

Query-by-Sketch: Scaling Shortest Path Graph Queries on Very Large Networks

Ye Wang
School of Computing,
Australian National
University
ye.wang2@anu.edu.au

Qing Wang
School of Computing,
Australian National
University
qing.wang@anu.edu.au

Henning Koehler
School of Fundamental
Sciences, Massey
University
h.koehler@massey.ac.nz

Yu Lin
School of Computing,
Australian National
University
yu.lin@anu.edu.au

ABSTRACT

Computing shortest paths is a fundamental operation in processing graph data. In many real-world applications, discovering shortest paths between two vertices empowers us to make full use of the underlying structure to understand how vertices are related in a graph, e.g. the strength of social ties between individuals in a social network. In this paper, we study the shortest-path-graph problem that aims to efficiently compute a shortest path graph containing exactly all shortest paths between any arbitrary pair of vertices on complex networks. Our goal is to design an exact solution that can scale to 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 computational complexity. To empirically verify the efficiency of QbS, we conduct experiments on 12 real-world datasets, among which the largest 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.

CCS CONCEPTS

• Theory of computation → Shortest paths.

KEYWORDS

Shortest paths; graphs; 2-hop cover; distance labelling; pruned landmark labelling; graph sketch; breadth-first search; algorithms

1 INTRODUCTION

Graphs are typical data structures used for representing complex relationships among entities, such as friendships in social networks, connections in computer networks, and links among web pages [6, 33, 35]. Computing shortest paths between vertices is a fundamental operation in processing graph data, and has been used in many algorithms for graph analytics [24, 29, 40]. 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 [2, 5, 14–16, 31, 32, 36, 38]. By leveraging specific

properties of road networks, such as hierarchical structures and near planarity [3, 13], previous works have proposed various exact and approximate methods for answering point-to-point shortest path queries [1, 10]. Nonetheless, these methods often do not perform well on complex networks (e.g., social networks, and web graphs) because complex networks exhibit different properties from road networks, such as small diameter and local clustering [3, 13, 15]. 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.

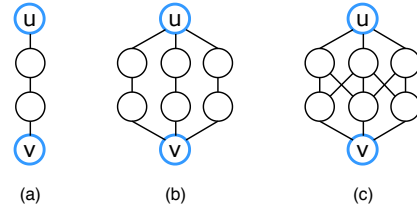


Figure 1: An illustration of shortest paths between two vertices u and v whose distance is 3: (a) one shortest path; (b) three shortest paths; (c) seven shortest paths.

Given two vertices u and v , as depicted in Figure 1(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(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 analyze how they are connected. 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. Accordingly, we term this problem as the *shortest-path-graph* problem (formally defined in Section 2).

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) [7, 22, 28], the Shortest Path Network Interdiction problem (i.e., to find critical edges and vertices whose removal can destroy all shortest paths between two vertices) [20, 23], and the variants such as the Shortest Path Common Links problem (i.e., to find links common

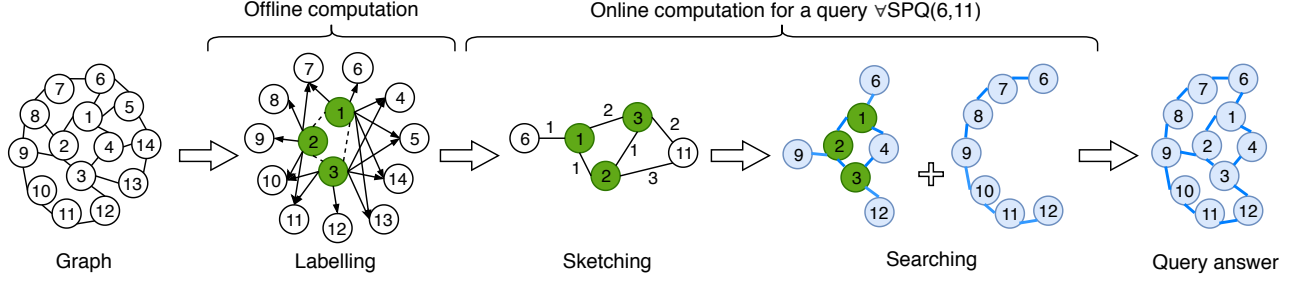


Figure 2: An illustration of our method Query-by-Sketch (QbS) for answering all shortest path queries.

to all shortest paths between two vertices) [18, 26]. These shortest path related problems are motivated by a wide range of real-world applications arising in designing and analyzing 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 helps 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 [11] or performing a breadth-first search (BFS) for unweighted graphs [9]. 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 [8]. However, for large graphs, storing even just shortest path distances of all pairs of vertices is prohibitive [3] and storing all shortest paths of all pairs is hardly feasible due to the demand for much more space overhead. Thus, the question we tackle in this paper is: *How to construct labels for shortest-path-graph queries that should be of reasonable size (e.g. not much larger than the original graph), within a reasonable time (e.g. not longer than one day), and can speed up query answering as much as possible?* In answering this question, we develop an efficient solution for shortest-path-graph queries. It is worth to note that: 1) we do not enumerate all shortest paths to produce a shortest path graph that contains *exactly all shortest paths* between two vertices; 2) our proposed solution can answer shortest-path-graph queries very efficiently, in microseconds for graphs with millions of edges and in less than half a second for graphs with billions of edges.

