SIEF: Efficiently Answering Distance Queries for Failure Prone Graphs

Yongrui Qin
School of Computer Science
The University of Adelaide
Adelaide, SA 5005, Australia
yongrui.qin@adelaide.edu.au

Quan Z. Sheng
School of Computer Science
The University of Adelaide
Adelaide, SA 5005, Australia
michael.sheng@adelaide.edu.au

Wei Emma Zhang
School of Computer Science
The University of Adelaide
Adelaide, SA 5005, Australia
wei.zhang01@adelaide.edu.au

ABSTRACT

Shortest path computation is one of the most fundamental operations for managing and analyzing graphs. A number of methods have been proposed to answer shortest path distance queries on static graphs. Unfortunately, there is little work on answering distance queries on *dynamic* graphs, particularly graphs with edge failures. Today's real-world graphs, such as the social network graphs and web graphs, are evolving all the time and link failures occur due to various factors, such as people stopping following others on Twitter or web links becoming invalid. Therefore, it is of great importance to handle distance queries on these failure-prone graphs. This is not only a problem far more difficult than that of static graphs but also important for processing distance queries on evolving or unstable networks. In this paper, we focus on the problem of computing the shortest path distance on graphs subject to edge failures. We propose SIEF, a Supplemental Index for Edge Failures on a graph, which is based on distance labeling. Together with the original index created for the original graph, SIEF can support distance queries with edge failures efficiently. By exploiting properties of distance labeling on static graphs, we are able to compute very compact distance labeling for all singe-edge failure cases on dynamic graphs. We extensively evaluate our algorithms using six real-world graphs and confirm the effectiveness and efficiency of our approach.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Graphs and networks*; H.2.8 [**Database management**]: Database Applications—*Graph Indexing and Querying*

General Terms

Algorithms, Experimentation, Performance

Keywords

Shortest paths, distance query, 2-hop labeling, edge failure

1. INTRODUCTION

Recent years have witnessed the fast emergence of massive graph data in many application domains, such as the World Wide Web, linked data technology, online social networks, and Web of Things [21, 19, 22, 25]. In a graph, one of the most fundamental challenges centers on the efficient computation of the shortest path or distance between any given pair of vertices. For instance, distances or the numbers of links between web pages on a web graph can be considered a robust measure of web page relevancy, especially in relevance feedback analysis in web search [21]. In RDF graphs of linked data, the shortest path distance from one entity to another is important for ranking entity relationships and keyword querying [19, 14]. For online social networks, the shortest path distance can be used to measure the closeness centrality between users [22].

A large body of indexing techniques have been recently proposed to process exact shortest path distance queries on graphs [10, 23, 9, 8, 2, 26, 15]. Among them, a significant portion of indexes are based on *2-hop* distance labeling, which is originally proposed by Cohen et al. [12]. The 2-hop distance labeling techniques pre-compute a label for each vertex so that the shortest path distance between any two vertices can be computed by giving their labels only. These labeling indexes, such as [10, 8, 2, 15], have been proved to be efficient, i.e., being able to answer a distance query within microseconds.

Motivation. The above mentioned approaches generally make the assumption that graphs are static. However, in reality, many graphs are subject to edge failures. In this paper, we refer to graphs that are not subject to edge failures as *stable graphs*, i.e., static graphs. Similarly, we refer to graphs that are subject to edge failures as *unstable graphs*. For example, the emerging social Web of Things calls for graph data management with edge failures because smart things are normally moving and their connectivity could be intermittent, leading to frequent and unpredictable changes in the corresponding graph models [11, 25]. Another example is web graphs. It is not uncommon that some web links become invalid as the web evolves. All these are examples of unstable graphs, which are common in the real-world, calling for efficient graph computations by considering link failures. We believe that it is imperative to design novel algorithms that can compute shortest path indexes for fast response on distance queries avoiding any failed edge. Some real-world applications/scenarios that require the computa-

10.5441/002/edbt.2015.14

tion of shortest path distance avoiding a failed edge are described in the following.

SCENARIO 1. *The* **most vital arc** *problem [17, 6] aims to identify the edge on a given shortest path and the removal of this edge results in the longest replacement path. Here, a replacement path means a shortest path from a source vertex to a destination vertex in a graph that avoids a specified edge. To find the most vital arc in a graph, we need to compute the shortest path distances efficiently when we are given an arc (i.e., an edge) to avoid.*

SCENARIO 2. *In the* **sensitivity analysis** *and in many analytical applications of transportation networks, government agencies need to evaluate different road segments (i.e., to find how much a road segment is worth) through Vickrey pricing [16], such that maintenance budget can be allocated accordingly, or the amount of tolls can be adjusted reasonably [24]. For example, if tolls are not charged appropriately and avoiding an expensive toll point causes only a small detour, then it is more likely that most drivers would take the detour, rather than pay for the toll.*

SCENARIO 3. *In order to develop* **game-theoretic and price-based mechanisms** *to share bandwidth and other network resources, a natural economic question is [16]: how much is an edge in a network worth to a user who wants to send data between two nodes along a shortest path? Or in other words, what is the penalty of avoiding an edge in the given network?*

These application scenarios reveal an urge for handling shortest path computations in a graph with single-edge failures. Here, single-edge failure refers to graph failures with only one failed edge at a time [5]. Note that, other types of edge failure, such as dual-failure in [13], may allow multiple failed edges at a time. But they are considered much harder than single-failure [13]. To shed light on these challenging issues, we focus on single-edge failures in this paper.

Contributions. Since 2-hop labeling has shown its power to support instant responses to shortest path distance queries on stable graphs, our work aims at extending this technique to support unstable graphs. Existing shortest path indexing techniques based on 2-hop labeling can be used to precompute the whole shortest path index for a graph. The resulted indexes can normally answer distance queries fast using moderate storage space [2, 15]. However, applying indexing techniques designed for static/stable graphs directly to evolving/unstable graphs may lead to inefficiency. When considering every single-edge failure case and constructing a corresponding index for each case, the size of all these indexes will become too big to manage. For instance, a snapshot of the Gnutella peer-to-peer (P2P) file sharing network in August 2002 contains more than 6,000 vertices[1] and 20,000 edges. Using state-of-the-art method, Pruned Landmark Labeling (PLL) [2], the index size is slightly more than 5 MB. However, suppose we want to construct such index for each single-edge failure case, the total index size would be more than $5 \times 20,000 = 10^5$ MB.

To address the deficiency of existing shortest path indexing techniques, this paper proposes a generic framework named SIEF, a Supplemental Index for Edge Failures on a graph, to construct compact shortest path indexes efficiently for unstable graphs where single-edge failures may exist. As an initial attempt on this challenging issue, we focus on unweighted, undirected graphs. Similar to other distance labeling based indexing methods [2, 15], our method can be extended to weighted and/or directed graphs. We highlight our main contributions in the following.

- We present the concept of *well-ordering 2-hop distance labeling* and identify its important properties that can be utilized to design algorithms for shortest path indexes on graphs with edge failures.
- We analyze shortest path index constructions on graphs with edge failures theoretically. We develop the corresponding theorems as well as novel algorithms to enable constructions of compact indexes for all the single-edge failure cases of the entire graph. By applying our approach to the aforementioned Gnutella P2P dataset, the size of the generated SIEF index together with the original index created for the original graph is merely 14 MB, which is much more compact than 10^5 MB by directly using PLL method [2] to construct indexes for each single-edge failure case.
- We conduct extensive experiments on six real-world graphs to verify the efficiency and effectiveness of our method. The results show that our method can efficiently answer shortest path distance queries avoiding a failed edge with very compact labeling indexes.

The rest of this paper is organized as follows. In Section 2, we review the related work. In Section 3, we present some preliminaries on 2-hop distance labeling. We then present the framework and the details of our approach in Section 4. In Section 5, we report the results of an extensive experimental study using six graphs from real-world. Finally, we present some concluding remarks in Section 6.

2. RELATED WORK

In this section, we review the major techniques that are most closely related to our work.

Distance labeling has been an active research area in recent years. In [10], Cheng and Yu exploit the strongly connected components property and graph partitioning to precompute 2-hop distance cover. However, the graph partitioning process introduces high cost because it has to find vertex separators recursively. Hierarchical hub labeling (HHL) proposed by Abraham et al. [1] is based on the partial order of vertices. Smaller labeling results can be obtained by computing labeling for different partial order of vertices. In [18], Jin et al. propose a highway-centric labeling (HCL) that uses a spanning tree as a highway. Based on the highway, a 2-hop labeling is generated for fast distance computation.

