously eliminated labels when they become non-redundant. Thus even

Algorithm 12: Batch Repair

```
   Input: G′, V_AFF, r_i, Γ, Γ′
   Output: Γ′
1  foreach v ∈ V_AFF do
2  |  D_BOU[v] ← d_BOU(v, V_AFF) // use Γ to compute
3  end
4  while V_AFF is not empty do
5  |  V_min ← {v ∈ V_AFF | D_BOU[v] is minimal}
6  |  remove V_min from V_AFF
7  |  foreach v ∈ V_min do
8  |  |  if v is prunable then
9  |  |  |  if v is a landmark then
10 |  |  |  |  δ′_H(r_i, v) ← D_BOU[v]
11 |  |  |  else
12 |  |  |  |  remove r-label from Γ′(v)
13 |  |  |  end
14 |  |  else
15 |  |  |  set r-label of Γ′(v) to (r_i, D_BOU[v])
16 |  |  |  foreach w ∈ N_G′(v) ∩ V_AFF do
17 |  |  |  |  D_BOU[w] ← min(D_BOU[w], D_BOU[v] + 1)
18 |  |  |  end
19 |  |  end
20 |  end
21 end
```

if the distance between r and v does not change, the highway labelling may need to be updated.

Example 21. *Consider the graphs in Figure 6.6. In all cases, vertex v is affected, but the distance between r and v does not change. For case (a) adding the edge (b, v) does not cause a label change for v because there does not exist any landmark in the newly created path between v and r. It does however for case (b) because b is a landmark in the newly created path between v and r which will cause the r-label of v to be deleted. Similarly, deletion of (b, v) does not cause a change on the label of v in case (c) because of the presence of landmark a in a shortest-path between v and r, but causes a change in case (d) where an r-label needs to be inserted because a path passing through landmark b no longer exists and there is not any landmark on a shortest-path between v and r.*

A core difficulty in identifying whether affected vertices have changes on their labels is that label changes can happen far away from updates, further computing the changed labels of such vertices may require the consideration of vertices whose labels do not change, as illustrated by the example below.

Example 22. *Consider the graph in Figure 6.7. The distance between r and c changes, but the label of c does not change. That is because the shortest-path between r and c goes through*

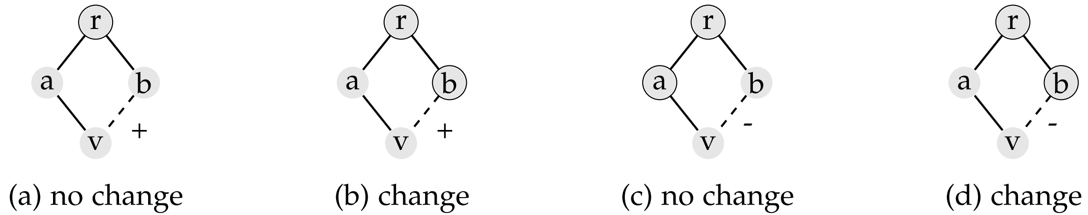


Figure 6.6: Example graphs where the landmarks are circled. (a) Adding the edge (b, v) does not cause a label change for v; (b) Adding the edge (b, v) causes the label of v to be changed where the r-label of v needs to be deleted; (c) Deleting the edge (b, v) does not cause a change on the label of v; (d) Deleting the edge (b, v) causes a change on the label of v where an r-label needs to be inserted in v.

landmark b without change. At the same time the label of v does change, as the edge (r, b) eliminates a shortest-path between r and v that passes through landmark b, similar to case (d) in Example 21. Although the label of c does not change, the changed distance between r and c is needed for computing the changed label of v. Therefore, c needs to be captured as well.

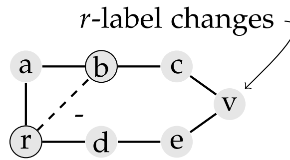


Figure 6.7: An example graph where r and b are landmarks and the edge (r, b) is deleted. The r-label of v is changed although the r-label of c does not change.

We thus need to reexamine exactly which vertices need to be returned. Firstly, this includes any vertex v for which the highway labelling must be updated. For non-landmarks the only possible change is to their r-label. For landmarks their distance to r is stored as part of the highway, and needs to be updated when it changes. Secondly, we must return any vertex for which the distance to r changes. That is because the batch repair algorithm computes the updated distance of a vertex to r from that of its neighbors, so using outdated values for a neighbor could lead to errors.

Example 23. *Consider the graph in Figure 6.8. Although the distances from r to a and b are both changed, the only vertex for which the highway cover labelling needs to be updated is a. For b, its r-label is still redundant. Using the old distance between r and b would cause our batch repair algorithm to compute $d_{G'}(r, a)$ as $d_G(r, b) + 1 = 3$.*

By considering vertices for which either label or distance changes, we can address both of the issues illustrated in Examples 22 and 23. This motivates the following definition.

Definition 23 (Landmark-Distance Affected)**.** *A vertex v is* landmark-distance-affected (LD-affected) *by a batch update B w.r.t. a landmark $r \in R$ iff it is*

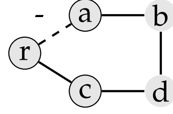


Figure 6.8: An example graph where r, a and c are landmarks and the edge (r, a) is deleted. The distances from r to a and b are both changed.

(i) label-affected: *the r-label of v changes, or*

(ii) distance-affected: *the distance between r and v changes.*

As seen in Example 21, changes to r-label without changes to distance happen whenever a new shortest-path passing through another landmark is created where none existed previously, or when the last such path is deleted. To identify such cases, we track whether a shortest-path to r passes through another landmark.

Definition 24 (Landmark Length)**.** *The* landmark length *of a path p starting from $r \in R$ is a tuple $(d, l) \in \mathbb{N} \times \mathbb{B}$ where*

- *d is the length of p (number of edges), and*
- *l is the* landmark flag, *with $l =$ True iff p passes through a landmark other than r.*

We denoted this landmark length as $|p|_{\text{L}}$. The landmark distance *between r and v in G is the minimal landmark length of paths between them, denoted as*

$$d_G^L(r, v) := \min \{ \, |p|_{\text{L}} \mid \, p \text{ is a path between } r \text{ and } v \text{ in } G \, \}$$

The ordering used to compare landmark length tuples is the lexicographical one, with True $<$ False*. The latter ensures that the landmark flag of $d_G^L(r, v)$ is set iff* any *of the shortest paths between r and v passes through another landmark.*

Lemma 17. *Let $d_{G'}^L(r, v) = (d, l)$. If $d = \infty$ or $l =$ True then v has no r-label in Γ'. Otherwise v has the r-label (r, d).*

Proof. If v has any r-label in Γ' it must be (r, d). As Γ' is minimal, this r-label exists iff it is not redundant. For $d = \infty$ redundancy of (∞, r) is obvious. Otherwise (d, r) is redundant iff the correct distance could also be computed using the highway. This happens iff a shortest-path between r and v passes through another landmark, which is indicated by the landmark flag. □

Lemma 18. *A vertex v is LD-affected iff $d_G^L(r, v) \neq d_{G'}^L(r, v)$.*

Proof. Let l_G and $l_{G'}$ denote the landmark flags of $d_G^L(r, v)$ and $d_{G'}^L(r, v)$, respectively. Condition (ii) of Definition 23 states $d_G(r, v) \neq d_{G'}(r, v)$. It suffices to show that for $d_G(r, v) = d_{G'}(r, v)$ condition (i) holds iff $l_G \neq l_{G'}$. This is trivial for $d_G(r, v) = d_{G'}(r, v) = \infty$. For finite distances it follow from Lemma 17. □

Like Algorithm 11, our improved batch search algorithm computes a superset of the set of all LD-affected vertices, albeit a smaller one. By Lemma 18 we need to return a vertex whenever its landmark distance changes. Thus we improve upon Algorithm 11 by tweaking the pruning conditions:

- *Insertion*: To affect the landmark distance, the landmark length of a new path p_{new} from r to v must be strictly smaller than the current landmark distance between r and v. Thus we check $|p_{\text{new}}|_{\text{L}} < d_G^L(r,v)$.
- *Deletion*: A deleted path p_{del} can only affect landmark distance if its landmark length was minimal, i.e., equal to the old landmark distance. This suggests checking $|p_{\text{del}}|_{\text{L}} = d_G^L(r,v)$. However, deleted paths may be obscured by shorter composite paths, so we check $|p_{\text{del}}|_{\text{L}} \leq d_G^L(r,v)$ instead.

The effects of these optimizations can be observed in Example 21, where v will not be returned for case (a) and case (c).

To apply the new pruning conditions, we must know the landmark length of a path we are following, and whether or not it passes through a deleted edge. Thus we track not only the length of each path, but also a landmark flag and a deletion flag.

Definition 25 (Extended Landmark Length)**.** *The* extended landmark length *of a path p starting from $r \in R$ is a tuple $(d,l,e) \in \mathbb{N} \times \mathbb{B} \times \mathbb{B}$ where*

- *(d,l) is the landmark length of p, and*
- *e is the* deletion flag, *with $e =$ True iff p passes through a deleted edge.*

We use lexicographical order for comparison, with True < False.

For ease of extending landmark length values we will flatten tuples implicitly, i.e., we treat $((d,l),e)$ as (d,l,e). The choice of the ordering True < False for the deletion flag is not arbitrary. When multiple search paths merge, we only track the length of the shorter one w.r.t. extended landmark length. To ensure that deleted paths will not be pruned using the stricter condition for insertion, we need to keep the deletion flag if *any* path has it, which is achieved by ordering True < False.

We apply our pruning conditions by comparing the extended landmark lengths computed for paths ending in v to the landmark distance of v in G. For this we identify the minimal extended landmark length that indicates LD-affectedness.