Contributions. In the following, we summarize the contributions of this paper with the key technical details:

(1) We observe that 2-hop distance cover is inadequate for labelling required by shortest-path-graph queries. To alleviate this limitation and achieve high scalability, we propose a scalable method for answering shortest-path-graph queries, called *Query-by-Sketch* (QbS). This method consists of three phases, as illustrated in Figure 2: (a) *labelling* - constructing a labelling scheme, which is compact and of a small size, using a small number of landmarks through pre-computation, (b) *sketching* - using labelling to efficiently compute a *sketch* that summarizes the important structure of shortest paths in a query answer, 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.

(2) We theoretically prove the correctness of our method *QbS*. In addition to this, we conduct the complexity analysis for *QbS* through analysing the time complexities of the algorithms for constructing a labelling scheme, computing a sketch, and performing a guided search for answering queries. We also prove that our labelling scheme is deterministic w.r.t. landmarks. This enables us to leverage the thread-level parallelism by performing BFSs from different landmarks simultaneously without considering an order of landmarks, which improves the efficiency of labelling construction and thus achieves better scalability.

(3) We have conducted experiments on 12 real-world datasets, among which the largest dataset ClueWeb09 has 1.7 billion vertices and 7.8 billion edges. It is shown that *QbS* has significantly better scalability than the baseline methods. The labelling construction of *QbS* can be parallelized, which takes 10 seconds for datasets with millions of edges and half an hour for the largest dataset ClueWeb09. The labelling sizes constructed by *QbS* are generally smaller than the original sizes of graphs. Further, *QbS* can answer queries much faster than the other methods. For graphs with billions of edges, it takes only around 0.01 - 0.5 seconds to answer a query.

2 PRELIMINARIES

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* $L(v) = \{(r_1, \delta_{vr_1}), \dots, (r_n, \delta_{vr_n})\}$, 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 $\text{size}(L) = \sum_{v \in V} |L(v)|$. 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.* [8] proposed *2-hop distance cover*, which has been widely used in labelling-based approaches for distance queries.

DEFINITION 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:

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

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 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 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 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.

3.1 2-Hop Path Cover

Originally, 2-hop distance cover was proposed for reachability and distance queries [8]. Below, we discuss why it is insufficient for shortest-path-graph queries.

EXAMPLE 3.1. Consider a query $SPG(3, 7)$ on a graph G depicted in Figure 3 (a). The query answer is colored in green. In Figure 3(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 3.2. [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 $p \in P_{uv}^G$ with $p \neq (u, v)$, the following holds:

$$d_G(u, v) = \min\{\delta_{ur} + \delta_{vr} \mid (r, \delta_{ur}) \in L(u), (r, \delta_{vr}) \in L(v), r \in V(p) \setminus \{u, v\}\}, \quad (1)$$

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 .

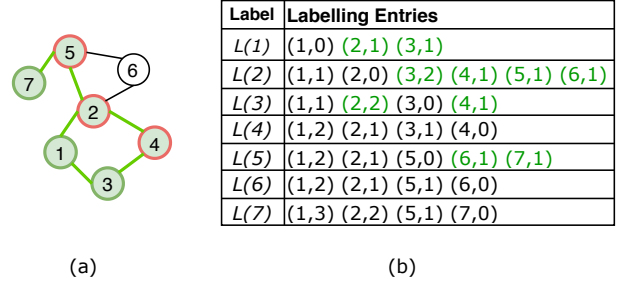


Figure 3: (a) A graph G in which the answer of $\text{VSPQ}(3, 7)$ is colored in green; (b) Labels over G , where labels for a 2-hop distance cover are colored in black and additional labels from a 2-hop path cover are colored in green.

EXAMPLE 3.3. Consider Figure 3 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 $SPG(3, 7)$.

3.2 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. $L(v) = \{(u, \delta_{vu}) \mid u \in V\}$, 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) [3], 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) [3], 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 $[v_1, v_2, \dots, v_{|V|}]$ 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 $\text{depth}[v]$ 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 $[v_1, \dots, v_{k-1}]$, 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 $d_{L_{k-1}}(v_k, u) < \text{depth}[u]$, the label $(v_k, \text{depth}[u])$ 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 $d_{L_{k-1}}(v_k, u) \geq \text{depth}[u]$ (Line 8). Note that, unlike PLL, in the case of $d_{L_{k-1}}(v_k, u) = \text{depth}[u]$, the label $(v_k, \text{depth}[u])$ cannot be pruned in PPL; otherwise, 2-hop path cover is not guaranteed, i.e., not all shortest

paths are covered by labels. When $d_{L_{k-1}}(v_k, u) \leq \text{depth}[u]$, 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).

