Efficient Top-k Closeness Centrality Search *

Paul W. Olsen Jr. #1, Alan G. Labouseur #2, Jeong-Hyon Hwang #3

Department of Computer Science, University at Albany – State University of New York
1400 Washington Avenue, Albany, NY 12222, USA

[1] polsen@cs.albany.edu
[2] alan@cs.albany.edu
[3] jhh@cs.albany.edu

***Abstract*—Many of today's applications can benefit from the discovery of the most central entities in real-world networks. This paper presents a new technique that efficiently finds the k most central entities in terms of closeness centrality. Instead of computing the centrality of each entity independently, our technique shares intermediate results between centrality computations. Since the cost of each centrality computation may vary substantially depending on the choice of the previous computation, our technique schedules centrality computations in a manner that minimizes the estimated completion time. This technique also updates, with negligible overhead, an upper bound on the centrality of every entity. Using this information, our technique proactively skips entities that cannot belong to the final result. This paper presents evaluation results for actual networks to demonstrate the benefits of our technique.**

I. Introduction

Consider a person planning to open a store. She would prefer a location closest, on average, to a large number of potential customers [1]. In the case of viral marketing, it is crucial to find a small number of people who can trigger the largest and fastest product adoption through social contact advertising [2]. In sociopolitical science and health care, researchers strive to understand opinion formation and disease propagation with a focus on the most influential and central people [3]. Other applications that benefit from the discovery of highly central entities in real-world networks include national security, power grid administration, policy making, and computer network management.

Each of the above networks can be represented as a graph G with a set V of vertices that represent entities and another set E of edges that represent relationships between entities. In this paper, we study the problem of finding, given a graph $G(V, E)$ and a positive integer k, the k most central vertices in G. We focus on one popular centrality metric, *closeness centrality* [4], [5], [6]. In terms of this metric, a vertex is highly central if it has paths to a large number of other vertices and the average shortest path length to these vertices is small (Section II). In a graph representing a road network, a vertex with the highest closeness centrality corresponds to a location closest, on average, to all other locations. In the context of viral marketing, such a vertex represents a person who, with the smallest number of intermediaries on average, can influence the greatest number of people. As we discuss in Section VII, other types of centrality can be efficiently computed (e.g., degree centrality, PageRank) or have unique limitations and complexities (e.g., eccentricity, betweenness). We leave the extension of our work to the latter as future research.

To find the k vertices with the highest closeness centrality values, one may consider constructing a distance matrix that stores the shortest distance for all pairs of vertices [7], [8] and then using that matrix to calculate the centrality of each vertex. This approach requires $\Omega(|V|^2)$ space and is thus impractical for large graphs (e.g., several terabytes of memory for a graph with 1 million vertices). A more practical approach is to compute the centrality of each vertex using a single-source shortest path algorithm [9], thereby obviating the need for constructing a distance matrix. Given a directed graph with nonnegative edge weights, this approach can complete in $O(|V| \cdot |E| + |V|^2 \log |V|)$ time with only $O(|V| + |E|)$ space by repeating an implementation of Dijkstra's algorithm [9] using a Fibonacci heap [10] for each vertex. There are also techniques that can reduce centrality computation time at the expense of accuracy [11], [12], [13]. These approximation techniques, however, support only *undirected* graphs [11], [13] or *unweighted* graphs [12]. They are also unsuitable when correct answers are required (e.g., the centrality measure is used to select the best paper over the last 10 years).

In this paper, we present a new solution to the aforementioned problem for both *directed* and *undirected* graphs with nonnegative edge weights. In our experiments, our solution was up to 142 times faster (e.g., 28.6 minutes vs. 67.5 hours) than the conventional approach which computes the centrality of every vertex independently. In one case, when the graph size was increased by a factor of 16, the completion time of our solution increased only by a factor of 46 (= $16^{1.38}$) while that of the conventional approach showed a super-quadratic increase (339 = $16^{2.1}$). Our technique also quickly generates *approximate answers* and then gradually refines them until it produces final, correct results. In most of our experiments, approximately 73% of vertices in the initial, approximate answers were correct (i.e., kept in the final results). Our solution also uses only $O(|V| + |E|)$ space.

A key principle of our approach is to materialize intermediate results while the centrality of a vertex is computed, and then reuse those results while the centrality of another vertex is computed. Our technique can apply this type of *sharing* to any pair of vertices with an edge between them. The cost of computing a vertex's centrality in this way heavily depends on the choice of the previous vertex. Therefore, our technique *schedules* (i.e., determines the order of) centrality computations striving to minimize the overall completion time.

* This work is supported by NSF CAREER award IIS-1149372.

To enable this optimization, our technique estimates, with low overhead, the centrality computation cost for each possible sharing scenario. This method can also estimate the centrality of each vertex, which enables early production of approximate top-k answers. Finally, our technique efficiently maintains an upper bound on the centrality of every vertex while the centrality values of other vertices are computed. Based on these bounds, it can proactively skip vertices that cannot belong to the final top-k result, thereby further improving performance.

In this paper, we make the following contributions:

- We develop a technique that efficiently shares intermediate results across centrality computations.
- We present a method for scheduling centrality computations in a manner that minimizes the estimated completion time.
- We provide an efficient technique for skipping vertices that cannot be among the k most central vertices.
- We describe a method for quickly producing and refining approximate top-k answers.
- We experimentally demonstrate the benefits of the above features using real-world networks.

The rest of this paper is organized as follows: Section II provides a formal definition of the problem mentioned above. Section III describes our approach for sharing intermediate results across centrality computations. Sections IV and V present our solutions for skipping unnecessary centrality computations and scheduling centrality computations, respectively. Section VI shows our evaluation results and Section VII summarizes related work. Section VIII concludes this paper.

II. Problem Statement

In this paper, we study the problem of finding the k most central vertices in a graph $G(V, E)$ where V and E denote the set of vertices and edges, respectively. To deal with various real-world networks, we consider both *directed graphs* (e.g., citation networks) and *undirected graphs* (e.g., coauthorship networks) with arbitrary nonnegative edge weights. Hereafter, we focus on directed graphs since any undirected graph can be converted into a directed graph by replacing each undirected edge $\{x, y\}$ of weight w with two directed edges (x, y) and (y, x), each of weight w.

The closeness centrality of a vertex represents how close the vertex is, on average, to other vertices [4], [5], [6]. If graph G is strongly connected, the closeness centrality of a vertex v, denoted by $c(v)$, is defined as

$$c(v) = \frac{|V| - 1}{\sum_{v' \in V} d(v, v')} \tag{1}$$

where $d(v, v')$ denotes the geodesic (i.e., shortest) distance from vertex v to v' (Table I). Practical graphs, however, including those examined in Section VI, may not be strongly connected. Therefore, we use a more general definition of closeness centrality [14], [15]:

Symbol	Description
V	set of vertices
E	set of edges
$\mathcal{V}_v$	set of vertices reachable from v
$\mathcal{E}_v$	set of edges reachable from v
$w(v, v')$	minimum weight of the edges from vertex v to vertex v' (∞ if no such edge exists)
$d(v, v')$	geodesic distance from vertex v to vertex v' (∞ if there is no path from v to v')
$c(v)$	closeness centrality of vertex v (Definition 1)
$\mathcal{V}_{v\|p}$	$\{v\} \cup \{v' \in \mathcal{V}_v : d(v, v') < w(v, p) + d(p, v')\}$
$\mathcal{E}_{v\|p}$	set of edges that emanate from a vertex in $\mathcal{V}_{v\|p}$

TABLE I. Summary of Notation

Definition 1: (Closeness Centrality) The closeness centrality of vertex v is defined as:

$$c(v) = \frac{(|\mathcal{V}_v| - 1)^2}{(|V| - 1) \sum_{v' \in \mathcal{V}_v} d(v, v')} \tag{2}$$

where $\mathcal{V}_v$ denotes the set of vertices reachable from v and $d(v, v')$ denotes the geodesic distance from vertex v to v' (Table I).