Lemma 19. *Let v be LD-affected w.r.t. r, and β defined as*

$$\beta(r,v) := \left(d_G^L(r,v),\ \text{True}\right)$$

Any composite path of minimal extended landmark length equals to $\beta(r,v)$ or less and pass through an updated edge.

Proof. In the following we shall always refer to composite paths from r to v. By Lemma 18 we have $d_G^L(r,v) \neq d_{G'}^L(r,v)$.

Algorithm 13: Improved Batch Search

```
    Input: G', B, r, Γ
    Output: V_AFF+
 1  foreach (a,b) ∈ B do
 2  |   e ← (a,b) is deleted
 3  |   if d_G(r,a) < d_G(r,b) then
 4  |   |   add (d^L_G(r,a) ⊕ b, e, b) to Q
 5  |   end
 6  |   else if d_G(r,a) > d_G(r,b) then
 7  |   |   add (d^L_G(r,b) ⊕ a, e, a) to Q
 8  |   end
 9  end
10  while Q is not empty do
11  |   remove minimal (d,l,e,v) from Q
12  |   if v ∉ V_AFF+ then
13  |   |   add v to V_AFF+
14  |   |   foreach w ∈ N_G'(v) do
15  |   |   |   d_w ← ((d,l) ⊕ w, e)
16  |   |   |   if d_w ≤ β(r,w) then
17  |   |   |   |   add (d_w, w) to Q
18  |   |   |   end
19  |   |   end
20  |   end
21  end
```

(1) If $d^L_G(r,v) < d^L_{G'}(r,v)$ then all paths of minimal landmark length must pass through a deleted edge. That makes their extended landmark length $\beta(r,v)$ or less.

(2) If $d^L_G(r,v) > d^L_{G'}(r,v)$ then all paths of minimal landmark length must pass through an inserted edge. Their landmark length is at most $d^L_{G'}(r,v)$, so their extended landmark length is strictly less than $\beta(r,v)$. □

Batch search with improved pruning is described in Algorithm 13. As we frequently need to update the landmark length of a path when appending another vertex, we define an operator for this:

$$(d,l) \oplus w := \begin{cases} (d+1, \text{True}) & \text{if } w \text{ is a landmark} \\ (d+1, l) & \text{otherwise} \end{cases}$$

We finally show the correctness of Algorithm 13, i.e., that all LD-affected vertices are included in its result set. Note that some additional vertices may be returned as well.

Lemma 20. *Algorithm 13 returns all LD-affected vertices.*

Proof sketch. Let v be LD-affected, and $P_{\min}$ be the set of all composite paths from r to v of minimal landmark length. By Lemma 19 these (and all their prefixes) meet the pruning condition in line 16 and pass through an updated edge. Thus the search in Algorithm 13 will follow them, starting from the last deleted or first inserted edge. While some paths may be pruned in line 12, the search will still follow at least one path $p \in P_{\min}$ with minimal landmark length. While its extended landmark length may not be minimal, this only causes p to be pruned if its landmark length equals $d^L_G(r,v)$ and p does not pass through a deleted edge. But in this case, v is not LD-affected. □

6.4.2 Improved Batch Repair

Now, we present improved batch repair method that repairs LD-affected vertices $V_{\text{AFF+}}$ returned by the improved batch search method. As we wish to eliminate redundant r-labels, we track landmark distance. Thus, we redefine distance bound as landmark distance bound for $v \in V_{\text{AFF+}}$.

Definition 26 (Landmark Distance Bound)**.** *Let $S \subset V \setminus \{r\}$ be a set of vertices. The* landmark distance bound *of v w.r.t. S is:*

$$d^L_{\text{BOU}}(v,S) := \min\{d^L_{G'}(r,w) \oplus v \mid w \in N_{G'}(v) \setminus S\};$$

The following lemma allows us to compute the landmark distance of vertices in $V_{\text{AFF+}}$ from r in G' using their landmark distance bounds.

Lemma 21. *Let $S \subset V \setminus \{r\}$ and $v \in S$ with minimal landmark distance bound. Then $d^L_{G'}(r,v) = d^L_{\text{BOU}}(v,S)$.*

Proof. For $d_{G'}(r,v) = \infty$ this is trivial. Otherwise let p be a shortest-path from r to v in G' w.r.t. landmark length, v' the first vertex in p that lies in S, and w its predecessor in p. Since $w \notin S$ we have $d^L_{\text{BOU}}(v',S) \leq d^L_{G'}(r,w) \oplus v = d^L_{G'}(r,v')$. If $v' \neq v$ then $d_{G'}(r,v') < d_{G'}(r,v) \leq d_{\text{BOU}}(v,S)$, and therefore $d_{\text{BOU}}(v',S) < d_{\text{BOU}}(v,S)$. This contradicts the minimality of $d_{\text{BOU}}(v,S)$, so $v' = v$. It follows that $d^L_{\text{BOU}}(v,S) = d^L_{G'}(r,v)$. □

Note that $d^L_{G'}(r,v) = d^L_{\text{BOU}}(r,S)$ does not generally hold for every boundary vertex v.

Algorithm 14 shows the pseudo-code of our improved batch repair algorithm. Given a graph G' and a set of all affected vertices $V_{\text{AFF+}}$, we first compute the landmark distance bounds of vertices in $V_{\text{AFF+}}$ using their unaffected neighbors. We then find vertices in $V_{\text{AFF+}}$ with minimal distance bounds and remove them from $V_{\text{AFF+}}$. By Lemma 21 their landmark distance to r in G' equals their landmark distance bounds. We use these landmark distances to update their r-labels, as well as their highway distances in the case of landmarks. Finally we update the landmark distance bounds of neighboring vertices in $V_{\text{AFF+}}$. We continue this process until $V_{\text{AFF+}}$ is empty.

Algorithm 14: Improved Batch Repair

Input: G', $V_{\text{AFF+}}$, r_i, Γ, Γ'
Output: Γ'

```
1  foreach v ∈ V_AFF+ do
2  |   D_BOU[v] ← d^L_BOU(v, V_AFF+) // use Γ to compute
3  end
4  while V_AFF+ is not empty do
5  |   V_min ← {v ∈ V_AFF+ | D_BOU[v].d is minimal}
6  |   remove V_min from V_AFF+
7  |   foreach v ∈ V_min do
8  |   |   if D_BOU[v].d = ∞ ∨ D_BOU[v].l then
9  |   |   |   remove r-label from Γ'(v)
10 |   |   else
11 |   |   |   set r-label of Γ'(v) to (r_i, D_BOU[v].d)
12 |   |   end
13 |   |   if v is a landmark then
14 |   |   |   δ'_H(r_i, v) ← D_BOU[v].d
15 |   |   end
16 |   |   foreach w ∈ N_G'(v) ∩ V_AFF+ do
17 |   |   |   D_BOU[w] ← min(D_BOU[w], D_BOU[v] ⊕ w)
18 |   |   end
19 |   end
20 end
```

6.4.3 Landmark Parallelism

Below, we show that BatchHL can be easily parallelized at the landmark level. Let $\Gamma = (H, L)$ be the unique minimal highway cover labelling over G. Then the unique minimal highway cover labelling $\Gamma' = (H', L')$ over G' may differ from Γ in:

(1) *highway:* H is changed to H'
To enable the parallelism on highway, we store H using a *highway matrix* such that $h_{ij} = h_{ji}$ for each pair of landmarks (r_i, r_j). Then, searches can be conducted in parallel to update the entries in this highway matrix.

(2) *labels:* L is changed to L'
For any vertex v, distance entries in $L(v)$ w.r.t. different landmarks are disjoint subsets. Thus updating distance entries in $L(v)$ w.r.t. different landmarks can be processed in parallel.

Putting them all together, for any batch update, we run batch search and batch repair w.r.t. each landmark in parallel to speed up the performance.

6.5 Analysis of BatchHL

The following example illustrates the individual steps that our BatchHL algorithm runs through.

Example 24. *Consider the graph and updates in Figure 6.5. The initial highway labelling* $\Gamma = (H, L)$ *will look like this:*

$$H = \{\delta_H(r_1, r_2) = 2\},$$

$L =$

a	b	c	d	e	f	g	h	i
$(r_1,1)$	$(r_1,1)$	$(r_1,1)$		$(r_1,2)$	$(r_1,1)$	$(r_1,2)$	$(r_1,3)$	
		$(r_2,1)$	$(r_2,1)$		$(r_2,2)$	$(r_2,1)$	$(r_2,2)$	$(r_2,2)$

BatchHL will initialize Γ' *as* Γ*, and then run BatchSearch and BatchRepair for both* r_1 *and* r_2*.*

- *For* r_1 *the basic BatchSearch described as Algorithm 11 returns*

$$V_{\text{AFF+}} = \{r_2, d, e, f, g, h, i\}$$

  Here vertex e is not actually affected, but still returned due to the composite path $r_1 - f - e$*.*

- *For* r_1 *the improved BatchSearch described as Algorithm 13 returns only*

$$V_{\text{AFF+}} = \{e, f, g, h\}$$

For r_2, d *and* i*, the new paths through a have the same landmark length as existing ones and are thus pruned. The eliminated path* $r_1 - f - g - h - i$ *has strictly greater landmark length than the existing path through* r_2*, and thus is pruned. Note that e is still returned due to the composite path* $r_1 - f - e$*, despite not being LD-affected.*

One of these sets is then used as input for BatchRepair described in Algorithm 14, say $V_{\text{AFF}} = \{e, f, g, h\}$*. The initial landmark bounds for this set are*