Algorithm 1: PrunedBFS

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

```

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

To answer a query $\text{SPG}(u, v)$, 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 $V_{uv} = \{r \mid r = \min(\delta_{ur} + \delta_{vr}), (r, \delta_{ur}) \in L(u), (r, \delta_{vr}) \in L(v)\}$. 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 $\forall \text{SPQ}(u, v)$ 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 3.4. When using PPL to answer the query $\forall \text{SPQ}(3, 7)$ on the graph G in Figure 3(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 $V_{7,1} = \{2\}$, 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 BFSs, PPL can 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 more than one shortest path exists between 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 in Example 3.4.

Path labelling with parents. One common technique to accelerate query time for shortest-path-graph queries is to keep additional parent information in labels so as to provide a clearer direction towards shortest paths. For example, Akiba *et al.* [3] 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 all shortest paths, this requires us to store all 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 store the “parent” vertices of v , rather than the “child” vertices of r_i , because landmarks often have a high degree [3]. To distinguish from PPL, we abbreviate this 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 6.

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.

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 4.1, 4.2 and 4.3, respectively. The main idea behind this method is to construct a labelling scheme through precomputation, and then answer shortest-path-graph queries by performing online computation that involves two steps: fast sketching and guided searching.

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 any pair of vertices $u, v \in V$, we can efficiently compute a sketch for answering $\text{SPG}(u, v)$.

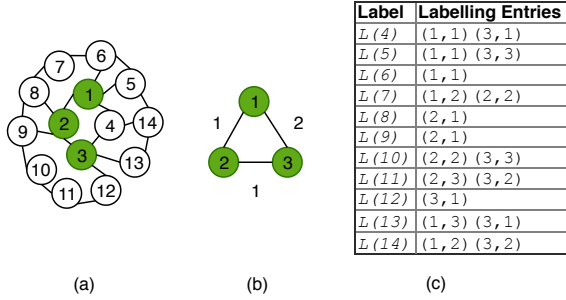


Figure 4: (a) A graph with three landmarks $\{1, 2, 3\}$ (highlighted in green), (b) a meta-graph, and (c) a path labelling.

DEFINITION 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 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 $u \in V \setminus R$ a label $L(u)$ s.t.

$$L(u) = \{(r, \delta_{ur}) \mid r \in R, \delta_{ur} = d_G(u, r), \exists p \in P_{ur}(V(p) \cap R = \{r\})\}. \quad (2)$$

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 4.3. Figure 4 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 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 4.4. Figure 5 shows how our algorithm conducts BFSs to construct labels. The BFS from landmark 1 is depicted in Figure 5(a), in which vertices $\{4, 5, 6, 7, 13, 14\}$ are labelled because the other vertices

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  $E_R \leftarrow \emptyset$ ;  $L(v) \leftarrow \emptyset$  for all  $v \in V$ 
2 for all  $r_i \in R$  do
3    $Q_L \leftarrow \emptyset$ ;  $Q_N \leftarrow \emptyset$ ;
4    $Q_L.push(r_i)$ ;
5    $depth[r_i] \leftarrow 0$ ;  $depth[v] \leftarrow \infty$  for all  $v \in V \setminus \{r_i\}$ ;
6    $n \leftarrow 0$ ;
7   while  $Q_L$  and  $Q_N$  are not empty do
8     for all  $u \in Q_L$  at depth  $n$  do
9       for all unvisited neighbors  $v$  of  $u$  do
10         $depth[v] \leftarrow n + 1$ ;
11        if  $v$  is a landmark then
12           $Q_N.push(v)$ ;
13           $E_R \leftarrow E_R \cup \{(r_i, v)\}$ ;
14           $\sigma(r_i, v) \leftarrow depth[v]$ ;
15        else
16           $Q_L.push(v)$ ;
17           $L(v) \leftarrow L(v) \cup \{(r_i, depth[v])\}$ ;
18     for all  $u \in Q_N$  at depth  $n$  do
19       for all unvisited neighbors  $v$  of  $u$  do
20         $depth[v] \leftarrow n + 1$ ;
21         $Q_N.push(v)$ ;
22    $n \leftarrow n + 1$ ;

```

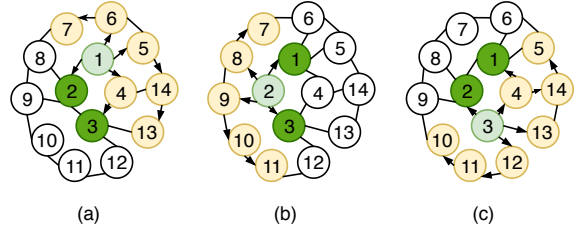


Figure 5: An illustration of labelling: (a), (b) and (c) describe the BFSs rooted at the landmarks 1, 2 and 3, respectively, where light and dark green vertices denote the landmarks, and yellow vertices denote those being labelled.

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 5(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 5(c), which works in a similar manner.

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

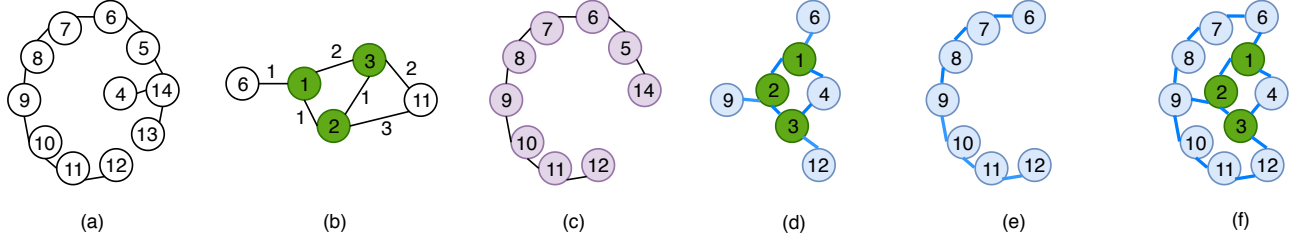


Figure 6: An illustration of sketching and searching: (a) the sparsified graph G^- of the graph G shown in Figure 4(a); (b) the sketch for $SPG(6,11)$ on the graph G ; (c) the bi-directional BFS on G^- , (d) the recover search based on \mathcal{L} , (e) the reverse search based on G^- , and (f) shows the query answer of $SPG(6,11)$.

