Exploring Shortest Paths on Large-scale networks

Ye Wang

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

December 2021

Except where otherwise indicated, this thesis is my own original work.

Ye Wang
7 December 2021

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Qing Wang, Yu Lin and Brendan McKay who were always willing and enthusiastic to assist in any way they could over the past years. All our meetings were pivotal in inspiring me to think comprehensively and objectively from multiple perspectives. Specially, I would like to show my deep appreciation to Henning Koehler who continuously provides me valuable suggestions.

Furthermore, I wish to thank members in our research group for their energy, understanding and accompanies during my time in Canberra. I also wish to acknowledge various kinds of help offered by several staffs in the research school of computing, the graduate house and the university house.

Finally, I cannot forget to thank my mother for her unconditional support, her patience and encouragement, without which I would not be confident to preserve with the meaningful thing I like.

Abstract

Networks, which are gaining much popularity in various disciplines, model connections between different real-world entities. Based on graph theory, we study a series of problems for understanding networks. Shortest path is among the most fundamental notions studied in graph theory. By exploring shortest path structure, we can better understand the connections between entities. However, it is often computationally expensive to compute shortest paths on large-scale networks which are common nowadays.

This thesis addresses two problems on large-scale networks in relating to shortest paths: (1) How are two vertices connected in a graph? We first define the notion of shortest path graph which contains exactly all shortest paths between two vertices. This notion empowers us to make full use of shortest path structures for analysing the connection between vertices. We formalize the shortest-path-graph problem and propose a novel method, called Query-by-Sketch, which can efficiently leverage offline labelling to guide online searching through a fast-sketching process for solving the shortest-path-graph problem on complex networks. (2) How is one source vertex connected to other vertices in a graph? Enumerating all shortest paths between a source vertex and all other vertices is less informative due the fact that the number of shortest paths is excessive. In addition to this, coverage centrality fails to measure how influential a vertex is in the information flow from a source vertex. Thus we extend the notion of coverage centrality to relative coverage, to measure the importance a vertex is w.r.t a source vertex. Then we formalize the top-k relative coverage problem and develop an efficient method to answer top-k relative coverage queries in a reduced space, and design a bit-parallel optimization to speed up the computation of relative coverage.

The thesis also theoretically discusses the complexity of proposed methods, and presents the experimental results of proposed methods using 12 and 6 real-world datasets for two problems respectively, which illustrates the efficiency and scalability of proposed methods.

Contents

Chapter 1

Introduction

Shortest path is one of the most fundamental notions used in graph theory. Exploring shortest paths between vertices help better understanding connections between entities in many ways in real-world networks. Specifically, a path between two vertices s and t is a shortest path if it has the minimum length among all paths between s and t. It is considered to capture the "closest" connections between vertices. For example, in a road network where endpoints or intersections of roads are represented as vertices and roads are represented as edges, the shortest path between two vertices which correspond with given locations finds applications in navigation systems [Goldberg and Harrelson, 2005]. Also due to the optimality shortest path has, it serves as a basis for tackling a series of interesting problems. For example, a company plans to do advertising on a social web with millions of individuals. Identifying influential users on shortest paths can help enhancing the spread of content since information spreads faster through shortest paths [Yu et al., 2015]. Moreover, consider a protein-protein interaction network in which vertices represent proteins and edges represent interactions between proteins. Shortest path gives important structural information about essential proteins which have more interactions with other proteins, and thus support the selection of possible targets for drug discovery [Estrada, 2006].

1.1 Problems

This thesis aims to answer two research questions based on the shortest path notion:

(1) How do two vertices connect in a graph? To answer this question, we propose a novel problem, the *shortest path graph problem*, to explore the shortest path structure between two vertices.

(2) How does one vertex connect with other vertices in a graph? We formalize the *top-k relative coverage problem* to identify the top-k vertices that best "cover" the shortest path from one given vertex to other vertices.

This thesis proposes exact and efficient algorithms for each problem. Notice that, in recent years, scalability for graph algorithms has become a necessity due to the emergency of large-scale networks with millions of vertices and edges. With this in mind, the methods proposed in this thesis are carefully designed to be scalable on

large-scale networks. Proposed methods for solving these two problems are illustrated by simple examples in this thesis. This thesis further presents experiments on real-world networks to evaluate the performance of all proposed methods and the discussion on the experimental results.

1.1.1 Shortest Path Graph

Computing shortest paths between vertices is a fundamental operation in processing graph data, and has been used in many algorithms for graph analytics [Yao et al., 2013; Opsahl et al., 2010; Kolaczyk et al., 2009]. These algorithms are often applied to support applications that require low latency on graphs with millions or billions of vertices and edges. Therefore, it is highly desirable – but challenging – to compute shortest paths efficiently on very large graphs.

Previously, the problem of point-to-point shortest path queries has been well studied, which is to find a shortest path between two vertices in a graph [Goldberg and Harrelson, 2005; Goldberg et al., 2006; Bast et al., 2007; Goldberg, 2007; Wagner and Willhalm, 2007; Abraham et al., 2010; Wu et al., 2012; Sankaranarayanan et al., 2009; Sanders and Schultes, 2005]. By leveraging specific properties of road networks, such as hierarchical structures and near planarity [Fu et al., 2013; Akiba et al., 2013], previous work has proposed various exact and approximate methods for answering point-to-point shortest path queries [Cowen and Wagner, 2004; Abraham et al., 2012]. Nonetheless, these methods often do not perform well on complex networks (e.g., social networks, web graphs, and computer networks), because complex networks exhibit different properties from road networks, such as small diameter and local clustering [Goldberg and Harrelson, 2005; Fu et al., 2013; Akiba et al., 2013]. Furthermore, existing methods for point-to-point shortest path queries were designed with the guarantee of finding only one shortest path, which limits their usability in practical applications.

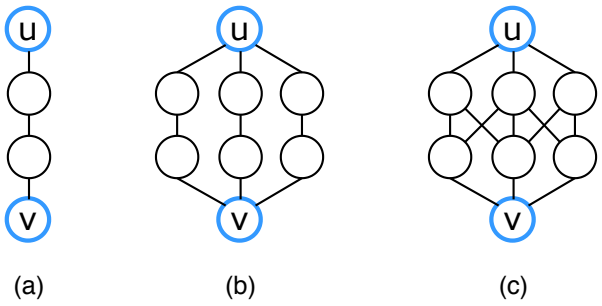


Figure 1.1: An illustration of shortest paths between two vertices u and v whose distance is 3: (a) one shortest path; (b) three shortest paths; (c) seven shortest paths.

Given two vertices u and v, as depicted in Figure 1.1(a)-(c), they have the same distance and cannot be distinguished from one another if only one shortest path is considered. However, when considering all shortest paths, the shortest paths between these two vertices indeed exhibit considerably different structures in Figure 1.1(a)-(c), which can not only distinguish vertices u and v in different scenarios, but

also empower us to make full use of such structures to analyse how they are connected. However, enumerating all the shortest paths between u and v is not only combinatorially challenging, but also leads to highly overlapped paths. Thus, in this paper, we study the problem of finding the structure of shortest paths between vertices. Specifically, we use the notion of "shortest path graph" to represent the structure of shortest paths between two vertices, which is a subgraph containing *exactly all shortest paths* between these two vertices.

Interestingly, shortest path graph manifests itself as a basis for tackling various shortest path related problems, particularly when investigating the structure of the solution space of a combinatorial problem based on shortest paths, for example, the Shortest Path Rerouting problem (i.e., to find a rerouting sequence from one shortest path to another shortest path that only differs in one vertex) [Kamiński et al., 2011; Bonsma, 2013; Nishimura, 2018], the Shortest Path Network Interdiction problem (i.e., to find critical edges and vertices whose removal can destroy all shortest paths between two vertices) [Khachiyan et al., 2008; Israeli and Wood, 2002], and the variants such as the Shortest Path Common Links problem (i.e., to find links common to all shortest paths between two vertices) [Labbé et al., 1995; Hansen et al., 1986]. These shortest path related problems are motivated by a wide range of real-world applications arising in designing and analysing networks. For example, identifying a rerouting sequence for shortest paths enables the robust design of networks with minimal cost for reconfiguration, and finding critical edges and vertices help defend critical infrastructures against cyberattacks.

However, computing shortest path graphs is computationally expensive since it requires to identify all shortest paths, not just one, between two vertices. A straightforward solution for answering shortest-path-graph queries is to compute on-the-fly all shortest paths between two vertices using Dijkstra algorithm for weighted graphs [Dijkstra et al., 1959] or performing a breadth-first search (BFS) for unweighted graphs [Cormen et al., 2009]. This is costly on graphs with millions or billions of vertices and edges. Another solution is to precompute all shortest paths for all pairs of vertices in a graph and then assign precomputed labels to vertices such that certain properties hold, e.g. 2-hop distance cover [Cohen et al., 2003]. However, for large graphs, storing even just shortest path distances of all pairs of vertices is prohibitive [Akiba et al., 2013] and storing all shortest paths of all pairs is hardly feasible due to the demand for much more space overhead.

In conclusion, the first problem we study in this thesis is the shortest path graph problem, which aims at finding the shortest path graph between two vertices on large-scale complex networks. More specifically, we need to answer the following question: *How to construct labels for shortest-path-graph queries which should be of reasonable size (e.g. not much larger than an original graph), within a reasonable time (e.g. not longer than one day), and can speed up query answering as much as possible?*

1.1.2 Top-k Relative Coverage

Centrality is a fundamental concept in analyzing real-world networks. It measures how important a vertex is in a network by taking into account its position in the topological structure of the network. Until now, a variety of graph-theoretic centrality measures have been proposed, and some of them are based on exploiting shortest path structure between vertices in a network, such as betweenness centrality [Freeman, 1977], closeness centrality [Bavelas, 1950], and coverage centrality [Yoshida, 2014].

Conceptually, betweenness centrality measures the extent to which a vertex lies on shortest paths between other vertices. Nonetheless, it is limited to revealing only certain aspects of vertices in a network and cannot discover different roles of a vertex under different perspectives. Consider the graph depicted in Figure 1.2(a) for example. According to betweenness centrality, vertex 7 is more important than vertex 3. However, vertex 3 appears on shortest paths between more distinct vertex pairs than vertex 7, which can be measured by coverage centrality. Moreover, instead of considering shortest paths between all vertex pairs, one may be more interested in discovering which vertices have influence on the information flow from a specific source (i.e., vertex) in a large-scale network. Figure 1.2(b) shows that vertex 2 appears to have more influence on vertex 5 than vertex 0, although vertex 0 is measured to be more important than vertex 5 according to coverage centrality.

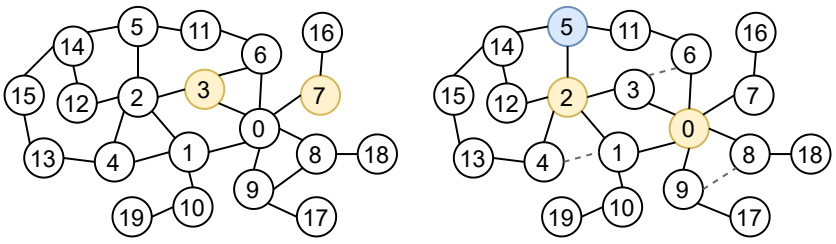


Figure 1.2: An illustrative example: (a) betweenness centrality and coverage centrality of vertices 3 and 7; (b) relative coverage centrality of vertices 2 and 0 w.r.t. vertex 5.

Thus, the problem we study is the top-k relative coverage problem, which is to find top-k vertices that best cover the shortest paths between a source vertex and other vertices in a network. Vertices with high coverage centrality are pivotal in the connections between vertices, and offer themselves as ideal candidates for landmarks in shortest path distance labelling methods [Akiba et al., 2013; Farhan et al., 2019]. In many real-world applications, ranking top-k vertices with highest coverage centralities are also highly sought after, rather than obtaining actual values of coverage centralities for all vertices. Below, we discuss two of these applications.

- *Road network upgrade* [Medya et al., 2018a]. Consider the transportation authority of a city, where a new public facility such as a stadium is opened. The top-k

relative coverage problem can help select locations which are most important in fast connecting the new stadium with other locations. Then the authority can upgrade the road network to reduce the travel time between the new facility and other locations to ease access to the stadium from other locations.

- *Content spread enhancement* [Yu et al., 2015]. Consider a social network site on which users may post content items (e.g. videos and photos) and expect others to receive content items being posted. The top-k relative coverage problem can help identify a small number of individuals that are most influential in connecting a user with others. Then strengthening the connections between users and their influential individuals can boost content spread.

A straightforward solution to the top-k relative coverage problem is to compute the relative coverage of each vertex in the graph and then select top-k vertices with maximum relative coverage. However, this is computationally expensive because it computes all shortest paths between a source vertex and all other vertices.

To sum up, the second problem we study in this paper is the top-k relative coverage problem, which detects vertices that best "cover" the shortest paths from one source vertex to other vertices. The main challenge is to develop an efficient and scalable method to solve this problem on large-scale networks.

1.2 Contributions

In this thesis, we study the problems introduced above, aiming to design scalable methods on unweighted graphs with millions of vertices and edges.

Shortest Path Graph. The first problem we study in this thesis is the shortest path graph problem which aims at explaining how two vertices are connected. Though computing shortest paths is a fundamental operation in processing graph data, computing shortest path graphs is computationally expensive since it requires to identify all shortest paths between two vertices. Our goal is to design an exact solution that can scale to undirected graphs with millions or billions of vertices and edges. To achieve high scalability, we propose a novel method, *Query-by-Sketch* (QbS), which efficiently leverages offline labelling (i.e., precomputed labels) to guide online searching through a fast sketching process that summarizes the important structural aspects of shortest paths in answering shortest-path-graph queries. We theoretically prove the correctness of this method and analyze its complexity. To empirically verify the efficiency of QbS, we conduct experiments on 12 real-world datasets, among which the large dataset has 1.7 billion vertices and 7.8 billion edges. The experimental results show that QbS can answer shortest-path-graph queries in microseconds for million-scale graphs and less than half a second for billion-scale graphs.

Top-k Relative Coverage. The second problem we study in this thesis is the top-k relative coverage problem which aims at explaining how one vertex connects with other vertices. Among a plethora of variations of centrality measures, coverage centrality measures the importance of a vertex by counting the number of distinct vertex pairs

whose shortest paths go through the vertex. However, coverage centrality fails to measure how influential a vertex is in the information flow from a specific source (i.e., a vertex). Therefore, we introduce relative coverage based on coverage centrality and study the top-k relative coverage problem. We propose an efficient method to answer top-k relative coverage queries, which only requires to compute relative coverage in a reduced search space. Then we design a bit-parallel optimization method to accelerate the computation of relative coverage. We theoretically analyse the complexity of our methods and experimentally verify their efficiency through experiments on 6 real-world networks.

Notice that, though methods in this thesis are developed for unweighted graphs, they can be simply extended to deal with directed graphs.

1.3 Thesis Outline

The thesis is organized as follows: We review previous works which are related to the problems studied in this thesis in Chapter 2. We study the shortest path graph problem in Chapter 3 and the top-k relative coverage problem in 4. We conclude this thesis in Chapter 5.

In both Chapter 3 and Chapter 4, we begin with an overview of the specific problem we study. We then propose and illustrate our methods for solving the problem. We conclude each chapter with a brief summary.

Chapter 2

Literature Review

Shortest-path computation is a fundamental operation in graph theory. A path between s and t is a shortest path if it is of minimum length among all paths between s and t, where the length of a path p is defined as the sum of the weights of all edges of p. The shortest path distance between two nodes s and t is defined as the minimum length of a path between s and t, i.e., the length of shortest path between s and t. In this chapter, we provide a literature review on shortest-path computation (i.e., several important problems which find shortest paths), and applying shortest-path-based coverage centrality to analyze graphs.

2.1 Shortest Path Graph

In this section, we briefly review the related work which study shortest-path computation, including single-source shortest-path, all-pair shortest-path and point-to-point shortest-path.

2.1.1 Single-source Shortest-path Problem and All-pairs Shortest-path Problem

The single-source shortest-path (SSSP) problem and the all-pairs shortest-path (ASSP) problem are two basic versions of the shortest-path computation. The goal of SSSP problem is to find all shortest paths or distances between a given source vertex s and all other vertices in a graph. It can be solved efficiently using Dijkstra which is implemented using Fibonacci heaps on weighted graphs in $O(|E| + |V|log|V|)$ time, or using breadth-first search on unweighted graphs in $O(|V| + |E|)$ time [Dijkstra et al., 1959; Fredman and Tarjan, 1987]. In both methods, all vertices are visited in ascending order of shortest path distance from s. Since 1959, theoretical developments for SSSP problem on general directed and undirected graphs have been based on Dijkstra and breadth-first search [Thorup, 1999]. The goal of ASSP problem is to compute the shortest paths or distances between all pairs of vertices in a graph. We can apply Dijkstra to compute all shortest paths by selecting each vertex as the source, or apply Bellman-Ford, Floyd-Warshall, etc [Cormen et al., 2009]. Specifically, while Dijkstra can only deal with graphs with negative weights, Bellman-Ford and

Floyd-Warshall can deal with graphs without negative cycles in $O(|V|^2|E|)$ time and $O(|V|^3)$ time, respectively.