Note that Equation (2) assigns a high centrality value to vertex v if v has a large number of reachable vertices and their geodesic distances from v are short. When G is strongly connected, Equation (2) reduces to Equation (1) since $\mathcal{V}_v = V$ for any $v \in V$.

In this paper, we consider the problem of finding a set of vertices whose centrality values are larger than or equal to the k-th highest centrality value. This set may contain more than k vertices if some of these vertices have identical centrality values. For example, if the centrality values of a, b, and c are 1, 0.9, and 0.9, respectively, and no other vertex has a higher centrality value, then our answer to a top-2 centrality problem is $\{a, b, c\}$ rather than $\{a, b\}$ or $\{a, c\}$. Such a top-k centrality problem can be formally defined as follows:

Definition 2: (Top-k Closeness Centrality Problem) Given graph $G(V, E)$, a top-k centrality problem is to find:

$$\arg\max_{V' \subseteq V,\ |V'| \geq k} (\min_{v \in V'} c(v), |V'|).$$

where the $\max$ function assumes a lexicographic order $>$ such that $(a, b) > (c, d)$ if and only if $(a > c) \vee ((a = c) \wedge (b > d))$.

In this paper, our goal is to develop solutions for quickly answering the top-k centrality problem. The key questions that we answer in Sections III, IV and V, respectively, are as follows:

- Is it possible to *share* results between centrality computations to reduce the overall completion time?
- Is it possible to find vertices that cannot be included in the final top-k result without computing their actual centrality values?
- In what order should we compute the centrality of vertices so that overall completion time is minimized?

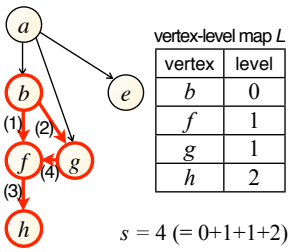
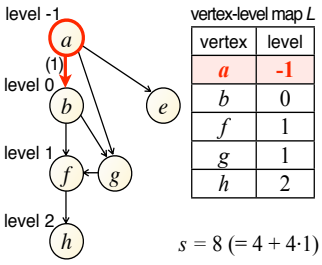
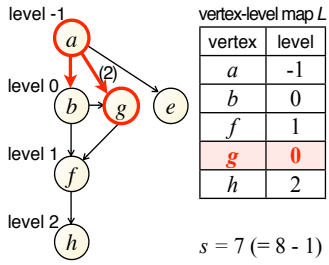


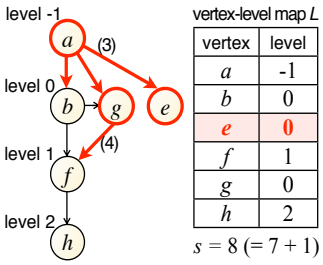
Fig. 1. PFS starts at b. Following edges in the specified order, it finds $d(b, v')$ for each $v' \in \mathcal{V}_b$.



(a) placing a at level -1 (e's level unknown)



(b) promoting g to level 0 (e's level unknown)



(c) placing e at level 0

Fig. 2. Δ-PFS starts at vertex a. Placing vertex a at level -1 and then following only the four edges labeled (1) through (4), it ensures that each vertex $v' \in \mathcal{V}_a$ is at level $d(a, v')$-1.

III. Sharing Across Centrality Computations

This section describes our technique for efficiently computing the centrality of each vertex by reusing intermediate results. Sections III-A and III-B discuss basic principles and algorithmic details of this technique, respectively.

A. Core Ideas

The closeness centrality of vertex v requires $\mathcal{V}_v$ and $d(v, v')$ for every $v' \in \mathcal{V}_v$ (Definition 1). This requirement can be met by Priority-First Search (PFS) algorithms which visit vertices in order of increasing geodesic distance from v (e.g., Dijkstra's [9] for weighted graphs and BFS [8] for unweighted graphs). Figure 1 shows an example that computes the closeness centrality of vertex b by starting a PFS at b. In this example, the weight of each edge is set to 1 to ease illustration. This PFS follows edges in the order of (1) – (4) finding that $\mathcal{V}_b = \{b, f, g, h\}$, $d(b, b) = 0$, $d(b, f) = d(b, g) = 1$, and $d(b, h) = 2$. Thus, the closeness centrality of b is computed as $c(b) = \frac{(|\mathcal{V}_b|-1)^2}{(|V|-1)\sum_{v' \in \mathcal{V}_b} d(b,v')} = \frac{(4-1)^2}{(6-1)(0+1+1+2)} = \frac{9}{20}$.

Consider starting a new PFS at a in Figure 1 to compute the centrality of vertex a. This PFS would follow all seven edges in the figure. It is possible to achieve the same effect more efficiently by leveraging the previous PFS started at b. This approach raises the challenge of identifying which past computations can be effectively reused. Our solution (i) assumes that the previous PFS placed every vertex $v' \in \mathcal{V}_b$ at level $d(b, v')$ and then (ii) places each vertex $v' \in \mathcal{V}_a$ at target level $d(a, v') - w(a, b)$ while skipping (i.e., not following the outgoing edges of) vertices already placed at the target level during the previous search. We call this solution Δ-PFS because it performs PFS while *skipping calculations done by the previous search*.

Figures 1 and 2 illustrate the core ideas mentioned above (table L and variable s are explained below). In Figure 1, PFS places b at level $d(b, b) = 0$, f at level $d(b, f) = 1$, g at level $d(b, g) = 1$, and h at level $d(b, h) = 2$. In Figure 2(a), Δ-PFS places a at level $d(a, a) - w(a, b) = -1$. Following a's outgoing edge to b, Δ-PFS finds that the previous search already placed b at the target level $d(a, b) - w(a, b) = 0$. In this case, it does not follow the outgoing edges of b since every vertex v' (e.g., f and h in Figures 1 and 2) having a shortest path from a via b is already at the target level $d(a, v') - w(a, b)$. After this step, Δ-PFS follows vertex a's edge to vertex g. In this case, Δ-PFS *promotes* g from level 1 (Figure 2(a)) to target level $d(a, g) - w(a, b) = 0$ (Figure 2(b)) and *newly places* e (which was not at any level) at level $d(a, e) - w(a, b) = 0$ (Figure 2(c)). Next, Δ-PFS follows g's outgoing edge to f finding that f is already at the target level $d(a, f) - w(a, b) = 1$. As in the case of examining b, it does not follow any outgoing edges of f. It then terminates since there are no more edges to follow.