Very recently, the pruned landmark labeling (PLL) [2] is proposed by Akiba et al. to pre-compute 2-hop distance labels for vertices by performing a breadth-first search from every vertex. The key idea is to prune vertices that have obtained correct distance information during breadth-first searches, which helps reduce the search space and sizes of labels. Further, query performance is also improved as the number of label entries per vertex is reduced. IS-Label (or ISL) is developed by Fu et al. in [15] to pre-compute 2-hop

[1] http://snap.stanford.edu/

distance label for large graphs in memory constrained environments. ISL is based on the idea of independent set of vertices in a large graph. By recursively removing an independent set of vertices from the original graph, and by augmenting edges that preserve distance information after the removal of vertices in the independent set, the remaining graph keeps the distance information for all remaining vertices in the graph. Apart from the 2-hop distance labeling technique, a multi-hop distance labeling approach [8] is also studied, which can reduce the overall size of labels at the cost of increased distance querying time.

Tree decomposition approach has been recently investigated [23, 4] for answering distance queries on graphs. Wei proposes TEDI [23], which first decomposes a graph into a tree and then constructs a tree decomposition for the graph. A tree decomposition of a graph is a tree with each vertex associated with a set of vertices in the graph, which is also called a *bag*. The shortest paths among vertices in the same bag are pre-computed and stored in bags. For any given source and target vertices, a bottom-up operation along the tree can be executed to find the shortest path. An improved TEDI index is proposed by Akiba et al. in [4] that exploits a core-fringe structure to improve index performance. However, due to the large size of some bags in the decomposed tree, the construction time for a large graph is costly and thus such indexing approaches cannot scale well.

Maintenance of 2-hop reachability labeling is also studied. For example, HOPI (2-HOP-cover-based Index) introduces some maintenance techniques for the constructed index. HOPI is developed by Schenkel et al. in [20] and is designed to speed up connection or reachability tests in XML documents based on the idea of 2-hop cover. HOPI is able to update indexes for insertions and deletions of nodes, edges or even XML documents. To the best of our knowledge, HOPI is the first work on maintenance of 2-hop labeling. Recently, maintenance of 2-hop labeling for large graphs has also been studied by Bramandia et al. in [7]. However, all these studies focus on reachability queries and are based on 2-hop labeling but not on 2-hop distance labeling.

Incremental maintenance of 2-hop distance labeling is also studied very recently by Akiba et al. in [3]. In that work, incremental updates (i.e., edge insertions) of 2-hop labeling indexes are investigated. To support fast incremental updates, outdated distance labels are kept, which will not affect the distance computation on the updated graphs in the incremental case. However, for the decremental case (i.e., edge deletions), this approach will not work, as outdated distance labels must be removed first and then some necessary labels of the 2-hop labeling index need to be recomputed. Hence, their update algorithms cannot be applied on edge deletions (i.e., edge failures), which will be discussed in this paper.

3. PRELIMINARIES

3.1 2-Hop Distance Labeling

The technique of 2-hop cover can be used to solve reachability problems (using reachability labels) and shortest path distance querying problems (using distance labels) on graphs [12]. Since our work focuses on the shortest path distance querying problems, we adopt distance labels with the 2-hop cover technique. We specifically refer to it as *2-hop distance labeling* or *2-hop distance cover*.

Assume a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. For each vertex $v \in V$, there is a precomputed label $L(v)$, which is a set of vertex and distance pairs (u, δ_{uv}). Here u is a vertex and δ_{uv} is the shortest path distance between u and v. Given such a labeling for all vertices in G, denoted by L, for any pair of vertices s and t in G, we have

$$\begin{aligned} dist(s,t,L) =& min\{\delta_{vs} + \delta_{vt} | (v, \delta_{vs}) \in L(s) \\ & \text{and } (v, \delta_{vt}) \in L(t)\} \end{aligned} \tag{1}$$

If $L(s)$ and $L(t)$ do not share any vertices, we have $dist(s,t,L)$ $=\infty$. The distance between any given vertices s and t in G is denoted by $d_G(s,t)$. If we have $d_G(s,t) = dist(s,t,L)$ for all s and t in G, we call the labeling result L a 2-hop distance cover.

3.2 Well-Ordering 2-Hop Distance Labeling

For a connected graph G, there exists a sequence of vertices $\sigma =< v_0, v_1, v_2, \ldots, v_{n-1} >$. We denote the order of any vertex v_i as $\sigma[v_i]$ and we have $\sigma[v_i] = i$ for the above given vertex sequence. Based on this, we can define *Well-Ordering 2-Hop Distance Labeling* in the following.

Definition 1 (***Well-Ordering 2-Hop Distance Labeling***). Suppose that (1) each vertex v_i has a distance labeling $L(v_i)$, and the labeling result L of all vertices forms a 2-hop distance cover of G; (2) for any pair of vertices v_i and v_j, given that $\sigma[v_i] < \sigma[v_j]$, then v_j is not in $L(v_i)$ and v_i may be in $L(v_j)$. We call such a 2-hop distance cover a *well-ordering 2-hop distance labeling.* Alternatively we say that a 2-hop distance cover has well-ordering property.

Similar concepts of well-ordering 2-hop distance labeling also appear in recent research efforts such as HHL [1], PLL [2], and ISL [15]. This confirms that well-ordering 2-hop distance labeling is important in the related research area. More importantly, we will show in this paper that the well-ordering property is also a basic concept in the design of index construction algorithms for distance labeling computation on unstable graphs where edges may fail.

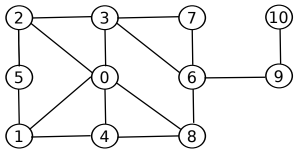


Figure 1: A graph example

In a graph containing multiple connected components, suppose its 2-hop labeling is L. For any pair of vertices u and v in different connected components, we can assert that $L(u)$ and $L(v)$ do not share any vertex according to the definition of 2-hop cover. Each connected component has its own vertex orders. For such a graph, we will have separate vertex orders for each connected component. We denote a connected component containing vertex u as $C(u)$. If u and v belong to the same connected component, we have $C(u) = C(v)$.

Figure 1 shows an example graph with 11 vertices and Table 1 shows a well-ordering 2-hop distance labeling result L for the graph (L can be constructed by PLL [2] using the same vertex ordering as that specified in the table). In the

Table 1: 2-Hop Distance Labeling L for Figure 1

Label	Entries
$L(0)$	(0,0)
$L(1)$	(0,1) (1,0)
$L(2)$	(0,1) (2,0)
$L(3)$	(0,1) (2,1) (3,0)
$L(4)$	(0,1) (1,1) (4,0)
$L(5)$	(0,2) (1,1) (2,1) (5,0)
$L(6)$	(0,2) (2,2) (3,1) (4,2) (6,0)
$L(7)$	(0,2) (2,2) (3,1) (6,1) (7,0)
$L(8)$	(0,1) (4,1) (6,1) (8,0)
$L(9)$	(0,3) (2,3) (3,2) (4,3) (6,1) (9,0)
$L(10)$	(0,4) (2,4) (3,3) (4,4) (6,2) (9,1) (10,0)

table, the order of vertices is $< 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 >$. Take $L(5)$ as an example to further explain the idea of well-ordering 2-hop distance labeling. $L(5)$ is the label of vertex 5. By the well-ordering property, label entries in $L(5)$ can only contain vertices $0, 1, 2, 3, 4$ and 5. Since label entries containing vertices 3 and 4 are redundant in $L(5)$ (this will be explained in more details later in this section), label entries in $L(5)$ only contain vertices $0, 1, 2$ and 5.

3.3 Properties of Well-Ordering 2-Hop Distance Labeling

Technically speaking, if we index shortest paths for all pairs using a labeling method, we will obtain an index that occupies $O(n^2)$ disk space. This index can be considered as a special 2-hop distance labeling. Obviously, the space complexity of this is too high for large graphs. Constructing a minimal 2-hop distance labeling has been proven to be NP-hard [12]. Therefore, an alternative way to obtain labeling results with reduced sizes is by using heuristic methods [10, 8, 2, 15]. Well-ordering 2-hop distance labeling is one of the techniques that can help to design efficient algorithms for constructing shortest path distance labeling indexes and for index maintenance. We identify its useful properties in the following.

Lemma 1. *Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $u \in G$ and $\sigma[u]$ is a minimum among all vertices in G, then for any vertex $v \in G$, we must have $(u, \delta_{uv}) \in L(v)$.*

Proof. It is trivial to prove this when $v = u$ since $(u, 0) \in L(u)$. We prove the case when $v \neq u$ by contradiction. Suppose there exists a vertex $v \in G$, $(u, \delta_{uv}) \notin L(v)$. By the definition of L, since $\sigma[u]$ is minimum, $L(u)$ will contain only one label entry $(u, 0)$. Then it is obvious that $L(u)$ and $L(v)$ do not share any vertex, which leads to $dist(u, v, L) = \infty$. This implies that u and v belong to different connected components, which is false. Therefore, the lemma is proved. □

Lemma 2. *Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $s, t, u \in G$ and $dist(s, t, L) = dist(s, u, L) + dist(u, t, L)$, then u must be an internal vertex of a certain shortest path between s and t.*

Proof. Since $dist(s, t, L) = dist(s, u, L) + dist(u, t, L)$, there must exist some shortest path that starts from s, passes u, and ends at t. Hence the lemma is proved. □

Take vertices 5, 6 and 2 in Figure 1 as an example. From Table 1, we have $dist(5, 6, L)$=3 and $dist(5, 2, L)$+$dist(2, 6, L)$ =1+2=3. From Figure 1, we can see that vertex 2 is an internal vertex on some shortest path, denoted as p, between vertex 5 and vertex 6. In this case, we have $p =< 5, 2, 3, 6 >$.