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 on very large networks.

DEFINITION 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:

$$d_{uv}^T = \min_{(r, r')} \{ \delta_{ru} + d_M(r, r') + \delta_{rv} \mid (r, \delta_{ru}) \in L(u), (r', \delta_{rv}) \in L(v) \}; \quad (3)$$

Accordingly, we have the following corollary.

COROLLARY 4.6. $d_{uv}^T \geq d_G(u, v)$ 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_{uv}^T (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 4.7. Figure 6(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 $d_{6,11}^T = 5$, and $d_{6,11}^T = d_G(6, 11)$.

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 $G[V \setminus R]$ by removing all landmarks in R and all edges incident to these landmarks from G . $d_{G[V \setminus R]}(u, v)$ 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_{uv}^T due to the fact that $d_G(u, v) = \min(d_{G[V \setminus R]}(u, v), d_{uv}^T)$.
- S_{uv} can guide how to conduct a bi-directional search on the sparsified graph $G[V \setminus R]$. Specifically, for $t \in \{u, v\}$, we have

$$d_t^* = \max_{(r, t) \in E_S} \sigma_S(r, t) - 1, \quad (4)$$

Algorithm 3: Computing a sketch S_{uv}

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

```

1  $V_S \leftarrow \emptyset, E_S \leftarrow \emptyset;$ 
2 for all  $\{r, r'\} \subseteq R$  do
3    $\pi_{rr'} \leftarrow +\infty;$ 
4   if  $(r, \delta_{ur}) \in L(u)$  and  $(r', \delta_{vr'}) \in L(v)$  then
5      $\pi_{rr'} \leftarrow \delta_{ur} + d_M(r, r') + \delta_{vr'};$ 
6  $d_{uv}^T \leftarrow \min\{\pi_{rr'} \mid \{r, r'\} \subseteq R\};$ 
7 for all  $\{r, r'\} \subseteq R$  and  $\pi_{rr'} = d_{uv}^T$  do
8    $E_S \leftarrow E_S \cup \{(u, r), (v, r')\};$ 
9    $\sigma_S(u, r) \leftarrow \delta_{ur}, \sigma_S(v, r') \leftarrow \delta_{vr'};$ 
10  for all  $(r_i, r_j)$  in the shortest path graph of  $(r, r')$  in  $M$  do
11     $E_S \leftarrow E_S \cup \{(r_i, r_j)\};$ 
12     $\sigma_S(r_i, r_j) \leftarrow \sigma(r_i, r_j);$ 
13  $V_S \leftarrow V(E_S);$ 
```

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 $G^- = G[V \setminus R]$ and the labelling scheme \mathcal{L} , guided by the sketch S_{uv} , as follows:

$$G_{uv} = \begin{cases} G_{uv}^{\mathcal{L}} & \text{if } d_{G^-}(u, v) > d_{uv}^T; \\ G_{uv}^- \cup G_{uv}^{\mathcal{L}} & \text{if } d_{G^-}(u, v) = d_{uv}^T; \\ G_{uv}^- & \text{otherwise.} \end{cases} \quad (5)$$

We use $G_{uv}^{\mathcal{L}}$ 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 [15], under the guide of S_{uv} w.r.t. Eq. 4. This search terminates when common vertices are found or the upper bound d_{uv}^T 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 $G_{uv}^{\mathcal{L}}$. As we do not know initially which of the

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

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

```

1  $d_{uv}^\top, d_u^*, d_v^* \leftarrow \text{get\_bound}(S_{uv});$ 
2  $P_u \leftarrow \emptyset, P_v \leftarrow \emptyset, d_u \leftarrow 0, d_v \leftarrow 0;$ 
3 Enqueue  $u$  to  $Q_u$  and  $v$  to  $Q_v$ ;
4  $\text{depth}_u[w] \leftarrow \infty, \text{depth}_v[w] \leftarrow \infty$  for all  $w \in V \setminus R$ ;
5  $\text{depth}_u[u] \leftarrow 0, \text{depth}_v[v] \leftarrow 0;$ 
6 while  $d_u + d_v < d_{uv}^\top$  do
7    $t \leftarrow \text{pick\_search}(P_u, P_v, d_u^*, d_v^*, d_u, d_v);$ 
8   if  $t = u$  then
9      $Q_u \leftarrow \text{forward\_search}(Q_u);$ 
10  if  $t = v$  then
11     $Q_v \leftarrow \text{backward\_search}(Q_v);$ 
12     $P_t \leftarrow P_t \cup Q_t; d_t \leftarrow d_t + 1;$ 
13     $\text{depth}_t[w] \leftarrow d_t$  for  $w \in Q_t$ ;
14    if  $P_u \cap P_v$  is not empty then
15      break;
16 if  $P_u \cap P_v \neq \emptyset$  then
17    $G_{uv}^- \leftarrow \text{reverse\_search}(P_u \cap P_v, G^-, \text{depth}_u, \text{depth}_v);$ 
18 if  $d_u + d_v = d_{uv}^\top$  then
19    $Z \leftarrow \emptyset;$ 
20   for all  $(r, t) \in E_S$  with  $t \in \{u, v\}$  do
21      $d_m \leftarrow \min\{\sigma_S(r, t) - 1, d_t\};$ 
22     for all  $w$  with  $\text{depth}_t[w] = d_m, (r, \delta_{wr}) \in L(w),$ 
23        $\delta_{wr} + d_m = \sigma_S(r, t)$  do
24          $Z \leftarrow Z \cup \{(w, r)\};$ 
25    $G_{uv}^\mathcal{L} \leftarrow \text{recover\_search}(S_{uv}, \mathcal{L}, Z, G^-, \text{depth}_u, \text{depth}_v);$ 
26  $G_{uv} \leftarrow G_{uv}^- \cup G_{uv}^\mathcal{L};$ 