The examples in Figures 1 and 2 also show how our technique maintains intermediate results to efficiently compute centrality values. These intermediate results are (i) a map L which contains each vertex v' reachable from the source (i.e., the vertex at which a PFS starts) and the level of v' (denoted $L[v']$) and (ii) a variable s which maintains the sum of the differences between the level of the source and the level of every vertex v' in L. For example, the PFS in Figure 1 inserts b, f, g, h and their levels into L. It also increases s by these levels since $L[v'] - L[b] = L[v'] - 0 = L[v']$ for each $v' \in \{b, f, g, h\}$. When Δ-PFS starts at a (Figure 2(a)), the source changes from b at level 0 (Figure 1) to a at level $-w(a, b) = -1$. In this case, for each $v' \in \{b, f, g, h\}$, the difference between the level of the source and the level of v' increases by 1. Thus, Δ-PFS increases s by $4 \cdot 1$. When Δ-PFS *promotes* g from level 1 (Figure 2(a)) to level 0 (Figure 2(b)), Δ-PFS decreases s by 1, the promotion distance. When Δ-PFS *newly visits* e and places it at level 0 (Figure 2(c)), $(e, 0)$ is inserted into L and s is increased by $d(a, e) = 1$ in response to this new addition.

When Δ-PFS completes (Figure 2(c)), L has entries for all of the vertices reachable from a (i.e., $|L| = |\mathcal{V}_a|$), and $s = \sum_{(v',l') \in L}(L[v'] - L[a]) = \sum_{(v',l') \in L}((d(a, v') - w(a, b)) - (d(a, a) - w(a, b))) = \sum_{v' \in \mathcal{V}_a} d(a, v')$. Using a variable tracking $|L|$, the centrality of a can be computed with negligible overhead as $\frac{(|\mathcal{V}_a|-1)^2}{(|V|-1)\sum_{v' \in \mathcal{V}_a} d(a,v')} = \frac{(|L|-1)^2}{(|V|-1)s} = \frac{(6-1)^2}{(6-1)8} = \frac{5}{8}$.

A detailed description of Δ-PFS and a proof of its correctness are included in Appendices A and B. Δ-PFS is an extension to PFS with additional features that use a map L and a variable s to quickly compute centrality values and skip calculations done by a previous search. Incremental search algorithms for dynamic graphs [16] and Δ-PFS bear similarity in that they reuse previous results. The former algorithms, however, are not applicable to searches which start at different vertices. Furthermore, they do not maintain intermediate results for fast centrality computation (e.g., L and s of Δ-PFS). In

Algorithm 1: $top\_centrality(G, k)$

input : graph $G(V, E)$, k
output : top-k list A
1 $\mathcal{S} \leftarrow schedule(G, k)$; *// Section V*
2 **for** v : $start(\mathcal{S})$ **do** *// every start vertex v*
3 $(L, s) \leftarrow$ PFS(v);
4 $process(v, L, s, A, \mathcal{S}, k)$; *// Algorithm 2*
5 **return** A;

Algorithm 2: $process(p, L, s, A, \mathcal{S}, k)$

input : vertex p, vertex-level map L, sum of geodesic distances s, top-k list A, schedule $\mathcal{S}$, k
1 $c(p) \leftarrow \frac{(|L|-1)^2}{(|V|-1)s}$; *// centrality of p (Definition 1)*
2 $update(A, p, c(p), k)$; *// update A using p, $c(p)$, and k*
3 **for** *each vertex v that follows p in schedule $\mathcal{S}$* **do**
4 $(L, s, \Lambda) = \Delta$-PFS(v, p, L, s);
5 $process(v, L, s, A, \mathcal{S}, k)$;
6 $rollback(L, \Lambda)$; *// restore state of L as of step 4*

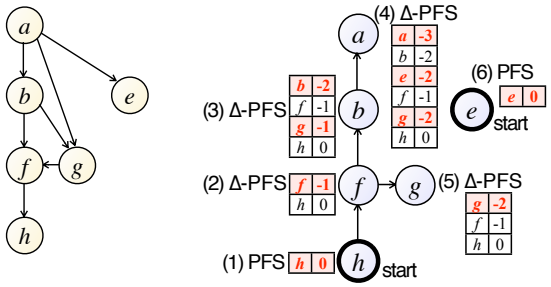
Figure 2, Δ-PFS follows only four edges rather than all seven edges. Section VI presents our experimental results where Δ-PFS achieves the effect of PFS by visiting a much smaller number (e.g., 0.1% – 20%) of vertices compared to PFS. The set of vertices visited by a Δ-PFS which starts at v and reuses the search started at p, can be expressed as $\mathcal{V}_{v|p} = \{v\} \cup \{v' \in \mathcal{V}_v : d(v, v') < w(v, p) + d(p, v')\}$ (Appendix B). While a PFS starting at v completes in $O(|\mathcal{E}_v| + |\mathcal{V}_v| \log |\mathcal{V}_v|)$ time, a Δ-PFS starting at v and reusing a search started at p completes in $O(|\mathcal{E}_{v|p}| + |\mathcal{V}_{v|p}| \log |\mathcal{V}_{v|p}|)$ where $\mathcal{V}_v$, $\mathcal{E}_v$, $\mathcal{V}_{v|p}$ and $\mathcal{E}_{v|p}$ are defined as in Table I (Appendix C).

B. Algorithmic Details

Algorithm 1 shows the overall operation of our technique. It begins by obtaining a *centrality computation schedule* S (line 1). Figure 3(b) shows an example schedule constructed for the graph in Figure 3(a). In Figure 3(b), vertex f follows h, suggesting the computation of f's centrality using a Δ-PFS which reuses a search started at h. Vertices not following any other vertex are called *start vertices* (e.g., e and h) and their centrality values are computed using PFS. Our technique for constructing such a schedule is presented in Section V.

After obtaining the schedule, Algorithm 1 picks a start vertex v (line 2) and then starts PFS at v (line 3). The PFS executed on line 3 corresponds to Dijkstra's algorithm [9] except that it (i) places each vertex $v' \in \mathcal{V}_v$ at level $d(v, v')$ while storing v' and $d(v, v')$ in L and (ii) initially sets s to 0 and then increases s by $d(v, v')$ for every $v' \in \mathcal{V}_v$, thereby ensuring that $s = \sum_{v' \in \mathcal{V}_v} d(v, v')$ when PFS completes. This PFS algorithm is presented in Appendix A. After obtaining L and s as above, the centrality values of v and its successors in the schedule are computed recursively (line 4) using Algorithm 2.

Algorithm 2 computes the centrality of vertex p using L and s (line 1) as explained in Section III-A. Next, it updates the top-k list A so that A keeps, among the vertices whose centrality values are computed, those having centrality values no less than the k-th highest centrality value (Definition 2). Then, for every vertex v that follows p in the schedule (line 3), Algorithm 2 updates L and s using Δ-PFS and returns them (line 4) in addition to a map Λ (explained below). Finally, it recursively uses Algorithm 2 to process v and its successors (line 5). The tables in Figure 3(b) illustrate how L is updated as the centrality of each vertex is computed in the order (1) – (6). In the tables for steps (2) – (5), the changed entries in L are shaded. These entries indicate the vertices that Δ-PFS visits. The rest of each table shows the benefit of Δ-PFS over PFS (i.e., the vertices that Δ-PFS skips, but PFS would visit if it were executed).



(a) Graph (b) Schedule

Fig. 3. Overview of Our Approach

Our method uses a separate map Λ (lines 4 and 6 in Algorithm 2) to handle multiple successors. For instance, vertex f in Figure 3(b) is followed by b and g. The Δ-PFS computations for b and g must use the same version of L (see table (2) in Figure 3(b)) although each of them needs to update L. Therefore, our technique updates an entry in L only after saving that entry in Λ. When we add a new vertex v' to L, we add the entry $(v', null)$ to Λ. This logging approach allows our centrality computation method to restore the previous version of L by rolling back the operations that have changed the levels of vertices (line 6 in Algorithm 2).