Lemma 3. *Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $s, t, u \in G$ and u has minimum vertex order $\sigma[u]$ among all shortest paths between s and t. Then we must have $(u, \delta_{us}) \in L(s)$ and $(u, \delta_{ut}) \in L(t)$ and $dist(s, t, L) = \delta_{us} + \delta_{ut}$.*

Proof. We prove this by contradiction. Without loss of generality, suppose $(u, \delta_{us}) \notin L(s)$. In order to calculate $dist(s, u, L)$, there must exist some vertex v other than u, where $(v, \delta_{vs}) \in L(s)$, $(v, \delta_{vu}) \in L(u)$ and $dist(s, u, L) = \delta_{vs} + \delta_{vu}$. According to Lemma 2, v must be an internal vertex of some shortest path between s and u. Hence v must also be an internal vertex of some shortest path between s and t. Meanwhile, by definition, we must have $\sigma[v] < \sigma[u]$. This contradicts our assumption that u has the minimum vertex order among all shortest paths between s and t. Hence, we must have $(u, \delta_{us}) \in L(s)$. Furthermore, u is an internal vertex of some shortest path between s and t, thus $dist(s, t, L) = \delta_{us} + \delta_{ut}$. Hence the lemma is proved. □

Take vertices 1 and 6 in Figure 1 as an example. Paths $p_1 =< 1, 0, 8, 6 >$, $p_2 =< 1, 0, 3, 6 >$ and $p_3 =< 1, 4, 8, 6 >$ are all the shortest paths between vertices 1 and 6. Vertex 0 is the one with minimum order along all these paths. From Table 1 we can see that both vertices 1 and 6 contain a label entry $(0, \delta)$. We can also easily check that $dist(1, 6, L) = \delta_{0,1} + \delta_{0,6} = 1 + 2 = 3$.

Lemma 4. *Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $\sigma[u] < \sigma[v]$. If there is a label entry $(u, \delta_{uv}) \in L(v)$, we must have for any label entry $(r, \delta_{rv}) \in L(v)$, (1) $\delta_{uv} \leq \delta_{rv} + dist(r, u, L)$; (2) if $\sigma[r] < \sigma[u]$ and $\delta_{uv} = \delta_{rv} + dist(r, u, L)$ then $(u, \delta_{uv}) \in L(v)$ is a redundant label entry.*

Proof. We first prove the first claim that $\delta_{uv} \leq \delta_{rv} + dist(r, u, L)$. By definition and the triangle inequalities we must have $\delta_{uv} = d_G(u, v) = dist(u, v, L) \leq \delta_{rv} + dist(r, u, L)$.

We then prove the second claim. We need to prove that if $\delta_{uv} = \delta_{rv} + dist(r, u, L)$, then for any vertex t, when we calculate $dist(v, t, L)$, (u, δ_{uv}) in $L(v)$ is not required. For t, there are three cases: (1) $(u, \delta_{ut}) \notin L(t)$; (2) $(u, \delta_{ut}) \in L(t)$ but $\delta_{uv} + \delta_{ut} > dist(v, t, L)$; (3) $(u, \delta_{ut}) \in L(t)$ and $\delta_{uv} + \delta_{ut} = dist(v, t, L)$. For Case (1) and Case (2), it is trivial since (u, δ_{uv}) in $L(v)$ is not required to calculate $dist(v, t, L)$. For Case (3), according to Lemma 2, u is an internal vertex of some shortest paths between v and t. Similarly, since $\delta_{uv} = \delta_{rv} + dist(r, u, L)$, r is an internal vertex of some shortest paths between u and v, which means r is also an internal vertex of some shortest paths between v and t. In such case, we prove in the following that there must exist a vertex s other than u and we have $(s, \delta_{sv}) \in L(v)$, $(s, \delta_{st}) \in L(t)$ where $\delta_{st} + \delta_{sv} = dist(v, t, L)$.

Suppose s is the vertex with minimum vertex order among all shortest paths between v and t. According to Lemma 3, we must have $(s, \delta_{sv}) \in L(v)$, $(s, \delta_{st}) \in L(t)$ and $\delta_{st} + \delta_{sv} = dist(v, t, L)$. Since $\sigma[s] \leq \sigma[r] < \sigma[u]$, s is not the same vertex of u. Therefore, (u, δ_{uv}) in $L(v)$ is not required to calculate $dist(v, t, L)$. Hence, the second claim is also proved. □

Take label entries of vertex 5 in Table 1 as an example. We have $\sigma(3) < \sigma(5)$ and $\sigma(2) < \sigma(3)$. We also have $\delta_{3,5} = 2 = \delta_{2,5} + \delta_{2,3}$. Therefore $(3, 2)$ is a redundant label entry in $L(5)$, which can be removed from $L(5)$.

4. THE SIEF APPROACH

In this section, we first provide an overview of our SIEF approach. We then analyze the 2-hop distance labeling computation on graphs with single-edge failures and introduce a set of algorithms to achieve fast and compact index constructions.

4.1 SIEF Overview

After an edge fails on a graph, we observe that distances of a considerable proportion of shortest paths between any pair of vertices remain unchanged. Therefore, to construct a new index for each single-edge failure case, we only need to compute new labels for those vertices with changed shortest path distances due to the edge failure. Overall, our index construction approach can be divided into two main stages. In the first stage, *IDENTIFY*, we identify affected vertices after an edge fails. In the second stage, *RELABEL*, we relabel all affected vertices with necessary additional label entries for the single-edge failed graph. These new label entries form a new part of the index, which is called a *supplemental index*.

Before the detailed discussions of our algorithms, suppose that the failed edge is (u, v) in G, and the new graph is G', we introduce a concept for the supplemental index construction:

Definition 2 (***Affected vertices*** $AV_{(u,v)}$). For any vertices s and t, if $d_{G'}(s,t) \neq d_G(s,t)$, then $s \in AV_{(u,v)}$ and $t \in AV_{(u,v)}$.

To be specific, $AV_{(u,v)}$ contains all vertices whose distance to some other vertex must have been changed due to the failed edge (u, v). It is quite clear that supplemental indexes should be constructed to maintain all new distances for each single-edge failure case. In other words, supplemental indexes are constructed based on all the vertices in $AV_{(u,v)}$. Further, in order to be compact, the supplemental indexes should only answer distances that cannot be answered by the original index.

4.2 Identification of Affected Vertices

Before we can start to construct supplemental indexes, we need to identify all the affected vertices in $AV_{(u,v)}$ first. A naive method would be to compare distances for any possible pair of affected vertices in the original graph G and the new graph G' with a failed edge (u, v), but that would be very time consuming as it will need to test distances of $O(n^2)$ pairs of vertices. In the following, we will try to identify some important properties for vertices in $AV_{(u,v)}$ for us to identify $AV_{(u,v)}$ more efficiently and accurately.

Lemma 5. *After removing the failed edge (u, v) from graph G, for any vertex s, t in G', we must have $d_{G'}(s,t) \geq dist(s,t, L)$.*

Proof. In the old graph G, there are only two types of shortest paths: (1) shortest paths containing edge (u, v); and (2) shortest paths not containing edge (u, v). For the former, we have $d_{G'}(s,t) \geq d_G(s,t) = dist(s,t,L)$. For the latter, we have $d_{G'}(s,t) = d_G(s,t) = dist(s,t,L)$. Thus the lemma is proved. □

Lemma 6. *After removing the failed edge (u, v) from graph G, for any vertex s, t in G', if $d_{G'}(s,t) > dist(s,t,L)$, and suppose a shortest path between s and t in G is $\pi_G(s,t)$, then we must have $uv \in \pi_G(s,t)$ or $vu \in \pi_G(s,t)$.*

Proof. This can be proved by contradiction. Suppose we have $d_{G'}(s,t) > dist(s,t,L)$ but $uv \notin \pi_G(s,t)$ and $vu \notin \pi_G(s,t)$, which means edge (u, v) does not appear in $\pi_G(s,t)$. In such case, there must exist a path $P_{G'}(s,t)$ in G' where $\pi_G(s,t) = P_{G'}(s,t)$. This means $d_{G'}(s,t)$ must be at most the length of $P_{G'}(s,t)$, i.e., the length of $\pi_G(s,t)$. Thus, we must have $d_{G'}(s,t) = dist(s,t,L') \leq d_G(s,t)$. This contradicts our assumption $d_{G'}(s,t) > dist(s,t,L)$. □