$d^L_{\text{BOU}}(r_1, \ldots) =$

e	f	g	h
$(2, \text{False})$	(∞, False)	$(3, \text{True})$	$(5, \text{True})$

Here e has the minimal distance bound, so we update $L(e)$ *by setting its* r_1*-label to* 2 *(which does not actually change* $L'(e)$*). Afterwards e is removed from* V_{AFF} *and the landmark bound of f is updated to* $(3, \text{False})$*. In the next iteration f and g are minimal, so the* r_1*-label in* $L(f)$ *is updated to* $(r_1, 3)$ *and the* r_1*-label in* $L'(g)$ *is removed. Finally* $d^L_{\text{BOU}}(r_1, h)$ *is updated to* $(4, \text{True})$ *and the* r_1*-label in* $L'(h)$ *is removed. This leaves* L' *as:*

$L' =$

a	b	c	d	e	f	g	h	i
$(r_1,1)$	$(r_1,1)$	$(r_1,1)$		$(r_1,2)$	$(r_1,3)$			
		$(r_2,1)$	$(r_2,1)$		$(r_2,2)$	$(r_2,1)$	$(r_2,2)$	$(r_2,2)$

Running BatchSearch for r_2 gives us one of

$$V_{\text{AFF}} = \{r_1, a, b, e\} \text{ or } V_{\text{AFF}} = \{a, e\}$$

depending on which algorithm (Algorithms 11 or 13) is used. Running BatchRepair Algorithm 14 on either of those inserts $(r_2, 1)$ into $L'(a)$ and $(r_2, 2)$ into $L'(e)$ for the final updated highway labelling

	a	b	c	d	e	f	g	h	i
$L' =$	$(r_1,1)$	$(r_1,1)$	$(r_1,1)$		$(r_1,2)$	$(r_1,3)$			
	$(r_2,1)$		$(r_2,1)$	$(r_2,1)$	$(r_2,3)$	$(r_2,2)$	$(r_2,1)$	$(r_2,2)$	$(r_2,2)$

6.6 Theoretical Results

In this section, we prove the correctness of our batch-dynamic method and show that our batch-dynamic method can preserve the minimality property of highway cover labelling. We also analyse the complexity of the proposed algorithms.

6.6.1 Proof of Correctness

Lemma 22. *Algorithm 11 returns the set of all CP-affected vertices.*

Proof. We show that a vertex is CP-affected iff it lies in $V_{\text{AFF+}}$ returned by Algorithm 11. We prove the "if" and "only if" below.

(if) Let $v \in V_{\text{AFF+}}$. Then there must exist a composite path p from r to v of length at most $d_G(r, v)$ that passes through at least one edge in B. If p lies in G then it lies in $P_G(r, v)$ but not in $P_{G'}(r, v)$, so v is affected. If p lies in G', then either it lies in $P_{G'}(r, v)$ or there exists an strictly shorter path p' in $P_{G'}(r, v)$. Neither p nor p' lies in $P_G(r, v)$, so v is affected. If p lies neither in G nor in G' then it must be significant. Thus v is CP-affected. in all cases.

(only if) Reversely, let v be CP-affected. If $P_G(r, v) \nsubseteq P_{G'}(r, v)$ then there exists a path p in G of length $d_G(r, v)$ that passes through a deleted edge. If $P_G(r, v) \subsetneq P_{G'}(r, v)$ then there exists a path p in G' of length at most $d_G(r, v)$ that passes through an inserted edge. Otherwise the exists a significant composite path of length at most $d_G(r, v)$. Thus, in all cases, there exists a composite path p of length at most $d_G(r, v)$ that passes through an edge in B.

Let (a, b) be either the last deleted edge that p passes through, or the first inserted edge, with $d_G(r, a) < d_G(r, b)$. Then p can be split into p_{ra} from r to a, (a, b) and p_{bv} from b to v such that p_{ra} lies in G and p_{bv} in G'. The search in Algorithm 11 starting at b will use $|p_{rb}| = d_G(r, a) + 1$ as the anchor distance for b, and proceed along p_{bv}. Thus for every vertex $w \in p_{bv}$, including v, it will obtain $|p_{rw}| \leq d_G(r, w)$ and add w to $V_{\text{AFF+}}$. □

6.6.2 Preservation of Minimality

Theorem 6. *The highway labelling Γ' returned by Algorithm 10 is the minimal highway labelling for G'.*

Proof. By Lemmas 22 and 20, the vertex set V_{AFF} returned by `BatchSearch` contains all LD-affected vertices, regardless of which Algorithm (11 or 13) is used. By Lemma 18 this means that for vertices outside of V_{AFF} the landmark distance to r_i does not change, so that in line 2 of Algorithm 10 the value of $d^L_{\text{BOU}}(v, V_{\text{AFF}})$ can be computed from Γ. From Lemma 21 it follows that $D_{\text{BOU}}[v] = d^L_{G'}(r_i, v)$ whenever vertex v lies in $V_{\min}$.

For each landmark r and each vertex LD-affected w.r.t. r we update the r-label of v in Γ' based on its landmark distance to r in G'. By Lemma 17 these updates are correct. As the r-labels of vertices outside of V_{AFF} do not change, and we initialized Γ' using Γ, this leave all vertices with correct r-labels, for all $r \in R$, so the distance labelling of Γ' is correct and minimal. Highway is updated for vertices in V_{AFF} as well, for all $r \in R$, and do not change for others by Definition 23. □

6.6.3 Complexity Analysis

Let a be the total number of affected vertices, s be the maximum label size and d be the maximum degree. In our method, a refers to CP-affected vertices in the batch-update setting which is different from FulFD [Hayashi et al. 2016] and FulPLL [D'angelo et al. 2019] in the single-update setting. We perform a number $|R|$ of BFSs to construct highway labelling in $O(|R| \cdot |V|)$ time and space. Then, we update highway labelling in Algorithm 10 where Algorithm 11 visits $O(a)$ vertices and for each affected vertex performs d queries to check its affected neighbors in $O(d \cdot s)$ time. Thus, the time complexity of Algorithms 11 and 13 is $O(a \cdot d \cdot s)$. Note that Algorithm 13 further reduces the total number of CP-affected vertices and is naturally faster than Algorithm 11. In practice, s and d are closer to the average values, and a is usually orders of magnitudes smaller than the total number of vertices in a graph. Next, Algorithm 12 repairs CP-affected vertices returned by Algorithm 11 which in the worse case could repair the labels of all CP-affected vertices. To decide whether the label of an affected vertex needs to be repaired, we check its neighbors in $O(d)$. Thus, the time complexity of Algorithm 14 is $(a \cdot d)$, and the overall time complexity of Algorithm 10 is $O(|R| \cdot a \cdot d \cdot s)$ using $O(|V|)$ space. We omit s from the time complexity of Algorithm 12 and 14 because we store distances for all unaffected neighbors of affected vertices in Algorithms 11 and 13.

6.7 Experimental Setup

In this section, we present the datasets, the baseline methods and the test data generation process for the evaluation of our proposed batch-dynamic method.

Table 6.2: Datasets, where $size(G)$ denotes the size of a graph G with each edge appearing in the forward and reverse adjacency lists and being represented by 8 bytes.

Dataset	Network	n	m	m/n	avg. deg.	max. deg.	avg. dist.	$size(G)$
Italianwiki	web (d)	1.2M	35M	16.6	33.25	81090	3.5	153 MB
Frenchwiki	web (d)	2.2M	59M	13.2	26.36	137021	3.9	223 MB

6.7.1 Datasets

We use 17 real-world large networks in our experiments. The detailed description and summary for 15 out of 17 of these datasets are provided in Section 4.7.1 and Table 4.2, respectively. The other two datasets are two dynamic real-world networks whose topology evolves over time. The summary of these two datasets i.e., Italianwiki and Frenchwiki, is provided in Table 6.2. These datasets are accessible at Koblenz Network Collection [Kunegis 2013]. We briefly describe them below.

- **Italianwiki:** This is a web graph which shows the evolution of hyperlinks between articles of the Italian Wikipedia. The articles are represented as nodes and the existence or removal of hyperlinks between the articles are represented as edges. An edge between a pair of articles indicates that a hyperlink was added or removed depending on the edge weight (-1 for removal or +1 for addition).

- **Frenchwiki:** This is a web graph which shows the evolution of hyperlinks between articles of the French Wikipedia. The nodes represent articles and edges represent hyperlinks between articles. An edge between a pair of articles indicates that a hyperlink was added or removed depending on the edge weight (-1 for removal or +1 for addition) [D'angelo et al. 2019].

6.7.2 Baseline Methods

We consider the following variants of our batch-dynamic algorithm, (1) BHL: which uses the batch search described in Algorithm 11 and the batch repair described in Algorithm 12, (2) BHL$^+$: which uses the improved batch search described in Algorithm 13 and the batch repair described in Algorithm 14, and (3) BHL^p: which is a parallel variant of BHL$^+$. We compare these variants with the state-of-the-art methods as follows:

- FulFD [Hayashi et al. 2016]: A fully dynamic method that incorporates two algorithms IncFD and DecFD to update distance labelling for edge insertions and deletions, and then combines it with a graph traversal algorithm to answer distance queries.

- FulPLL [D'angelo et al. 2019]: A fully dynamic 2-hop cover labelling method which is composed of two separate dynamic algorithms. The first algorithm

was proposed in [Akiba et al. 2014] to answer distance queries on graphs undergoing edge insertions and the second algorithm was proposed in [D'angelo et al. 2019] to answer distance queries on graphs undergoing edge deletions. This method is based on the pruned landmark labelling (PLL) [Akiba et al. 2013].

– PSL* [Li et al. 2019]: A parallel algorithm which constructs pruned landmark labelling for static graphs to answer distance queries.

– BiBFS [Hayashi et al. 2016]: An online bidirectional BFS algorithm which answers distance queries using an optimized strategy to expand searches from the direction with fewer vertices.