Our technique preserves one instance of Λ for every level of recursion (Algorithm 2). In all of our evaluations, each instance of Λ stores information about a small fraction (e.g., $< 5\%$) of vertices and the depth of recursion is relatively small (e.g., 30, given a graph containing 1,000,000 vertices). Our technique can also be extended to control memory utilization by adding code, after line 2 in Algorithm 2, that changes the current vertex p's successors to start vertices if memory utilization is higher than a threshold. In this case, because further recursion is disallowed, no more instances of Λ are created. However, the vertices that became start vertices must be processed using PFS which is usually slower than Δ-PFS.

IV. Pruning

This section presents our technique for proactively skipping vertices that cannot be included in the final top-k answer. Section IV-A provides an overview of this technique. Section IV-B describes this technique in detail.

A. Core Ideas

Suppose that our solution presented in Section III has inserted the centrality values of k vertices into top-k list A. Let $\theta(A)$ denote the minimum of the current centrality values in A. Then, any vertex f whose centrality $c(f)$ is less than $\theta(A)$

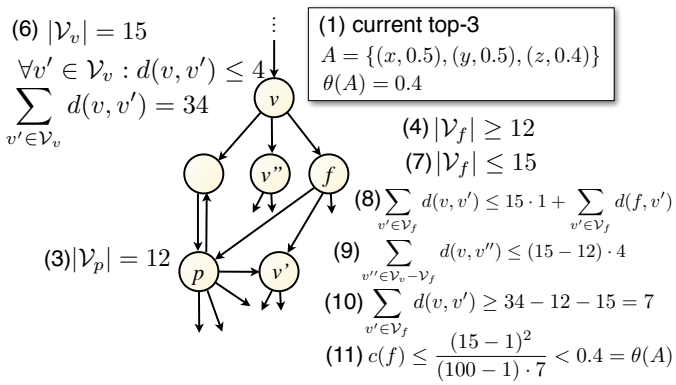


Fig. 4. Pruning Example ($|V| = 100$)

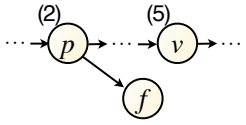


Fig. 5. Schedule before Skipping f

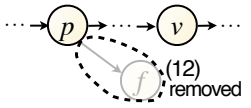


Fig. 6. Schedule after Skipping f

cannot be included in the final top-k answer. It is possible to proactively skip such a vertex f if an upper bound on $c(f)$ is found before computing $c(f)$ and if that bound is less than $\theta(A)$. For this reason, we call $\theta(A)$ the *pruning threshold*. A crucial requirement of this pruning approach is the maintenance of these bounds with low computational overhead. To effectively skip vertices, these bounds must be close to the actual centrality values.

Consider the example in Figure 4 where $\theta(A)$ is 0.4 (see (1) in Figure 4) and an upper bound on $c(f) = \frac{(|\mathcal{V}_f|-1)^2}{(|V|-1)s_f}$ is derived after obtaining an upper bound on $|\mathcal{V}_f|$ and a lower bound on $s_f = \sum_{v' \in \mathcal{V}_f} d(f,v')$. In this example, $c(p)$ is computed (3) according to the schedule in Figure 5 (2) finding that $|\mathcal{V}_p| = 12$ (i.e., 12 vertices are reachable from p). At this point, $|\mathcal{V}_p| = 12$ can be used as a lower bound on $|\mathcal{V}_f|$ for any vertex f that has a path to p (4) since $\mathcal{V}_p \subseteq \mathcal{V}_f$ (i.e., every vertex reachable from p is also reachable from f). Next, according to the schedule (5), $c(v)$ is computed (6) finding that $|\mathcal{V}_v| = 15$, $d(v,v') \leq 4$ for every $v' \in \mathcal{V}_v$ and $s_v = \sum_{v' \in \mathcal{V}_v} d(v,v') = 34$. In this case, $|\mathcal{V}_v| = 15$ can be used as an upper bound on $|\mathcal{V}_f|$ for any vertex f reachable from v since $\mathcal{V}_f \subseteq \mathcal{V}_v$ (7).

Assume that $d(v,f) = 1$ and $|V| = 100$. Then, a lower bound on $s_f = \sum_{v' \in \mathcal{V}_f} d(f,v')$ can be found as follows: First, $\sum_{v' \in \mathcal{V}_f} d(v,v') \leq \sum_{v' \in \mathcal{V}_f} \big(d(v,f) + d(f,v')\big) = \sum_{v' \in \mathcal{V}_f} 1 + \sum_{v' \in \mathcal{V}_f} d(f,v') = |\mathcal{V}_f| \cdot 1 + s_f \leq 15 \cdot 1 + s_f$ (8). Since $|\mathcal{V}_v - \mathcal{V}_f| = |\mathcal{V}_v| - |\mathcal{V}_f| \leq 15 - 12$ and $d(v,v') \leq 4$ for all $v' \in \mathcal{V}_v$ (by (6)), $\sum_{v'' \in \mathcal{V}_v - \mathcal{V}_f} d(v,v'') \leq (15-12) \cdot 4 = 12$ (9). Then, by (8), $s_f \geq \sum_{v' \in \mathcal{V}_f} d(v,v') - 15 \cdot 1$. Furthermore, since $\sum_{v' \in \mathcal{V}_v} d(v,v') = \sum_{v' \in \mathcal{V}_f} d(v,v') + \sum_{v'' \in \mathcal{V}_v - \mathcal{V}_f} d(v,v'')$, $s_f \geq \Big(\sum_{v' \in \mathcal{V}_v} d(v,v') - \sum_{v'' \in \mathcal{V}_v - \mathcal{V}_f} d(v,v'')\Big) - 15 \cdot 1 \geq 34 - 12 - 15$ by (9). Therefore, $s_f \geq 7$ (10). In this case, $c(f) \leq \frac{(15-1)^2}{(100-1)\cdot 7} < 0.4 = \theta(A)$ (11) and f can be safely skipped (12).

The above properties can be formally expressed as follows:

Lemma 1: For any vertex f reachable from vertex v,

$$s_f = \sum_{v' \in \mathcal{V}_f} d(f,v') \geq s_v - (|\mathcal{V}_v| - \perp_f) \cdot \delta_v - d(v,f) \cdot \top_f$$

where $s_v = \sum_{v' \in \mathcal{V}_v} d(v,v')$, $\perp_f$ and $\top_f$ are lower and upper bounds on $|\mathcal{V}_f|$, and δ_v is an upper bound such that $\delta_v \geq d(v,v')$ for all $v' \in \mathcal{V}_v$.