2.1.2 Point-to-point Shortest-path Problem

As a generalization of SSSP problem and APSP problem, the point-to-point shortest-path problem finds one shortest path or the distance between two specific vertices. Both Dijkstra's algorithm and BFS are very inefficient in computing point-to-point shortest paths on large networks, since the search space of these two methods contains a large number of unnecessary vertices which are not on the shortest paths. A simple strategy for reducing search space is to employ bi-directional search which performs two searches from two given vertices, respectively, based on certain heuristic assumptions [Goldberg and Harrelson, 2005; Jin et al., 2013]. To further accelerate shortest path computation, a number of methods have been proposed to precompute and store precomputation information so as to answer point-to-point shortest path queries online in a shorter time. This leads to a trade-off between the time needed for precomputation, the space needed for storing the precomputed information and the online query time.

We categorize algorithms for solving point-to-point shortest path problem on real-world networks into two types: exact algorithms and approximate algorithms.

2.1.2.1 Exact Methods

Exact methods have been proposed to solve point-to-point shortest path problem on two types of real-world networks, road networks and complex networks, by exploiting the properties each type of networks has.

Road networks. Due to the fact that a large number of methods have been proposed for finding a point-to-point shortest path on road networks [Goldberg, 2007; Wagner and Willhalm, 2007; Wu et al., 2012; Sommer, 2014; Li et al., 2017], we can only highlight influential methods that help to put our methods into perspective. Notice that, these methods mainly suppose that road networks are weighted and directed. A* search works like Dijkstra's algorithm, except that it selects a vertex v to traverse next according to an estimated distance, defined as the sum of exact distance between the source vertex and v and estimated distance between v and the goal vertex [Hart et al., 1968]. Lauther proposed to avoid unnecessary traversal according to geographic background (i.e., coordinates of vertices) [Lauther, 2004]. They divided the network into regions and used *arc-flag* for edges to indicate the region each edge route to. To guide the search on networks without additional information, Gutman et al. defined the notion of *reach* which encodes the length of shortest paths a vertex lies in [Gutman, 2004]. Vertices are excluded from the search if they can not be on the paths long enough for the current query, based on upper bounds on vertex reaches and lower bounds on distance between the source vertex and the goal vertex. Almost at the same time, aiming at strengthening the A* search, Goldberg et al. proposed ALT (A*, landmark and triangle inequality) algorithm which uses

the landmark-based distance lower bound in combination with the triangle inequality [Goldberg and Harrelson, 2005]. Though better estimation on distances leads to fewer vertices being traversed, precomputed distances between landmarks and all other vertices dominate the space required by ALT. To save the space needed for storing precomputed information, Sanders et al. took the advantage of inherent road hierarchy to restrict the search to a smaller subgraph [Sanders and Schultes, 2005]. They introduced the notion of *shortcut* for representing shortest paths between a vertex pair and constructed *highway hierarchies* which consists of levels contracted from the origin graph, based on the fact that vertices in road networks have constant low degrees. Later, Goldberg et al. further proposed *REAL* which improved *reach* based search by adding shortcuts and combining it with ALT [Goldberg et al., 2006]. However, it makes the pre-processing more complex (i.e., contains two pre-processing algorithms, one for reach and one for ALT). Bast et al. proposed *transit node routing* which further adapts the highway hierarchies [Bast et al., 2007]. They proposed to guide the search using transit nodes, which is a set of vertices on sufficiently long shortest paths and can be detected in top levels of highway hierarchies. Geisberger et al. obtained contraction hierarchies [Geisberger et al., 2008]. Shortcuts between vertices are sequentially computed in a total order of vertices and stored as labels. However, the resulting path may contain several shortcuts, thus additional processing is needed to transform shortcuts to real paths. Bauer et al. proposed *SHARC* by combining ideas from *highway hierarchies*, *arc-flag*, and *REAL* [Bauer and Delling, 2010]. Abraham et al. proposed a variant of *transit node routing* based on modeling road networks as graphs with low highway dimensions [Abraham et al., 2010]. They showed how low highway dimensions guarantee the efficiency for previous algorithms (e.g., [Gutman, 2004; Sanders and Schultes, 2005; Bast et al., 2007; Geisberger et al., 2008]). It can be seen that, methods above targeted at finding a point-to-point shortest path between two vertices on a road network with specific properties, such as weighted edges, low vertex degree, low highway dimensions, hierarchical structures, and near planarity. For networks without such properties, these methods fail to well perform [Abraham et al., 2010].

Complex networks. Unfortunately, complex networks are known to have different properties from road networks, such as small-world properties and high highway dimensions [Abraham et al., 2010], which has thus led to a line of research on point-to-point shortest path (distance) queries for complex networks [Xiao et al., 2009; Wei, 2010]. Xiao et al. [Xiao et al., 2009] exploited graph symmetry to label shortest paths. Though the size of labels has been compressed depending on the symmetric property, the space cost is still high. Later, Wei [Wei, 2010] introduced a method based on tree decomposition for point-to-point shortest path queries. However, most of complex networks have a large component in which vertices are densely connected, making it hard to be decomposed into tree-like structures. Several methods have been proposed for finding shortest path distances on complex networks (e.g., [Akiba et al., 2013; Fu et al., 2013; Akiba et al., 2012; Hayashi et al., 2016; Farhan et al., 2019]). Some of them considered answering point-to-point shortest path queries as an exten-

sion of answering distance queries, although they did not provide any experiments. For example, Akiba et al. [Akiba et al., 2013] proposed pruned landmark labelling (PLL) which constructs a 2-hop labelling for distance queries by conducting pruned BFSs. Fu et al. [Fu et al., 2013] proposed IS-label, a labelling for distance queries on weighted graphs based on an independent set of vertices. Both of these methods discussed labellings for point-to-point shortest path queries by extending labellings for distance queries with parent information, which however require a high space overhead and do not scale to large graphs.

In this thesis, we study the shortest-path-graph problem, which is computationally more difficult than the point-to-point shortest path problem, and little attention has previously been given. Our method precomputes a small-sized distance labelling and can handle complex networks with up to billions of vertices.

2.1.2.2 Approximate Algorithms

Due to the high computational costs of computing point-to-point shortest paths or distances, a number of approximate methods for point-to-point shortest path queries have been proposed in the past, including landmark-based methods [Gubichev et al., 2010; Zhao et al., 2011; Tretyakov et al., 2011]. Specifically, Gubichev et al. [Gubichev et al., 2010] proposed to pre-compute shortest paths from each vertex to each landmark, and then concatenate shortest paths from two given vertices to the same landmarks to compute approximate shortest paths. They also proposed cycle elimination and tree-based sketch to boost the accuracy. Zhao et al. [Zhao et al., 2011] proposed a method, called Rigel, to estimate shortest path distances between vertices. They also extended Rigel for approximating shortest paths. Tretyakov et al. [Tretyakov et al., 2011] used shortest path trees rooted at landmarks to approximate shortest path distances and search for one shortest path.

2.2 Coverage Centrality

In this section, we briefly discuss the related work on shortest-path based centralities and the computation on coverage centrality.

Centrality measures have been extensively studied in the last few decades. As shortest paths are useful to indicate the fast communication between vertices, several centrality measures based on shortest paths have been proposed, such as betweenness centrality, closeness centrality and coverage centrality [Bavelas, 1950; Freeman, 1977; Yoshida, 2014]. Each of them reveals certain aspects of vertices in a network, and can discover different roles of a vertex under different perspectives.

The notion of (shortest-path) coverage centrality was first introduced by Yoshida [Yoshida, 2014] in the context of finding a set of k vertices with the maximum group centrality. However, the exact methods for computing such a set w.r.t. group coverage centrality take $O(k|V|^2|E|)$ time, where $|V|$ is the number of vertices and $|E|$ is the number of edges in a graph. Thus, Yoshida proposed an approximation method

for finding such vertices in almost linear time [Yoshida, 2014]. Later, Takaguchi *et al.* [Takaguchi et al., 2016] extended the notion of coverage centrality to temporal networks as they considered such a notion as an important indicator for understanding the structure of temporal networks. To compute the centrality of temporal vertices, they proposed both exact and approximate methods, which require $O(|V|^2 log|E|)$ and $O(log^2|V|)$ queries to a reachability oracle, respectively. Based on defining a restricted sample space, Chehreghani *et al.* proposed an approximate algorithm which can estimate coverage centrality of one vertex in a directed graph in $O(|E| + |V|log|V|)$ time [Chehreghani et al., 2018]. Another line of research has focused on controlling the coverage centrality of vertices via modifications to a graph [Medya et al., 2018b; D'Angelo et al., 2019; Medya et al., 2017]. For example, Medya et al. [Medya et al., 2018b] studied the coverage centrality maximization problem aiming to improve the group centrality of a target vertex set via adding a small number of edges into a graph, and D'Angelo et al. [D'Angelo et al., 2019] examined the coverage centrality maximization problem specifically on undirected networks to address the challenge relating to the lack of submodularity in undirected networks. As these problems are NP-hard and APX-hard, only approximation methods were proposed [Medya et al., 2018b; D'Angelo et al., 2019].

Chapter 3

Shortest Path Graph

3.1 Overview

To understand how two vertices are connected in a graph, we study the shortest path graph problem on large-scale complex networks in this chapter.

Computing a shortest path graph is computationally expensive since all shortest paths between two vertices should be found. For search-based methods such as Dijkstra [Dijkstra et al., 1959] and breadth-first search which compute all the shortest paths on the fly without precomputation, their query time complexities are high, thus these methods are not feasible when we require low latency on large-scale networks. In order to speed up the online computation of shortest paths, labelling-based methods have been proposed. These methods first precompute labels such that certain properties hold, and then answer a point-to-point shortest-path (distance) query by examining the labelling information without searching the graph. The state-of-the-art work is pruned landmark labelling, in which a 2-hop distance cover is constructed to encode shortest distance information [Akiba et al., 2013]. But the 2-hop distance cover is inadequate to characterize labels required by shortest-path-graph queries. We extend pruned landmark labelling and propose pruned path labelling to support the computation of shortest path graph. We also propose a method, called parent pruned path labelling, which keep additional parent information in labels to further accelerate query time. But storing labels for computing every shortest path is hardly feasible due to the demand for much more space, and as a result, the labelling sizes of these two methods are hundreds of times larger than the origin graph. This shows the need for a trade-off between query time and labelling size, and motivates us to develop a hybrid method.

To achieve high scalability, we propose a novel method, *Query-by-Sketch* (QbS), which efficiently leverages offline labelling (i.e., precomputed labels) to guide online searching through a fast sketching process that summarizes the important structural aspects of shortest paths in answering shortest-path-graph queries. This method consists of three phases, as illustrated in Figure 3.1: (a) *labelling* - constructing a labelling scheme, which is compact and of a small size, using a small number of landmarks through precomputation, (b) *sketching* - using labelling to efficiently compute a *sketch* that summarizes the important structural aspects of shortest paths in a query answer,

Table 3.1: Frequent used notations.

Notation	Description
$G = (V, E)$	A graph
$d_G(u, v)$	Distance between u and v in G
R	Set of landmarks
$L(v)$	Label of v
$V(G)$	Set of vertices in G
$E(G)$	Set of edges in G
P_{uv}^G	Set of all shortest paths between u and v in G
G_{uv}	Shortest path graph between u and v

and (c) *searching* - computing shortest paths on a sparsified graph under the "guide" of the sketch. We develop efficient algorithms for these phases, and combine them effectively to handle shortest-path-graph queries on very large graphs. We theoretically prove the correctness of our proposed methods, and analyze the complexity of each algorithm. Then we experimentally evaluate the performance of our proposed methods on 12 real-world networks, among which the largest one has billions of vertices and edges. It is shown that *QbS* has significantly better scalability than the baseline methods.

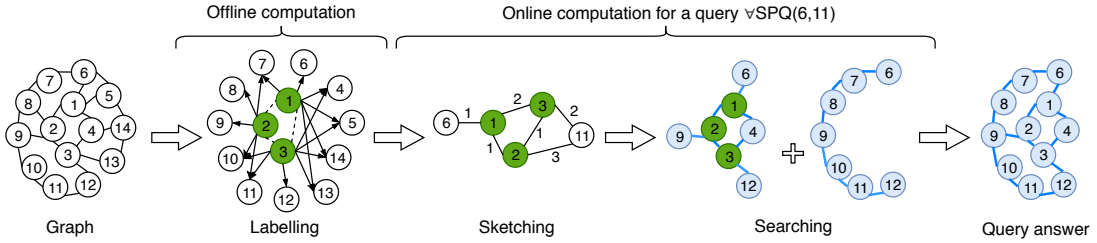


Figure 3.1: An illustration of our method Query-by-Sketch (QbS) for answering shortest-path-graph queries.

Outline. The rest of this chapter is organized as follows. In Section 3.2, we present the preliminaries and problem definition. Section 3.3 discusses several labelling-based methods and their limitations. Our Query-by-Sketch method is introduced in Section 3.4. Section 3.5 presents the proof of correctness and the analysis of complexity. Section 3.6 discusses the experimental results. We conclude this chapter in Section 3.7.

3.2 Preliminaries

Table 3.1 lists the notations that are frequently used in this section. Let $G = (V, E)$ be an unweighted graph, where V and E represent the set of vertices and edges in G, respectively. Without loss of generality, we assume that G is undirected and

connected since our work can be easily extended to directed or disconnected graphs. We use $V(G)$ and $E(G)$ to refer to the set of vertices and edges in G, respectively, P_{uv} the set of all shortest paths between u and v, and $d_G(u,v)$ the shortest path distance between u and v in G.

Distance labelling. Let $R \subseteq V$ be a subset of special vertices in G, called *landmarks*. For each vertex $v \in V$, the *label* of v is a set of *labelling entries* $L(v) = \{(r_1, \delta_{vr_1}), \ldots, (r_n, \delta_{vr_n})\}$, where $r_i \in R$ and $\delta_{vr_i} = d_G(v, r_i)$. We call $L = \{L(v)\}_{v \in V}$ a *labelling* over G. The *size* of a labelling L is defined as *size*(L)=$\Sigma_{v \in V}|L(v)|$. In viewing that each labelling entry (r_i, δ_{vr_i}) corresponds to a *hop* from a vertex v to a landmark r_i with the distance δ_{vr_i}, Cohen *et al.* [Cohen et al., 2003] proposed *2-hop distance cover*, which has been widely used in labelling-based approaches for distance queries.

Definition 3.2.1. [2-HOP DISTANCE COVER] *A labelling L over a graph $G = (V, E)$ is a* 2-hop distance cover *iff, for any two vertices $u, v \in V$, the following holds:*

$$d_G(u,v) = min\{\delta_{ur} + \delta_{vr} | (r, \delta_{ur}) \in L(u), (r, \delta_{vr}) \in L(v)\}.$$

Informally, 2-hop distance cover requires that, for any two vertices in a graph, their labels must contain at least one common landmark r that lies on one of their shortest paths.

Shortest-path-graph problem. In this work, we study shortest-path-graph queries. We first define the notion of *shortest path graph*.

Definition 3.2.2. [SHORTEST PATH GRAPH] *Given any two vertices u and v in a graph G, the shortest path graph (SPG) between u and v is a subgraph G_{uv} of G, where (1) $V(G_{uv}) = \bigcup_{p \in P_{uv}} V(p)$ and (2) $E(G_{uv}) = \bigcup_{p \in P_{uv}} E(p)$.*

A shortest path graph G_{uv} is different from an induced subgraph $G[V']$ where $V' = \bigcup_{p \in P_{uv}} V(p)$. Every edge in G_{uv} must lie on at least one shortest path between u and v, whereas $G[V']$ may contain edges that do not lie on any shortest path between u and v.

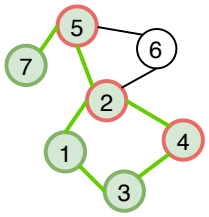
Definition 3.2.3. [SHORTEST-PATH-GRAPH PROBLEM] *Let $G = (V, E)$ and $u, v \in V$. Then the* shortest-path-graph problem *is, given a query $SPG(u, v)$, to find the shortest path graph G_{uv} over G.*

3.3 Shortest Path Labelling

In this section, we discuss several labelling-based methods for the shortest path graph problem. The purpose is to discuss their limitations and possible sources of difficulties.

3.3.1 2-Hop Path Cover

Originally, 2-hop distance cover was proposed for reachability and distance queries [Cohen et al., 2003]. Below, we discuss how it can be used to find shortest paths and



Label	Labelling Entries
L(1)	(1,0) (2,1) (3,1)
L(2)	(1,1) (2,0) (3,2) (4,1) (5,1) (6,1)
L(3)	(1,1) (2,2) (3,0) (4,1)
L(4)	(1,2) (2,1) (3,1) (4,0)
L(5)	(1,2) (2,1) (5,0) (6,1) (7,1)
L(6)	(1,2) (2,1) (5,1) (6,0)
L(7)	(1,3) (2,2) (5,1) (7,0)

(a) (b)

Figure 3.2: (a) A graph G in which the answer of $\forall SPQ(3,7)$ is colored in green; (b) Labels over G, where labels for a 2-hop distance cover are colored in black and additional labels from a 2-hop path cover are colored in green.

why it is insufficient for all-shortest-path queries.

Example 3.3.1. *Consider a query $SPG(3,7)$ on a graph G depicted in Figure 3.2 (a). The query answer is colored in green. In Figure 3.2(b), labels of a 2-hop distance cover over G are colored in black. Starting from vertices 3 and 7, we can find vertex 1 because $(1,1) \in L(3)$ and $(1,3) \in L(7)$, $d_G(3,7) = 1+3 = 4$. Then, we have to stop since the label of vertex 1 does not contain entries to other vertices. Thus, using the labels of the 2-hop distance cover can compute only one shortest path between 3 and 7, failing to find vertices 2, 4 and 5 in the answer.*

Finding a shortest path graph that exactly contains all shortest paths between two vertices requires us to accurately encode *every* shortest path between two vertices into labels. Thus, to answer shortest-path-graph queries, we generalize 2-hop distance cover to a property called *2-hop path cover*.