```

three cases of Eq. 5 holds, a bi-directional search is always performed. This search provides us with $d_{G^-}(u, v)$, though we abort once $d_{G^-}(u, v) > d_{uv}^\top$ can be guaranteed. Then depending on the values of $d_{G^-}(u, v)$ and d_{uv}^\top , a reverse search, a recover search, or both of them are performed to compute G_{uv}^- and $G_{uv}^\mathcal{L}$ as in Eq. 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 , 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 [19] 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 $\text{depth}_u[x] - 1 = \text{depth}_u[x']$, 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 $\text{depth}_t[w] = \min\{\sigma_S(r, t) - 1, d_t\}$ and $\sigma_S(r, t) = \delta_{wr} + \text{depth}_t[w]$ (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 $(r, \delta_{wr'}) \in L(w')$ and $\delta_{wr'} + 1 = \delta_{wr}$. The shortest paths between w and u (resp. v) is computed according to $\text{depth}_u[\]$ (resp. $\text{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 4.8. Figure 6(c)-(e) illustrates how our guided searching finds the answer for a query $\text{SPG}(6, 11)$. The sparsified graph G^- is depicted in Figure 6(a) and the sketch is depicted in Figure 6(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 6(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 6(e), which goes back to 6 and 11 from $P_6 \cap P_{11} = \{8\}$. The recover search is depicted in Figure 6(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 6(f).

5 THEORETICAL DISCUSSION

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

5.1 Proof of Correctness

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

THEOREM 5.1. *Given any query $\text{SPG}(u, v)$ 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 4.2. Suppose that we conduct a BFS rooted from $r \in R$. Given a landmark $r' \in R \setminus \{r\}$, if $\exists p \in P_{rr'}(V(p) \cap R = \{r, r'\})$ holds, there must exist $w \in Q_L$ with $\text{depth}[w] + 1 = \text{depth}[r']$ and $(w, r') \in E$ (Lines 8-9, 11), and accordingly an edge (r, r') is added into M (Lines 13-14). Otherwise,

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

Table 1: 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.

r' is directly pushed into Q_N (Lines 19-21). Given a vertex $v \in V \setminus R$ 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 $\text{depth}[w] + 1 = \text{depth}[v]$ and $(w, v) \in E$ (Lines 8-9, 15), and accordingly a label $(r, \text{depth}[v])$ 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 4.5. First, Algorithm 3 (Lines 2-7) finds pairs of landmarks (r, r') that minimise $\{\delta_{ur} + d_M(r, r') + \delta_{r'v} \mid (r, \delta_{ur}) \in L(u) \text{ and } (r', \delta_{r'v}) \in L(v)\}$ (i.e., satisfying Eq. (3) in Definition 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). \square

5.2 Complexity Analysis

The time complexity of constructing a BFS from one landmark in Algorithm 2 is $O(|E|)$ and the overall time complexity of Algorithm 2 is $O(|R||E|)$. 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 time complexity of conducting a guided search in Algorithm 4 is $O(|E| + |R||V|)$.

Note that, in our work, the number of landmarks is small, i.e., $|R| = 20$ by default, which is much smaller than the number of vertices or edges in the original graph. Thus, we can see that, constructing a labelling 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 .

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 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 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 $R \setminus \{r, r'\}$. From Definition 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 $R \setminus \{r\}$. Therefore, \mathcal{L} is deterministic w.r.t G and R . \square

For a fixed set of landmarks, the labelling construction in Algorithm 2 yields the same labelling scheme, regardless of the ordering of landmarks. This deterministic nature of labelling scheme enables us to speed up the construction of labelling scheme by parallelising 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.

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 such as construction time, labelling size and query time?

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.

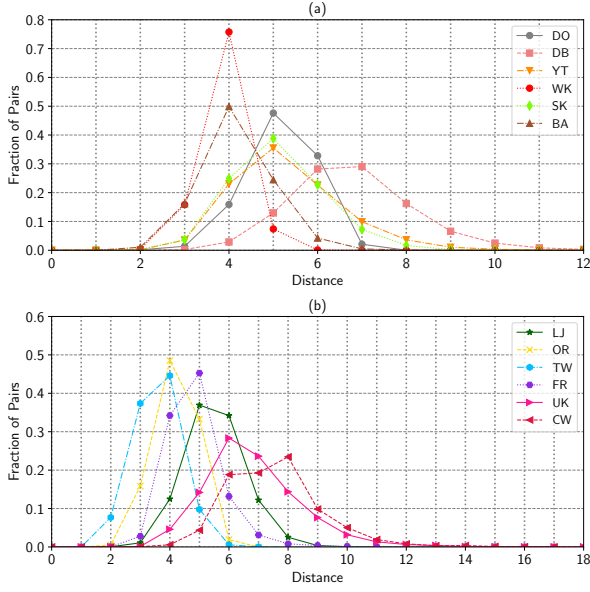


Figure 7: Distance distribution of 10,000 randomly selected pairs of vertices on all the datasets.