Proof: For any vertex f reachable from v, $\mathcal{V}_f \subseteq \mathcal{V}_v$ and thus $\mathcal{V}_v = \mathcal{V}_f \cup (\mathcal{V}_v - \mathcal{V}_f)$ and $\mathcal{V}_f \cap (\mathcal{V}_v - \mathcal{V}_f) = \varnothing$. Therefore,

$$(i)\ s_v = \sum_{v' \in \mathcal{V}_v} d(v,v') = \sum_{v' \in \mathcal{V}_f} d(v,v') + \sum_{v' \in \mathcal{V}_v - \mathcal{V}_f} d(v,v').$$

Here,

$$(ii) \sum_{v' \in \mathcal{V}_v - \mathcal{V}_f} d(v,v') \leq (|\mathcal{V}_v| - \perp_f) \cdot \delta_v$$

because $d(v,v') \leq \delta_v$ for each $v' \in \mathcal{V}_v - \mathcal{V}_f$ and $|\mathcal{V}_v - \mathcal{V}_f| = |\mathcal{V}_v| - |\mathcal{V}_f|$ (due to $\mathcal{V}_v \supseteq \mathcal{V}_f$) $\leq |\mathcal{V}_v| - \perp_f$. Furthermore,

$$(iii) \sum_{v' \in \mathcal{V}_f} d(v,v') \leq d(v,f) \cdot \top_f + s_f$$

since $\sum_{v' \in \mathcal{V}_f} d(v,v') \leq \sum_{v' \in \mathcal{V}_f} \big(d(v,f) + d(f,v')\big) = d(v,f) \cdot |\mathcal{V}_f| + \sum_{v' \in \mathcal{V}_f} d(f,v') \leq d(v,f) \cdot \top_f + s_f$. Then,

$$\begin{aligned} s_f &\geq \sum_{v' \in \mathcal{V}_f} d(v,v') - d(v,f) \cdot \top_f \quad \text{by } (iii) \\ &= \Big(s_v - \sum_{v' \in \mathcal{V}_v - \mathcal{V}_f} d(v,v')\Big) - d(v,f) \cdot \top_f \quad \text{by } (i) \\ &\geq s_v - (|\mathcal{V}_v| - \perp_f) \cdot \delta_v - d(v,f) \cdot \top_f \quad \text{by } (ii). \end{aligned}$$

■

Theorem 1: For any vertex f,

$$c(f) \leq \frac{(\top_f - 1)^2}{(|V|-1)\Big(s_v - (|\mathcal{V}_v| - \perp_f) \cdot \delta_v - d(v,f) \cdot \top_f\Big)}$$

where s_v, δ_v, $\perp_f$ and $\top_f$ are as in Lemma 1.

Proof: $c(f) = \frac{(|\mathcal{V}_f|-1)^2}{(|V|-1)s_f} \leq \frac{(\top_f-1)^2}{(|V|-1)s_f} \leq \frac{(\top_f-1)^2}{(|V|-1)\Big(s_v-(|\mathcal{V}_v|-\perp_f)\cdot\delta_v-d(v,f)\cdot\top_f\Big)}$ by Lemma 1. ■

B. Algorithmic Details

After the centrality values of k vertices are inserted into top-k list A, our pruning approach can be enabled by running Algorithm 3 right after line 2 in Algorithm 2. Algorithm 3 uses intermediate results L and s obtained from a search started at vertex v (Section III), as well as pruning threshold $\theta(A)$, schedule $\mathcal{S}$, and δ_v. To ensure that $\delta_v \geq d(v,v')$ for each $v' \in \mathcal{V}_v$, the value of δ_v is determined as follows: If $c(v)$ was computed using PFS (Algorithm 4 in Appendix A), δ_v is set to $\max_{v' \in \mathcal{V}_v} d(v,v')$. If $c(v)$ was computed using Δ-PFS which reused a search started at p (Algorithm 5 in Appendix A), δ_v is assigned the maximum of (i) $w(v,p) + \delta_p$ where $\delta_p \geq d(p,v')$ for $v' \in \mathcal{V}_p$ is from the search started at p and (ii) $\max_{v' \in \mathcal{V}_v - \mathcal{V}_p} d(v,v')$. In this case, (i) for each $v' \in \mathcal{V}_p$, $\delta_v \geq w(v,p) + \delta_p \geq w(v,p) + d(v,p) \geq d(v,v')$ and (ii) for each $v' \in \mathcal{V}_v - \mathcal{V}_p$, $\delta_v \geq d(v,v')$.

Our method uses maps $\top : V \to \mathbb{N}$ and $\perp : V \to \mathbb{N}$ to store, for every $f \in V$, upper and lower bounds on $|\mathcal{V}_f|$, respectively. Given $|L| = |\mathcal{V}_v|$, it visits each vertex f such that the value of

Algorithm 3: $prune(v, L, s, \theta(A), \mathcal{S}, \delta_v)$

input : vertex v, vertex-level map L, sum of distances s, threshold $\theta(A)$, schedule $\mathcal{S}$, distance upper bound δ_v

1 ensure that $\bot[f] \geq |L|$ for every f with a path to v;
2 ensure that $\top[f] \leq |L|$ for every f with a path from v;
3 insert $(v, 0)$ into priority queue Q (priority: 0);
4 **while** $|Q| > 0$ **do**
5 $(f, l) \leftarrow remove\_min(Q)$; // *dequeue* $\arg\min_{(v', l') \in Q} l'$
6 $s' = s - (|L| - \bot[f]) \cdot \delta_v - l \cdot \top[f]$; // *Lemma 1*
7 $c' \leftarrow \frac{(\top[f]-1)^2}{(m-1) \cdot s'}$; // *upper bound on $c(f)$: Theorem 1*
8 **if** $c' - \theta(A) < \phi[f]$ **then** // *c' decreased or $\theta(A)$ increased*
9 $\phi[f] \leftarrow \theta(A) - c'$; // *save $\theta(A) - c'$ for vertex f*
10 **for** *each vertex f' with an edge from f* **do**
11 insert $(f', l + w(f, f'))$ into Q;
12 **if** $c' < \theta(A)$ **and** *f has no successors in $\mathcal{S}$* **then**
13 remove f and its incoming edge from $\mathcal{S}$;

$\bot[f]$ is less than $|L|$ and there is a path from f to v. For each visited vertex f, $\bot[f]$ is set to $|L|$ (line 1 in Algorithm 3). This approach incurs low overhead since it visits a vertex f only when a tighter bound value for $\bot[f]$ is available. In an undirected graph, $\bot[f]$ is updated only once for every vertex f because $|\mathcal{V}_p| = |\mathcal{V}_f|$ for every vertex p with a path to and from f. Next, for every vertex f such that the value of $\top[f]$ is greater than $|L|$ and there is a path from v to f, our method sets $\top[f]$ to $|L|$ (line 2).

After the above steps, our method skips vertices that cannot be included in the final top-k answer. Since the pruning condition for vertex f requires $d(v, f)$ (Lemma 1 and Theorem 1), our method performs a PFS which starts at vertex v (lines 3-13). While visiting each vertex f in order of increasing geodesic distance from v, it computes a lower bound s' on s_f (line 6) and an upper bound c' on $c(f)$ (line 7). It then checks if c' has decreased or $\theta(A)$ has increased, by comparing $c' - \theta(A)$ to $\phi[f]$ which stores the minimum among the past values of $c' - \theta(A)$ (line 8). If so, it sets $\phi[f]$ to $c' - \theta(A)$ (line 9) and inserts into Q vertices whose pruning condition may hold due to the changes reflected in c' and $\theta(A)$ (lines 10 and 11). If vertex f cannot be in the final top-k answer (i.e., $c' < \theta(A)$) and f is not needed for computing the centrality of any other vertex (i.e., f has no successors in $\mathcal{S}$), then f is safely skipped (lines 12-13). Our method visits vertex f only if the difference between the upper bound on $c(f)$ and $\theta(A)$ decreases. Section VI provides detailed evaluation results on our method's effectiveness in skipping centrality computations with economical use of computation resources.

V. Scheduling

This section describes our techniques for determining the order of centrality computations (Section V-A) and estimating centrality computation time (Sections V-B and V-C). The technique mentioned in Section V-B can also estimate the centrality of vertices.