Definition 3.3.2. [2-HOP PATH COVER] *Let $G = (V,E)$ be a graph and L a labelling over G. We say L is a* 2-hop path cover *iff L is a 2-hop distance cover and, for any two vertices $u,v \in V$ and any path $p \in P^G_{uv}$ with $p \neq (u,v)$, the following holds:*

$$d_G(u,v) = min\{\delta_{ur} + \delta_{vr} | (r,\delta_{ur}) \in L(u), (r,\delta_{vr}) \in L(v), r \in V(p) \backslash \{u,v\}\}, \tag{3.1}$$

Compared with 2-hop distance cover, 2-hop path cover further requires that, for any shortest path p between any two vertices u and v that contains more than one edge, the labels of u and v should contain a common landmark r that lies on p, but not be u or v.

Example 3.3.3. *Consider Figure 3.2 again, in which a 2-hop path cover contains labels colored both in black and in green. According to the labels of vertices 1 and 7, vertex 2 can be found. Then by the labels of 2 and 7, we can further find vertex 5. Similarly, vertex 4 can be found through the labels of 2 and 3. Thus, using the labels of the 2-hop path cover, we can find the query answer for $SPG(3,7)$.*

3.3.2 Path Labelling Methods

To answer shortest-path-graph queries, a naive labelling-based method is, for each vertex $v \in V$, to conduct a breadth-first search (BFS) from v and store the distances between v and all other vertices in the label of v, i.e. $L(v) = \{(u, \delta_{vu}) | u \in V\}$, which is a 2-hop path labelling. Although shortest-path-graph queries can be answered using L, it is inefficient, particularly when a graph is large. The time and space complexity of constructing such labels are $O(|V||E|)$ and $O(|V|^2)$ respectively. Answering one shortest-path-graph query would cost $O(|V|^2)$ in the worst case. A question that naturally arises is: can we follow the idea of Pruned Landmark Labelling (PLL) [Akiba et al., 2013], which has been shown to be successful for distance queries, to develop a pruning strategy for shortest-path-graph queries for improving efficiency? We will thus introduce two pruned path labelling methods for shortest-path-graph queries in the following.

Pruned path labelling. Inspired by Pruned Landmark Labelling (PLL) [Akiba et al., 2013], we conduct pruning during the breadth-first searches, i.e. *pruned* BFSs, for shortest-path-graph queries. We abbreviate this pruned path labelling method by PPL.

PPL works as follows. Given a pre-defined landmark order $[v_1, v_2, \ldots, v_{|V|}]$ over all vertices in G, we conduct a pruned BFS from each vertex one by one as described in Algorithm 1. In each pruned BFS rooted at v_k, we use $depth[v]$ to denote the distance between v_k and v. Further, L_{k-1} refers to the labels that have been constructed through the previous pruned BFSs from vertices $[v_1, \ldots, v_{k-1}]$, and $d_{L_{k-1}}(v_k, u)$ denotes the distance between v_k and u being queried using labels in L_{k-1}. When $d_{L_{k-1}}(v_k, u) < depth[u]$, the label $(v_k, depth[u])$ is pruned (Lines 6-7) because labels in L_{k-1} have already covered the shortest paths between v_k and u. In other words, v_k is only added into the labels of vertices u when $d_{L_{k-1}}(v_k, u) \geq depth[u]$ (Line 8). Note that, unlike PLL, in the case of $d_{L_{k-1}}(v_k, u) = depth[u]$, the label $(v_k, depth[u])$ cannot be pruned in PPL; otherwise, 2-hop path cover is not guaranteed, i.e., not all shortest paths are covered by labels. When $d_{L_{k-1}}(v_k, u) \leq depth[u]$, no further edges are traversed from u because paths in this expansion have already been covered by labels in L_k (Lines 6-7 and 9-10).

To answer a query $SPG(u, v)$, we need to compute vertices and edges of G_{uv} from a pruned path labelling L recursively. Assume that $d_G(u, v) \neq 1$; otherwise we finish with G_{uv} containing only one edge (u, v). We begin with $E(G_{uv}) = \emptyset$. We find the common landmarks in their labels that are on the shortest paths, e.g., computing a set $V_{uv} = \{r | r = min(\delta_{ur} + \delta_{vr}), (r, \delta_{ur}) \in L(u), (r, \delta_{vr}) \in L(v)\}$. Then we query the shortest paths between u, v and these common landmarks, i.e., (u, r) and (v, r) for each $r \in V_{uv}$. The query $\forall SPQ(u, v)$ is computed by combining the shortest paths between u, v and the landmarks, i.e., $E(G_{uv}) = \bigcup_{r \in V_{uv}} (E(G_{ur}) \cup E(G_{vr}))$.

Example 3.3.4. *When using PPL to answer the query* $\forall SPQ(3, 7)$ *on the graph* G *in Figure 3.2(a), we start with* $(3, 7)$ *and obtain* $V_{3,7} = \{1, 2\}$. *This leads to four new queries* $(3, 1), (7, 1), (3, 2)$ *and* $(7, 2)$. *The distance between* 3 *and* 1 *is 1. Thus,* $E(G_{3,7}) = \{(1, 3)\} \cup E(G_{7,1}) \cup E(G_{3,2}) \cup E(G_{7,2})$. *For the new query* $(7, 1)$, *we obtain* $V_{7,1} = \{2\}$,

Algorithm 1: PrunedBFS

Input: $G = (V, E)$; a landmark v_k; a labelling L_{k-1}

```
1  Q ← ∅;
2  Q.push(v_k);
3  depth[v_k] ← 0, depth[v] ← ∞ for all v ∈ V\{v_k};
4  L_k(v) ← L_{k-1}(v) for all v ∈ V;
5  while Q is not empty do
6      dequeue u from Q;
7      if d_{L_{k-1}}(v_k, u) < depth[u] then
8          continue;
9      L_k(u) ← L_k(u) ∪ {(v_k, depth[u])};
10     if d_{L_{k-1}}(v_k, u) = depth[u] then
11         continue;
12     for all (u, v_i) ∈ E s.t. depth[v_i] = ∞ do
13         depth[v_i] ← depth[u] + 1;
14         enqueue v_i to Q;
15 return L_k;
```

leading to another queries $(7,2)$ *and* $(1,2)$*. Similarly, for* $(3,2)$ *and* $(7,2)$ *we obtain queries* $(1,2)$*,* $(2,3)$*,* $(2,5)$ *and* $(2,7)$*. Note that the labels of vertex* 3 *are visited more than once, i.e. when querying* $(3,7)$ *and* $(3,2)$*. Further, because* 3 *and* 7 *have multiple shortest paths between them, more than one common vertex on their shortest paths are found from their labels, i.e.* $\{1,2\}$*. As a result, edges* $(2,5)$ *and* $(5,7)$ *are handled multiple times, i.e., when querying* $(2,7)$ *and* $(1,7)$*.*

PPL has the same time and space complexity for constructing labels as the naive labelling-based method. However, due to pruning in the BFSs, PPL can often construct labels more efficiently with a significantly reduced labelling size. Nonetheless, the query time of PPL is still slow because all shortest paths between two vertices can only be found through searching vertices and edges using labels in a recursive manner. When there exists more than one shortest path between the query vertices, labels of some vertices are searched repeatedly and edges are found repeatedly, leading to unnecessary computational cost, e.g., vertex 3 and edges $\{(2,5)(5,7)\}$ as discussed in Example 3.3.4.

Path labelling with parents. One common technique to accelerate query time for shortest path queries is to keep additional parent information in labels so as to provide a clearer direction towards shortest paths. For example, Akiba *et al.* [Akiba et al., 2013] extended the label of each vertex $v \in V$ to a set of triples (r, δ_{vr}, w_{vr}) where w_{vr} is the "parent" vertex of r on a shortest path from v to r. To find shortest path graphs, this would require us to store all the parent vertices of a vertex, rather than just one parent vertex as in the previous work for finding one shortest path. To be precise, we store a set of triples $\{(r_i, \delta_{vr_i}, W_{vr_i})\}_{1 \leq i \leq |V|}$ where W_{vr_i} is a set of "parent" vertices

of v on a shortest path from v to a landmark r_i. To reduce space overhead, for each of such shortest paths, we choose to store the "parent" vertices of v, rather than the "child" vertices of r_i, because landmarks often have a high degree [Akiba et al., 2013]. To distinguish from PPL, we abbreviate this path labelling method with additional parent information by ParentPPL.

The time complexity of ParentPPL for constructing labels remains to be $O(|V||E|)$ but the space complexity becomes $O(|V||E|)$. In practice, additional parent information only helps speed up query time on small graphs. Even for a graph with millions of vertices and edges, ParentPPL would run out of time (same as PPL) or space, failing to construct labels. We will discuss this further in Section 3.6.

3.3.3 Discussion

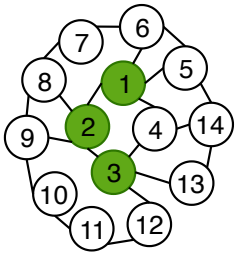
For 2-hop labelling-based methods such as PPL and ParentPPL, the structure (i.e., shortest paths) of a graph is encoded into distance information of labels under the guarantee of 2-hop path cover. Although shortest paths can be recovered through computing distances between pairs of vertices, these methods are inefficient. This is because they recursively split each path into two sub-paths and compute vertices on sub-paths via distance information in labels, which leads to redundant or unnecessary searches. Although storing parent information can often accelerate query time, it makes labelling size larger and does not scale over large networks. Therefore, we need to find a method for which (1) the labelling size is small, (2) the structure of shortest paths can be recovered in an efficient way, i.e., reducing redundant and unnecessary computation, and (3) it can scale over large networks.

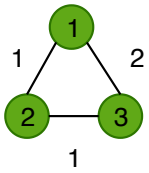
3.4 Query-by-Sketch

In this section, we present an efficient and scalable method for solving the shortest-path-graph problem, called *Query-by-Sketch* (QbS). Conceptually, this method consists of three key components: *labelling*, *sketching* and *searching*, which will be discussed in Sections 3.4.1, 3.4.2 and 3.4.3, respectively. The main idea behind this method is to construct a labelling scheme through pre-computation, and then answer shortest-path-graph queries by performing online computation that involves two steps: fast sketching and guided searching.

3.4.1 Labelling Scheme

Let $G = (V, E)$ be a graph, $R \subseteq V$ be a set of landmarks, and $|R| \ll |V|$ (i.e., $|R|$ is sufficiently smaller than $|V|$). We first preprocess the graph G to obtain a compact representation of the shortest paths among landmarks, called a *meta-graph* of G. Then, based on such a meta-graph, we define a labelling scheme to assign a label to each vertex in G such that, given a pair of any vertices $u, v \in V$, we can efficiently compute a sketch for answering $SPG(u, v)$.





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

(a) (b) (c)

Figure 3.3: (a) A graph with three landmarks $\{1,2,3\}$ (highlighted in green), (b) a meta-graph, and (c) a path labelling.

Definition 3.4.1. [Meta-graph] *A* meta-graph *is* $M = (R, E_R, \sigma)$ *where* R *is a set of landmarks,* $E_R \subseteq R \times R$ *is a set of edges s.t.* $(r, r') \in E_R$ *iff at least one shortest path between* r *and* r' *does not go through any other landmarks, and* $\sigma : E_R \mapsto \mathbb{N}$ *assigns each edge in* E_R *a weight, i.e.* $\sigma(r, r') = d_G(r, r')$.

Conceptually, a meta-graph represents how landmarks are connected through their shortest paths in a graph G.

Definition 3.4.2. [Labelling scheme] *A* labelling scheme $\mathcal{L} = (M, L)$ *consists of a meta-graph* M *and a path labelling* L *that assigns to each vertex* $u \in V \backslash R$ *a label* $L(u)$ *s.t.*

$$L(u) = \{(r, \delta_{ur}) | r \in R, \delta_{ur} = d_G(u, r), \\ \exists p \in P_{ur}(V(p) \cap R = \{r\})\}. \tag{3.2}$$

Note that, to accurately present how vertices are linked to landmarks, we only allow that (r, δ_{ur}) is in the label $L(u)$ iff there exists at least one shortest path between u and r that does not contain other landmarks.

Example 3.4.3. *Figure 3.3 depicts a graph (a) and the meta-graph (b) and the path labelling (c) of this graph. The edge* $(1,3)$ *in the meta-graph is assigned with a weight* 2*, i.e.* $\sigma(1,3) = 2$*, since there is one shortest path between* 1 *and* 3 *which goes through* 4*. The label of* 4 *in the path labelling contains* $(1,1)$ *and* $(3,1)$*. The labelling entry* $(2,2)$ *is not included in the label of* 4 *because every shortest path between* 4 *and* 2 *goes through another landmark, i.e.* 1 *or* 3*.*

Algorithm 2 describes the pseudo-code of our algorithm for constructing a labelling scheme. Given a graph G and a set of landmarks R, we conduct a BFS from each landmark $r_i \in R$. We use two queues Q_L and Q_N to keep track of visited vertices, which respectively need to be labeled and not to be labeled. All vertices, except for r_i, are initialized as being unvisited (Line 5). For each vertex $u \in Q_L$ at the n-th level of the BFS, we set its unvisited neighbors v being visited (Line 10). If v is a landmark, we push v into Q_N and add an edge into E_R and store the distance between

Algorithm 2: Constructing a labelling scheme $\mathcal{L}$

Input: $G = (V, E)$; a set of landmarks $R \subseteq V$
Output: A labelling scheme $\mathcal{L} = (M, L)$ with $M = (R, E_R, \sigma)$.

```
1  E_R ← ∅;
2  L(v) ← ∅ for all v ∈ V
3  for all r_i ∈ R do
4      Q_L ← ∅;
5      Q_N ← ∅;
6      Q_L.push(r_i);
7      depth[r_i] ← 0; depth[v] ← ∞ for all v ∈ V\{r_i};
8      n = 0;
9      while Q_L and Q_N are not empty do
10         for all u ∈ Q_L at depth n do
11             for all unvisited neighbors v of u do
12                 depth[v] ← n + 1;
13                 if v is a landmark then
14                     Q_N.push(v);
15                     E_R ← E_R ∪ {(r_i, v)};
16                     σ(r_i, v) ← depth[v];
17                 else
18                     Q_L.push(v);
19                     L(v) ← L(v) ∪ {(r_i, depth[v])};
20         for all u ∈ Q_N at depth n do
21             for all unvisited neighbors v of u do
22                 depth[v] ← n + 1;
23                 Q_N.push(v);
24         n ← n + 1;
```

r_i and v to the edge in σ. Otherwise, we push v into Q_L and add a label in L for v (Lines 11-17). Then, We check unvisited neighbors of each vertex $u \in Q_N$ at the n-th level, and push v into Q_N without adding a label in L or an edge in M (Lines 18-21). This process is conducted level-by-level on the BFS (Line 22).

Example 3.4.4. *Figure 3.4 shows how our algorithm conducts BFSs to construct labels. The BFS from landmark* 1 *is depicted in Figure 3.4(a), in which vertices* $\{4, 5, 6, 7, 13, 14\}$ *are labelled because the other vertices are either landmarks or have landmarks in all their shortest paths to landmark* 1*. We add edges* $(1, 2)$ *and* $(1, 3)$ *into the meta-graph. In the BFS from landmark* 2 *in Figure 3.4(b), vertices* $\{7, 8, 9, 10, 11\}$ *are labelled because the shortest paths between* 2 *and vertices in* $\{4, 5, 6, 12, 13, 14\}$ *all go through landmark* 1 *or* 3*. The BFS from landmark* 3 *is depicted in Figure 3.4(c), which works in a similar manner.*

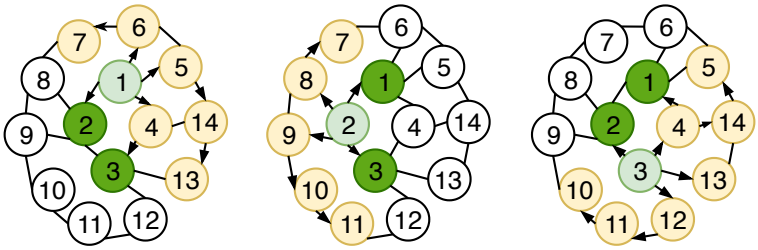


Figure 3.4: An illustration of labelling: (a), (b) and (c) describe the BFSs rooted at the landmarks 1, 2 and 3, respectively, where light and dark green vertices denote the landmarks, and yellow vertices denote those being labelled.

3.4.2 Fast Sketching

Let $\mathcal{L} = (M, L)$ be a labelling scheme on a graph G. For a given query $SPG(u, v)$, we proceed to answer $SPG(u, v)$ in two steps; (1) computing a sketch for two vertices u and v from the labelling scheme $\mathcal{L}$ efficiently; (2) computing the exact answer by conducting a guided search based on the sketch for two vertices u and v. Hence, the purpose of such a sketch is to provide an efficient and principled way of searching the answer of $SPG(u, v)$, which is particularly important in very large networks.