Note that FulFD and FulPLL can handle only a single edge insertion/deletion at a time. Thus, for a fair comparison, we also consider a unit-update variant of our algorithm: treating our method BHL$^+$ in the unit update setting by performing one update at a time. We call this unit-update variant as UHL$^+$. The code for FulFD, FulPLL and PSL* was provided by their authors and implemented in C++. We use the same parameter settings as suggested by their authors unless otherwise stated. For a fair comparison, we also select high degree landmarks and set them to 20 in the same way as FulFD for our methods. We set the number of threads to 20 for PSL* as well as for the parallel variant of our method BHL^p.

6.7.3 Test Data Generation

For our batch-dynamic variants, we generate 10 batches for the first 15 datasets, where each batch contains 1,000 edges that are randomly selected. We consider three batch update settings for testing:

(1) *Decremental* - delete these batches and measure the average deletion time;

(2) *Incremental* - add these batches followed by decremental updates and measure the average insertion time;

(3) *Fully dynamic* - randomly select 50% updates in each of these 10 batches to delete and then measure the average update time after applying these batches.

For the two datasets that are real-world dynamic networks, we select 10 batches in the order of their timestamps, each containing 1,000 real-world inserted/deleted edges and measure the average update time after applying them in a streaming fashion.

For the methods FulFD, FulPLL and UHL$^+$, we randomly sample 1000 edges and follow the same update settings as above to measure the update time of performing updates one by one. These settings enable us to explore the impacts of edge insertions and edge deletions respectively, in addition to their combined effect.

In Figure 6.9, we report the distance distribution of edges in these batches after deleting. As we can see, the distances in all datasets are small ranging from 1 to 6. This shows that the updates are mostly from densely connected components of these

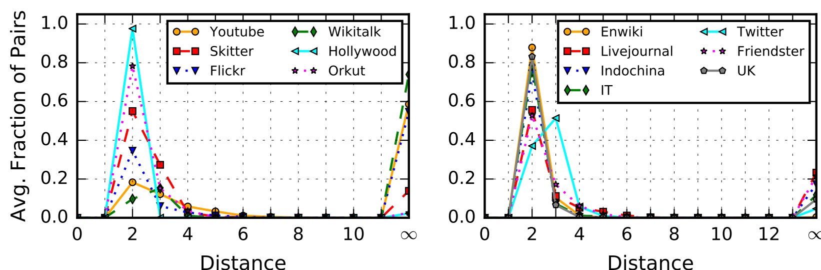


Figure 6.9: Distance distribution of batch updates.

networks which may cause fewer vertices to be affected in the *incremental* setting. Further, only a small number of updates are disconnected (i.e., have distance ∞) in most of these datasets.

We use the same sampling method as described in Section 4.7.3 to generate queries in order to evaluate the query performance on graphs being changed by fully dynamic batch updates. We also report the average labelling size produced by our batch-dynamic method BHL$^+$ after performing fully dynamic batch updates.

6.8 Results and Discussion

Now we present our experimental results and discuss them in detail.

6.8.1 Performance Comparison

We compare our methods against the labelling-based methods in terms of update time, labelling size and query time.

Update Time

Table 6.4 present the average update time in the fully dynamic setting and Table 6.3 present the average update time in the incremental and decremental settings of our proposed method and the baseline methods.

Fully dynamic setting. From Table 6.3, we see that our proposed methods BHL^p, BHL$^+$, and BHL significantly outperform FulFD and FulPLL on all datasets w.r.t. update time. In particular, our methods BHL^p and BHL$^+$ are over 15 times faster than FulFD on most of the datasets and several orders of magnitude faster than FulPLL. FulPLL only works on four graphs and fails to scale to large graphs with more than 100 millions. Further, the performance difference of BHL$^+$ and BHL is due to the fact that our improved batch search in BHL$^+$ can further prune away affected vertices that do not need to be repaired, and in practice they are significant in amount as can be seen in Table 6.6. Our methods also significantly outperform FuLFD on the real-world dynamic networks: Italianwiki and Frenchwiki. We can

Table 6.3: Comparing update time and query time of our methods BHL^p, BHL$^+$, and BHL with the state-of-the-art dynamic methods, where the batch size is 1,000 and thus the update time reported for every method is for 1,000 updates.

Dataset	Fully Dynamic Batch Update Time (sec.)						Query Time [ms]			
	BHL^p	BHL$^+$	BHL	UHL$^+$	FulFD	FulPLL	BHL$^+$	FulFD	FulPLL	PSL*
Youtube	0.046	0.070	0.208	0.091	1.249	9.110	0.005	0.010	0.045	0.002
Skitter	0.147	0.601	0.902	1.587	5.986	8.770	0.029	0.020	0.082	0.007
Flickr	0.024	0.026	0.130	0.099	2.152	6.300	0.007	0.013	0.102	0.005
Wikitalk	0.029	0.025	0.101	0.134	2.926	4.550	0.006	0.008	0.031	0.001
Hollywood	0.008	0.014	0.115	0.056	4.423	–	0.026	0.036	–	0.143
Orkut	0.537	1.775	5.855	4.539	13.30	–	0.102	0.156	–	0.203
Enwiki	0.508	1.681	10.50	3.952	121.7	–	0.053	0.051	–	0.021
Livejournal	0.221	0.306	0.873	0.379	4.736	–	0.043	0.051	–	0.047
Indochina	0.543	1.181	1.547	9.575	20.63	–	0.788	0.767	–	0.007
IT	1.494	4.502	5.433	26.57	129.6	–	1.173	1.103	–	0.059
Twitter	13.29	49.62	115.7	125.6	5103	–	0.868	0.174	–	–
Friendster	0.409	0.410	0.811	21.93	23.27	–	0.815	0.902	–	–
UK	14.45	41.46	40.79	56.50	110.1	–	1.174	5.233	–	–
Clueweb09	–	–	–	–	–	–	–	–	–	–
Clueweb12	–	–	–	–	–	–	–	–	–	–
Italianwiki	0.001	0.001	0.025	0.051	6.623	–	0.008	0.014	–	0.006
Frenchwiki	0.003	0.004	0.067	0.098	5.289	–	0.009	0.016	–	0.006

also observe that the average update time of BHL$^+$, BHL and BHL^p is always by far smaller than recomputing labelling from scratch, i.e., construction time of BHL$^+$ in Table 6.5. Notice that, we consider the same construction time for BHL^p and BHL as BHL$^+$, which is smaller than the construction time of baseline methods FulFD [Hayashi et al. 2016] and PSL* [Li et al. 2019] on all datasets. We can see that the parallel variant of PLL (PSL*) still failed to construct labelling for the largest three datasets.

Incremental setting. Table 6.4 also shows that our methods BHL$^+$, BHL^p are considerably faster than the baseline methods IncFD and IncPLL. Even though IncFD and IncPLL do not preserve the minimality of distance labellings and thus do not spend time to delete outdated and redundant label entries, they are still slower than our methods. We can also see BHL$^+$ and BHL^p in the batch update setting are significantly faster than UHL$^+$ in the unit update setting. This is because UHL$^+$ requires extra usage of resource for each single update and involves in repeated and unnecessary computations. Here it is also to note that BHL^p does not perform well on the last four datasets as compared to BHL. This is because there only exist a very small number of average affected vertices against the total number of affected vertices as shown in Table 6.6. This confirms that the parallel variant of our method works very well when a large number of vertices are affected by batch updates; otherwise it may introduce unneeded thread overhead.

Decremental setting. It is evident from Table 6.4 that our methods BHL$^+$ and BHL^p are much faster than DecFD and DecPLL on all the datasets in this setting. Espe-

Table 6.4: Comparing update time of our methods BHL^p and BHL$^+$ with the state-of-the-art dynamic methods, where the batch size is 1,000 and thus the update time reported for every method is for 1,000 updates.

Dataset	Incremental Batch Update Time (sec.)					Decremental Batch Update Time (sec.)				
	BHL^p	BHL$^+$	UHL$^+$	IncFD	IncPLL	BHL^p	BHL$^+$	UHL$^+$	DecFD	DecPLL
Youtube	0.003	0.008	0.048	0.154	0.194	0.070	0.169	0.239	3.181	9.850
Skitter	0.002	0.006	0.069	0.117	1.312	0.163	0.751	2.382	14.15	31.50
Flickr	0.003	0.008	0.072	0.053	1.259	0.030	0.041	0.107	3.364	13.40
Wikitalk	0.002	0.005	0.097	0.029	0.081	0.046	0.044	0.147	5.674	9.820
Hollywood	0.001	0.002	0.046	0.090	27.53	0.017	0.031	0.071	8.401	–
Orkut	0.005	0.014	0.127	0.367	–	0.677	0.035	5.921	23.94	–
Enwiki	0.008	0.012	0.168	0.316	4.916	0.770	3.079	8.194	251.2	–
Livejournal	0.006	0.010	0.202	0.244	–	0.299	0.570	0.731	4.736	–
Indochina	0.015	0.011	0.308	0.141	4.680	0.553	1.346	19.20	44.92	–
IT	0.101	0.033	13.21	0.147	–	1.505	4.699	42.75	285.6	–
Twitter	0.125	0.024	13.09	0.263	–	19.17	68.85	231.8	9460	–
Friendster	0.163	0.035	20.96	0.254	–	0.420	0.738	21.87	30.38	–
UK	0.218	0.055	4.349	0.258	–	14.99	42.29	75.20	257.3	–
Clueweb09	–	–	–	–	–	–	–	–	–	–
Clueweb12	–	–	–	–	–	–	–	–	–	–
Italianwiki	–	–	–	–	–	–	–	–	–	–
Frenchwiki	–	–	–	–	–	–	–	–	–	–

cially, BHL and BHL^p can achieve outstanding performance on networks which have a high average degree such as Twitter, Flickr and Hollywood. Due to inherent complexity of edge deletion on graphs (i.e., increasing distances), DecFD and DecPLL take very long in identifying and updating labels of affected vertices. As we can see, DecPLL does not have results on 8 out of 12 datasets. This is because while applying decremental updates their software either crashed or did not finish when the datasets are large that is why we don't have query time and labelling size after updates for these datasets in Table 6.3 and 6.5. Furthermore, our methods BHL$^+$ and BHL^p outperform UHL$^+$ because both leverage the benefit of handling updates in a batch and significantly reduce repeated computations during identifying and repairing the labels of affected vertices.