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 1 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 [25], Stanford Networks Analysis Project [27], Dynamically Evolving Large-scale Information Systems Project¹ and the Lemur Project².

Queries. We randomly sampled 10,000 pairs of vertices from all pairs of vertices in each graph to evaluate the average query time. Figure 7 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 [3]. We thus use the methods *Pruned Path Labelling* (PPL) and *Pruned Path Labelling with Parent information* (ParentPPL) as discussed in Section 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 [15]. We denote it as Bi-BFS.

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 $|R| \times 8$ 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 [30]. We set $|R| = 20$ in QbS by default.

6.2 Performance Comparison

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

6.2.1 Construction Time. Table 2 shows that our method QbS can efficiently construct a labelling 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.

6.2.2 Labelling Size. Table 3 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 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) are small in QbS, compared with the sizes of labelling (i.e. $size(\mathcal{L})$ in Table 3). 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.

6.2.3 Query Time. Table 2 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

¹See <http://law.di.unimi.it/datasets.php> for datasets

²See <https://lemurproject.org/clueweb09/index.php>

Dataset	Construction Time (sec.)				Average Query Time (ms.)			
	QbS-P	QbS	PPL	ParentPPL	QbS	PPL	ParentPPL	Bi-BFS
Douban	0.05	0.3	154	2,736	0.037	1.414	0.038	0.585
DBLP	0.12	1.1	2,610	11,049	0.097	1.782	0.052	2.995
Youtube	0.47	4.4	22,601	DNF	0.218	5.314	-	23.809
WikiTalk	0.61	4.9	8,662	DNF	0.693	3.536	-	6.984
Skitter	1.51	12.7	86,326	DNF	0.951	16.978	-	44.685
Baidu	2.04	18.9	DNF	OOE	0.845	-	-	174.412
LiveJournal	6.48	52.2	DNF	OOE	1.095	-	-	84.967
Orkut	10.85	73.2	DNF	OOE	4.237	-	-	207.541
Twitter	199.8	1,345	DNF	OOE	164.333	-	-	4,817.774
Friendster	416.5	2,354	DNF	OOE	11.972	-	-	3,600.362
uk2007	178.5	1,485	OOE	OOE	77.830	-	-	5,264.101
ClueWeb09	1,819	17,060	OOE	OOE	480.443	-	-	DNF

Table 2: Comparison of construction time and query time. DNF and OOE refer to running out of time (>24 hours) and running out of memory, respectively.

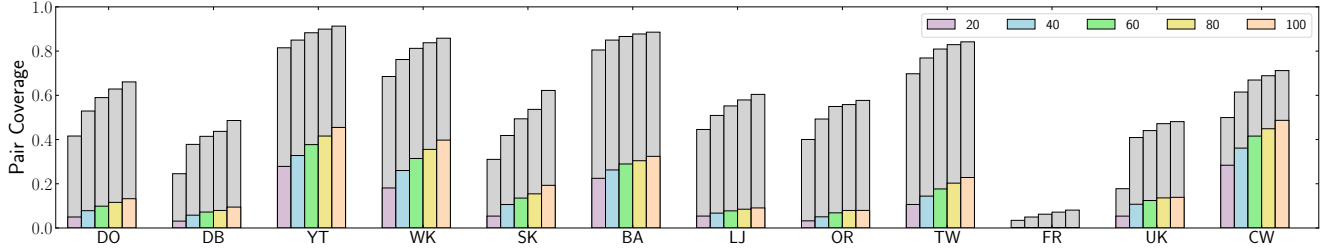


Figure 8: Pair coverage ratios using our method QbS under 20-100 landmarks where light color denotes the ratio of all the shortest paths between a vertex pair go through landmarks and grey color denoted the ratio of some but not all shortest paths between a vertex pair go through landmarks.

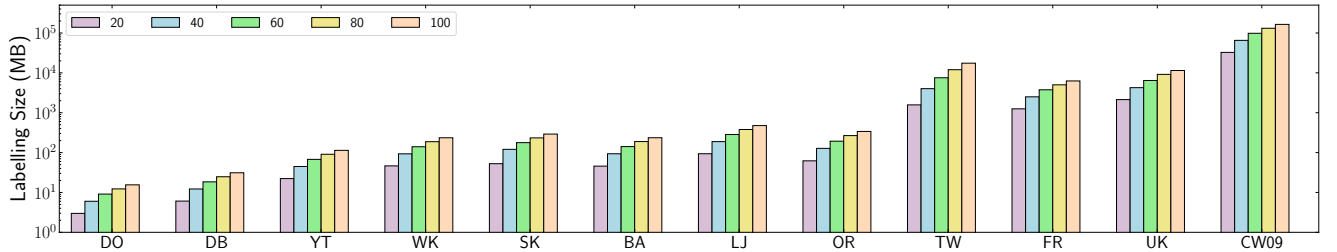


Figure 9: Labelling sizes using QbS under 20-100 landmarks on all the datasets.

has larger shortest path graphs as shown by $size(\Delta)$ in Table 3 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 ParentPPL 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.