Definition 3.4.5. [Sketch] *A* sketch *for $SPG(u, v)$ on $\mathcal{L}$ is $S_{uv} = (V_S, E_S, \sigma_S)$ where $V_S = \{u, v\} \cup R$ is a set of vertices, E_S is a set of edges, and $\sigma_S : E_S \mapsto \mathbb{N}$ with $\sigma_S(u', v') = d_G(u', v')$, satisfying the condition that E_S contains only edges lying on the paths between u and v with the minimal length as defined below:*

$$d^{\top}_{uv} = \min_{(r,r')} \{\delta_{ru} + d_M(r, r') + \delta_{r'v} | (r, \delta_{ru}) \in L(u), (r', \delta_{r'v}) \in L(v)\}; \tag{3.3}$$

Accordingly, we have the following corollary.

Corollary 3.4.6. *$d^{\top}_{uv} \geq d_G(u, v)$ holds.*

Algorithm 3 describes how to construct a sketch. Let u and v be a pair of vertices. We start with $V_S = \emptyset$ and $E_S = \emptyset$. Then, for each pair of landmarks $\{r, r'\}$, we compute the minimum length $\pi_{rr'}$ of paths between u and v that go through r and r' using the labels in L and the meta graph M (Lines 2-5). After that, we obtain the minimum length of paths between u and v that go through at least one landmark, i.e., $d^{\top}_{uv}$ (Line 6), and add the edges in these paths into E_S, the vertices in these paths into V_S, and the corresponding distances are associated with the edges (Lines 7-13).

Example 3.4.7. *Figure 3.5(b) shows the sketch between two vertices 6 and 11. The sketch has the edges $(1, 6)$, $(1, 3)$, $(3, 11)$, $(2, 3)$ $(1, 2)$ and $(2, 11)$ because we have the following shortest paths between 6 and 11 with $\delta_{6,1} + d_M(1, 3) + \delta_{11,3} = 5$ and $\delta_{6,1} + d_M(1, 2) + \delta_{11,2} = 5$. We thus have $d^{\top}_{6,11} = 5$, and $d^{\top}_{6,11} = d_G(6, 11)$.*

Algorithm 3: Computing a sketch S_{uv}

Input: $\mathcal{L} = (M, L)$, two vertices u and v.
Output: A sketch $S_{uv} = (V_S, E_S, \sigma_S)$

$V_S \leftarrow \emptyset$, $E_S \leftarrow \emptyset$;
for *all* $\{r, r'\} \subseteq R$ **do**
  $\pi_{rr'} \leftarrow +\infty$;
  if $(r, \delta_{ur}) \in L(u)$ *and* $(r', \delta_{vr'}) \in L(v)$ **then**
    $\pi_{rr'} \leftarrow \delta_{ur} + d_M(r, r') + \delta_{vr'}$;
$d_{uv}^{\top} \leftarrow \min\{\pi_{rr'} | \{r, r'\} \subseteq R\}$;
for *all* $\{r, r'\} \subseteq R$ *and* $\pi_{rr'} = d_{uv}^{\top}$ **do**
  $E_S \leftarrow E_S \cup \{(u, r), (v, r')\}$;
  $\sigma_S(u, r) \leftarrow \delta_{ur}$, $\sigma_S(v, r') \leftarrow \delta_{vr'}$;
  for *all* (r_i, r_j) *in the shortest path graph of* (r, r') *in* M **do**
    $E_S \leftarrow E_S \cup \{(r_i, r_j)\}$;
    $\sigma_S(r_i, r_j) \leftarrow \sigma(r_i, r_j)$;
  $V_S \leftarrow V(E_S)$;

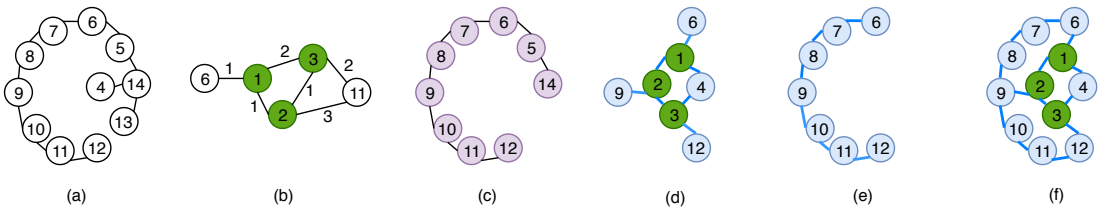


Figure 3.5: An illustration of sketching and searching: (a) the sparsified graph G^- of the graph G shown in Figure 3.3(a); (b) the sketch for SPG(6,11) on the graph G; (c) the bi-directional BFS on G^-, (d) the recover search based on $\mathcal{L}$, (e) the reverse search based on G^-, and (f) shows the query answer of SPG(6,11)

.

3.4.3 Guided Searching

Guided by S_{uv}, we conduct a search to compute the exact answer of $SPG(u, v)$, based on the following observations:

- Such a search can be conducted on a sparsified graph $G[V \backslash R]$ by removing all landmarks in R and all edges incident to these landmarks from G. $d_{G[V \backslash R]}(u, v)$ may potentially be greater than $d_G(u, v)$; however, the number of search steps in this sparsified graph can be upper bounded by $d^{\top}_{uv}$ due to the fact that $d_G(u, v) = min(d_{G[V \backslash R]}(u, v), d^{\top}_{uv})$.
- S_{uv} can guide how to conduct a bi-directional search on the sparsified graph $G[V \backslash R]$. Specifically, for $t \in \{u, v\}$, we have

$$d^*_t = \max_{(r,t) \in E_S} \sigma_S(r, t) - 1, \tag{3.4}$$

  which suggests the number of search steps from the u and v sides, respectively. Here, we subtract 1 because r can be found via labels of vertices in at most $\sigma_S(r, t) - 1$ steps.

Given a query $SPG(u, v)$ on a graph G, the answer G_{uv} can thus be computed by searching over the sparsified graph $G^- = G[V \backslash R]$ and the label scheme $\mathcal{L}$, guided by the sketch S_{uv}, as follows:

$$G_{uv} = \begin{cases} G^{\mathcal{L}}_{uv} & \text{if } d_{G^-}(u, v) > d^{\top}_{uv}; \\ G^-_{uv} \cup G^{\mathcal{L}}_{uv} & \text{if } d_{G^-}(u, v) = d^{\top}_{uv}; \\ G^-_{uv} & \text{otherwise.} \end{cases} \tag{3.5}$$

We use $G^{\mathcal{L}}_{uv}$ to refer to shortest paths between u and v that go through at least one landmark in R.

Generally, a guided search has three stages: (1) *Bi-directional search*, which has a *forward search* from the u side and a *backward search* from the v side [Goldberg and Harrelson, 2005], under the guide of S_{uv} w.r.t. Eq. 3.4. This search terminates when common vertices are found or the upper bound $d^{\top}_{uv}$ is reached. (2) *Reverse search*, which reverses the previous bi-directional search back to u and v in order to compute shortest paths in G^-_{uv}. (3) *Recover search*, which recovers the relevant labelling information under the guide of S_{uv} in order to compute shortest paths in $G^{\mathcal{L}}_{uv}$. As we do not know initially which of the three cases of Eq. 3.5 holds, a bi-directional search is always performed. This search provides us with $d_{G^-}(u, v)$, though we abort once $d_{G^-}(u, v) > d^{\top}_{uv}$ can be guaranteed. Then depending on the values of $d_{G^-}(u, v)$ and $d^{\top}_{uv}$, a reverse search, a recover search, or both of them are performed to compute G^-_{uv} and $G^{\mathcal{L}}_{uv}$ as in Eq. 3.5.

Algorithm 4 presents our guided search algorithm. We maintain two queues P_u and P_v which contain the set of all vertices traversed from u and v, respectively. d_u and d_v indicate the levels of traversal being conducted in the BFSs rooted at u and v,

Algorithm 4: Searching on $G[V \backslash R]$

Input: $G^- = G[V \backslash R]$, S_{uv}, $\mathcal{L} = (M, L)$
Output: A shortest path graph G_{uv}

$d^\top_{uv}, d^*_u, d^*_v \leftarrow get\_bound(S_{uv})$;
$P_u \leftarrow \emptyset$, $P_v \leftarrow \emptyset$, $d_u \leftarrow 0$, $d_v \leftarrow 0$;
Enqueue u to Q_u and v to Q_v;
$depth_u[w] \leftarrow \infty$, $depth_v[w] \leftarrow \infty$ for all $w \in V \backslash R$;
$depth_u[u] \leftarrow 0$, $depth_v[v] \leftarrow 0$;
while $d_u + d_v < d^\top_{uv}$ **do**
    $t \leftarrow pick\_search(P_u, P_v, d^*_u, d^*_v, d_u, d_v)$;
    if $t = u$ **then**
        $Q_u \leftarrow forward\_search(Q_u)$;
    if $t = v$ **then**
        $Q_v \leftarrow backward\_search(Q_v)$;
    $P_t \leftarrow P_t \cup Q_t$
; $d_t \leftarrow d_t + 1$;
$depth_t[w] \leftarrow d_t$ **for** $w \in Q_t$;
if $P_u \cap P_v$ *is not empty* **then**
    break;
if $P_u \cap P_v \neq \emptyset$ **then**
    $G^-_{uv} \leftarrow reverse\_search(P_u \cap P_v, G^-, depth_u, depth_v)$;
if $d_u + d_v = d^\top_{uv}$ **then**
    $Z \leftarrow \emptyset$;
    for *all* $(r, t) \in E_S$ *with* $t \in \{u, v\}$ **do**
        $d_m \leftarrow \min\{\sigma_S(r, t) - 1, d_t\}$;
        for *all* w *with* $depth_t[w] = d_m$, $(r, \delta_{wr}) \in L(w)$, $\delta_{wr} + d_m = \sigma_S(r, t)$ **do**
            $Z \leftarrow Z \cup \{(w, r)\}$;
    $G^{\mathcal{L}}_{uv} \leftarrow recover\_search(S_{uv}, \mathcal{L}, Z, G^-, depth_u, depth_v)$;
$G_{uv} \leftarrow G^-_{uv} \cup G^{\mathcal{L}}_{uv}$;

respectively. Two queues Q_u and Q_v keep vertices being searched from u and v at the d_u and d_v level, respectively. Initially P_u and P_v are empty, and u and v are enqueued into Q_u and Q_v respectively. $depth_u$ and $depth_v$ denote the depths of all vertices in the BFSs rooted at u and v.

A bi-directional search is first conducted (Lines 6-15). In each iteration, the bi-directional search is guided by d_u^* and d_v^* as well as the relative sizes of P_u and P_v to decide the next step (Line 7). We choose t where $d_t^* > d_t$ and $t \in \{u, v\}$. If both u and v satisfy this condition, or none of them satisfy this condition, then the choice of a forward search ($t = u$) and a backward search ($t = v$) is determined by the sizes of P_u and P_v. Accordingly, P_u or P_v are expanded (Line 12). The bi-directional search terminates either when $d_u + d_v$ reaches the upper bound $d_{uv}^\top$ or $P_u \cap P_v$ is not empty. This approach extends the *Optimized Bidirectional BFS* algorithm of [Hayashi et al., 2016] by incorporating bounds obtained from our sketch.

If $P_u \cap P_v$ is not empty, we have $d_{G^-}(u, v) \leq d_{uv}^\top$ and thus start a reverse search (Lines 16-17). For each vertex $x \in P_u \cap P_v$, we compute the shortest paths between u and x and between v and x according to the depths of vertices in $depth_u$ and $depth_v$, respectively. For example, a neighbour x' of x in G^- is on the shortest path between x and u if $depth_u[x] - 1 = depth_u[x']$, and thus we find such x' and compute shortest paths between x' and u in the same manner. If $d_u + d_v = d_{uv}^\top$, we have $d_{G^-}(u, v) \geq d_{uv}^\top$ and start a recover search (Lines 18-24). For each edge (r, t) in the sketch S_{uv} and $t \in \{u, v\}$, we search for all vertices w with $depth_t[w] = \min\{\sigma_S(r, t) - 1, d_t\}$ and $\sigma_S(r, t) = \delta_{wr} + depth_t[w]$ (Lines 19-23). Each w is a vertex closest to landmark r among all vertices on at least one shortest path between r and t in our previous bi-directional search. Z stores (w, r) pairs to guide the recover searches. In the recover search (Line 24), for each edge (r, r') in S_{uv} where $r, r' \in R$, we recover the shortest paths between r and r' according to $\mathcal{L}$. For each $(w, r) \in Z$, we find shortest paths between w and r according to G^- and labelling information $\mathcal{L}$. For example, for a neighbour w' of w in G^-, w' is on the shortest path between w and r if $(r, \delta_{w'r}) \in L(w')$ and $\delta_{w'r} + 1 = \delta_{wr}$. The shortest paths between w and u (resp. v) is computed according to $depth_u[\,]$ (resp. $depth_v[\,]$), but the search for parts of shortest paths that have already been found in the reversed search can be skipped. We also compute the shortest paths between relevant landmarks.

Example 3.4.8. *Figure 3.5(c)-(e) illustrates how our guided searching finds the answer for a query SPG(6,11). The sparsified graph G^- is depicted in Figure 3.5(a) and the sketch is depicted in Figure 3.5(b). The sketch provides the upper bound $d_{6,11}^\top = 5$, $d_6^* = 0$ and $d_{11}^* = 2$ because $\sigma_S(1, 6) = 1$ and $\sigma_S(2, 11) = 3$, respectively. The bi-directional BFS is depicted in Figure 3.5(c), in which $d_6 = 2$, $d_{11} = 3$, $P_6 = \{5, 7, 8, 14\}$, and $P_{11} = \{10, 12, 9, 8\}$. The queues P_6 and P_{11} meet at vertex 8, and thus $d_{G^-}(6, 11) = 5$. The reverse search is depicted in Figure 3.5(e), which goes back to 6 and 11 from $P_6 \cap P_{11} = \{8\}$. The recover search is depicted in Figure 3.5(d), which finds shortest paths going through the landmarks $\{1, 2, 3\}$ with $Z = \{(12, 3), (9, 2), (6, 1)\}$ and recovers shortest paths between landmarks in the sketch. The final query answer is depicted in Figure 3.5(f).*

3.5 Theoretical Discussion

We prove the correctness of QbS and analyze its complexity. We also discuss how to parallelize the labelling construction process.

3.5.1 Proof of Correctness

In the following, we prove the theorem for the correctness of QbS.

Theorem 3.5.1. *Given any query SPG(u,v) on a graph G, the answer G_{uv} can be computed using QbS.*

Proof sketch. We first prove that a labelling scheme constructed by Algorithm 2 satisfies Definition 3.4.2. Suppose that we conduct a BFS rooted from $r \in R$. Given a landmark $r' \in R \backslash \{r\}$, if $\exists p \in P_{rr'}(V(p) \cap R = \{r, r'\})$ holds, there must exist $w \in Q_L$ with $depth[w] + 1 = depth[r']$ and $(w, r') \in E$ (Lines 8-9, 11), and accordingly an edge (r, r') is added into M (Lines 13-14). Otherwise, r' is directly pushed into Q_N (Lines 19-21). Given a vertex $v \in V \backslash R$ that is not a landmark, if $\exists p \in P_{rv}(V(p) \cap R = \{r\})$ holds, there must exist $w \in Q_L$ with $depth[w] + 1 = depth[v]$ and $(w, v) \in E$ (Lines 8-9, 15), and accordingly a label $(r, depth[v])$ is added into L (Lines 16-17). Otherwise, v is directly pushed into Q_N (Lines 19-21).

Now we prove that a sketch constructed by Algorithm 3 satisfies Definition 3.4.5. First, Algorithm 3 (Lines 2-7) finds pairs of landmarks (r, r') that minimise $\{\delta_{ur} + d_M(r, r') + \delta_{r'v} | (r, \delta_{ur}) \in L(u)$ and $(r', \delta_{r'v}) \in L(v)\}$ (i.e., satisfying Eq. (3) in Definition 3.4.5). Then it adds $(u, r), (r', v)$ and all edges on the shortest paths between (r, r') on a meta-graph into the sketch (Lines 8-12).

Finally, we prove that G_{uv} can be constructed by Algorithm 4. Each shortest path between u and v that does not go through any landmark can be constructed from G^- using a bi-directional BFS and its reverse search (Lines 6-15 and 16-17). For each shortest path between u and v that goes through at least one landmark, all such landmarks must be included in S_{uv} and such shortest paths are computed using the recover search (Lines 18-24). □

3.5.2 Complexity Analysis

The time complexity of constructing a BFS from one landmark in Algorithm 2 is $O(|V| + |E|)$ and the overall time complexity of Algorithm 2 is $O(|R||V| + |R||E|)$. The time complexity of constructing a sketch in Algorithm 3 is $O(|R|^4)$ and can be reduced to $O(|R|^2)$ by precomputing shortest path distances and shortest paths between landmarks on a meta-graph constructed by Algorithm 3, i.e., computation on Lines 10-12 is saved. The space complexities for storing meta-graph will increase from $O(|R|^2)$ to $O(|R|^4)$. The time complexity of conducting a guided search in Algorithm 4 is $O(|E| + |R||V|)$.

Note that, in this work, the number of landmarks is small, i.e., $|R| = 20$ by default, which is much smaller than the number of nodes or edges in the original

graph. Thus, we can see that, constructing a label scheme by Algorithm 2 is indeed $O(|E|)$, computing a sketch is constant time, and performing a guided search becomes $O(|E^*| + |V|)$ where $|E^*|$ denotes the number of edges in the sparsified graph after removing edges incident to landmarks from G.