Labelling Size

Table 6.5 shows that BHL$^+$ has significantly smaller labelling size than FulFD, FulPLL and PSL* on all the datasets. When an update occurs, the labelling size of FulFD remains unchanged because they store complete shortest-path trees at all times. In contrast, BHL$^+$ stores pruned shortest-path trees preserving the property of minimality. Nonetheless, the labelling size of BHL$^+$ remains stable in practice because the average label size is bounded by a constant, i.e., the number of landmarks. The labelling size of FulPLL may increase significantly because IncPLL does not remove outdated and redundant distance entries and there is also no bound on labelling size. The parallel variant of PLL (PSL*) which exploit PLL properties to reduce labelling

Table 6.5: Comparing performance of our method BHL^{+} with the baseline methods in terms of the construction time and labelling size. Note that when a method did not finish the labelling construction in 24 hours, we denote it as "–".

Dataset	Construction Time (CT) [s]				Labelling Size (LS)			
	BHL^{+}	FulFD	FulPLL	PSL*	BHL^{+}	FulFD	FulPLL	PSL*
Youtube	1.46	3.56	84	4	20 MB	83 MB	3.14 GB	318 MB
Skitter	2.68	8.31	511	21	42 MB	153 MB	11.9 GB	1.01 GB
Flickr	3.17	10.8	546	23	34 MB	152 MB	13.1 GB	0.98 GB
Wikitalk	1.93	4.68	92	4	41 MB	74 MB	5.22 GB	160 MB
Hollywood	6.32	24.7	9,782	377	27 MB	263 MB	–	4.15 GB
Orkut	24.6	90.3	–	26,310	70 MB	711 MB	–	121 GB
Enwiki	24.4	91.1	7,382	389	82 MB	608 MB	–	7.04 GB
Livejournal	20.3	48.3	–	4,441	122 MB	663 MB	–	50.5 GB
Indochina	9.06	30.1	3,205	86	85 MB	838 MB	–	3.39 GB
IT	76.4	231	–	10,377	854 MB	4.73 GB	–	130 GB
Twitter	540	2,010	–	–	1.14 GB	3.83 GB	–	–
Friendster	1,202	3,476	–	–	2.43 GB	9.14 GB	–	–
UK	176	625	–	–	1.78 GB	11.8 GB	–	–
Clueweb09	46,366	–	–	–		–	–	–
Clueweb12	22,370	–	–	–		–	–	–
Italianwiki	6	15	–	215	23 MB	159 MB	–	0.81 GB
Frenchwiki	11	25	–	433	46 MB	272 MB	–	1.54 GB

size still produces labelling of very large size as compared to BHL^{+}.

Query Time

Table 6.3 shows that the average query time of BHL^{+} is comparable with FulFD and faster than FulPLL. It has been previously shown [D'angelo et al. 2019] that the average query time is largely dependent on labelling size. Since the dynamic operations do not considerably affect the labelling size for BHL^{+} and FulFD, their query times remain stable. On Twitter, the query time of BHL^{+} underperforms FulFD because FulFD also maintains the shortest-path information for the neighborhood of landmarks and we can see that the maximum degree of Twitter is very high which might cause many pairs to be covered by landmarks. However, the query time for FulPLL may considerably increase over time because they do not remove outdated entries, leading to labelling of increasing sizes.

Although the query time of PSL* in Table 6.3 is better than BHL^{+} on some datasets, it only handles static graphs. For dynamic graphs, it has the following limitations: (1) the cost of re-constructing labelling from scratch after each batch update is too high to afford, particularly when batch updates are frequent or when underlying dynamic graph is large which is evident from Table 6.5, (2) the labelling size is much larger than BHL^{+}. As we can see in Table 6.5, PSL* produces the labelling of size almost 99% larger than the labelling of BHL^{+} for Orkut thus possess a high query cost as well. Considering the overall performance w.r.t. three main factors

Table 6.6: Comparing the average number of vertices affected by BHL$^+$ and BHL after performing batch updates on all the datasets.

Dataset	$\|V\|$	BHL$^+$			BHL
		Delete	Add	Mix	Mix
Youtube	1.1 M	366 K	23 K	166 K	476 K
Skitter	1.7 M	971 K	11 K	834 K	1,266 K
Flickr	1.7 M	55 K	22 K	42 K	157 K
Wikitalk	2.4 M	127 K	16 K	81 K	474 K
Hollywood	1.1 M	14 K	2 K	7 K	41 K
Orkut	3.1 M	503 K	3 K	293 K	982 K
Enwiki	4.2 M	1,220 K	4 K	712 K	3,587 K
Livejournal	4.8 M	276 K	12 K	156 K	454 K
Indochina	7.4 M	2,079 K	15 K	200 K	3,085 K
IT	41 M	5,878 K	28 K	5,681 K	6,759 K
Twitter	42 M	10,622 K	2 K	8,341 K	20,705 K
Friendster	66 M	66 K	6 K	36 K	80 K
UK	106 M	54,515 K	12 K	54,026 K	54,864 K
Clueweb09	–	–	–	–	–
Clueweb12	–	–	–	–	–
Italianwiki	1.2 M	–	–	337	9 K
Frenchwiki	2.2 M	–	–	3 K	45 K

Table 6.7: Comparing update time, construction time (CT), query time (QT) and labelling size (LS) on directed graphs.

Dataset	BHL^p [s]	BHL$^+$ [s]	BHL [s]	CT [s]	QT [ms]	LS
Wikitalk	0.02	0.04	0.17	2.03	0.001	54 MB
Enwiki	2.98	12.5	28.0	46.8	0.023	177 MB
Livejournal	7.54	18.9	15.1	44.6	0.050	222 MB
Twitter	16.2	64.4	142	931	0.312	1.7 GB

i.e., query time, labelling size and construction time, BHL$^+$ stands out in claiming the best trade-off between these factors.

6.8.2 Performance under Varying Landmarks

Figure 6.10 shows how the update time of our method BHL$^+$ in the fully dynamic setting behaves when increasing the number of landmarks. We can see that the update time for almost all datasets grows till 30 landmarks and then either decreases or remains stable. This is because selecting a larger number of landmarks can better leverage the pruning power of our method. On Twitter, we observe that the update time grows linearly due to its very high average degree which leads to a large fraction of vertices to be affected as can be seen in Table 6.6 for 20 landmarks. We can also see in Figure 6.11 that the query time decreases or remains the same for almost

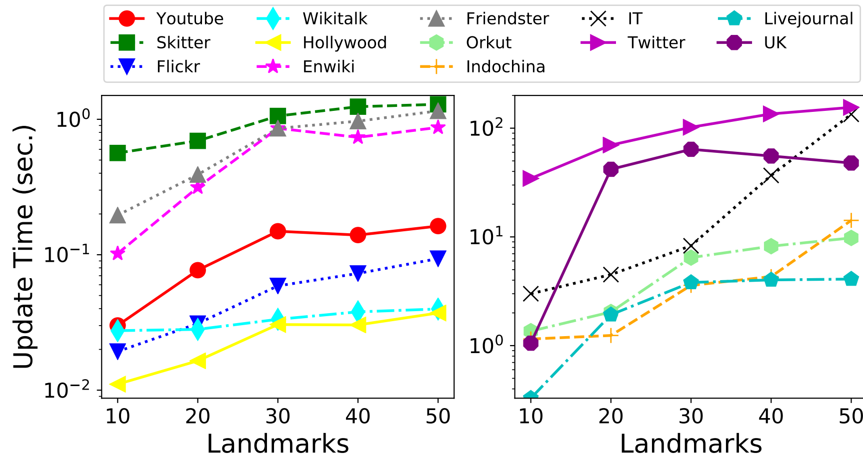


Figure 6.10: Update time under 10-50 landmarks.

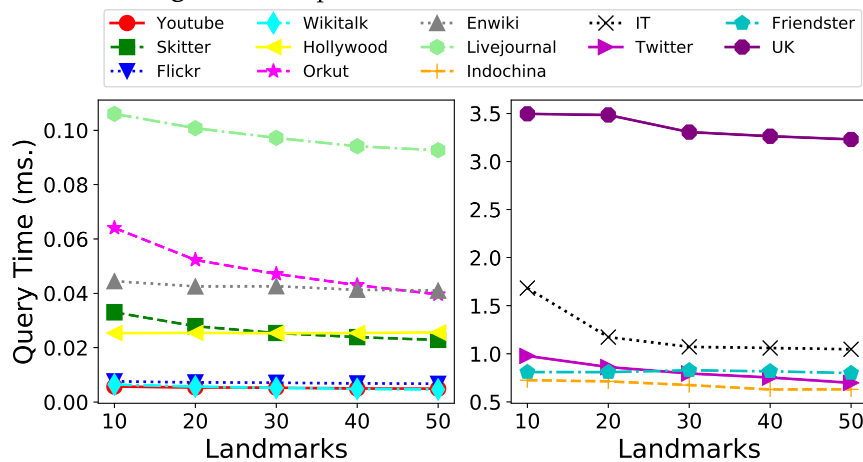


Figure 6.11: Query time under 10-50 landmarks.

all datasets with the increased number of landmarks. Particularly, the query time of Twitter, Orkut and Livejournal decreases because they have a very high average degree and selecting a larger number of high degree landmarks contributes greatly towards shortest-path coverage and makes querying process faster.