6.3 Effects of Sketching

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

Figure 8 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 8, 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

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: 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.

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.

6.4 Performance with Varying Landmarks

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

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 10. Generally, the construction time grows linearly. In Figure 10(a)-(b), for datasets with millions of edges, QbS can construct labels under 100 landmarks within at most a few minutes. In Figure 10 (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.

6.4.2 Labelling Size. We compared the labelling sizes of QbS against different numbers of landmarks in Figure 9. For a labelling scheme $\mathcal{L} = (M, L)$, we use $|R| \cdot 8$ bits to store labels of each vertex. For M , as discussed in Section 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, 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.

6.4.3 Query Time. The impact of varying landmarks on query time is shown in Figure 11. 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.

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 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.

7 RELATED WORK

Exact algorithms. One of the most classical methods for shortest path computation is Dijkstra’s algorithm [11]. It computes a single-source shortest path tree on a weighted graph in time complexity $O(|E| + |V|\log|V|)$. For unweighted graphs, breadth-first search (BFS) computes a single-source shortest path tree in $O(|E|)$. However, these methods are very inefficient on large networks. A simple strategy for reducing search space is to employ bi-directional BFS which performs two searches from two given vertices, respectively, based on certain heuristic assumptions [15, 21]. To further accelerate shortest path computation, a number of methods have been proposed to pre-compute a labelling so as to answer point-to-point shortest path queries online in a shorter time [2, 5, 14–16, 31, 32, 36–39]. For example, Xiao et al. [39] exploited graph symmetry to label

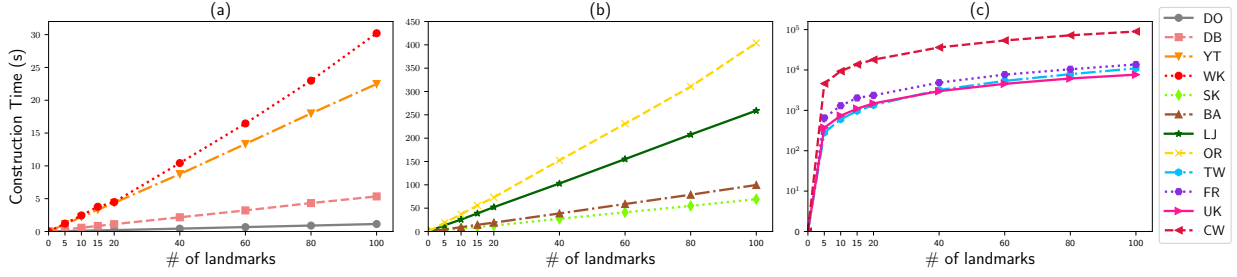


Figure 10: Construction times using QbS under 0-100 landmarks on all the datasets.

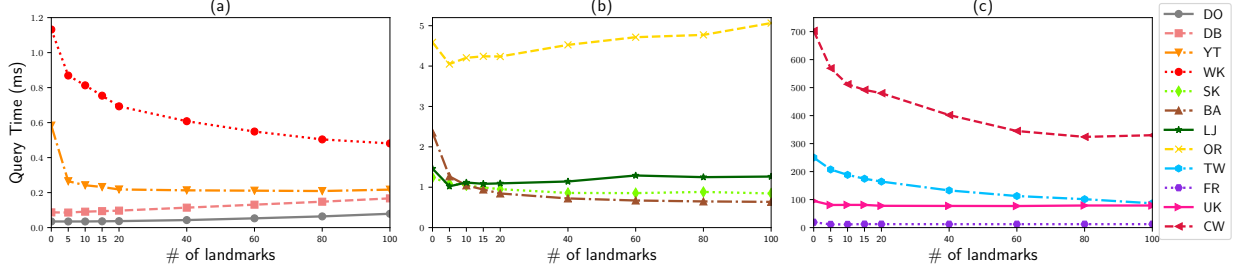


Figure 11: Average query times using our method QbS under 0-100 landmarks on all the datasets.

shortest paths. Though the size of labels has been compressed depending on the symmetric property, the space cost is still high. Later, Wei [37] 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., [3, 4, 12, 13, 19]). Some of them considered answering point-to-point shortest path queries as an extension of answering distance queries, although they did not provide any experiments. For example, Akiba et al. [3] proposed pruned landmark labelling (PLL) which constructs a 2-hop labelling for distance queries by conducting pruned BFSs. Fu et al. [13] 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 work, 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 pre-computes a small-sized distance labelling and can handle complex networks with up to billions of vertices.

Approximate algorithms. Due to the high computational costs of computing shortest paths, a number of approximate methods for point-to-point shortest path queries have been proposed in the past, including landmark-based methods with acceptable accuracy [17, 34, 41]. Specifically, Gubichev et al. [17] proposed to pre-compute shortest paths from each vertex to each landmark, and then concatenate shortest paths from two vertices to the same landmarks to approximate shortest paths. They also proposed cycle elimination and tree-based sketch to boost accuracy. Zhao et al.

[41] proposed a method, called Rigel, to estimate shortest path distances. They also extended Rigel for approximating shortest paths. Tretyakov et al. [34] used shortest path trees rooted at landmarks to approximate shortest path distances and search for one shortest path. Unlike these approximate algorithms, our work here aims to develop an exact method to accurately compute a shortest path graph that contains all shortest paths between two given vertices.