A. Overview

If there is an edge from a vertex v to a vertex p, the closeness centrality of v can be obtained from Δ-PFS which reuses a search started at p (Section III-A). Each solid arrow in Figure 7 represents such a centrality computation scenario for the graph in Figure 3(a). In Figure 7, the weight of an edge is the estimated time to finish the corresponding computation. For example, the weight $\hat{t}_{a|b}$ of the edge from b to a represents the estimated time to compute the centrality of a using Δ-PFS which reuses a search started at b. The centrality of each vertex v can also be computed using PFS. In Figure 7, this case is represented as a dotted edge from a virtual vertex o to vertex v and the edge weight $\hat{t}_v$ is set to the estimated time to compute the centrality of v using PFS.

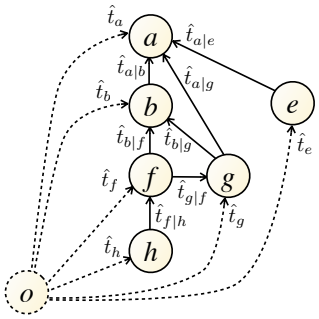


Fig. 7. Initial Schedule

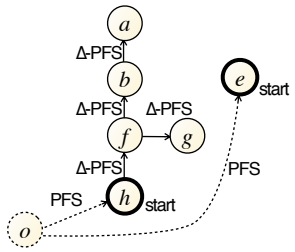


Fig. 8. Optimized Schedule

If all of the possible centrality computation scenarios are represented as above, a directed minimum spanning tree rooted at virtual vertex o (Figure 8) has the following properties: (i) every vertex v in the tree has a path from o, meaning that the centrality of v can be computed using a series of PFS and Δ-PFS operations represented as the edges on the path from o, and (ii) the sum of edge weights (i.e., the estimated completion time) is no larger than those of other spanning trees rooted at o. In other words, this spanning tree represents a centrality computation schedule which minimizes the estimated completion time. This tree can be obtained in $O(|E| \log |V|)$ time [17].

B. Approximate Top-k Answers

Our scheduling approach (Section V-A) requires an estimated completion time for every possible scenario of computing a vertex's centrality. As explained in Section V-C, these completion times can be estimated if, for each vertex v, $|\mathcal{V}_v|$ (i.e., the number of vertices reachable from v) and $s_v = \sum_{v' \in \mathcal{V}_v} d(v, v')$ are known. In this section, we describe our technique that estimates $|\mathcal{V}_v|$ and s_v with low computational and space overhead. Using the estimates of $\mathcal{V}_v$ and s_v, this technique estimates the centrality of every vertex and presents the k vertices with the highest estimates as an approximate top-k answer.

Our method for estimating $|\mathcal{V}_v|$ and s_v for every v extends an approximate centrality computation algorithm by Kang et. al [12]. This algorithm uses a fixed-size bitmap $\hat{\mathcal{V}}_{v,i}$, called an *FM-sketch* [18], to represent the set of vertices reachable from v within i hops. First, it initializes, for each vertex v, a sketch $\hat{\mathcal{V}}_{v,0}$ using the ID of v. At iteration i, for every vertex v, $\hat{\mathcal{V}}_{v,i}$ is updated using the bitwise OR operation $\biguplus$ with $\hat{\mathcal{V}}_{v,i-1}$ and $\hat{\mathcal{V}}_{p,i-1}$ for every vertex p with an edge from v. The $\biguplus$ operation supports duplicate-insensitive counting (i.e., has the

property that if $count(\hat{V}_1) \approx |V_1|$ and $count(\hat{V}_2) \approx |V_2|$, then $count(\hat{V}_1 \uplus \hat{V}_1) \approx |V_1 \cup V_2|$ where $count(\hat{V}_i)$ is the number of distinct values estimated from sketch $\hat{V}_i$). The above process is then repeated until $\hat{\mathcal{V}}_{v,i} = \hat{\mathcal{V}}_{v,i-1}$ for every vertex v (i.e., all of the vertices reachable from v are reflected in $\hat{\mathcal{V}}_{v,i}$). Then, $|\mathcal{V}_v|$ and s_v are estimated as $count(\hat{\mathcal{V}}_v)$ and $\hat{s}_v = \sum_{j=1}^{i} j \cdot \hat{\mathcal{V}}_{v,j}$ where $\hat{\mathcal{V}}_v = \hat{\mathcal{V}}_{v,i}$.

The algorithm by Kang et al. cannot effectively support *weighted* graphs. To overcome this limitation with low space and time costs, our approach merges sketches that are sent to the same vertex during different iterations. At iteration i, our approach incorporates, for every edge (p, v), sketch $\hat{\mathcal{V}}_{v,i}$ into $\hat{\mathcal{V}}_{p,\tau}$ rather than $\hat{\mathcal{V}}_{p,i+1}$, where τ is $round_\mu\Big(round_{w(p,v)}\big(i + w(p,v)\big)\Big)$, $\mu = \min_{(p,q)\in E} w(p,q)$ is the minimum edge weight, and $round_\mu(t)$ is a multiple of μ closest to t. In this way, $\hat{\mathcal{V}}_{p,\tau}$ can approximate a set of vertices whose shortest distance from p is no longer than τ. A concrete description of the above method can be found in our technical report [19].

Our approach requires only $O(\psi|V| + \psi|E|)$ memory space where ψ is the size of each FM-sketch. The reason for this benefit is that our approach maintains one sketch for each vertex and up to two sketches for each edge since $round_\mu\Big(round_{w(p,v)}\big(i + w(p,v)\big)\Big) < round_{w(p,v)}\big(i + w(p,v)\big) + \mu < \Big(\big(i + w(p,v)\big) + w(p,v)\Big) + \mu \leq i + 3 \cdot w(p,v)$. The running time of this approach is $O(\psi \frac{\delta}{\mu}|E|)$ where δ is the diameter of the graph such that $\delta = \max_{v\in V}(\max_{v'\in\mathcal{V}_v} d(v,v'))$. In many real-world networks, the diameter tends to decrease as the network size increases [20]. On road networks defined in a 2-dimensional coordinate space, δ in general increases in proportion to $\sqrt{|V|}$ (Section VI). If δ is extremely large compared to μ, the running time of our approach can also be reduced at the expense of accuracy by suppressing the variation of edge weights (e.g., changing each edge weight $w(p,v)$ to $w(p,v)^\epsilon$ for some ϵ such that $0 \leq \epsilon < 1$).

When $\hat{\mathcal{V}}_v$ and $\hat{s}_v$ are obtained as above, the centrality of v can be estimated as $\frac{(count(\hat{\mathcal{V}}_v)-1)^2}{(|V|-1)\hat{s}_v}$ with negligible overhead. Then, the k vertices with the highest estimated centrality values can also be found in $O(|V| \log k)$ time using a priority queue of size k. The utility of these vertices as an approximate top-k answer is experimentally demonstrated in Section VI.

C. Search Cost Estimation

We now discuss how the estimates of $|\mathcal{V}_v|$ and s_v (Section V-B) can be used to estimate the completion times of PFS and Δ-PFS.