6.8.3 Performance under Varying Batch Size

We also compare the total time of querying and updating on dynamic graphs. To make a fair comparison, the total time of our methods BHL$^+$ and BHL^p, and the baseline method FuLFD is the total time to perform a batch update plus the query time to perform 1000 queries after the batch update and then averaged over 1000 queries, denoted as BHL$^+$+QT, BHL^p+QT and FuLFD+QT, respectively. We conduct the experiments for 5 randomly sampled fully dynamic batch updates of varying sizes, i.e., 500 to 10,000 in each batch.

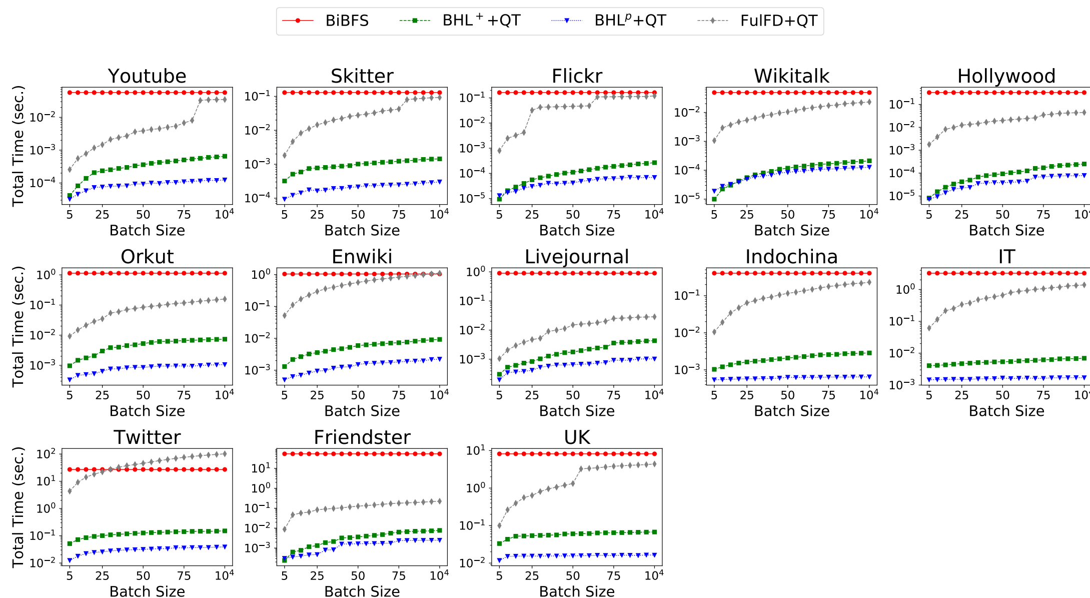


Figure 6.12: Comparing the total time of querying and updating by the proposed methods against online search methods.

Figure 6.12 presents the results. For the baseline method BiBFS, we take only the query time averaged over 1000 queries after applying a batch update. We see that, the overall performance of our methods is significantly better than the baseline methods on all the datasets. It is worth noticing that BHL^p is not only more efficient than BHL$^+$, but also their efficiency gap becomes larger when the size of batch updates increases. This shows that the parallelism power of BHL^p can be better leveraged for batch updates of larger sizes. We can also observe that the update time along with the query time of our methods grows fast for batches of smaller sizes (with up to 1000 updates) and then grows very slowly when batch sizes become very large which shows that our methods are robust w.r.t the increased batch size.

6.8.4 Performance on Directed Graphs

We also conduct experiments on directed graphs. We can see in Table 6.7 that the update time of our methods is significantly smaller than the construction time of labelling from scratch. The update time of our optimized method BHL$^+$ is faster than the method BHL on all datasets except Livejournal. On Livejournal, the amount of affected vertices traversed by both BHL and BHL$^+$ is the same; however, due to additional overhead of computing extended landmark lengths, BHL$^+$ under-performs BHL. BHL^p is still the fastest among all methods. Our methods are also efficient in performing queries and have small labelling sizes.

6.9 Summary

In this chapter, we have studied the distance query problem on batch-dynamic graphs that undergo rapid changes such as edge insertions or deletions in their topological structure. We proposed a novel method that can efficiently handle such changes in batches on graphs in order to answer distance queries correctly. Our proposed method is also based on the properties of the highway cover distance labelling presented in Chapter 4. It explores the combinatorial nature of interactions occurring between different types of updates (edge insertions and edge deletions) in a batch for efficiently maintaining a highway cover distance labelling that reflects batch updates on graphs. We further developed a parallel variant to speed up the performance. We proved the correctness and analyzed the complexity of our proposed method. We also showed that our proposed method can preserve the minimality property of a highway cover distance labelling. We empirically verified the efficiency and scalability of our approach on 17 real-world networks from a variety of domains. The results showed that our proposed algorithms significantly outperform the state-of-the-art methods.

Chapter 7

Extensions

In this chapter, we provide a brief discussion on extending the proposed methods to directed and weighted graphs.

7.1 Directed graphs

The methods proposed in this thesis can be extended to handle directed graphs. For directed graphs, we redefine $d_G(s,t)$ as the distance from vertex s to vertex t. Then, we store two sets of labels, namely *forward label* $L_f(v)$ and *backward label* $L_b(v)$ for each vertex $v \in V$. These forward and reverse labels contain pairs $(r_i, \delta_{r_i v})$ that are computed by performing forward and backward pruned BFSs w.r.t. each landmark $r_i \in R$, respectively. We also store a *forward highway* $H_f = (R, \delta_{H_f})$ and a *backward highway* $H_b = (R, \delta_{H_b})$, where for any two landmarks $r_i, r_j \in R$, $\delta_{H_f}(r_i, r_j) = d_G(r_i, r_j)$ and $\delta_{H_b}(r_j, r_i) = d_G(r_j, r_i)$. To answer a distance query (s,t), we can use $L_f(s)$ and $L_b(t)$ to compute the upper bound distance from s to t in the same way as described in Equation 4.2.

To repair the labels and highways under single-update setting (proposed in Chapter 5), we conduct JP-BFSs twice: once in the forward direction and once in the backward direction. Similarly, in the batch-update setting (proposed in Chapter 6), we perform batch search and batch repair methods twice: once in the forward direction and once in the backward direction. Then the correct upper bound for a distance query (s,t) can be computed using the updated labels $L_f(s)$, $L_b(t)$, and highways δ_{H_f}, δ_{H_b} in the same way as described in Equation 4.2.

7.2 Weighted graphs

The methods proposed in this thesis can also be extended to handle non-negative weighted graphs.

- For the method proposed in Chapter 4, we conduct pruned Dijkstra's algorithm in place of pruned BFSs for constructing the labelling.

- For the method proposed in Chapter 5, we conduct jumped-and-pruned flavour of Dijkstra's algorithm in place of JP-BFSs to repair the labels and highways

under single-update setting.

- Similarly, for the batch-update setting proposed in Chapter 6, we use Dijkstra's algorithm in our batch search and batch repair algorithms.

For non-negative weighted graphs, we consider updates in the form of edge weight increase or decrease instead of edge insertion or deletion. Our methods can then handle weight increases in a similar way to edge deletions, and weight decreases in a similar way to edge insertions.

Chapter 8

Conclusions and Future Work

In this thesis, we have studied the distance query problem which is a fundamental problem in graph theory. The existing algorithms to address this problem suffer from scalability. Thus, we first proposed a highly scalable method to answer distance queries on static graphs that may have billions of vertices and edges. Then, we developed dynamic methods that consider both the single-update setting and the batch-update setting to perform graph updates on labelling efficiently for fast and accurate distance querying on dynamic graphs. In the following we summarise our conclusions for each of these problems.

- In Chapter 4, we have studied the distance query problem on large static graphs. Our solution combines an offline distance labelling and online searching to efficiently answer exact distance queries on large static graphs. We present a novel property called *highway cover labelling* and propose a labelling construction algorithm which enables fast construction of highway cover distance labelling for billion-scale networks. Then, we formulate a querying framework that combines a highway cover distance labelling with distance-bounded shortest-path searches to enable fast distance computation. We prove that our proposed labelling construction algorithm can construct a unique highway cover distance labelling independent of the order of landmarks being applied for construction; further, such a unique highway cover distance labelling is highway cover minimal. Due to these nice highway cover labelling properties, we also develop a parallel algorithm to speed up the highway cover distance labelling construction process by conducting BFSs simultaneously w.r.t. multiple landmarks. We showed in our experiments that the proposed method is efficient and scalale, outperforming the state-of-the-art methods on 15 real-world networks from a variety of domains.

- In Chapter 5, we have studied the distance query problem on large dynamic graphs that undergo changes such as edge insertions and deletions one by one. We develop two algorithms called *incremental algorithm* and *decremental algorithm* against these two fundamental types of changes in dynamic graphs, i.e., edge insertions and edge deletions, respectively. The proposed algorithms further exploit the properties of highway cover labelling in order to efficiently

maintain a highway cover distance labelling for dynamic networks. Then, we combine these two algorithms to propose a fully dynamic solution that can reflect both edge insertions and deletions efficiently into a graph. We theoretically prove that the proposed fully dynamic method is correct and can preserve the minimality property of a highway cover distance labelling after updating the highway cover distance labelling to reflect changes into a graph. We conduct extensive experiments to empirically verify the efficiency and scalability of the proposed algorithms. The results showed that the proposed algorithms can significantly outperform the state-of-the-art methods on 15 real-world networks from a variety of domains.