3.5.3 Parallelization

Given a graph G and a set of landmarks R in G, a nice property of our labelling scheme $\mathcal{L}$ is that there is only one such labelling scheme. Formally, we prove the lemma below.

Lemma 3.5.2. *Let $\mathcal{L}$ be a labelling scheme on a graph G w.r.t. a set of landmarks R. $\mathcal{L}$ is deterministic.*

Proof sketch. A labelling scheme $\mathcal{L}$ consists of a meta-graph $M = (R, E_R, \sigma)$ and a path labelling L. From Definition 3.4.1, an edge $(r, r') \in E_R$ if and only if there exists at least one shortest path between r and r' that does not go through any other landmarks in $R \backslash \{r, r'\}$. From Definition 3.4.2, a label $(r, \delta_{ur}) \in L(u)$ if and only if there exists at least one shortest path between u and r that does not go through any other landmarks in $R \backslash \{r\}$. Therefore, $\mathcal{L}$ is deterministic w.r.t G and R. □

For a fixed set of landmarks, the labelling construction in Algorithm 2 yields the same label scheme, regardless of the ordering of landmarks. This deterministic nature of labelling scheme enables us to speed up the construction of labelling scheme by paralleling Algorithm 2. If we use one thread for constructing labels from one landmark, then we can leverage the thread-level parallelism to perform BFSs from different landmarks simultaneously.

3.6 Experiments

We evaluated our method *QbS* to answer the following questions:

(**Q1**) How efficiently can our proposed method answer shortest-path-graph queries, while still achieving construction time efficiency and low labelling space overhead?

(**Q2**) How well can sketching help improve the performance of answering shortest-path-graph queries?

(**Q3**) How does the number of landmarks affect the performance (construction time, labelling size and query time) of our proposed method?

3.6.1 Experimental Setup

We implemented our proposed methods in C++ 11 and compiled using g++. We performed all experiments on a Linux server which has Intel Xeon W-2175 with 2.5GHz and 512GB of main memory.

(a)

Dataset	Network	Type
Douban (DO)	social	undirected
DBLP (DB)	co-authorship	undirected
Youtube (YT)	social	undirected
WikiTalk(WK)	communication	directed
Skitter (SK)	computer	undirected
Baidu (BA)	web	directed
LiveJournal (LJ)	social	directed
Orkut (OR)	social	undirected
Twitter (TW)	social	directed
Friendster (FR)	social	undirected
uk2007 (UK)	web	directed
ClueWeb09 (CW)	computer	directed

(b)

Dataset	$\|V\|$	$\|E\|$	$\|E^{un}\|$	max. deg	avg. deg	avg. dist	$\|G\|$
Douban	0.2M	0.3M	0.3M	287	4.2	5.2	2.5MB
DBLP	0.3M	1.1M	1.1M	343	6.6	6.8	8.0MB
Youtube	1.1M	3.0M	3.0M	28,754	5.27	5.3	23MB
WikiTalk	2.4M	5.0M	4.7M	100,029	3.89	3.9	36MB
Skitter	1.7M	11.1M	11.1M	35,455	13.08	5.1	85MB
Baidu	2.1M	17.8M	17.0M	97,848	15.89	4.1	130MB
LiveJournal	4.8M	68.5M	43.1M	20,334	17.79	5.5	329MB
Orkut	3.1M	117M	117M	33,313	76.28	4.2	894MB
Twitter	41.7M	1.5B	1.2B	2,997,487	57.74	3.6	9.0GB
Friendster	65.6M	1.8B	1.8B	5,214	55.06	4.8	13.0GB
uk2007	106M	3.7B	3.3B	979,738	62.77	5.6	24.8GB
ClueWeb09	1.7B	7.8B	7.8B	6,444,720	9.27	7.5	58.2GB

Table 3.2: Details of datasets, where $|E^{un}|$ is the number of edges in a graph being treated as undirected, and $|G|$ denotes the size of a graph G with each edge appearing in the adjacency lists and being represented by 8 bytes.

Datasets. We conducted experiments on 12 real-world graph datasets from various types of complex large networks, including social networks, computer networks, web networks, co-authorship networks and communication networks. Table 3.2 presents the details of these datasets, among which the largest one has 1.7 billion vertices and 7.8 billion edges. We treated graphs in these datasets as being undirected. All the datasets used in our experiments are publicly available from Koblenz Network Collection [Kunegis, 2013], Stanford Networks Analysis Project [Leskovec and

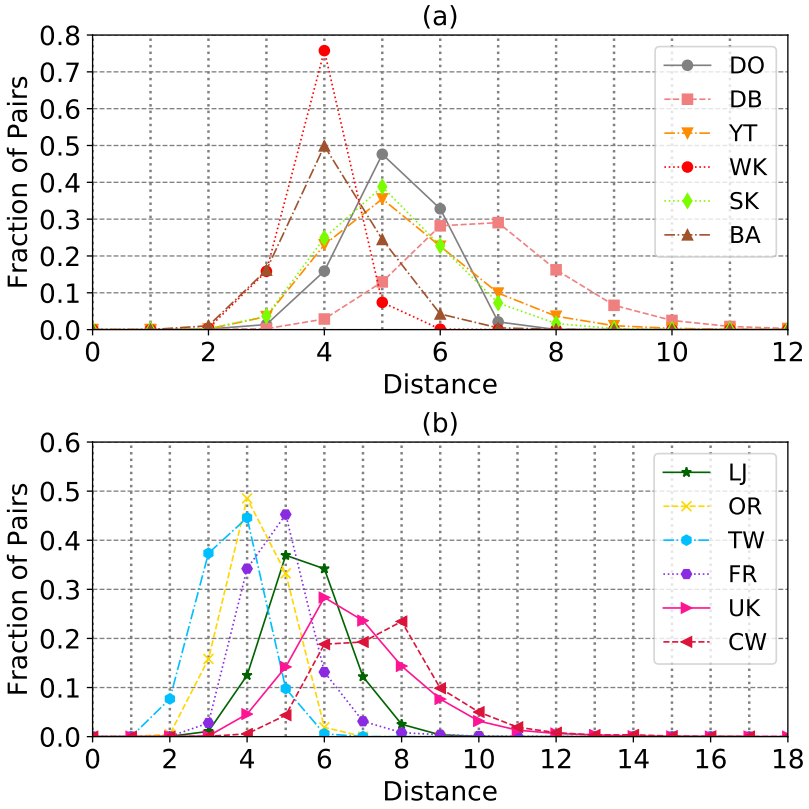


Figure 3.6: Distance distribution of 10,000 randomly selected pairs of vertices on all the datasets.

Krevl, 2014], Dynamically Evolving Large-scale Information Systems Project [1] and the Lemur Project[2].

Queries. We randomly sampled 10,000 pairs of vertices from all pairs of vertices in each graph, i.e., $V \times V$, to evaluate the average query time on each graph. Figure 3.6 shows the distance distribution of these 10,000 randomly sampled pairs of vertices in each graph dataset. We can see that the distances of these pairs of vertices mostly fall into the range of 2-9.

Baselines. We considered the following baselines:

(1) **Labelling-based methods**. Pruned landmark labelling (PLL) is the state-of-the-art method for computing exact distance queries [Akiba et al., 2013]. We thus use the methods *Pruned Path Labelling* (PPL) and *Pruned Path Labelling with Parent information* (ParentPPL) as discussed in Section 3.3 as our baselines.

(2) **Search-based methods.** We use bi-directional BFS as the baseline which conducts search from the directions of two vertices alternatively [Goldberg and Harrelson, 2005]. We denote it as Bi-BFS.

[1]See http://law.di.unimi.it/datasets.php for datasets
[2]See https://lemurproject.org/clueweb09/index.php

To evaluate the parallel speed-up of construction time, we use QbS to refer to our method with a sequential labelling construction and QbS-P to refer to our method with a parallel labelling construction, with up to 12 threads in our experiments. In PPL and ParentPPL, we use 32 bits and 8 bits to represent a landmark and a distance in their labels, respectively, and 32 bits to store each parent in ParentPPL. In QbS and QbS-P, we use $|R|$*8 bits to store the label of each vertex.

Landmarks. In PPL and ParentPPL, landmarks are ordered in descending order of degrees. In QbS, we choose vertices with the largest degrees as landmarks for two reasons: (1) removing high-degree vertices sparsifies a graph much more than low-degree vertices; (2) computing distances from two vertices to high-degree landmarks provides a good estimation of the shortest distance between these two vertices [Potamias et al., 2009]. We set $|R| = 20$ in QbS by default.

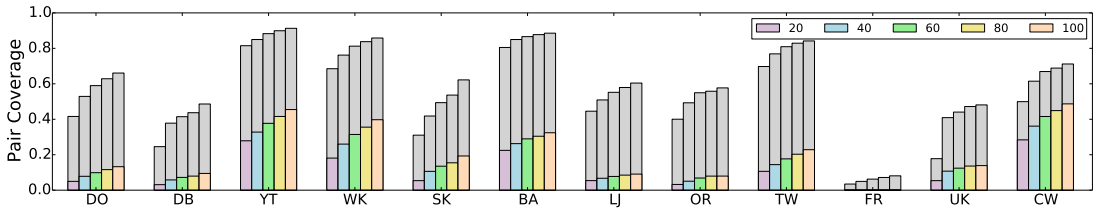


Figure 3.7: Pair coverage ratios using our method QbS under 20-100 landmarks where light color denotes the ratio of all the shortest paths between a vertex pair go through landmarks and grey color denoted the ratio of some but not all shortest paths between a vertex pair go through landmarks.

Dataset	Construction Time (sec.)			
	QbS-P	QbS	PPL	ParentPPL
Douban	0.05	0.3	154	2,736
DBLP	0.12	1.1	2,610	11,049
Youtube	0.47	4.4	22,601	DNF
WikiTalk	0.61	4.9	8,662	DNF
Skitter	1.51	12.7	86,326	DNF
Baidu	2.04	18.9	DNF	OOE
LiveJournal	6.48	52.2	DNF	OOE
Orkut	10.85	73.2	DNF	OOE
Twitter	199.8	1,345	DNF	OOE
Friendster	416.5	2,354	DNF	OOE
uk2007	178.5	1,485	OOE	OOE
ClueWeb09	1,819	17,060	OOE	OOE

Table 3.3: Comparison of construction time. DNF and OOE refer to running out of time (>24 hours) and running out of memory, respectively.

Dataset	Average Query Time (ms.)			
	QbS	PPL	ParentPPL	Bi-BFS
Douban	0.037	1.414	0.038	0.585
DBLP	0.097	1.782	0.052	2.995
Youtube	0.218	5.314	-	23.809
WikiTalk	0.693	3.536	-	6.984
Skitter	0.951	16.978	-	44.685
Baidu	0.845	-	-	174.412
LiveJournal	1.095	-	-	84.967
Orkut	4.237	-	-	207.541
Twitter	164.333	-	-	4,817.774
Friendster	11.972	-	-	3,600.362
uk2007	77.830	-	-	5,264.101
ClueWeb09	480.443	-	-	DNF

Table 3.4: Comparison of construction time and query time. DNF refers to running out of time (>24 hours).

3.6.2 Performance Comparison

We conducted experiments to compare construction time, labelling size and query time of our method against the baselines.

3.6.2.1 Construction Time

Table 3.3 shows that our method QbS can efficiently construct a label scheme on all the datasets, scaling over large networks with billions of vertices and edges. Compared with PPL and ParentPPL, our method QbS uses a significantly less amount of time (i.e., 2-4 orders of magnitude faster) to construct labelling information. Moreover, PPL failed to construct labels for 7 out of 12 datasets and ParentPPL failed for 10 out of 12 datasets. This is because these methods need to meet the 2-hop path cover property. The reason why ParentPPL is much slower than PPL is because a vertex often has more than one parent and finding all parents takes more time though the time complexity remains unchanged. We can also see that, compared with QbS, QbS-P can further improve construction time (i.e., 6-12 times faster), leading to much better scalability than QbS.

3.6.2.2 Labelling Size

Table 3.5 presents the comparison results for the labelling sizes of QbS, PPL and ParentPPL on all the datasets. We use $size(\Delta)$ to denote the size of precomputed shortest path graphs between landmarks as discussed in Section 3.5.2. We observe that: 1) the labelling sizes of QbS are hundreds of times smaller than the labelling sizes of PPL and ParentPPL; 2) the labelling sizes of ParentPPL are about twice as the labelling sizes of PPL. For dense graphs, such as Twitter, the sizes of precomputed

shortest paths in QbS are relatively larger than the ones in sparse graphs. This is due to the existence of many shortest paths between landmarks in dense graphs. Nonetheless, it is important to notice that, the sizes of precomputed shortest paths between landmarks (i.e. $size(\Delta)$ in Table 3.5) are small in QbS, compared with the sizes of labelling (i.e. $size(\mathcal{L})$ in Table 3.5). For meta-graphs, since each meta-graph contains at most $|R|^2$ edges, the space overhead for storing edges and weights of a meta-graph is very small. Indeed, even when we have $|R|$=100, the size of a meta-graph would still be smaller than 0.01MB. In summary, these results show that QbS can scale well over very large networks in terms of the labelling size.

Dataset	QbS		PPL	ParentPPL
	$size(\mathcal{L})$	$size(\Delta)$		
Douban	2.95MB	0.03MB	0.4GB	0.8GB
DBLP	6.05MB	0.03MB	1.2GB	2.4GB
Youtube	21.6MB	0.6MB	1.7GB	–
WikiTalk	45.7MB	0.7MB	2.1GB	–
Skitter	32.4MB	20.3MB	9.2GB	–
Baidu	40.8MB	4.8MB	–	–
LiveJournal	92.5MB	1.1MB	–	–
Orkut	58.6MB	3.5MB	–	–
Twitter	0.78GB	0.76GB	–	–
Friendster	1.22GB	0.01GB	–	–
uk2007	1.98GB	0.08GB	–	–
ClueWeb09	31.4GB	0.48GB	–	–

Table 3.5: Comparison of labelling sizes. $size(\mathcal{L})$ denotes the size of a labelling scheme $\mathcal{L}$ and $size(\Delta)$ the size of precomputed shortest-path graphs between landmarks in QbS.

3.6.2.3 Query Time

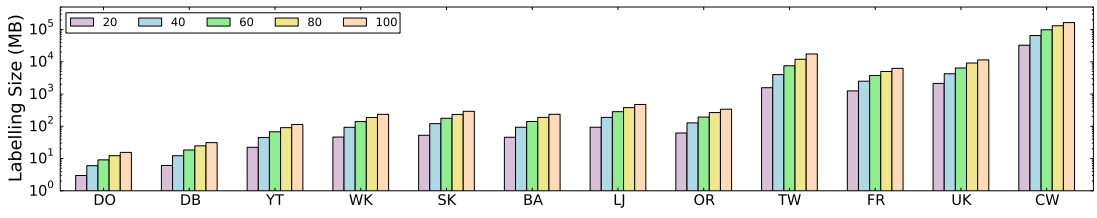


Figure 3.8: Labelling sizes using QbS under 20-100 landmarks on all the datasets.

Table 3.4 presents the comparison results of our method with the baselines in terms of query time. Compared with the search-based method Bi-BFS, our method QbS can answer queries much more efficiently, i.e., 10-300 times faster than Bi-BFS. Particularly, QbS is able to answer queries within milliseconds for 8 out of 12 datasets, and less than 0.5 seconds for the other datasets which have up to 1.7 billion vertices and 7.8 billion edges. We notice that, Twitter has significantly higher query time than

Friendster and uk2007. This is because, compared with the other graphs, Twitter has larger shortest-path graphs as shown by $size(\Delta)$ in Table 3.5 due to densely connected vertices with very high degrees. For labelled-based methods, the query times of both PPL and ParentPPL are much faster than Bi-BFS. However, neither PPL nor ParentPPL is scalable. PPL can only answer queries for the first 5 datasets, while ParenetPPL can only answer queries for the first 2 datasets which have less than 1 million vertices. This is because that constructing labelling information required by these methods is computationally expensive for very large graphs.

3.6.3 Effects of Sketching

We conducted an experiment to understand how sketching improves the performance of query answering in our method.

Figure 3.7 presents the pair coverage ratios of our method QbS using 20-100 landmarks. Here, pair coverage ratio refers to the proportion of queries in which the shortest paths between two vertices go through at least one landmark, among 10,000 queries used in our experiments. We distinguish two cases: (i) Queries in which *all shortest paths* between two vertices go through at least one landmark; (ii) Queries in which *some but not all shortest paths* between two vertices go through at least one landmark. Pair coverage ratios reflect the effectiveness of sketching used in our method QbS since a sketch cannot guide queries in which none of shortest paths between two vertices go through landmarks.

From Figure 3.7, we can see that: (1) When the number of landmarks increases, the pair coverage ratios go up for both Case (i) and Case (ii); nonetheless, the increasing rate generally slows down. (2) For datasets in which graphs have high degree vertices compared with their other vertices, such as Youtube, WikiTalk, Baidu, Twitter, and ClueWeb09, their pair coverage ratios are generally higher than the other datasets. This is because these high degree vertices are more likely on the shortest paths of the other vertices. For Friendster, as it does not have high degree vertices, the pair coverage ratios are quite low. (3) For datasets in which graphs are sparse after removing landmarks that are vertices of high degrees, such as Youtube, WikiTalk, Baidu and ClueWeb09, the percentage of pair coverage ratio for Case (i) among pair coverage ratios for both cases is higher than the other datasets. In Friendster, the degrees of vertices are more evenly distributed; hence, landmarks hardly capture all shortest paths between two vertices and the pair coverage ratios for Case (i) are extremely low. However, the reasons why query time on Friendster is still fast are twofold: (1) QbS does not store parent information for reverse search since most parent vertices do not lead to shortest paths being recovered, and (2) QbS uses sketches to guide which side to expand for bi-directional searches.