According to Lemma 6 and the definition of affected vertices, if we have $d_{G'}(s,t) > dist(s,t,L) = d_G(s,t)$, we must have that $s, t \in AV_{(u,v)}$. This further means the shortest path(s) between s and t in the original graph G must contain the failed edge (u, v). Then, after edge (u, v) fails, take any one of these shortest paths (if multiple shortest paths exist; if not, we will have one and only one shortest path containing (u, v)) as an example, denoted as $\pi_G(s,t)$. Then it is easy to imagine that $\pi_G(s,t)$ will become two segments: one segment ends at u, denoted as Seg_u and the other segment ends at v, denoted as Seg_v. Without loss of generality, suppose s falls on the Seg_u and t falls on Seg_v. Since Seg_u and Seg_v must also be shortest paths from s to u and from t to v, respectively, this means we must have $d_G(s,u) = d'_G(s,u)$ and $d_G(t,v) = d'_G(t,v)$. But in the meantime, we must have $d_G(s,v) \neq d'_G(s,v)$ and $d_G(t,u) \neq d'_G(t,u)$ since otherwise we will have $d_{G'}(s,t) = d_G(s,t)$, which is impossible. Based on this observation, we can see that vertices in $AV_{(u,v)}$ form two disjoint sets: one set is $AV_{(u,v)}(u)$ and the other set is $AV_{(u,v)}(v)$, where for $\forall s \in AV_{(u,v)}(u)$ and $\forall t \in AV_{(u,v)}(v)$, we must have $d_{G'}(s,t) > d_G(s,t)$, $d_G(s,u) = d'_G(s,u)$ and $d_G(t,v) = d'_G(t,v)$. Since (u, v) is the failed edge, obviously, we must have $u \in AV_{(u,v)}(u)$ or $v \in AV_{(u,v)}(v)$. Further, it should be noted that, $\forall s, t \in AV_{(u,v)}(u)$, we must have $d_G(s,t) = d'_G(s,t)$. The same conclusion can be made on $\forall s, t \in AV_{(u,v)}(v)$.

Next, we are going to show that all vertices $s \in AV_{(u,v)}(u)$ form a tree rooted at u and similarly, all vertices $t \in AV_{(u,v)}(v)$ also form a tree rooted at v.

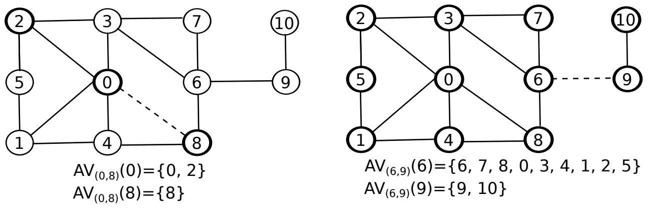
Lemma 7. *After removing the failed edge (u, v), for any vertex w in G', suppose w is an affected vertex, i.e. $w \in AV_{(u,v)}(u)$ or $w \in AV_{(u,v)}(v)$. Without loss of generality, we assume $w \in AV_{(u,v)}(u)$. Then we must have $d_G(w,v) = d_G(w,u) + 1$.*

Proof. Since $w \in AV_{(u,v)}(u)$, we must have that $d_G(w,v) \neq d'_G(w,v)$, which means that any shortest path between w and v in G, denoted as p_{wv} must contain the failed edge (u, v), which must also be a shortest path between w and v. Hence, there must exist a certain shortest path between w and v containing edge (u, v) in the original graph and we can denote it as $p_{wv} = p_{wu} + (u, v)$. Hence, we must have $d_G(w,v) = d_G(w,u) + 1$. □

Lemma 8. *After removing the failed edge (u, v), suppose w in G' is an affected vertex, i.e. $w \in AV_{(u,v)}(u)$ or $w \in AV_{(u,v)}(v)$. Without loss of generality, we assume $w \in AV_{(u,v)}$ (u). Then there must exist a certain shortest path between w and v containing edge (u, v) in the original graph, where each internal vertex is an affected vertex in $AV_{(u,v)}(u)$.*

Algorithm 1 Identify affected vertices

```
Input: $G$, $(u, v)$, distance vectors $d_u, d_v, d'_u, d'_v$
Output: $AV_{(u,v)}(u)$, $AV_{(u,v)}(v)$
Initialize flag $m[t] \leftarrow 0$ for any vertex $t$ in $G$
$m[u] \leftarrow 1$
$Q \leftarrow \emptyset$
Enqueue $u$ into $Q$
while $Q$ is not empty do
    Dequeue $t$ from $Q$
    for all neighbor vertex $r$ of $t$ do
        if $m[r] = 0$ then
            if $d_v[r] = d_u[r] + 1$ and $d'_v[r] \neq d_u[r] + 1$ then
                $AV_{(u,v)}(u) \leftarrow AV_{(u,v)}(u) \cup \{r\}$
                Enqueue $r$ into $Q$
            $m[r] \leftarrow 1$
Repeat the above steps by mapping $u \leftarrow v$ and $v \leftarrow u$
    to identify $AV_{(u,v)}(v)$
```



Case (a): The failed edge is (0,8)   Case (b): The failed edge is (6,9)

Figure 2: Affected vertices identification

Proof. Since $w \in AV_{(u,v)}(u)$, then according to Lemma 7, we must have the fact that any shortest path between w and u, denoted as p_{wu}, plus edge (u, v) in the original graph must be a shortest path between w and v. Then, there must exist a certain shortest path between w and v containing edge (u, v) in the original graph and we can denote it as $p_{wv} = p_{wu} + (u, v)$.

It is clear that the internal vertices of p_{wv} must also be on some shortest path p_{wu}. And all shortest paths from these internal vertices to vertex v must contain edge (u, v), which means, their distances to vertex v must have changed in the new graph G'. Therefore, they must also be affected vertices in $AV_{(u,v)}(u)$ like w. □

Note that, according to Lemma 8, $AV_{(u,v)}(u)$ and $AV_{(u,v)}$ (v) can be considered as trees rooted at u and v, respectively. Moreover, we must have $AV_{(u,v)}(u) \bigcap AV_{(u,v)}(v) = \emptyset$. This is because otherwise, any vertex r in $AV_{(u,v)}(u) \bigcap AV_{(u,v)}(v)$ must have $d_G(r, v) = d_G(r, u)+1$ and $d_G(r, u) = d_G(r, v)+1$, which is impossible. Lemma 8 forms the basis of Algorithm 1. Note that, in Algorithm 1, we need to calculate distance vectors d_u, d_v, d'_u and d'_v for each single-edge failure case. Here, d_u stores distances from all vertices in G to vertex u while d'_u stores distances from all vertices in G' to vertex u. Distance vectors d_v and d'_v are similar. The calculations can be done efficiently using a BFS algorithm. To reduce the calculation cost, we will fix an end point of failed edges, i.e., we will firstly compute affected vertices for all edges attached to u then we move to other vertices for processing the rest single-edge failure cases.

Figure 2 shows two examples of identifying affected vertices. It uses the same graph in Figure 1. In this figure, the first example is Case (a), where the failed edge is $(0, 8)$. The second example is Case (b), where the failed edge is $(6, 9)$. In Case (a), starting from vertex 0, we identify the affected vertex set rooted at 0 as $AV_{(0,8)}(0) = \{0, 2\}$ since only vertices 0 and 2 have changed their distance to vertex 8. Meanwhile, starting from vertex 8, we identify the affected vertex set rooted at 8 as $AV_{(0,8)}(8) = \{8\}$ since only vertex 8 has changed its distance to vertex 0. Differently, in Case (b), as can be observed in the figure, the original graph will become two connected components rooted at vertices 6 and 9, respectively. In this case, it is obvious that we have $AV_{(6,9)}(6) = \{6, 7, 8, 0, 3, 4, 2, 5\}$ and $AV_{(6,9)}(9) = \{9, 10\}$.

4.3 Relabeling: Supplemental Index Construction

After identifying all affected vertices, we can start relabeling the affected vertices in order for fast computation of shortest path distances on the graph with single-edge failures. Only supplemental indexes will be created, i.e., only changed distance information will be captured in supplemental indexes. All the unchanged distance information will be still computed using the original indexes (such as the distance labeling in Table 1 for the example graph in Figure 1). We develop two relabeling algorithms for the supplemental index construction, namely the BFS AFF algorithm and the BFS ALL algorithm. Detailed descriptions of these two algorithms are presented in the following.