- In Chapter 6, we have studied the distance query problem on batch-dynamic graphs that undergo rapid changes such as edge insertions or deletions in a batch. We propose a batch-dynamic method that can efficiently process edge insertions and deletions together in large batches in order to answer distance queries accurately when the topological structure of a graph has been changed. The proposed method explores the combinatorial nature of interactions occurring between different types of updates in a batch in order to efficiently maintain a highway cover distance labelling for reflecting batches of updates on graphs. We also develop a parallel variant that can parallelize the update on a highway cover distance labelling with respect to different landmarks so as to speed up the performance. We prove the correctness and analyze the complexity of the proposed method. We show that the proposed method can still preserve the minimality property of highway cover labelling. We empirically verify the efficiency and scalability of the proposed method on 17 real-world networks from a variety of domains (i.e., 15 real-world networks as in Chapters 4 and 5 plus two additional dynamic real-world networks). The results show that the proposed method can significantly outperform the state-of-the-art methods.

For future work, one of the interesting directions to explore further is road networks. All the methods presented in this thesis have used a novel highway structure as an integral part of their solutions. In the literature, a number of methods targeted for addressing the distance query problem [Akiba et al. 2014; Jin et al. 2012; Abraham et al. 2010] on road networks have originally been motivated by the idea of highway structure that commonly exists in road networks. Thus, it would be interesting to explore: how well the highway cover distance labelling method proposed in Chapter 4 can perform for road networks. Moreover, real-world road networks are weighted and their topological structures often remain unchanged but their edge weights are dynamically updated under dynamic conditions such as changing traffic conditions. Hence, it would be interesting to extend and explore the applicability of the dynamic methods proposed in Chapter 5 and Chapter 6 to weighted road networks in a way that updates are modeled in terms of edge weight increase or edge weight decrease instead of removal or insertion of edges.

Another interesting direction to explore in future research is the selection of

highly central landmarks. It has been previously known that landmarks with high centrality have the potential to cover the shortest paths between a significant fraction of vertex pairs in a graph and are thus desirable for being selected for distance labelling. In our work, highly central landmarks can help produce highway cover distance labellings of a smaller size (with a smaller average label size as well), thereby leading to reduced search space and boosting the query performance. Thus, it would be interesting to investigate and develop an efficient method for selecting a number of highly central landmarks that could considerably save us searching on a sparsified graph with much better quality of upper bounds in our querying framework presented in Chapter 4. Further, re-selection of highly central landmarks could also be required after a certain amount of changes occurring on the topological structure of a dynamic network. This would help optimize the size of a highway cover distance labelling and query performance. Therefore, it is also interesting to explore the problems such as: after how much changes on the topological structure of a dynamic graph, re-positioning of landmarks is required.

Bibliography

Abraham, I., Delling, D., Goldberg, A. V., and Werneck, R. F. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th International Conference on Experimental Algorithms* (2011), pp. 230–241. (pp. 2, 6, 15, 18, 19, 32)

Abraham, I., Delling, D., Goldberg, A. V., and Werneck, R. F. 2012. Hierarchical hub labelings for shortest paths. In *Proceedings of the 20th Annual European Conference on Algorithms* (2012), pp. 24–35. (pp. 2, 6, 12, 15, 16, 17, 25, 26, 32, 38, 43)

Abraham, I., Fiat, A., Goldberg, A. V., and Werneck, R. F. 2010. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10 (USA, 2010), pp. 782–793. Society for Industrial and Applied Mathematics. (pp. 18, 116)

Acar, U. A., Anderson, D., Blelloch, G. E., and Dhulipala, L. 2019. Parallel batch-dynamic graph connectivity. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19 (New York, NY, USA, 2019), pp. 381–392. Association for Computing Machinery. (p. 8)

Acar, U. A., Anderson, D., Blelloch, G. E., Dhulipala, L., and Westrick, S. 2020. Parallel batch-dynamic trees via change propagation. In *ESA* (2020). (p. 8)

Akiba, T., Iwata, Y., Kawarabayashi, K.-i., and Kawata, Y. 2014. Fast shortest-path distance queries on road networks by pruned highway labeling. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)* (2014), pp. 147–154. SIAM. (pp. 6, 19, 116)

Akiba, T., Iwata, Y., and Yoshida, Y. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 349–360. (pp. xv, 3, 4, 6, 7, 9, 15, 16, 17, 20, 25, 26, 30, 32, 33, 34, 37, 38, 42, 43, 44, 52, 70, 71, 103)

Akiba, T., Iwata, Y., and Yoshida, Y. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proceedings of the 23rd International Conference on World Wide Web* (2014), pp. 237–248. (pp. xix, 6, 7, 20, 21, 41, 51, 52, 70, 84, 103)

Akiba, T., Sommer, C., and Kawarabayashi, K.-i. 2012. Shortest-path queries for complex networks: Exploiting low tree-width outside the core. In *Proceedings of the 15th International Conference on Extending Database Technology* (2012), pp. 144–155. (pp. 1, 3, 6, 16)

Backstrom, L., Huttenlocher, D., Kleinberg, J., and Lan, X. 2006. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD (2006), pp. 44–54. (pp. 1, 42)

Bast, H., Funke, S., Sanders, P., and Schultes, D. 2007. Fast routing in road networks with transit nodes. *Science 316*, 5824, 566–566. (p. 18)

Batz, G. V., Delling, D., Sanders, P., and Vetter, C. 2009. Time-dependent contraction hierarchies. In *Proceedings of the Meeting on Algorithm Engineering and Experiments* (USA, 2009), pp. 97–105. Society for Industrial and Applied Mathematics. (p. 22)

Batz, G. V., Geisberger, R., Neubauer, S., and Sanders, P. 2010. Time-dependent contraction hierarchies and approximation. In *Proceedings of the 9th International Conference on Experimental Algorithms*, SEA'10 (Berlin, Heidelberg, 2010), pp. 166–177. Springer-Verlag. (p. 22)

Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., and Wagner, D. 2010. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *ACM J. Exp. Algorithmics 15*. (p. 18)

Bernstein, A. 2009. Fully dynamic (2 + epsilon) approximate all-pairs shortest paths with fast query and close to linear update time. In *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*, FOCS (2009), pp. 693–702. (p. 21)

Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., and Hwang, D.-U. 2006. Complex networks: Structure and dynamics. *Physics Reports 424*, 4-5 (April), 175–308. (pp. 1, 3)

Boldi, P., Rosa, M., Santini, M., and Vigna, S. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*, WWW (2011), pp. 587–596. (pp. 41, 42)

Boldi, P., Santini, M., and Vigna, S. 2008. A large time-aware graph. *SIGIR Forum 42*, 2, 33–38. (p. 42)

Boldi, P. and Vigna, S. 2004. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW (2004), pp. 595–602. (pp. 5, 41, 42)

Callaway, D. S., Newman, M. E. J., Strogatz, S. H., and Watts, D. J. 2000. Network robustness and fragility: Percolation on random graphs. *Phys. Rev. Lett. 85*, 5468–5471. (p. 16)

Chang, L., Yu, J. X., Qin, L., Cheng, H., and Qiao, M. 2012. The exact distance to destination in undirected world. *The VLDB Journal 21*, 6 (Dec.), 869–888. (pp. 6, 15, 16, 17)

Cheng, J. and Yu, J. X. 2009. On-line exact shortest distance query processing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT (2009), pp. 481–492. (p. 16)

Cohen, E., Delling, D., Fuchs, F., Goldberg, A. V., Goldszmidt, M., and Werneck, R. F. 2013. Scalable similarity estimation in social networks: Closeness, node labels, and random edge lengths. In *Proceedings of the First ACM Conference on Online Social Networks*, COSN '13 (New York, NY, USA, 2013), pp. 131–142. Association for Computing Machinery. (p. 1)

Cohen, E., Halperin, E., Kaplan, H., and Zwick, U. 2002. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2002), pp. 937–946. (pp. 1, 6, 7, 11, 12, 15, 16, 32)

Das Sarma, A., Gollapudi, S., Najork, M., and Panigrahy, R. 2010. A sketch-based distance oracle for web-scale graphs. WSDM '10 (New York, NY, USA, 2010), pp. 401–410. Association for Computing Machinery. (pp. 17, 18)

Delling, D., Goldberg, A. V., Pajor, T., and Werneck, R. F. 2014. Robust distance queries on massive networks. In *European Symposium on Algorithms* (Berlin, Heidelberg, 2014), pp. 321–333. (pp. 6, 20, 38, 43)

Delling, D., Goldberg, A. V., Pajor, T., and Werneck, R. F. 2017. Customizable route planning in road networks. *Transportation Science 51*, 2, 566–591. (p. 22)

Demetrescu, C. and Italiano, G. F. 2004. A new approach to dynamic all pairs shortest paths. *J. ACM 51*, 6 (Nov.), 968–992. (p. 21)

Dhulipala, L., Durfee, D., Kulkarni, J., Peng, R., Sawlani, S., and Sun, X. 2020. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *SODA* (2020), pp. 1300–1319. (p. 7)

Dhulipala, L., Liu, Q. C., and Shun, J. 2020. Parallel batch-dynamic k-clique counting. *arXiv preprint arXiv:2003.13585*. (p. 8)

D'angelo, G., D'emidio, M., and Frigioni, D. 2019. Fully dynamic 2-hop cover labeling. *J. Exp. Algorithmics 24*, 1 (Jan.). (pp. xix, 6, 7, 20, 21, 51, 52, 63, 70, 72, 73, 84, 101, 102, 103, 107)

D'Emidio, M. 2020. Faster algorithms for mining shortest-path distances from massive time-evolving graphs. *Algorithms 13*, 8, 191. (p. 21)

Farhan, M., Wang, Q., Lin, Y., and McKay, B. D. 2019. A highly scalable labelling approach for exact distance queries in complex networks. In *22nd International Conference on Extending Database Technology EDBT* (2019), pp. 13–24. (pp. 15, 52, 67, 70)

Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J. 2005. Graph distances in the streaming model: the value of space. In *SODA*, Volume 5 (2005), pp. 745–754. Citeseer. (p. 21)

Freeman, L. C. 1977. A set of measures of centrality based on betweenness. *Sociometry*, 35–41. (p. 1)