3.6.4 Performance with Varying Landmarks

We also conducted experiments to evaluate how the number of landmarks may affect the performance of our method.

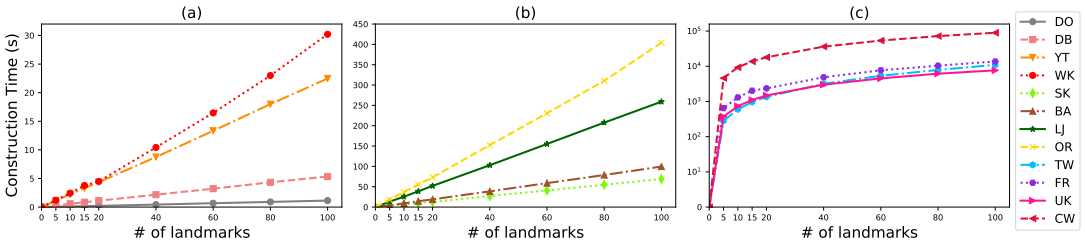


Figure 3.9: Construction times using QbS under 0-100 landmarks on all the datasets.

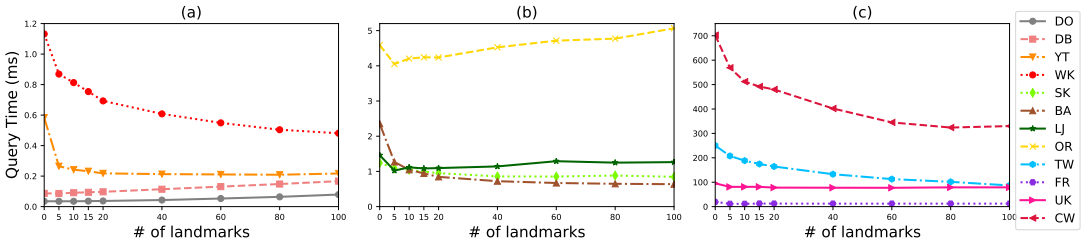


Figure 3.10: Average query times using our method QbS under 0-100 landmarks on all the datasets.

3.6.4.1 Construction Time

The construction times of our method QbS against different numbers of landmarks (from 20 to 100) are shown in Figure 3.9. Generally, the construction time grows linearly. In Figure 3.9(a)-(b), for datasets with millions of edges, QbS can construct labels under 100 landmarks within at most a few minutes. In Figure 3.9 (c), for datasets with billions of edges, QbS can construct labels within a few hours. It can be seen that the construction time is almost linear in the number of landmarks on each dataset, which confirms the scalability of QbS.

3.6.4.2 Labelling Size

We compared the labelling sizes of QbS against different numbers of landmarks in Figure 3.8. For a labelling scheme $\mathcal{L} = (M, L)$, we use $|R|$*8 bits to store labels of each vertex. For M, as discussed in Section 3.6.2.2, the labelling size of a meta-graph is very small, compared with the labelling size of Δ and L. It increases when the number of landmarks becomes larger. Nonetheless, even when $|R|$=100, the labelling size of a meta-graph would still be smaller than 0.01MB. For Δ, since we store the shortest paths between $|R|^2$ pairs, it grows fast when the number of landmarks increases. However, compared with the size of labels in L as shown in Table 3.5, Δ is small. The sizes of shortest paths between vertices with lower degrees are smaller than the ones between vertices with higher degrees. Thus, the labelling size of Δ does not increase quadratically in the number of landmarks. The sizes of path labelling L are linear in terms of the number of landmarks.

3.6.4.3 Query Time

The impact of varying landmarks on query time is shown in Figure 3.10. When the number of landmarks increases, there are generally three cases: 1) the query times increase, e.g., Douban, DBLP and Orkut; 2) the query times decrease, e.g., WikiTalk, Twitter and ClueWeb09; 3) the query times have no significant changes, e.g., LiveJournal and uk2007. If a graph has very high degree vertices, selecting more landmarks often decreases query times because removing more landmarks can further sparsify the graph significantly. For example, in Twitter, 38 million edges are incident to 20 landmarks, while 100 landmarks have around 123 million edges; accordingly, the query time under 100 landmarks is half as the query time under 20 landmarks. If degrees of vertices in a graph are evenly distributed such as Orkut, more landmarks do not necessarily improve query time; instead, due to increased computational cost for computing a sketch, query time often increases.

3.6.5 Remarks

In general, QbS has three sources of efficiency gains when answering shortest-path-graph queries:

(1) QbS enables queries to traverse on a graph whose parts with high centrality are sparsified. Thus, although removing a small number of landmarks alone does not significantly reduce the number of edges in a whole graph (e.g., 3.2% of edges are removed with 20 landmarks in Twitter), the number of edges traversed by queries is significantly reduced (e.g., around 30% less of edges being traversed by queries in QbS against Bi-BFS).

(2) QbS uses a sketch to guide the search for each query, further reducing the number of edges being traversed. Take Twitter for example, after adding the guide of sketches on a sparsified graph, 66% less of edges are traversed in QbS against Bi-BFS.

(3) QbS can avoid the computation of shortest paths between high-degree landmarks when two or more landmarks appear on one shortest path, since these shortest paths can be precomputed as discussed in Section 3.5.2.

In our experiments, the performance of QbS varies in datasets, depending on how the characteristics of datasets support these sources of gains to speed up query efficiency.

3.7 Conclusions

In this chapter, we have proposed a novel method, *Query-by-Sketch* (QbS), to answer shortest-path-graph queries on large graphs. QbS constructs a labelling scheme through pre-computation, and then answers queries by performing online computation that involves fast sketching and guided searching. We have analysed the complexity and correctness of our method. We have proven that our labelling scheme is

deterministic and can be constructed through a parallelized process. We have conducted experiments on 12 large real-world graphs to empirically verify the scalability and efficiency of QbS.

Chapter 4

Top-k Relative Coverage

4.1 Overview

The shortest path structure between one vertex and other vertices can be used to analyse how this vertex connects with other vertices in a graph. Previously, the Dijkstra and breath-first search algorithms can be applied to compute the shortest paths between a specific vertex and all other vertices. However, for large-scale networks, it is not only computationally expensive to compute these paths, but also hard to extract useful information (e.g., important vertices or edges) from these paths. Though several centrality measures have been proposed to identify important vertices that frequently appears on shortest paths (e.g., closeness centrality [Bavelas, 1950], betweenness centrality [Freeman, 1977] and coverage centrality [Yoshida, 2014]), vertices that are identified as important according to these centralities may not be important to a specific vertex.

Therefore, based on coverage centrality [Yoshida, 2014], we formally define the relative coverage. We study a novel problem, called the top-k relative coverage problem, which finds vertices that are influential in connecting one vertex with all other vertices based on shortest paths. By exploiting the problem structure, we propose an efficient method which only requires to compute the relative coverage of vertices in a small candidate set, i.e., eliminating unnecessary computations on vertices that cannot be in a top-k query answer. Then, to speed up the computation, we further propose a bit-parallel method which, different from thread-level parallelism, can compute the relative coverage of up to 64 vertices simultaneously using compact bit vector encoding. To evaluate the performance of our methods, we conduct experiments on 6 real-world datasets, among which the largest one has millions of vertices and edges.

Outline. The rest of this chapter is organized as follows. In Section 4.2, we define relative coverage based on the coverage centrality and formalize the top-k relative coverage problem. We first develop an efficient algorithm to tackle the top-k relative coverage problem in Section 4.3 and then propose an optimisation method to speed up the computation of relative coverage in Section 4.4. Section 4.5 discusses the experimental result. We conclude this chapter in Section 4.6.

Notations	Definitions and Descriptions
$G = (V, E)$	a graph with a vertex set V and an edge set E
$N(v)$	neighbors of v in G
$d(s, t)$	shortest path distance between s and t
P_{st}	set of vertices on any shortest path between s and t
$CC(u)$	coverage centrality of u
$RC(u\|s)$	relative coverage of u w.r.t. s
$Pred_s(v)$	set of predecessors of v w.r.t. s
$Succ_s(v)$	set of successors of v w.r.t. s

Table 4.1: Frequent used notations.

4.2 Preliminaries

Let $G = (V, E)$ denote an unweighted graph where V and E represent the set of vertices and edges, respectively. Without loss of generality, we assume that G is undirected since our work can be easily extended to directed graphs. We use $N(v)$ to denote the set of neighbors of v. Given two vertices $s, t \in V$, we use $d(s, t)$ to denote the length of the shortest paths between s and t, and P_{st} the set of all vertices on the shortest paths between s and t.

Given a vertex $u \in V$, the *coverage centrality* of u measures the importance of u by the number of distinct vertex pairs whose shortest paths contain vertex u [Yoshida, 2014]. Formally, it is defined as:

$$CC(u) = |\{(s, t)|s, t \in V, u \in P_{st}\}|. \tag{4.1}$$

In this work, we define *relative coverage* to describe the importance of a vertex u w.r.t. a given vertex s:

$$RC(u|s) = |\{(s, t)|t \in V, u \in P_{st}\}|. \tag{4.2}$$

Intuitively, the relative coverage of a vertex u w.r.t. a given vertex s measures the relative importance of u w.r.t. s, which corresponds to the number of distinct vertices which have at least one shortest path to s going through the vertex u. For clarity, we call such a set of vertices the *cover set* of $RC(u|s)$ on a graph G.

Based on the terminology above, we formalize the top-k relative coverage query.

Problem 4.2.1. *Given a graph $G = (V, E)$, a vertex $s \in V$ and $k \ll |V|$, a* top-k relative coverage query*, denoted as $Q = (G, s, k)$, is to find a sequence of vertices $\{v_1, v_2 \ldots, v_k\}$ from $V \backslash \{s\}$such that:*

(1) $\sum_{1 \leq i \leq k} RC(v_i|s)$ is maximized, and

(2) $RC(v_i|s) \geq RC(v_{i+1}|s)$ for $i = 1, \ldots, k-1$.

4.3 Proposed Method

In this section, We first discuss how to compute relative coverage. Then, we propose a method to answer top-k relative coverage queries efficiently. We analyze the complexity of our method.

Given a vertex $s \in V$, we define the predecessors $Pred_s(v)$ and successors $Succ_s(v)$ of a vertex v w.r.t. the vertex s as:

$$Pred_s(v) = \{u \in N(v) |\ d(s,u) = d(s,v) - 1\}; \tag{4.3}$$

$$Succ_s(v) = \{u \in N(v) |\ d(s,u) = d(s,v) + 1\}. \tag{4.4}$$

The following lemma states that, if a vertex u lies on one shortest path between two vertices s and v, then u must also lie on at least one shortest path between s and any successor of v.

Lemma 4.3.1. *If $u \in P_{sv}$, then $u \in P_{st}$ for each $t \in Succ_s(v)$.*

Proof. Since $u \in P_{sv}$ implies that $d(s,u) + d(u,v) = d(s,v)$, we know u lies on one shorest path between s and v. By $t \in Succ_s(v)$, we know that $d(s,t) = d(s,v) + 1$. Therefore, we have $d(s,u) + d(u,v) + d(v,t) = d(s,t)$ and $u \in P_{st}$. □

Based on Lemma 4.3.1, we propose an algorithm for computing $RC(u|s)$, which performs a BFS rooted at s to detect vertices in its cover set. Let T denote the set of all vertices in the cover set and $\overline{T} = V - T$ (i.e., vertices that are not in the cover set). We maintain two queues Q_T and $Q_{\overline{T}}$, which keeps track of vertices in T and $\overline{T}$, respectively. Algorithm 5 describes the pseudo-code of the algorithm, where *depth* indicates the levels being conducted in the BFS rooted in s and $d_s[\ ]$ denotes the depth of all vertices in the BFS rooted in s. First, $d_s[\ ]$ of each $v \in V$, except for s, is initialized as $+\infty$. During the search on nth-level, we find each vertex u with $d_s[u] = n$ in Q_T and then add their neighbor v with $d_s[v] = +\infty$ into Q_T. Similarly, we find each vertex u with $d_s[u] = n$ in Q_R and add their neighbor v with $d_s[v] = +\infty$ into Q_R. $RC[u]$ equals to the number of vertices in T.

Example 4.3.2. *Figure 4.1 illustrates how our algorithm conducts a BFS rooted at vertex* 5 *to compute* $RC(3|5)$. *The given graph is depicted in Figure 4.1(a). The cover set of* $RC(3|5)$ *is depicted in Figure 4.1(b), i.e.,* $T = \{0, 3, 7, 8, 9, 16, 17, 18\}$ *and* $|T| = 8$. *The arrows indicate the traversal from vertices to their successors w.r.t the root vertex* 5.

Next, we discuss how to answer a top-k relative coverage query in an efficient way. A naive method for finding top-k vertices with maximum relative coverage is to compute the relative coverage value of each vertex in a graph and then pick k vertices with highest values. However, this naive method is inefficient since it requires to enumerate the relative coverage values of all vertices for answering one top-k relative coverage query. A question arising is: *can we reduce the search space for top-k vertices in the answer of a top-k relative coverage query by considering only a small subset of vertices in a graph while still guaranteeing to find the correct answer?*

Algorithm 5: Single-RC

Input: G, s, u.
Output: $RC[u]$

```
1  d_s[v] ← +∞ for all v ∈ V;
2  d_s[s] ← 0;
3  Q_T̄.add(s);
4  depth ← 0;
5  while Q_T ∪ Q_T̄ ≠ V do
6      for v ∈ Q_T s.t. d_s[v] = depth do
7          for neighbor w of v s.t. d_s[w] = +∞ do
8              d_s[w] ← d_s[v] + 1;
9              Q_T.add(w);
10     for v ∈ Q_T̄ s.t. d_s[v] = depth do
11         for neighbor w of v s.t. d_s[w] = +∞ do
12             d_s[w] ← d_s[v] + 1;
13             if v = u then
14                 Q_T.add(v);
15                 continue;
16             Q_T̄.add(w);
17     depth ← depth + 1;
18 RC[u] ← |Q_T|;
```

Given a top-k relative coverage query $Q = (G, s, k)$, the answer of this query is a sequence of vertices $\{v_1, v_2 \ldots, v_k\}$ that must meet the two conditions stipulated in Problem 4.2.1. Since $RC(v_i|s) \geq RC(v_j|s)$ holds for $1 \leq i < j \leq k$, we observe a property where each vertex v_i in the sequence "depends on" the previous vertices $(v_1, \ldots, v_{i-1})$. The following lemma formulates this property.

Lemma 4.3.3. *For a query $Q = (G, s, k)$, the top-k vertices in its answer $\{v_1, v_2 \ldots, v_k\}$ must satisfy the condition $Pred_s(v_i) \subseteq \{s\} \cup \bigcup_{j=1}^{i-1}\{v_j\}$ where $i \in [1, k]$.*

Proof. We know that, for each $u \in Pred_s(v)$, $RC(v|s) < RC(u|s)$ must hold. Thus, if v_i is the i-th vertex in the answer, its predecessors $Pred_s(v_i)$ must be in $\{s\} \cup \{v_1, \ldots, v_{i-1}\}$ since these predecessors have higher relative coverage values w.r.t. vertex s. □

Thus, when answering a top-k relative coverage query, there is no need to compute the relative coverage for all the vertices and then select the top-k vertices. Instead, by Lemma 4.3.3, we can eliminate vertices whose predecessors do not depend on s and the previously selected vertices in the sequence, thereby restricting the search space of finding vertices in the top-k sequence.

For clarity, let $Basis(v_i) = \{s\} \cup \bigcup_{j=1}^{i-1}\{v_j\}$ denote the *basis* of the i-th vertex v_i in the top-k sequence. Then, we define that each i-th vertex v_i is associated with a

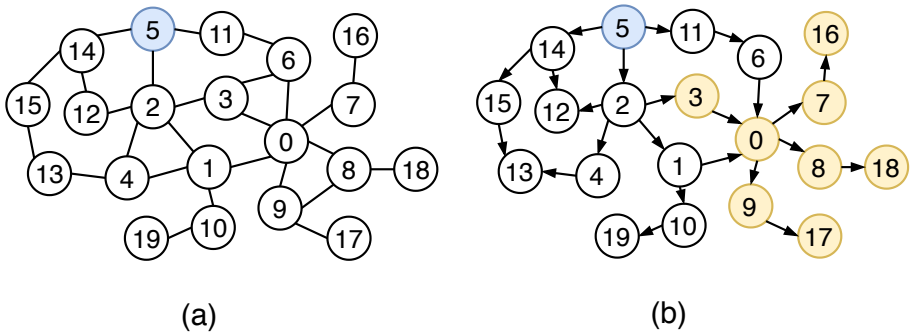


Figure 4.1: An illustration of computing $RC(3|5)$ using Algorithm 5, where the source vertex 5 is colored in blue and the vertices in the cover set are colored in yellow. $RC(3|5) = 8$ since there are 8 vertices in its cover set.

candidate set $C_i = \{v \in V | Pred_s(v) \subseteq Basis(v_i)\} - Basis(v_i)$. Intuitively, the candidate set C_i is a subset of vertices in V that are the candidates for the i-th vertex v_i. By Lemma 4.3.3, we know that C_i must contain the i-th vertex v_i in the answer of a top-k relative coverage query. This thus enables us to iteratively compute the top-k vertices, and in each i-th iteration only compute the relative coverage for vertices in the candidate set C_i and then select the vertex with the highest relative coverage as x_i. Further, since candidate sets in different iterations may be overlapping, to avoid repeated computations, we only need to compute relative coverage for each newly added vertex in $\Delta C_i = C_i - C_{i-1}$ at each i-th iteration and start with $\Delta C_1 = C_1$ as the base case. This design can greatly reduce the computational needs for answering top-k relative coverage queries.