4.3.1 BFS AFF algorithm

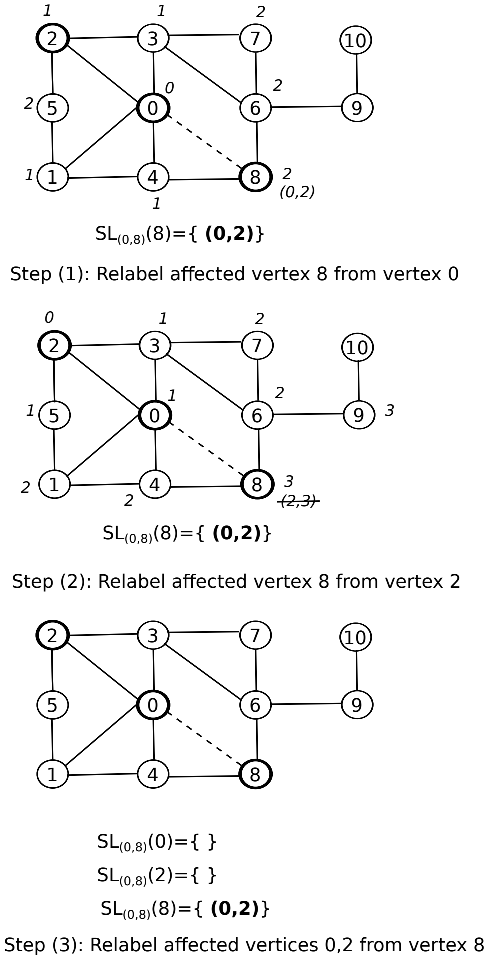
The BFS AFF algorithm relabels affected vertices using the traditional BFS algorithm. The BFS AFF algorithm uses a *late label-pruning* strategy which can save memory usage during the relabeling process. The detail steps are shown in Algorithm 2. To help understand the main idea of the BFS AFF algorithm, Figure 3 also depicts an example of the supplemental index construction process using the BFS AFF algorithm.

The failed edge is (0,8) in this example and there are three steps in Figure 3. Each step relabels one affected vertex. At Step (1), BFS AFF algorithm performs BFS from vertex 0. The number beside each node is the distance from that node to the BFS root, vertex 0. In this step, vertex 8 is the only affected vertex in $AV_{(0,8)}(8)$ that has larger vertex order than vertex 0. Therefore, the BFS process starting from vertex 0 will stop at distance 2 and will not examine vertices 9 and 10. After the BFS process stops, we add a supplemental label entry to the supplemental label of vertex 8, resulting in $SL_{(0,8)}(8) = \{(0, 2)\}$. At Step (2), BFS process starts from vertex 2. Note that, the distance information has been discarded at this step. Similarly, vertex 8 is the only affected vertex in $AV_{(0,8)}(8)$ that has larger vertex order than vertex 2. Then the BFS process starting from vertex 2 will stop at distance 3. Then we may want to add another label entry $(2, 3)$ into $SL_{(0,8)}(8)$. But based on the original index shown in Table 1 and the current supplemental label $SL_{(0,8)}(8) = \{(0, 2)\}$, we find that $(2, 3)$ is a redundant label entry in $SL_{(0,8)}(8) = \{(0, 2)\}$ since the distance between vertex 2 and vertex 8 can be computed based on $SL_{(0,8)}(8) = \{(0, 2)\}$ and the original index in Table 1. We call this the late-pruning strategy. Finally, at Step (3), the BFS process will start from vertex 8. However, since no vertex in $AV_{(0,8)}(0)$ has smaller vertex order than vertex 8, no label entry will be added to the supplemental index at this step. The final supplemental index that is constructed

Algorithm 2 BFS AFF algorithm

```
Input: $G$, $(u,v)$, $AV_{(u,v)}(u)$, $AV_{(u,v)}(v)$
Output: The supplemental index $SI_u$ and $SI_v$ for the edge
    failure case of $(u,v)$
$G' \leftarrow G - \{(u,v)\}$
//Construct $SI_u$ for vertices in $AV_{(u,v)}(u)$
$SI_u \leftarrow \emptyset$
for all $r \in AV_{(u,v)}(u)$ (in ascending vertex order) do
    Initialize supplemental label for $r$: $SL \leftarrow \emptyset$
    Start BFS algorithm to compute all the distances from
    $r$ to any vertices in $AV_{(u,v)}(v)$ that have larger vertex
    order than $\sigma(r)$
    for all vertex t in $AV_{(u,v)}(v)$ that has $\sigma(t) > \sigma(r)$ do
        if $(t, d_{G'}(t,r))$ is not a redundant label entry in $SL$
        then
            $SL \leftarrow SL \cup (t, d_{G'}(t,r))$
    $SI_u \leftarrow SI_u \cup (r, SL)$
//Construct $SI_v$ for vertices in $AV_{(u,v)}(v)$
$SI_v \leftarrow \emptyset$
for all $r \in AV_{(u,v)}(v)$ (in ascending vertex order) do
    Initialize supplemental label for $r$: $SL \leftarrow \emptyset$
    Start BFS algorithm to compute all the distances from
    $r$ to any vertices in $AV_{(u,v)}(u)$ that have larger vertex
    order than $\sigma(r)$
    for all vertex t in $AV_{(u,v)}(u)$ that has $\sigma(t) > \sigma(r)$
    do
        if $(t, d_{G'}(t,r))$ is not a redundant label entry in $SL$
        then
            $SL \leftarrow SL \cup (t, d_{G'}(t,r))$
    $SI_v \leftarrow SI_v \cup (r, SL)$
```

for the failed edge $(0,8)$ on the graph shown in Figure 1 is shown at Step (3). We will show later in Section 4.4 that such supplemental index is adequate for distance query evaluation.



$SL_{(0,8)}(8)=\{$ **(0,2)**$\}$

Step (1): Relabel affected vertex 8 from vertex 0

$SL_{(0,8)}(8)=\{$ **(0,2)**$\}$

Step (2): Relabel affected vertex 8 from vertex 2

$SL_{(0,8)}(0)=\{\ \}$

$SL_{(0,8)}(2)=\{\ \}$

$SL_{(0,8)}(8)=\{$ **(0,2)**$\}$

Step (3): Relabel affected vertices 0,2 from vertex 8

Figure 3: Supplemental index construction: BFS AFF on failed edge $(0,8)$

4.3.2 *BFS ALL algorithm*

The BFS ALL algorithm is very similar to the BFS AFF algorithm. The major diference is that the BFS ALL algorithm uses an *early label-pruning* strategy which consumes more memory during the relabeling process but gains acceleration of the relabeling process. The detail steps are shown in Algorithm 3. Figure 4 also depicts an example of the supplemental index construction process using the BFS ALL algorithm.

The failed edge is also (0,8) in this example and there are three steps in Figure 4. The main difference between BFS ALL an BFS AFF algorithms is that, in the BFS ALL algorithm, the distance information will be kept at each BFS step, using a set of temporary labels stored in TL. This distance information in TL can be used to prune label entries at the later BFS steps of the index construction process for all vertices in the graph and some vertices can be pruned during a BFS process. For example, at Step (2) in Figure 4, the number of vertices we need to visit (the vertices with bold label entries) is only seven, while at Step (2) in Figure 3, that number is 10 (by counting the vertices with distance information). Therefore, three vertices can be pruned at Step (2) in the BFS ALL algorithm. We call this the early-pruning strategy. It is obvious that the BFS ALL algorithm introduces more memory usage since BFS ALL has TL while BFS AFF does not. But the benefit of TL is that it can prune vertices at an early stage, and as will be shown in Section 5, this can speed up the BFS process greatly. Nevertheless, the final supplemental index constructed by the BFS ALL algorithm is the same as that constructed by the BFS AFF algorithm as the construction of SI_u and SI_v in both algorithms is the same.

4.4 Distance Query Evaluation on SIEF

For each single-edge failure case, we classify all possible distance queries into different types. Suppose the graph is G, the original labeling index is L, the failed edge is (u,v), the affected vertices are in $AV_{(u,v)}(u)$ and $AV_{(u,v)}(v)$, and the supplemental index is $SI_{(u,v)}$ (here, $SI_{(u,v)} = SI_u \cup SI_v$). We also denote $G' = G - \{(u,v)\}$. Given any pair of vertices s,t, we would like to compute the distance between s,t on G', denoted as $d_{G'}(s,t)$. Then we have the following different cases:

- Case 1: $s \notin AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$ and $t \notin AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$
- Case 2: $s \notin AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$ and $t \in AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$, or similarly, $s \in AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$ and $t \notin AV_{(u,v)}(u) \cup AV_{(u,v)}(v)$