8 CONCLUSIONS

We have proposed a novel method 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 analyzed the complexity and correctness of our method. 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. For future work, we plan to extend QbS on road networks by leveraging their specific properties and study landmark selection strategies to improve the performance.

REFERENCES

- [1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*. 24–35.
- [2] Ittai Abraham, Amos Fiat, Andrew V Goldberg, and Renato F Werneck. 2010. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. 782–793.
- [3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 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.
- [4] Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi. 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.
- [5] Holger Bast, Stefan Funke, Domagoj Matijević, Peter Sanders, and Dominik Schultes. 2007. In transit to constant time shortest-path queries in road networks. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*. 46–59.
- [6] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. 2006. Complex networks: Structure and dynamics. *Physics reports* 424, 4-5 (2006), 175–308.
- [7] Paul Bonsma. 2013. The complexity of rerouting shortest paths. *Theoretical computer science* 510 (2013), 1–12.
- [8] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.
- [9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.
- [10] Lenore J Cowen and Christopher G Wagner. 2004. Compact roundtrip routing in directed networks. *Journal of Algorithms* 50, 1 (2004), 79–95.
- [11] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.
- [12] Muhammad Farhan, Qing Wang, Yu Lin, and Brendan McKay. 2019. A Highly Scalable Labelling Approach for Exact Distance Queries in Complex Networks. In *Proceedings of the 22th International Conference on Extending Database Technology*.
- [13] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 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.
- [14] Andrew V Goldberg. 2007. Point-to-point shortest path algorithms with preprocessing. In *International Conference on Current Trends in Theory and Practice of Computer Science*. 88–102.
- [15] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. 156–165.
- [16] Andrew V Goldberg, Haim Kaplan, and Renato F Werneck. 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.
- [17] Andrey Gubichev, Srikanth Bedathur, Stephan Seufert, and Gerhard Weikum. 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.
- [18] Pierre Hansen, Jacques-François Thisse, and Richard E Wendell. 1986. Efficient points on a network. *Networks* 16, 4 (1986), 357–368.
- [19] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management*. 1533–1542.
- [20] Eitan Israeli and R Kevin Wood. 2002. Shortest-path network interdiction. *Networks: An International Journal* 40, 2 (2002), 97–111.
- [21] Ruoming Jin, Ning Ruan, Bo You, and Haixun Wang. 2013. Hub-accelerator: Fast and exact shortest path computation in large social networks. *arXiv preprint arXiv:1305.0507* (2013).
- [22] Marcin Kamiński, Paul Medvedev, and Martin Milanič. 2011. Shortest paths between shortest paths. *Theoretical Computer Science* 412, 39 (2011), 5205–5210.
- [23] Leonid Khachiyan, Endre Boros, Konrad Borys, Khaled Elbassioni, Vladimir Gurvich, Gabor Rudolf, and Jihui Zhao. 2008. On short paths interdiction problems: Total and node-wise limited interdiction. *Theory of Computing Systems* 43, 2 (2008), 204–233.
- [24] Eric D Kolaczyk, David B Chua, and Marc Barthélemy. 2009. Group betweenness and co-betweenness: Inter-related notions of coalition centrality. *Social Networks* 31, 3 (2009), 190–203.
- [25] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. 1343–1350.
- [26] Martine Labbé, Dominique Peeters, and Jacques-François Thisse. 1995. Location on networks. *Handbooks in operations research and management science* 8 (1995), 551–624.
- [27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.
- [28] Naomi Nishimura. 2018. Introduction to reconfiguration. *Algorithms* 11, 4 (2018), 52.
- [29] Tore Opsahl, Filip Agneessens, and John Skvoretz. 2010. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social networks* 32, 3 (2010), 245–251.
- [30] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*. 867–876.
- [31] Peter Sanders and Dominik Schultes. 2005. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms*. 568–579.
- [32] Jagan Sankaranarayanan, Hanan Samet, and Houman Alborzi. 2009. Path oracles for spatial networks. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1210–1221.
- [33] John Scott. 1988. Social network analysis. *Sociology* 22, 1 (1988), 109–127.
- [34] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano Garcia-Bañuelos, Jaak Vilo, and Marlon Dumas. 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.
- [35] Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. 2008. Searching the wikipedia with contextual information. In *Proceedings of the 17th ACM conference on Information and knowledge management*. 1351–1352.
- [36] Dorothea Wagner and Thomas Willhalm. 2007. Speed-up techniques for shortest-path computations. In *Annual Symposium on Theoretical Aspects of Computer Science*. 23–36.
- [37] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 99–110.
- [38] Lingkun Wu, Xiaokui Xiao, Dingxiang Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. 2012. Shortest path and distance queries on road networks: An experimental evaluation. *Proceedings of the VLDB Endowment* 5, 5 (2012), 406–417.
- [39] Yanghua Xiao, Wentao Wu, Jian Pei, Wei Wang, and Zhenying He. 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.
- [40] Bin Yao, Feifei Li, and Xiaokui Xiao. 2013. Secure nearest neighbor revisited. In *2013 IEEE 29th International Conference on Data Engineering*. 733–744.
- [41] Xiaohan Zhao, Alessandra Sala, Haitao Zheng, and Ben Y Zhao. 2011. Efficient shortest paths on massive social graphs. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 77–86.