FU, A. W.-C., WU, H., CHENG, J., AND WONG, R. C.-W. 2013. Is-label: An independent-set based labeling scheme for point-to-point distance querying. *Proc. VLDB Endow. 6*, 6 (April), 457–468. (pp. 3, 6, 15, 16, 17, 25, 26, 43, 44)

GEISBERGER, R., SANDERS, P., SCHULTES, D., AND DELLING, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms* (2008), pp. 319–333. Springer Berlin Heidelberg. (p. 18)

GEISBERGER, R., SANDERS, P., SCHULTES, D., AND VETTER, C. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science 46*, 3 (aug), 388–404. (pp. 2, 22)

GOLDBERG, A. V. AND HARRELSON, C. 2005. Computing the shortest path: A search meets graph theory. In *SODA* (2005), pp. 156–165. (p. 19)

GUBICHEV, A., BEDATHUR, S., SEUFERT, S., AND WEIKUM, G. 2010. Fast and accurate estimation of shortest paths in large graphs. In *CIKM* (2010), pp. 499–508. (pp. 6, 16, 17, 18)

GUTENBERG, M. P. AND WULFF-NILSEN, C. 2020. Fully-dynamic all-pairs shortest paths: Improved worst-case time and space bounds. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA (2020), pp. 2562–2574. (p. 21)

HAYASHI, T., AKIBA, T., AND KAWARABAYASHI, K.-I. 2016. Fully dynamic shortest-path distance query acceleration on massive networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (2016), pp. 1533–1542. (pp. xv, xix, 6, 7, 19, 20, 23, 25, 26, 30, 36, 37, 38, 41, 43, 44, 51, 52, 69, 70, 101, 102, 103, 105)

JIANG, M., FU, A. W.-C., WONG, R. C.-W., AND XU, Y. 2014. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *Proc. VLDB Endow. 7*, 12 (Aug.), 1203–1214. (pp. 20, 25, 26, 43)

JIN, R., RUAN, N., XIANG, Y., AND LEE, V. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), pp. 445–456. (pp. 3, 6, 15, 17, 43, 116)

KLEINBERG, J. M. 2000. Navigation in a small world. *Nature 406*, 6798, 845–845. (p. 2)

KONTOGIANNIS, S., MICHALOPOULOS, G., PAPASTAVROU, G., PARASKEVOPOULOS, A., WAGNER, D., AND ZAROLIAGIS, C. 2015. Analysis and experimental evaluation of time-dependent distance oracles. In *Proceedings of the Meeting on Algorithm Engineering and Experiments*, ALENEX '15 (USA, 2015), pp. 147–158. Society for Industrial and Applied Mathematics. (p. 22)

KONTOGIANNIS, S., MICHALOPOULOS, G., PAPASTAVROU, G., PARASKEVOPOULOS, A., WAGNER, D., AND ZAROLIAGIS, C. 2016. Engineering oracles for time-dependent road networks. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)* (2016), pp. 1–14. SIAM. (p. 22)

KONTOGIANNIS, S., WAGNER, D., AND ZAROLIAGIS, C. 2015. Hierarchical oracles for time-dependent networks. *CoRR, abs/1502.05222*. (p. 22)

KUMAR, R., NOVAK, J., AND TOMKINS, A. 2006. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD (2006), pp. 611–617. (p. 3)

KUNEGIS, J. 2013. Konect: The koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW (2013), pp. 1343–1350. (p. 102)

LESKOVEC, J., KLEINBERG, J., AND FALOUTSOS, C. 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. KDD (2005). (p. 41)

LESKOVEC, J., LANG, K. J., DASGUPTA, A., AND MAHONEY, M. W. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6*, 1, 29–123. (p. 42)

LESKOVEC, J. AND SOSIČ, R. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol. 8*, 1 (July). (pp. 4, 41)

LI, W., QIAO, M., QIN, L., ZHANG, Y., CHANG, L., AND LIN, X. 2019. Scaling distance labeling on small-world networks. In *Proceedings of the 2019 International Conference on Management of Data* (2019), pp. 1060–1077. (pp. 4, 6, 16, 17, 21, 41, 70, 74, 103, 105)

LI, Y., U, L. H., YIU, M. L., AND KOU, N. M. 2017. An experimental study on hub labeling based shortest path algorithms. *Proc. VLDB Endow. 11*, 4 (Dec.), 445–457. (pp. 4, 6, 38)

LIBEN-NOWELL, D. AND KLEINBERG, J. 2003. The link prediction problem for social networks. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03 (New York, NY, USA, 2003), pp. 556–559. Association for Computing Machinery. (p. 1)

MADKOUR, A., AREF, W. G., REHMAN, F. U., RAHMAN, M. A., AND BASALAMAH, S. 2017. A survey of shortest-path algorithms. *arXiv preprint arXiv:1705.02044*. (p. 1)

MCGREGOR, A. 2014. Graph stream algorithms: A survey. *SIGMOD Rec. 43*, 1 (May), 9–20. (p. 21)

MISLOVE, A., MARCON, M., GUMMADI, K. P., DRUSCHEL, P., AND BHATTACHARJEE, B. 2007. Measurement and analysis of online social networks. In *IMC* (2007), pp. 29–42. (pp. 41, 42)

MYERS, S. A. AND LESKOVEC, J. 2014. The bursty dynamics of the twitter information network. In *Proceedings of the 23rd International Conference on World Wide Web* (2014), pp. 913–924. (p. 3)

NEWMAN, M. E. J., STROGATZ, S. H., AND WATTS, D. J. 2001. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E 64*, 026118. (p. 16)

Ouyang, D., Qin, L., Chang, L., Lin, X., Zhang, Y., and Zhu, Q. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18 (New York, NY, USA, 2018), pp. 709–724. Association for Computing Machinery. (p. 19)

Ouyang, D., Yuan, L., Qin, L., Chang, L., Zhang, Y., and Lin, X. 2020. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *Proc. VLDB Endow. 13*, 5 (Jan.), 602–615. (pp. 20, 22)

Pacaci, A., Bonifati, A., and Özsu, M. T. 2020. Regular path query evaluation on streaming graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20 (New York, NY, USA, 2020), pp. 1415–1430. Association for Computing Machinery. (p. 21)

Pohl, I. 1969. Bi-directional and heuristic search in path problems (1969). (pp. 15, 25)

Potamias, M., Bonchi, F., Castillo, C., and Gionis, A. 2009. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management* (2009), pp. 867–876. (pp. 1, 6, 16, 17, 18)

Qiao, M., Cheng, H., Chang, L., and Yu, J. X. 2014. Approximate shortest distance computing: A query-dependent local landmark scheme. *IEEE TKDE 26*, 1, 55–68. (pp. 3, 6, 17, 18)

Qin, Y., Sheng, Q. Z., Falkner, N. J., Yao, L., and Parkinson, S. 2017. Efficient computation of distance labeling for decremental updates in large dynamic graphs. *World Wide Web 20*, 5 (Sept.), 915–937. (pp. 7, 20, 21, 32)

Robertson, N. and Seymour, P. D. 1984. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B 36*, 1, 49–64. (p. 16)

Roditty, L. and Zwick, U. 2004. On dynamic shortest paths problems. In S. Albers and T. Radzik Eds., *European Symposium on Algorithms* (2004), pp. 580–591. Springer. (p. 21)

Rossi, R. A. and Ahmed, N. K. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI (2015), pp. 4292–4293. (pp. 41, 42)

Sabidussi, G. 1966. The centrality index of a graph. *Psychometrika 31*, 4, 581–603. (p. 1)

Sanders, P. and Schultes, D. 2005. Highway hierarchies hasten exact shortest path queries. In *ESA* (2005), pp. 568–579. (p. 18)

Tarjan, R. E. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 3600 University City Science Center Philadelphia, PA, United States. (pp. 1, 15, 25)

Tretyakov, K., Armas-Cervantes, A., García-Bañuelos, L., Vilo, J., and Dumas, M. 2011. Fast fully dynamic landmark-based estimation of shortest path distances

in very large graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM (2011), pp. 1785–1794. (pp. 3, 6, 16, 17, 18, 21)

Ukkonen, A., Castillo, C., Donato, D., and Gionis, A. 2008. Searching the wikipedia with contextual information. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM (2008), pp. 1351–1352. (p. 1)

Vieira, M. V., Fonseca, B. M., Damazio, R., Golgher, P. B., Reis, D. d. C., and Ribeiro-Neto, B. 2007. Efficient search ranking in social networks. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM (2007), pp. 563–572. (pp. 1, 3, 18)

Watts, D. J. and Strogatz, S. H. 1998. Collective dynamics of 'small-world'networks. *nature 393*, 6684, 440–442. (p. 2)

Wei, F. 2010. Tedi: Efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), pp. 99–110. (pp. 6, 15, 16, 43)

Xu, B., Huang, Y., Kwak, H., and Contractor, N. 2013. Structures of broken ties: Exploring unfollow behavior on twitter. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW (2013), pp. 871–876. (p. 3)

Yahia, S. A., Benedikt, M., Lakshmanan, L. V. S., and Stoyanovich, J. 2008. Efficient network aware search in collaborative tagging sites. *Proc. VLDB Endow. 1*, 1 (Aug.), 710–721. (pp. 1, 3)

Yang, J. and Leskovec, J. 2015. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst. 42*, 1 (Jan.), 181–213. (p. 42)

Zhang, M., Li, L., Hua, W., Mao, R., Chao, P., and Zhou, X. 2021. Dynamic hub labeling for road networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), pp. 336–347. IEEE. (p. 22)

Zhang, M., Li, L., Hua, W., and Zhou, X. 2021. Efficient 2-hop labeling maintenance in dynamic small-world networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), pp. 133–144. (p. 21)