The running time of PFS which starts at v is $O(|\mathcal{E}_v| + |\mathcal{V}_v| \log |\mathcal{V}_v|)$ (Section III-A and Appendix C). Furthermore, when $\mathcal{V}_v$ contains a relatively large number of vertices, $|\mathcal{E}_v|$ can be approximated as $\frac{|E|}{|V|} \cdot |\mathcal{V}_v|$. Therefore, given sketch $\hat{\mathcal{V}}_v$ such that $count(\hat{\mathcal{V}}_v) \approx |\mathcal{V}_v|$ (Section V-B), we define the estimated PFS time $\hat{t}_v$ as $count(\hat{\mathcal{V}}_v) \log(count(\hat{\mathcal{V}}_v))$ while assuming that the unit time is the average amount of time that PFS would spend per visited vertex. If an unweighted graph is given, $\hat{t}_v$ is defined to be $count(\hat{\mathcal{V}}_v)$ since PFS can complete in $O(|\mathcal{E}_v| + |\mathcal{V}_v|)$ (Appendix C).

name	type	degree	meaning of edge from v_1 to v_2
RI	weighted, undirected	2.1	a road segment with length $w(v_1, v_2)$ between locations v_1 and v_2 in Rhode Island [21]
Web	unweighted, directed	5.8	web page v_2 has a hyperlink to v_1 [22]
Wiki	unweighted, directed	2.1	user v_2 has edited a Wikipedia Talk page of user v_1 [22]
DBLP	unweighted, undirected	5.3	researchers v_1 and v_2 have coauthored papers [23]

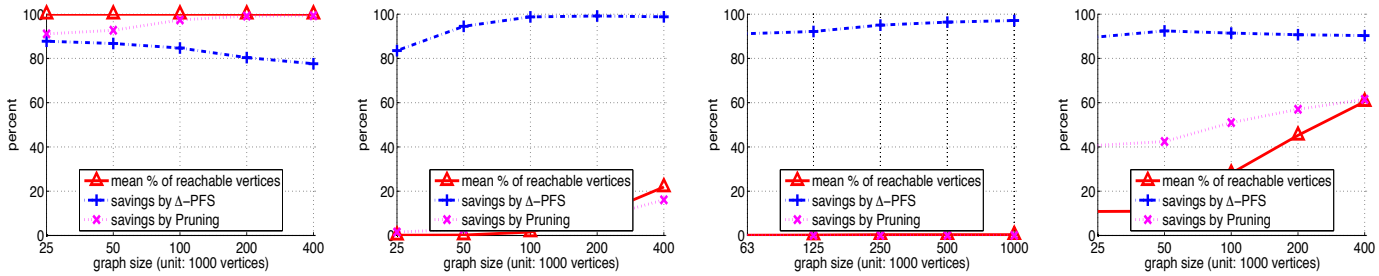
TABLE II. DATA SETS

A Δ-PFS which starts at v and reuses a search started at p completes in $O(|\mathcal{E}_{v\wr p}| + |\mathcal{V}_{v\wr p}| \log |\mathcal{V}_{v\wr p}|)$ time (Section III-A and Appendix C). Making assumptions on $|\mathcal{E}_{v\wr p}|$ as in the case of $\hat{t}_v$, we set the estimated Δ-PFS time $\hat{t}_{v\wr p}$ to $\gamma|\mathcal{V}_{v\wr p}| \log |\mathcal{V}_{v\wr p}|$ where γ denotes the ratio of the time that Δ-PFS spends per visited vertex to the time that PFS spends per visited vertex. In all of our evaluations (Section VI), γ was 1.79 or slightly smaller since Δ-PFS performs, for each visited vertex n, additional operations which save the previous location of n in Λ and restore the previous location of n using Λ (line 6 in Algorithm 2). Despite this higher overhead per visited vertex, Δ-PFS generally outperforms PFS since a vertex v in a real-world graph is likely to have an adjacent vertex p such that $\mathcal{V}_{v\wr p}$ is much smaller than $\mathcal{V}_v$ (Section VI). Furthermore, our scheduling approach selects PFS when it has a shorter expected completion time than Δ-PFS (Section V-A).

To obtain $\hat{t}_{v\wr p}$ as above, our method estimates $|\mathcal{V}_{v\wr p}|$ (i.e., the number of vertices that Δ-PFS visits) by adding (i) the estimated number of vertices that Δ-PFS would newly visit (e.g., vertex e in Figure 2(c)) and (ii) the estimated number of vertices that Δ-PFS would promote (e.g., vertex g in Figure 2(b)). The former (i.e., $|\mathcal{V}_v|$ - $|\mathcal{V}_p|$) can be approximated as $count(\hat{\mathcal{V}}_v) - count(\hat{\mathcal{V}}_p)$. The latter is approximated as $0.82\frac{\hat{\sigma}_{v\wr p}^{0.96} \cdot count(\hat{\mathcal{V}}_p)^{0.23}}{(w(v,p))^{0.83} \cdot \hat{s}_p^{0.16}}$ where $\hat{\sigma}_{v\wr p}$ is the estimated sum of promotion distances. This formula is obtained through linear regression that identified the relationship among the above variables based on actual Δ-PFS executions (Section VI). We define $\hat{\sigma}_{v\wr p}$ as $w(v,p) \cdot count(\hat{\mathcal{V}}_p) + \hat{s}_p - \frac{count(\hat{\mathcal{V}}_p)}{count(\hat{\mathcal{V}}_v)}\hat{s}_v$ for the following reasons: (i) For every promoted vertex v', $v' \in \mathcal{V}_p$. (ii) For every promoted vertex v', the promotion distance is $w(v,p) + d(p,v') - d(v,v')$. (iii) Δ-PFS skips every vertex $v' \in \mathcal{V}$ such that $w(v,p) + d(p,v') - d(v,v') = 0$. (iv) By (i), (ii) and (iii), the sum of the promotion distances can be expressed as $\sum_{v'\in\mathcal{V}_p}\big(w(v,p) + d(p,v') - d(v,v')\big) = \sum_{v'\in\mathcal{V}_p} w(v,p) + \sum_{v'\in\mathcal{V}_p} d(p,v') - \sum_{v'\in\mathcal{V}_p} d(v,v') = w(v,p)|\mathcal{V}_p| + s_p - \sum_{v'\in\mathcal{V}_p} d(v,v')$. (v) If $\mathcal{V}_p$ contains a relatively large number of vertices from $\mathcal{V}_v$, $\sum_{v'\in\mathcal{V}_p} d(v,v') \approx \frac{|\mathcal{V}_p|}{|\mathcal{V}_v|}\sum_{v'\in\mathcal{V}_v} d(v,v') = \frac{|\mathcal{V}_p|}{|\mathcal{V}_v|} s_v$.