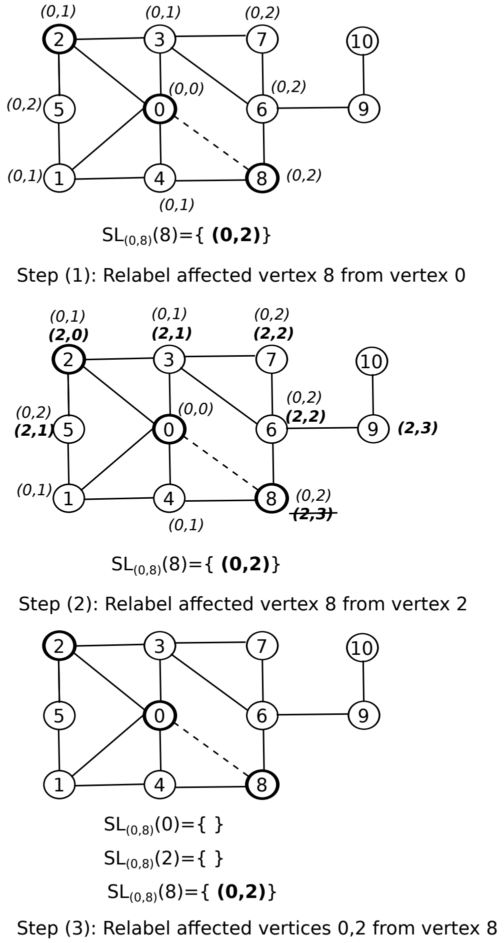
```
Algorithm 3 BFS ALL algorithm
Input: $G$, $(u,v)$, $AV_{(u,v)}(u)$, $AV_{(u,v)}(v)$
Output: The supplemental index $SI_u$ and $SI_v$ for the edge
    failure case of $(u,v)$
$G' \leftarrow G - \{(u,v)\}$
//Construct $SI_u$ for vertices in $AV_{(u,v)}(u)$
$SI_u \leftarrow \emptyset$
Initialize temporary labels $TL \leftarrow \emptyset$
for all $r \in AV_{(u,v)}(u)$ (in ascending vertex order) do
    Initialize supplemental label for $r$: $SL \leftarrow \emptyset$
    Start BFS algorithm to compute all the distances from
    $r$ to any vertices in $AV_{(u,v)}(v)$ that have larger vertex
    order than $\sigma(r)$ and record all temporary labels for all
    encountered vertices in $TL$; during the BFS process,
    if a new temporary label entry for a vertex $w$ is redundant
    in $TL$, all neighbor vertices of $w$ can be ignored
    by BFS
    for all vertex t in $AV_{(u,v)}(v)$ that has $\sigma(t) > \sigma(r)$ and
    has been searched by the above BFS process do
        if $(t, d_{G'}(t,r))$ is not a redundant label entry in $SL$
        then
            $SL \leftarrow SL \cup (t, d_{G'}(t,r))$
    $SI_u \leftarrow SI_u \cup (r, SL)$
//Construct $SI_v$ for vertices in $AV_{(u,v)}(v)$
$SI_v \leftarrow \emptyset$
Initialize temporary labels $TL \leftarrow \emptyset$
for all $r \in AV_{(u,v)}(v)$ (in ascending vertex order) do
    Start BFS algorithm to compute all the distances from
    $r$ to any vertices in $AV_{(u,v)}(u)$ that have larger vertex
    order than $\sigma(r)$ and record all temporary labels for
    all encountered vertices in $TL$; during the BFS process,
    if a new temporary label entry for a vertex $w$ is
    redundant in $TL$, then all neighbor vertices of $w$ will
    not be searched by BFS
    for all vertex t in $AV_{(u,v)}(u)$ that has $\sigma(t) > \sigma(r)$
    and has been searched by the above BFS process do
        if $(t, d_{G'}(t,r))$ is not a redundant label entry in $SL$
        then
            $SL \leftarrow SL \cup (t, d_{G'}(t,r))$
    $SI_v \leftarrow SI_v \cup (r, SL)$
```

- Case 3: $s \in AV_{(u,v)}(u)$ and $t \in AV_{(u,v)}(u)$, or similarly, $s \in AV_{(u,v)}(v)$ and $t \in AV_{(u,v)}(v)$
- Case 4: $s \in AV_{(u,v)}(u)$ and $t \in AV_{(u,v)}(v)$, or similarly, $s \in AV_{(u,v)}(v)$ and $t \in AV_{(u,v)}(u)$

Case 1 is trivial and we must have $d_{G'}(s,t) = d_G(s,t) = dist(s,t,L)$.

In Case 2 and in Case 3, according to Lemma 6 and the definition of affected vertices (see analysis in Section 4.2), we must also have $d_{G'}(s,t) = d_G(s,t) = dist(s,t,L)$.

In Case 4, suppose $s \in AV_{(u,v)}(u)$ and $t \in AV_{(u,v)}(v)$ (the other case can be analyzed in the same way). Obviously, distance between s and t changes to a larger value due to the failed edge. If s and t become disconnected to each other in G', both will not have labels in $SI_{(u,v)}$, then we have $d_{G'}(s,t) = \infty$. If s and t is still connected in G' and without loss of generality, suppose the vertex order is $\sigma(s) < \sigma(t)$, then at least vertex t contains supplemental label entries. This is because in both the BFS AFF algorithm and the BFS ALL algorithm, the affected vertex with minimum vertex order in $AV_{(u,v)}(u)$ (which is at most $\sigma(s)$) must produce one supplemental label entry for vertex t in $SI_{(u,v)}$ (see Lemma 3 for related details). For vertex s itself, if it does not produce any supplemental label entry for vertex t in $SI_{(u,v)}$, then it must be because the produced label entry is a redundant label. This means, in either case, the label entries of the supplemental label for vertex t in $SI_{(u,v)}$ must already contain adequate distance information for the computation of $d_{G'}(s,t)$. For example, to calculate $d_{G'}(2,8)$ in Figure 4, $SL_{(0,8)}(8) = \{(0,2)\}$ combining with $L(2) = \{(0,1)\ (2,0)\}$ in Table 1 is adequate and we can see that $d_{G'}(2,8) = 1 + 2 = 3$.



Step (1): Relabel affected vertex 8 from vertex 0

Step (2): Relabel affected vertex 8 from vertex 2

Step (3): Relabel affected vertices 0,2 from vertex 8

Figure 4: Supplemental index construction: BFS ALL on failed edge $(0,8)$

4.5 Some Remarks

Initial Index Construction. Pruned Landmark Labeling (PLL) technique presented in [2] is a state-of-the-art indexing technique for large static graphs. Indexes constructed by PLL [2] already have well-ordering property defined in Section 3. Therefore we use indexes constructed by PLL as the initial indexes for all original graphs in our experiments.

Time Complexity. Our algorithms can be directly applied on indexes constructed by PLL. Let w be the tree width [2] of G, n be the number of vertices and m be the number of edges in G. Also let (u,v) be the failed edge. If let $p = |AV_{(u,v)}(u) \cup$

$AV_{(u,v)}(v)|$ for each single-edge failure case (on average), the time complexity of the BFS AFF algorithm is $O(pn + pm)$ as it requires to perform p times BFS to compute SI_u and SI_v. Further, according to analysis of PLL in [2], the number of label entries per vertex is $O(w \log n)$. Then the time complexity of the BFS ALL algorithm is $O(nw \log n + p^2 w \log n)$, where $O(nw \log n)$ is the time upper bound to build temporary labels TL (note that p BFS rounds are enough to build the TL index that contains at most $nw \log n$ label entries) and $O(p^2 w \log n)$ is the time upper bound for redundancy tests.

5. EXPERIMENTS

We evaluated the performance of our proposed SIEF approach and this section reports the results. All experiments were performed under Linux (Ubuntu 10.04) on a server provided by eResearch SA[2]. The server was running on Dell R910 with 32 processing cores (four 8-core Intel Xeon E7-8837 CPUs at 2.67 GHz), 1024 GB main memory and 3 TB local scratch disk. All methods were implemented in C++ (the code of PLL [2] was obtained from the first author's code repository on GitHub[3]) using the same gcc compiler (version 4.4.6) with the optimizer option O3. It is worth mentioning that although we have a large amount of main memory on the server, the memory usage of our approach is in fact quite small and as observed during our experiments, the memory usage was within 12 GB for all datasets.

5.1 Datasets

Table 2 lists the six real-world datasets used in our experiments, which are briefly introduced as follows:

- `Gnutella` is a snapshot of the Gnutella peer-to-peer file sharing network collected in August 2002. Vertices represent hosts in the Gnutella network topology and edges represent connections between the hosts.
- The dataset `Facebook` consists of *circles* (or *friends lists*) from Facebook, which were collected from survey participants using a Facebook app called `Social Circles`.
- `Wiki-Vote` contains all Wikipedia voting data from the inception of Wikipedia till January 2008.
- `Oregon` is a graph of Autonomous Systems (AS) peering information inferred from Oregon route-views on May 26 2001.
- `Ca-HepTh` collaboration network of Arxiv High Energy Physics Theory category (there is an edge if authors coauthored at least one paper). The data covers papers in the period from January 1993 to April 2003 (124 months).
- `Ca-GrQc` collaboration network of Arxiv General Relativity category. Like `Ca-HepTh`, the data covers papers in the period from January 1993 to April 2003 (124 months).