Algorithm 6 describes the pseudo-code of our proposed algorithm for answering top-k relative coverage queries. We use *Ans* to denote the top-k sequence which is the answer to the query $Q = (G, s, k)$. $RC[\,]$ stores the relative coverage values of vertices. The algorithm conducts the iterations in Lines 4-9. At each iteration, the relative coverage value of each newly added vertex in the candidate set (i.e., ΔC_i) is computed, and then the vertex $x \in C_i$ with the highest relative coverage value is picked and added into *Ans* (in Lines 5-8). The iterations terminate when the top-k vertices are selected.

Example 4.3.4. *Consider the graph depicted in Figure 4.2 and a top-k relative coverage query* $Q = (G, 5, 4)$ *(i.e.* $s = 5$ *and* $k = 4$*) on the graph. We have* $C_1 = \{2, 11, 14\}$ *in Figure 4.2(a). Since vertex* 2 *has the largest relative coverage w.r.t vertex* 5 *in* C_1*, vertex 2 is selected, i.e.,* $x_1 = 2$*. Then* $C_2 = C_1 \backslash \{2\} \cup \{1, 4\}$ *since* $Pred_5(1) = Pred_5(4) = \{2\}$*, as shown in Figure 4.2(b), vertex* 12 *is not in* C_2 *because vertex* 14 *in* $Pred_5(12)$ *is not previously selected. Then we pick vertex* 1 *as* x_2*. Similarly,* $C_3 = C_2 \backslash \{1\} \cup \{10\}$ *as shown in Figure 4.2(c), and we pick vertex* 11 *as* x_3*. Finally, we have* $C_4 = C_3 \backslash \{11\}$ *in Figure 4.2(d) and select vertex* 6 *for* x_4*. Hence, the answer for this query* $Q = (G, 5, 4)$ *is* $(2, 1, 11, 6)$.

Complexity analysis. The time complexity of Algorithm 5 for computing the relative coverage of one vertex is $O(|V| + |E|)$. Accordingly, the time complexity of Algorithm 6 for answering a top-k relative coverage query is $O(|V|^2 + |V||E| + |V|log|V|)$ in the

Algorithm 6: RC-Query

Input: G, s, k.
Output: Ans.

```
1 Ans ← an empty list;
2 RC[v] ← 0 for each v ∈ V;
3 i ← 1;
4 while i ≤ k do
5     for each u ∈ ΔC_i do
6         RC[u] ← Single-RC(G,s,u);
7     x ← arg max_{u∈C_i} RC[u];
8     Ans.add(x);
9     i ← i + 1;
```

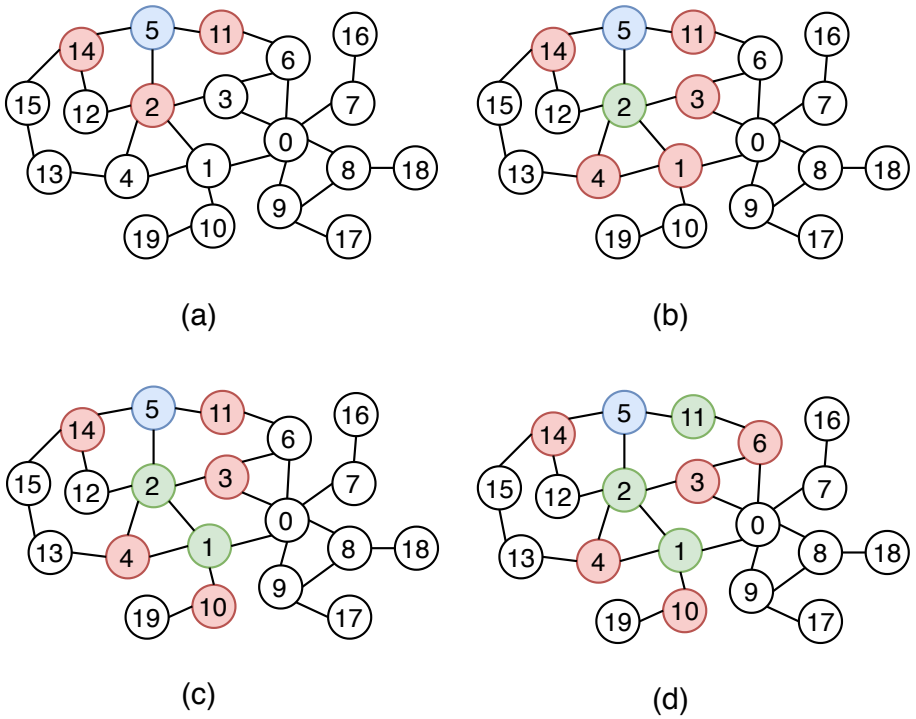


Figure 4.2: An illustration of how to answer a top-k relative coverage query $Q = (G, 5, 4)$. Red vertices are in the candidate set. Green vertices are with the largest relative coverage among all vertices in the previous candidate set.

worst case.

4.4 Optimization

Although candidate sets help avoid unnecessary computation on relative coverage of vertices, the size of a candidate set may be large, particularly in complex networks whose vertices are densely connected. Therefore, in this section, we present an optimization technique to speed up the computation of relative coverage. Inspired by the bit-parallel BFSs in [Akiba et al., 2013], we compute relative coverage of multiple vertices (i.e., at most 64 vertices) simultaneously by traversing all vertices in a graph once while performing bit-wise operations on bit-vectors.

Let ΔC_i be a set of newly added candidate vertices at the i-th iteration. We partition ΔC_i into subsets of size b, i.e., $\Delta C_i = \bigcup_{j=1}^{n} U_j$, where $b \leq 64$. Then, for each subset U_j, we compute $RC(u|s)$ for $u \in U_j$ by maintaining an array I_j such that:

$$I_j[v] = \{u \in U_j | u \in P_{sv}\}. \tag{4.5}$$

In other words, $u \in I_j[v]$ indicates that v is in the cover set of a candidate vertex u. We represent the value of each $I_j[v]$ by a bit vector with b bits, where each bit indicates the inclusion (i.e., "1") or exclusion (i.e., "0") of v in the cover set of a candidate vertex $u \in U_j$ since $|U_j| = b$.

The pseudo code for bit-paralleling the computation of relative coverage is described in Algorithm 7. For each subset U_j in the partition of ΔC_i, $I_j[v]$ for each $v \in V$ is initialized. Then $I_j[u]$ for each $u \in U_j$ is assigned with the value $\{u\}$. As in Algorithms 5 and 6, $RC[]$ stores the relative coverage values of vertices w.r.t. s, $d_s[]$ and $Pred_s[]$ denote the distances between vertices and s and predecessors of vertices w.r.t. s, respectively, and $depth$ indicates the levels in the BFS rooted at s. We traverse vertices according to $d_s[]$ in ascending order. Notice that for a vertex v' with $d_s[v'] \leq d_s[u]$ for all $u \in U_j$, $I[v']$ is guaranteed to be empty. Thus, in this algorithm we traverse from the level $min_{u \in U_j} d_s[u]$ (Line 5). During level-by-level traverse, for each vertex v, we add vertices in $I_j[v]$ into $I_j[w]$ where w is the successor of v w.r.t s (Lines 9-10). When the traversal is finished, we check whether $u \in I_j[v]$ for each $v \in V$ to obtain the relative coverage value for each $u \in U_j$ w.r.t. s (Lines 12-13). Note that $d_s[]$ and $Succ_s[]$ in Algorithm 7 only need to be computed once by traversing a BFS rooted at s.

Complexity analysis. Since $I_j[v]$ for each $v \in V$ can be represented using a 64-bit vector, and accordingly a set union operation between $I_j[v]$ and $I_j[w]$ (Line 10 of Algorithm 7) can be computed in $O(1)$ time by bit-wise OR and AND on bit vectors, the time complexity and space complexity of Algorithm 7 are $O(|E| + b|V|)$ and $O(|V|)$, respectively.

Algorithm 7: BitParallel-RC

Input: G, s, U_j.
Output: $RC[u]$ for $u \in U_j$.

$I_j[v] \leftarrow \emptyset$ for each $v \in V$;
for *each* $u \in U_j$ **do**
  $I_j[u] \leftarrow \{u\}$;
  $RC[u] \leftarrow 0$;
$depth \leftarrow min_{u \in U_j} d_s[u]$;
$d_{max} \leftarrow max_{v \in V} d_s[v]$;
while $depth < d_{max}$ **do**
  for *each* v *s.t.* $d_s[v] = depth$ **do**
    for *each* $w \in Succ_s[v]$ **do**
      $I_j[w] \leftarrow I_j[v] \cup I_j[w]$;
  $depth \leftarrow depth + 1$;
for *each* $v \in V$ **do**
  $RC[u] \leftarrow RC[u] + 1$ for each $u \in U_j$;

4.5 Experiments

We have conducted experiments to evaluate our proposed method, aiming to answer the following questions:

(Q1) How efficiently can our proposed method in Section 4.3 answer top-k relative coverage queries?

(Q2) How effectively can our optimization method in Section 4.4 improve the performance of our proposed method?

(Q3) How does k affect the sizes of candidate sets and query performance?

(Q4) How does the performance of our proposed method vary across different types of networks?

4.5.1 Experimental Setup

We implement all algorithms in C++ 11 and compile them using g++. All experiments are conducted on a Linux sever which has Intel Xeon W-2175 with 2.5GHz and 512GB of main memory.

Datasets. Table 4.2 shows the details of 6 real-world datasets we use in our experiments, among which the largest dataset YouTube has over 1.13 million vertices and 2.99 million edges. All of the graphs in these datasets are unweighted. We treat these graphs as being undirected by ignoring the direction of each edge and as being connected by only using the largest connected component as G. For each graph,

(a)

Dataset	Network Type	Source
EuroRoad	road network	[Rossi and Ahmed, 2015]
Facebook	social network	[Leskovec and McAuley, 2012]
CondMat	collaboration network	[Leskovec et al., 2007]
USRoad	road network	[Rossi and Ahmed, 2015]
EmailEu	communication network	[Leskovec et al., 2007]
YouTube	social network	[Yang and Leskovec, 2015]

(b)

Dataset	|V|	|E|	avg.deg	max.deg	avg.dist	diameter
EuroRoad	1.04K	1.31K	2.5	10	18.7	62
Facebook	4.04K	88.23K	43.7	1,045	43.7	8
CondMat	21.36K	91.31K	8.5	280	5.3	15
USRoad	126.15K	161.95K	2.6	7	223.8	617
EmailEu	224.83K	340.36K	3.0	7,636	4.1	14
YouTube	1.13M	2.99M	5.3	28,754	5.3	24

Table 4.2: Details of datasets.

we randomly select 1,000 vertex pairs to compute its average distance which is also presented in Table 4.2.

Queries. For each dataset, we randomly sample 500 vertices to compute top-k relative coverage queries. We choose $k \in \{5, 10, 15, 20\}$ in our experiments for answering the question Q3, and set $k = 10$ in the other experiments.

Methods. We compare the following methods in our experiments: (1) AllRC: we perform Algorithm 5 for all vertices in a graph and then sort all vertices according to their relative coverage values to pick top-k vertices for answering a query; (2) CandRC: we perform Algorithm 6 to answer queries, which restricts the search space to candidate sets; (3) optCandRC: we perform an optimised version of Algorithm 6 which uses the bit-parallel algorithm, i.e., Algorithm 7, to speed up the computation of relative coverage (in Lines 5-6 of Algorithm 6), where $b = 64$.

4.5.2 Performance Comparison

To verify the efficiency of our proposed method, we conduct experiments to explore the sizes of candidate sets generated by our proposed method. We also compare the query times of different methods to answer the questions Q1 and Q2.

4.5.2.1 Candidate Sets

Table 4.3 presents the ratio of the average size of candidate sets to the total number of vertices in a network, i.e., $|\bigcup_{i=1}^{k} C_i|/|V|$, on all datasets. We observe that our

Dataset	$\frac{\lvert\cup_{i=1}^{k} C_i\rvert}{\lvert V\rvert}$	Average Query Time		
		AllRC	CandRC	optCandRC
EuroRoad	2.36%	42.74ms	1.30ms	0.75ms
Facebook	45.54%	5.924s	2.68s	0.038s
CondMat	1.50%	56.913s	0.939s	0.055s
USRoad	0.01%	DNF	0.248s	0.160s
EmailEu	1.37%	DNF	76.628s	4.259s
YouTube	1.74%	DNF	DNF	132.758s

Table 4.3: The average ratio of candidate sets to the total number of vertices and average query time for answering top-k relative queries, where $k = 10$ and DNF means not finishing 500 queries in 24 hours.

proposed method can significantly reduce the search space on all datasets. Generally, candidate sets contain 1% – 3% vertices for these datasets except Facebook and USRoad. Facebook has nearly half of vertices on average (i.e. 45.54%) included in a candidate set, while USRoad has only 0.01% vertices on average in a candidate set. The reason why Facebook and USRoad have such different ratios from the other datasets is due to their different graphs structures. Facebook is a dense graph with vertices being densely connected, which is evidenced by the highest average degree (i.e. 43.7) and the lowest average distance (i.e. 3.7) in comparison with the others, as depicted in Table 4.2. On the contrary, USRoad is a sparse graph with a low average degree (i.e. 2.6) and the highest average distance (i.e. 223.8) among all the datasets.

4.5.2.2 Query Time

Table 4.3 also shows the average query time for answering top-k relative queries. We can see the following: (1) optCandRC is the fastest method and CandRC is faster than AllRC. Compared to AllRC, CandRC can answer queries over 40 times faster except for Facebook in which CandRC is only around 2 times faster. This is because that, CandRC only computes relative coverage of vertices in candidate sets while AllRC computes relative coverage of all vertices in a graph. Moreover, optCandRC is around 2-70 times faster than CandRC due to the fact that optCandRC speeds up the computation of CandRC by bit-paralleling the computation of relative coverage for at most 64 vertices a time, thereby reducing repeated vertex traversal. (2) Neither AllRC nor CandRC scale to large-scale networks; however, optCandRC can scale to networks with millions of vertices and edges. Specifically, AllRC and CandRC fail to answer queries for 3 datasets and 1 dataset among all 6 datasets, respectively. In contrast, optCandRC can answer queries in less than 3 minutes on average for large-scale networks such as YouTube, and such an average query time (i.e., 3 minutes) is reasonably acceptable for real-world applications.

4.5.3 Performance under Varying k

To answer the question Q3, we conduct experiments under different k to compare the query time of CandRC and optCandRC, and discuss how the size of candidate sets changes under different k.

4.5.3.1 Candidate Sets

The average sizes of candidate sets $|\bigcup_{i=1}^{k} C_i|$ under different k are shown in Figure 4.3. We have the following observations. (1) It shows that the average size of candidate sets grows almost linearly on all datasets. This indicates the scalability of our proposed method in terms of k. For example, the size of candidate sets grows fastest on the largest dataset YouTube; nonetheless, only less than 3% of vertices are included in the candidate set even when k increases to 20. (2) We also confirm that a network with a larger average degree and a larger maximum degree often lead to a larger candidate set under the same k. For example, the average size of candidate sets of Facebook is larger than the average sizes of candidate sets of CondMat and USRoad because the average degree of Facebook is larger, although CondMat and USRoad have larger numbers of vertices than Facebook.

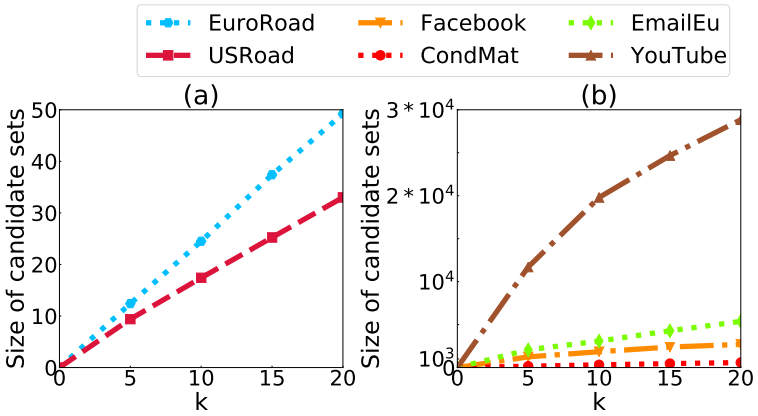


Figure 4.3: The average size of candidate sets under different k on all datasets.

4.5.3.2 Query Time

The average query times under different k on all datasets are illustrated in Figure 4.4. (1) The average query times of both CandRC and optCandRC grow almost linearly. This is because their average sizes of candidate sets grow almost linearly. (2) The faster the average size of candidate sets grows, the bigger performance gap optCandRC and CandRC may have. This is because optCandRC can not display the superiority of its bit-parallel computation when the number of vertices newly added into a candidate set in an iteration is small. For example, optCandRC improves the query time of CandRC over 10 times on CondMat, but only improves around 2 times on USRoad.