More details on these datasets can be found at the Stanford Network Analysis Project website[4]. Similar to [3, 2], we treat all graphs as undirected, unweighted graphs.

It should be noted that, in Table 2, $|V|$ refers to the number of vertices and $|E|$ refers to the number of edges. In addition, IT denotes the indexing time or index construction time (in seconds) and LN denotes the average number of label entries of each vertex. We obtained these IT and LN results by using the Pruned Landmark Labeling (PLL) technique presented in [2]. As mentioned, we applied our index construction algorithms directly on the indexes constructed by PLL in our experiments.

Table 2: Real-world Datasets and Their Statistics

Dataset	|V|	|E|	IT (s)	LN
Gnutella	6,301	20,777	0.825	163.647
Facebook	4,039	88,234	0.173	25.887
Wiki-Vote	7,115	103,689	0.525	69.915
Oregon	11,174	23,409	0.080	11.189
Ca-HepTh	9,877	51,971	0.557	75.311
Ca-GrQc	5,242	28,980	0.141	43.828

5.2 Performance Evaluation

We have conducted extensive experiments to validate our proposed approach. In the experiments, we compared the numbers of affected vertices (Section 5.2.3), the average label entry numbers with and without considering edge failures (Section 5.2.1). We performed queries with and without SIEF indexes (Section 5.2.4) and great efficiency improvement was observed if using SIEF indexes. We also studied the impact of our approach in terms of index size, identification time, and relabeling time for each dataset (Section 5.2.2 to 5.2.6). Note that, we construct SIEF indexes by computing supplemental indexes for all single-edge failure cases of a given graph.

5.2.1 Supplemental Label Entry Numbers

Figure 5 shows the difference between the original label entry number (OLEN) without support of single-edge failures and the supplemental label entry number (SLEN) with support of single-edge failures. SLEN and OLEN of `Wiki-Vote`[5] have the biggest gap, i.e., the ratio of SLEN to OLEN is observed around 80. SLEN and OLEN of `Facebook` have the second biggest gap and the ratio of SLEN to OLEN is around 40. For other datasets, the ratios of SLEN to OLEN are all under 10. This means, compared with the total number of label entries needed for the original graphs without considering edge failures, in the case of edge failures, the total extra number of label entries (in supplemental indexes) is less than 10 times of the number of the label entries in the original index. These results indicate that the SIEF indexes are very compact.

5.2.2 Index Size

Figure 6 shows the original index size for the graphs with no failed edges and the supplemental index size when considering edge failures. The sum of the original index size and the supplemental index size is the total index size for handling shortest path distances on graphs with all single-edge failure cases. From the figure, the `Gnutella` dataset shows comparatively smaller proportion of its supplemental index over its total index size while the `Facebook` dataset shows

[2] http://www.ersa.edu.au/

[3] https://github.com/iwiwi/pruned-landmark-labeling

[4] http://snap.stanford.edu/

[5] We use the first three letters in the names of each dataset (e.g., `Wik` for `Wiki-vote`) for better illustration in the figure.

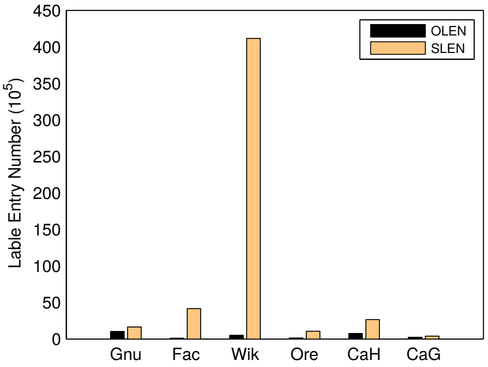


Figure 5: Comparisons between supplemental label entry numbers (SLENs) and original label entry numbers (OLENs)

largest proportion of its supplemental index over the related total index size. The `Wiki-Vote` dataset has the largest supplemental index size due to the fact that each single-failure case incurs a large number of affected vertices as well as a relatively large number of supplemental label entries (for more details, please see Table 3).

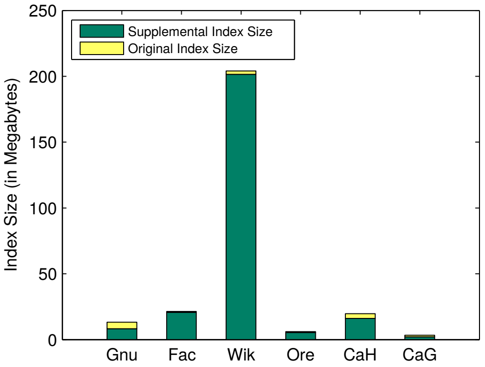


Figure 6: Index Size

5.2.3 *Affected Vertices*

Table 3 presents the relationship between affected vertices and average supplemental label entry number. *Avg* $|AU|/|V|$ represents the average percentage of affected vertices in a single-edge failure case, showing the impact of a single-edge failure on a graph. It is also the average proportion of affected vertices of the original graphs. *Avg* $|AU|$ represents the average number of affected vertices from the graph and *Avg SLEN* denotes the average number of supplemental label entries in a single-edge failure case.

From the table, we can see that the smallest percentage and the smallest average number of affected vertices are both observed in the `Ca-GrQc` dataset, with values of 1.486% and 77.884, respectively. We can also see from the table that the average supplemental label number decreases (or increases) together with the average number of the affected vertices. Also around 36% of vertices are affected in the `Wiki-Vote` dataset, which is the largest proportion. The largest average number of affected vertices is observed in the `Oregon` dataset, which is around 2,861 affected vertices for one failed edge. However, no clear linear relationship is found between the two. The largest gap occurs in the `Oregon` dataset, which indicates that the label pruning process on the affected vertices is quite powerful, leading to much fewer label entries per affected vertex. Meanwhile, the smallest gap happens in the `Gnutella` dataset and this indicates that label pruning is not very effective in this dataset.

Note that, although the proportion of affected vertices for a single-edge failure case could be large, as having been clarified in Figure 6, the final SIEF index for all single-edge failure cases is still of moderate sizes compared with the original index.

Table 3: Affected Vertices

Dataset	Avg $\|AU\|/\|V\|$	Avg $\|AU\|$	Avg SLEN
Gnutella	6.053%	381.386	78.445
Facebook	16.099%	650.241	47.042
Wiki-Vote	35.841%	2,550.090	396.971
Oregon	25.605%	2,861.070	45.323
Ca-HepTh	2.743%	270.881	51.095
Ca-GrQc	1.486%	77.884	13.064

5.2.4 *Query Time*

Table 4 shows the average BFS query time and the average SIEF query time. The former represents query time without using indexes proposed in this work, while the latter represents the query time when using SIEF indexes. From the table, we can see that the difference for `Oregon` dataset is the least, which still achieves at least 40 times faster when using SIEF indexes compared with the traditional BFS query approach. The largest gap occurs in the `Facebook` dataset, where the average BFS query time is around 500 times more than the SIEF query time. These results show that when using SIEF indexes, the query efficiency can be improved significantly and the query response times are normally no more than 5 μs. As mentioned in Section 4, we use supplemental indexes to support edge failures, the query process needs to examine the supplemental indexes first. When examining the supplemental indexes, SIEF checks whether the querying source and querying destination are both affected vertices given the edge failure constraint using binary search strategy. Based on the searching result, SIEF knows whether we can compute the shortest path distance based only on the supplemental indexes or based only on the original indexes. Nevertheless, the querying process is still much faster. The main reason is that the number of affected vertices for each single-edge failure case is typically small (more details are presented in Section 5.2.3) and hence the binary search process finishes quickly. This results in fast query responses in SIEF.

5.2.5 *Identification Time*

Table 5 shows the total time for identifying affected vertices for all single-edge failure cases. From the figure, we can see that, for the most datasets, the identification process can be done fairly fast and is normally finished within 80 seconds. The exception is `Wiki-Vote`, which requires a bit more than 600 seconds. The fast identification time is

Table 4: Average Query Time

Dataset	BFS Query Time	SIEF Query Time
Gnutella	140.329 μs	0.452 μs
Facebook	243.060 μs	0.522 μs
Wiki-Vote	284.867 μs	1.100 μs
Oregon	163.465 μs	4.985 μs
Ca-HepTh	325.196 μs	0.689 μs
Ca-GrQc	159.412 μs	0.479 μs

mainly because the affected vertices can be identified in a BFS manner and we only need to examine the distances between the affected vertices to one of the end vertices of a failed edge.