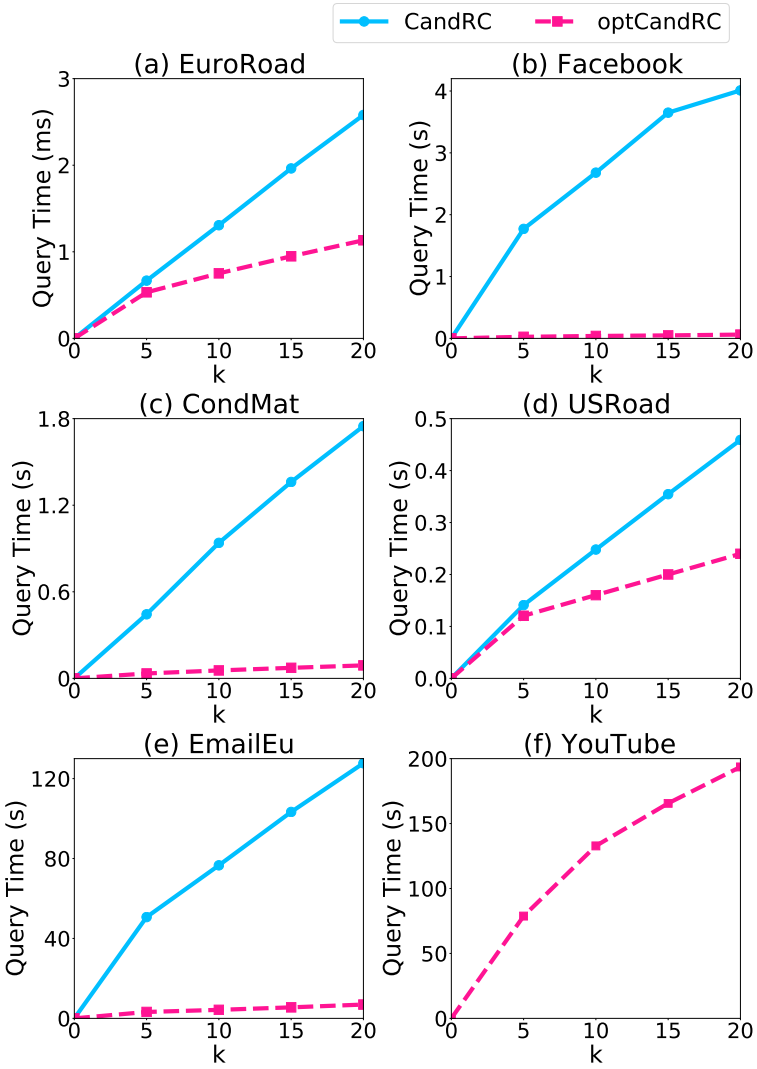


Figure 4.4: Comparison of the average query time for CandRC and optCandRC under different k on all datasets. The average query time for CandRC on YouTube is not shown in (f) since 500 queries cannot be finished in 24 hours.

4.5.4 Analysis on Network Types

We analyse how CandRC and optCandRC perform across different types of networks to answer the question Q4.

In Table 4.2, two road networks EURoad and USRoad distinguish themselves from the other networks (social, collaboration and communication networks) which are generally referred to as complex networks in the literature [Akiba et al., 2013]. We can see that, the maximum degrees of road networks are significantly (indeed orders of magnitude) smaller than the maximum degrees of complex networks, and the average distances of road networks are much larger than the average distances of complex networks. These indicate that vertices in complex networks are more densely connected than road networks.

In our proposed method CandRC, newly added vertices in ΔC_i are restricted to neighboring vertices of the most recently selected vertex x_{i-1}, i.e., the $i-1$-th vertex in the top-k sequence. Thus, $|\Delta C_i|$ must be not greater than the maximum degree in a dataset. Moreover, vertices that has shorter distances to the given vertex s in a query are more likely to be in the top-k sequence. Therefore, for the road networks EuroRoad and USRoad, their average sizes of candidate sets are order of magnitude smaller than the ones for the other networks under the same value of k and also grow slowly since they do not have high-degree vertices, i.e., the maximum degrees in EuroRoad and USRoad are 10 and 7, respectively, as shown in Table 4.2. As a result, as depicted in Figure 4.4, EuroRoad and USRoad have a smaller performance gap between CandRC and optCandRC in comparison with the other networks.

4.6 Conclusions

We have defined the relative coverage w.r.t a source vertex s – to quantify the importance of a vertex v via the number of vertices that have the vertex v lying on their shortest paths to the source vertex s. Then, we have studied the top-k relative coverage problem by proposing a method to answer top-k relative coverage queries efficiently, and further developing an optimization method using bit-parallel technique. We have also conducted experiments on 6 real-world datasets to evaluate the performance of our proposed methods.

Chapter 5

Conclusion and Future Work

In this thesis, we studied two novel problems which are related to shortest paths on large-scale networks. Specifically, we solved the shortest path graph problem on complex networks and top-k relative coverage problem. In the following we summarise our conclusions for each of these problems.

5.1 Shortest Path Graph

In Chapter 3, we aimed to understand how two vertices are connected in a graph. We studied a novel problem, the shortest path graph problem, which finds a subgraph that contains exactly all shortest paths between two vertices. We proposed a scalable method for answering shortest-path-graph queries, called Query-by-Sketch (QbS). It is a hybrid method which efficiently combines labelling and searching to handle shortest-path-graph queries on very large graphs. This method consists of three phases: labelling, sketching and searching. We proved the correctness of QbS and discussed its complexities. We conducted experiments on 12 real-world datasets to verified its efficiency and scalability.

Previously, the point-to-point shortest-path problem which finds one shortest path or shortest distance between two vertices has been widely studied. But vertices with same distance may be connected by different shortest path structures. Compared with one shortest path or distance which has limited usability, the shortest path graph shows the important topological structure between two vertices in detail, and can serve as a basis for solving problems related to shortest paths, e.g., identifying important vertices or edges.

5.2 Top-k Relative Coverage

In Chapter 4, we aimed to understand how one vertex is connected with other vertices in a graph. We extended the coverage centrality to relative coverage, to measure the importance of a vertex w.r.t a given source vertex. Then we studied a novel problem, the top-k relative coverage, which finds k vertices that best “cover" the shortest paths between a specific source vertex and all other vertices in the graph. We introduced a method for efficiently answering top-k relative coverage on large-scale net-

works, which only requires to compute relative coverage in a reduced search space. We further designed a bit-parallel optimization method to accelerate the computation of relative coverage. We theoretically analyse the complexity of our methods and experimentally verify their efficiency through experiments on 6 real-world networks.

When a graph is large, enumerating all shortest paths from a source vertex is less informative since the number of paths is excessive and these paths are highly similar, the top-k relative coverage problem provides us a way to efficiently identify influential vertices based on shortest paths.

5.3 Future Work

One of the interesting directions for future research is to study shortest path graph problem on road networks. Previously, a large number of methods have been proposed for finding point-to-point shortest paths on road network. However, one shortest path is insufficient for analysing how two vertices are connected. For example, one may be interested in the safest route such that the traffic is smooth, and picking one shortest path between two locations at random may lead him to a busy district in which traffic jam happens. The shortest path graph gives more choices in rerouting, or detecting the hub of transportation.

Another direction is to extend the top-k relative coverage to identify the most influential vertices w.r.t a set of vertices, instead of just one source vertex. It has a wider range of applications than the top-k relative coverage problem. For example, the transportation authority may be interested in strengthening the connection between several public places.

Bibliography

Abraham, I.; Delling, D.; Goldberg, A. V.; and Werneck, R. F., 2012. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*, 24–35. (cited on page 2)

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*, 782–793. (cited on pages 2 and 9)

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*, 349–360. (cited on pages 2, 3, 4, 9, 10, 13, 17, 18, 19, 30, 45, and 51)

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*, 144–155. (cited on page 9)

Bast, H.; Funke, S.; Matijevic, D.; Sanders, P.; and Schultes, D., 2007. In transit to constant time shortest-path queries in road networks. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, 46–59. (cited on pages 2 and 9)

Bauer, R. and Delling, D., 2010. Sharc: Fast and robust unidirectional routing. *Journal of Experimental Algorithmics (JEA)*, 14 (2010), 2–4. (cited on page 9)

Bavelas, A., 1950. Communication patterns in task-oriented groups. *The journal of the acoustical society of America*, 22, 6 (1950), 725–730. (cited on pages 4, 10, and 39)

Bonsma, P., 2013. The complexity of rerouting shortest paths. *Theoretical computer science*, 510 (2013), 1–12. (cited on page 3)

Chehreghani, M. H.; Bifet, A.; and Abdessalem, T., 2018. Novel adaptive algorithms for estimating betweenness, coverage and k-path centralities. *arXiv preprint arXiv:1810.10094*, (2018). (cited on page 11)

Cohen, E.; Halperin, E.; Kaplan, H.; and Zwick, U., 2003. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32, 5 (2003), 1338–1355. (cited on pages 3 and 15)

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C., 2009. *Introduction to algorithms*. MIT press. (cited on pages 3 and 7)

Cowen, L. J. and Wagner, C. G., 2004. Compact roundtrip routing in directed networks. *Journal of Algorithms*, 50, 1 (2004), 79–95. (cited on page 2)

Dijkstra, E. W. et al., 1959. A note on two problems in connexion with graphs. *Numerische mathematik*, 1, 1 (1959), 269–271. (cited on pages 3, 7, and 13)

D'Angelo, G.; Olsen, M.; and Severini, L., 2019. Coverage centrality maximization in undirected networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 501–508. (cited on page 11)

Estrada, E., 2006. Virtual identification of essential proteins within the protein interaction network of yeast. *Proteomics*, 6, 1 (2006), 35–40. (cited on page 1)

Farhan, M.; Wang, Q.; Lin, Y.; and Mckay, B., 2019. A highly scalable labelling approach for exact distance queries in complex networks. In *Proceedings of the 22th International Conference on Extending Database Technology*. (cited on pages 4 and 9)

Fredman, M. L. and Tarjan, R. E., 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34, 3 (1987), 596–615. (cited on page 7)

Freeman, L. C., 1977. A set of measures of centrality based on betweenness. *Sociometry*, (1977), 35–41. (cited on pages 4, 10, and 39)

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. *Proceedings of the VLDB Endowment*, 6, 6 (2013), 457–468. (cited on pages 2, 9, and 10)

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D., 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, 319–333. Springer. (cited on page 9)

Goldberg, A. V., 2007. Point-to-point shortest path algorithms with preprocessing. In *International Conference on Current Trends in Theory and Practice of Computer Science*, 88–102. (cited on pages 2 and 8)

Goldberg, A. V. and Harrelson, C., 2005. Computing the shortest path: A* search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, 156–165. (cited on pages 1, 2, 8, 9, 24, and 30)

Goldberg, A. V.; Kaplan, H.; and Werneck, R. F., 2006. Reach for A*: Efficient point-to-point shortest path algorithms. In *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments*, 129–143. (cited on pages 2 and 9)

Gubichev, A.; Bedathur, S.; Seufert, S.; and Weikum, G., 2010. Fast and accurate estimation of shortest paths in large graphs. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, 499–508. (cited on page 10)

Gutman, R. J., 2004. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. *ALENEX/ANALC*, 4 (2004), 100–111. (cited on pages 8 and 9)

Hansen, P.; Thisse, J.-F.; and Wendell, R. E., 1986. Efficient points on a network. *Networks*, 16, 4 (1986), 357–368. (cited on page 3)

Hart, P. E.; Nilsson, N. J.; and Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4, 2 (1968), 100–107. (cited on page 8)

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*, 1533–1542. (cited on pages 9 and 26)

Israeli, E. and Wood, R. K., 2002. Shortest-path network interdiction. *Networks: An International Journal*, 40, 2 (2002), 97–111. (cited on page 3)

Jin, R.; Ruan, N.; You, B.; and Wang, H., 2013. Hub-accelerator: Fast and exact shortest path computation in large social networks. *arXiv preprint arXiv:1305.0507*, (2013). (cited on page 8)

Kamiński, M.; Medvedev, P.; and Milanič, M., 2011. Shortest paths between shortest paths. *Theoretical Computer Science*, 412, 39 (2011), 5205–5210. (cited on page 3)

Khachiyan, L.; Boros, E.; Borys, K.; Elbassioni, K.; Gurvich, V.; Rudolf, G.; and Zhao, J., 2008. On short paths interdiction problems: Total and node-wise limited interdiction. *Theory of Computing Systems*, 43, 2 (2008), 204–233. (cited on page 3)

Kolaczyk, E. D.; Chua, D. B.; and Barthélemy, M., 2009. Group betweenness and co-betweenness: Inter-related notions of coalition centrality. *Social Networks*, 31, 3 (2009), 190–203. (cited on page 2)

Kunegis, J., 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, 1343–1350. (cited on page 29)

Labbé, M.; Peeters, D.; and Thisse, J.-F., 1995. Location on networks. *Handbooks in operations research and management science*, 8 (1995), 551–624. (cited on page 3)

Lauther, U., 2004. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. *Geoinformation und Mobilität-von der Forschung zur praktischen Anwendung*, 22 (2004), 219–230. (cited on page 8)

Leskovec, J.; Kleinberg, J.; and Faloutsos, C., 2007. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1, 1 (2007), 2–es. (cited on page 47)

Leskovec, J. and Krevl, A., 2014. Snap datasets: Stanford large network dataset collection. (cited on page 29)

Leskovec, J. and McAuley, J., 2012. Learning to discover social circles in ego networks. *Advances in neural information processing systems*, 25 (2012), 539–547. (cited on page 47)

Li, Y.; U, L. H.; Yiu, M. L.; and Kou, N. M., 2017. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11, 4 (2017), 445–457. (cited on page 8)

Medya, S.; Bogdanov, P.; and Singh, A., 2018a. Making a small world smaller: Path optimization in networks. *IEEE Transactions on Knowledge and Data Engineering*, 30, 8 (2018), 1533–1546. (cited on page 4)

Medya, S.; Silva, A.; Singh, A.; Basu, P.; and Swami, A., 2017. Maximizing coverage centrality via network design: Extended version. *arXiv preprint arXiv:1702.04082*, (2017). (cited on page 11)

Medya, S.; Silva, A.; Singh, A.; Basu, P.; and Swami, A., 2018b. Group centrality maximization via network design. In *Proceedings of the 2018 SIAM International Conference on Data Mining*, 126–134. SIAM. (cited on page 11)

Nishimura, N., 2018. Introduction to reconfiguration. *Algorithms*, 11, 4 (2018), 52. (cited on page 3)

Opsahl, T.; Agneessens, F.; and Skvoretz, J., 2010. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social networks*, 32, 3 (2010), 245–251. (cited on page 2)

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*, 867–876. (cited on page 31)

Rossi, R. A. and Ahmed, N. K., 2015. The network data repository with interactive graph analytics and visualization. In *AAAI*. http://networkrepository.com. (cited on page 47)

Sanders, P. and Schultes, D., 2005. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms*, 568–579. (cited on pages 2 and 9)

Sankaranarayanan, J.; Samet, H.; and Alborzi, H., 2009. Path oracles for spatial networks. *Proceedings of the VLDB Endowment*, 2, 1 (2009), 1210–1221. (cited on page 2)

Sommer, C., 2014. Shortest-path queries in static networks. *ACM Computing Surveys*, 46, 4 (2014), 45. (cited on page 8)

Takaguchi, T.; Yano, Y.; and Yoshida, Y., 2016. Coverage centralities for temporal networks. *The European Physical Journal B*, 89, 2 (2016), 1–11. (cited on page 11)

Thorup, M., 1999. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46, 3 (1999), 362–394. (cited on page 7)

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*, 1785–1794. (cited on page 10)

Wagner, D. and Willhalm, T., 2007. Speed-up techniques for shortest-path computations. In *Annual Symposium on Theoretical Aspects of Computer Science*, 23–36. (cited on pages 2 and 8)

Wei, F., 2010. TEDI: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 99–110. (cited on page 9)

Wu, L.; Xiao, X.; Deng, D.; Cong, G.; Zhu, A. D.; and Zhou, S., 2012. Shortest path and distance queries on road networks: An experimental evaluation. *Proceedings of the VLDB Endowment*, 5, 5 (2012), 406–417. (cited on pages 2 and 8)

Xiao, Y.; Wu, W.; Pei, J.; Wang, W.; and He, Z., 2009. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 493–504. (cited on page 9)

Yang, J. and Leskovec, J., 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42, 1 (2015), 181–213. (cited on page 47)

Yao, B.; Li, F.; and Xiao, X., 2013. Secure nearest neighbor revisited. In *2013 IEEE 29th International Conference on Data Engineering*, 733–744. (cited on page 2)

Yoshida, Y., 2014. Almost linear-time algorithms for adaptive betweenness centrality using hypergraph sketches. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 1416–1425. (cited on pages 4, 10, 11, 39, and 40)

Yu, Z.; Wang, C.; Bu, J.; Wang, X.; Wu, Y.; and Chen, C., 2015. Friend recommendation with content spread enhancement in social networks. *Information Sciences*, 309 (2015), 102–118. (cited on pages 1 and 5)

Zhao, X.; Sala, A.; Zheng, H.; and Zhao, B. Y., 2011. Efficient shortest paths on massive social graphs. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 77–86. (cited on page 10)