Table 5: Average Identification Time

Dataset	Identification Time
Gnutella	43.3708 s
Facebook	80.6844 s
Wiki-Vote	612.522 s
Oregon	35.6307 s
Ca-HepTh	36.2022 s
Ca-GrQc	4.32942 s

5.2.6 *Labeling Time*

Figure 7 shows the time for relabeling the affected vertices, which need extra distance label information to maintain correct distances to some other vertices due to a single-edge failure. Here, we used the estimated time for naive method (shown as "Estd Time for Naive Method" in the figure) as the baseline. The naive method refers to the method that we recompute a complete distance labeling index for each single-edge failure case. The process of labeling a new graph with a single-edge failure should be almost the same as the process of labeling the original graph. Therefore, the total labeling time of the naive method can be estimated by multiplying the total edge number in the original graph, i.e., the total number of single-edge failure cases, with the index time of the original graph (see Table 2).

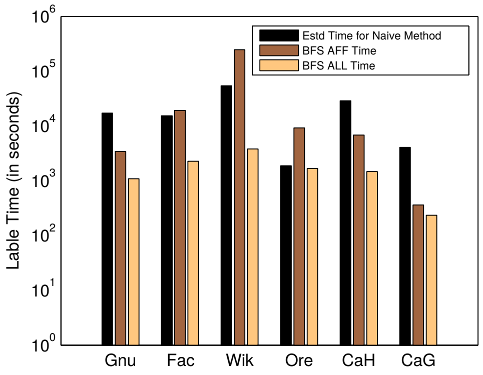


Figure 7: Labeling Time

Then, we compared the labeling times of the naive method and the two labeling methods proposed in our work: BFS AFF and BFS ALL. Recall that BFS AFF uses a late-label-pruning strategy and avoids labeling any unaffected vertices while BFS ALL uses an early-label-pruning strategy which needs labeling the unaffected vertices. From the figure, we can see that for some datasets, such as `Gnutella`, `Ca-HepTh` and `Ca-GrQc`, BFS AFF outperforms the naive method because the label-pruning process incurs some overhead when labeling unaffected vertices compared with the pure BFS process. However, BFS AFF is beaten by the naive method in terms of labeling time for other datasets, including `Facebook`, `Wiki-Vote` and `Oregon`, which contain a large number of vertices and/or a large number of edges. Hence, the late-label-pruning strategy in BFS AFF does not work well on all datasets. These results indicate that although label-pruning incurs some overhead on top of the BFS process, the label-pruning approach is quite effective in some datasets, especially datasets with more vertices and edges.

In contrast, BFS ALL performs the best on all datasets. For some datasets, such as `Facebook`, `Wiki-Vote` and `Ca-Hepth`, BFS ALL even performs orders of magnitude faster than both the naive method and the BFS AFF method. This confirms that the early-label-pruning strategy works very well on various datasets and the overhead on labeling unaffected vertices can be ignored due to the substantial label-pruning power it brings (for more details, please refer to Section 4.3).

6. CONCLUSION

This paper has studied the problem of computing the shortest path distance on graphs with single-edge failures based on 2-hop distance labeling techniques. The concept of well-ordering 2-hop distance labeling and its properties have been defined and analyzed. We have particularly focused on the constructions of compact distance labeling for all possible single-edge failure cases, a challenging problem that remains open, to the best of our knowledge. A generic framework, SIEF, has been designed for this purpose. Based on the most recent technique Pruned Landmark Labeling (PLL) [2] that handles only static graphs, we have implemented an extended version using the SIEF framework developed in this paper. Extensive experiments have also been performed on six real-world graphs to confirm its effectiveness and efficiency. SIEF is able to support compact index construction for all single-edge failure cases on graphs efficiently. Specifically, the SIEF index size is comparable to that of the indexes constructed for original static graphs, which is very compact. SIEF can answer distance queries with edge failure constraints several orders of magnitude faster than traditional Breadth-First-Search (BFS) algorithms.

In our future work, we will further investigate several aspects of answering distance queries on graphs with edge failures. The first one centers on how to support distance queries with more complex edge failure constraints, i.e., dual-failure on edges. The second aspect is to further speed up the index construction process in order to process larger graphs. Finally, it is also interesting to investigate the problem of answering distance queries on graphs with node failures, which is even more challenging than edge failures.

7. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. Hierarchical Hub Labelings for Shortest

Paths. In *Proc. of the 20th Annual European Symposium on Algorithms (ESA 2012)*, pages 24–35, Ljubljana, Slovenia, 2012.

[2] T. Akiba, Y. Iwata, and Y. Yoshida. Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*, pages 349–360, New York, NY, USA, 2013.

[3] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proc. of the 23rd International World Wide Web Conference (WWW 2014)*, pages 237–248, Seoul, Republic of Korea, 2014.

[4] T. Akiba, C. Sommer, and K. Kawarabayashi. Shortest-Path Queries for Complex Networks: Exploiting Low Tree-Width Outside the Core. In *Proc. of the 15th International Conference on Extending Database Technology, (EDBT 2012)*, pages 144–155, Berlin, Germany, 2012.

[5] S. Baswana, U. Lath, and A. S. Mehta. Single source distance oracle for planar digraphs avoiding a failed node or link. In *Proc. of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 223–232, 2012.

[6] C. Bazgan, S. Toubaline, and D. Vanderpooten. Efficient Algorithms for Finding the k Most Vital Edges for the Minimum Spanning Tree Problem. In *Proc. of the 5th International on Conference Combinatorial Optimization and Applications (COCOA 2011)*, pages 126–140, 2011.

[7] R. Bramandia, B. Choi, and W. K. Ng. Incremental Maintenance of 2-Hop Labeling of Large Graphs. *IEEE Trans. Knowl. Data Eng.*, 22(5):682–698, 2010.

[8] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao. The exact distance to destination in undirected world. *VLDB J.*, 21(6):869–888, 2012.

[9] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient Processing of Distance Queries in Large Graphs: A Vertex Cover Approach. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2012)*, pages 457–468, Scottsdale, AZ, USA, 2012.

[10] J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT*, pages 481–492, 2009.

[11] A. Ciortea, O. Boissier, A. Zimmermann, and A. M. Florea. Reconsidering the social web of things: position paper. In *Proc. the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2013) (Adjunct Publication)*, pages 1535–1544, Zurich, Switzerland, 2013.

[12] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 937–946, San Francisco, CA, USA, 2002.

[13] R. Duan and S. Pettie. Dual-failure distance and connectivity oracles. In *Proc. of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2009)*, pages 506–515, 2009.

[14] H. K. Farsani, M. A. Nematbakhsh, and G. Lausen. SRank: Shortest paths as distance between nodes of a graph with application to RDF clustering. *J. Information Science*, 39(2):198–210, 2013.

[15] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. IS-LABEL: an Independent-Set based Labeling Scheme for Point-to-Point Distance Querying. *Proc.of the VLDB Endowment*, 6(6):457–468, 2013.

[16] J. Hershberger and S. Suri. Vickrey Prices and Shortest Paths: What is an Edge Worth? In *Proc. of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*, pages 252–259, 2001.

[17] K. Iwano and N. Katoh. Efficient Algorithms for Finding the Most Vital Edge of a Minimum Spanning Tree. *Inf. Process. Lett.*, 48(5):211–213, 1993.

[18] R. Jin, N. Ruan, Y. Xiang, and V. E. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2012)*, pages 445–456, Scottsdale, AZ, USA, 2012.

[19] P. K., S. P. Kumar, and D. Damien. Ranked answer graph construction for keyword queries on RDF graphs without distance neighbourhood restriction. In *Proc. of the 20th International Conference on World Wide Web (WWW 2011, Companion Volume)*, pages 361–366, Hyderabad, India, 2011.

[20] R. Schenkel, A. Theobald, and G. Weikum. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *Proc. of the 21st International Conference on Data Engineering (ICDE 2005)*, pages 360–371, Tokyo, Japan, 2005.

[21] S. Vassilvitskii and E. Brill. Using Web-Graph Distance for Relevance Feedback in Web Search. In *Proc. of the 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR 2006)*, pages 147–153, Seattle, Washington, USA, 2006.

[22] K. Wehmuth and A. Ziviani. DACCER: Distributed Assessment of the Closeness CEntrality Ranking in complex networks. *Computer Networks*, 57(13):2536–2548, 2013.

[23] F. Wei. TEDI: efficient shortest path query answering on graphs. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2010)*, pages 99–110, Indianapolis, Indiana, USA, 2010.

[24] K. Xie, K. Deng, S. Shang, X. Zhou, and K. Zheng. Finding Alternative Shortest Paths in Spatial Networks. *ACM Trans. Database Syst.*, 37(4):29, 2012.

[25] L. Yao and Q. Z. Sheng. Exploiting Latent Relevance for Relational Learning of Ubiquitous Things. In *Proc. of the 21st ACM International Conference on Information and Knowledge Management (CIKM 2012)*, Maui, Hawaii, USA, 2012.

[26] A. D. Zhu, X. Xiao, S. Wang, and W. Lin. Efficient Single-Source Shortest Path and Distance Queries on Large Graphs. In *Proc. of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2013)*, pages 998–1006, Chicago, IL, USA, 2013.