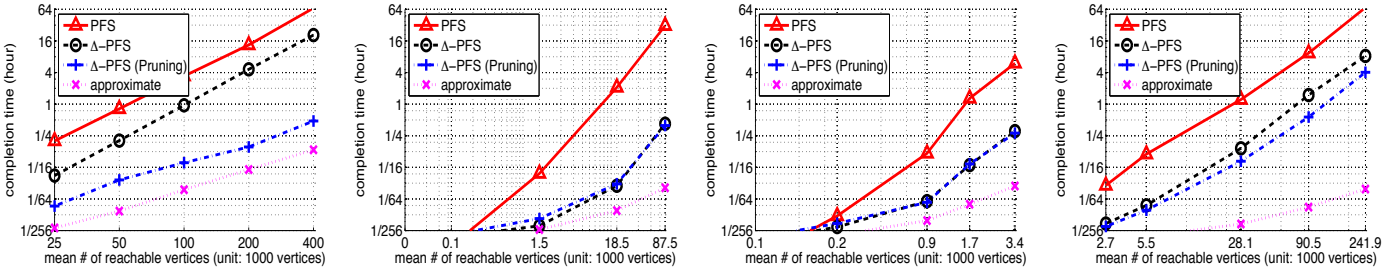
VI. EVALUATION

This section presents experimental results obtained by running three methods for finding the k most central vertices in a graph. One method, called PFS, computes the centrality of each vertex using an implementation of Dijkstra's algorithm [9] with a Fibonacci heap [10]. To the best of our knowl-



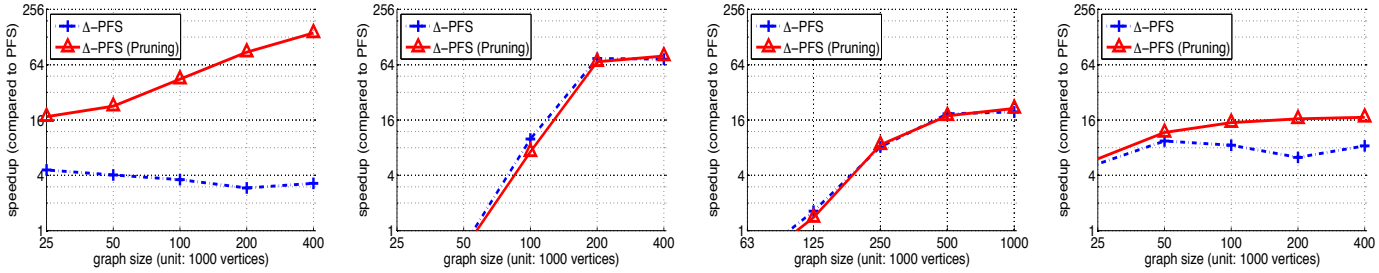
(a) RI (b) Web (c) Wiki (d) DBLP

Fig. 9. Mean % of Reachable Vertices and Savings by Δ-PFS (k = 10)



(a) RI (b) Web (c) Wiki (d) DBLP

Fig. 10. Benefits of Δ-PFS, Pruning, and Scheduling (k = 10)



(a) RI (b) Web (c) Wiki (d) DBLP

Fig. 11. Speedup

edge, PFS is the fastest method in the literature for finding the k most central vertices (Section VII). Another method, referred to as Δ-PFS, performs centrality computations as described in Sections III and V. This method can also be used with the pruning technique described in Section IV. We call this combination Δ-PFS (Pruning). These methods were evaluated using the data sets summarized in Table II. From each data set, we derived a series of five graphs. Each graph in a series contained twice as many vertices as the previous graph. For example, the smallest and largest graphs from the Wiki data set consisted of 62,500 vertices and 1,000,000 vertices, respectively (Figure 9(c)). When graphs were larger than the above (e.g., 2 million vertices), PFS did not complete within a week. Furthermore, it is not possible to accurately predict how long this method would take in such situations since the properties of a graph (e.g., the percentage of vertices that are reachable from a vertex) significantly change as the size of the graph increases. Therefore, we were not able to present accurate comparisons between the techniques for these large graphs. The results in this section are averaged over 10 runs executed on Quad-Core Xeon E5430 2.67 GHz CPUs.

A. *Effectiveness of Δ-PFS and Pruning*

We measured the performance of PFS, Δ-PFS, and Δ-PFS (Pruning). On the graphs constructed from the RI data set, the completion time of PFS increased by a factor of 339 (from 12 minutes to 67.5 hours) as the graph size varied from 25,000 to 400,000 vertices (Figure 10(a)). On the other hand, the completion time of PFS increased at a rate of 2,470 (from 1.7 minutes to 69.2 hours) in graphs from the DBLP data set. The reason is that while all of the vertices in the RI data set are reachable from each other, the fraction of vertices reachable from a vertex (i.e., those visited by PFS) increased substantially (from 10.8% to 60.5%) in the case of the DBLP data set (see "mean % of reachable vertices" in Figure 9(d)).

As mentioned above, the varying percentage of reachable vertices in a graph may significantly affect top-k centrality computation time. In order to illustrate general trends in performance despite this complexity, Figure 10 shows the overall completion time as a function of the average number of reachable vertices in a graph. In each observed case, Δ-PFS (Pruning) demonstrated significant performance benefits.

In particular, it outperformed PFS by a factor of 142 (28.6 minutes vs. 67.5 hours) on a graph containing 400,000 vertices from the RI data set. In this case, while the graph size was increased by a factor of 16, the running time of Δ-PFS (Pruning) only increased by a sub-quadratic factor of 46 (=$16^{1.38}$). On the other hand, PFS showed a super-quadratic increase (339 = $16^{2.1}$) in its completion time.

Figure 11 demonstrates how the benefit of Δ-PFS varies significantly depending on the data set. Δ-PFS without pruning outperforms PFS by a factor of 3.3 on a graph with 400,000 vertices from the RI data set. On the other hand, Δ-PFS reduces execution time by a factor of more than 73.7 in a graph with 400,000 vertices from the Web data set. In this graph, each vertex v tends to have a neighboring vertex p such that many shortest paths from v to other vertices pass through p (i.e., Δ-PFS which starts at vertex v and reuses a search started at p can skip a large number of vertices). As shown in Figure 9(b), Δ-PFS skipped up to 99% of vertices in the graphs obtained from the Web data set (see "savings by Δ-PFS").

In highly connected graphs (e.g., graphs from the RI data set), our pruning technique can obtain a relatively tight upper bound on the centrality of every vertex and evaluate the pruning condition along more paths. Therefore, it can proactively skip more vertices, particularly overcoming relatively few sharing opportunities (Figure 9(a)). Our pruning technique usually skips more vertices in larger graphs since, as graph size increases, more paths tend to exist between vertices [20]. In Figure 9, curves labeled "savings by Pruning" show the actual percentage of skipped centrality calculations.

B. Cost Analysis

Our technique mentioned in Section V determines the order of centrality computations and produces an approximate top-k answer. Our experimental results on the time overhead of this technique are presented in Section VI-D. Figure 12 shows the memory utilization of this technique. In all observed cases, memory utilization increased by a factor of 3 or 4 due to the use of sketches and the construction of the centrality computation schedule (Section V). When the schedule is initially constructed (Figure 7), it has the same size as the original graph. Its size then decreases as it is optimized (Figure 8). Given the aforementioned schedule, our technique performs a series of Δ-PFS operations (Section III). During this phase, the memory overhead of keeping the instances of Λ (Section III-B) is low (see the curves labeled "stack" in Figure 12). Furthermore, our technique can keep the memory utilization under a user-specified bound (Section III-B).

As Figure 10(c) shows, our pruning method incurs negligible computational overhead. In this case, the overall completion time with pruning did not increase noticeably despite adversarial conditions for pruning (i.e., there were no significant savings by pruning as shown in Figure 9(c)).

C. Impact of Parameters

While we increased the size of the sketches from 4 bytes to 1,024 bytes, we did not observe noticeable improvement in the performance of Δ-PFS (Pruning). However, sketches larger than 64 bytes incurred perceivable space and time overhead. Therefore, we use 64-byte sketches.

We examined the impact of k on the overall completion time. The completion time was minimized when k = 1. As k increases, the pruning threshold decreases. Thus, our pruning technique skipped fewer vertices (Figures 13 and 14).

D. Benefits of Approximate Results

Our technique mentioned in Section V-B has the advantage of producing approximate top-k answers. In contrast to previous approximation techniques that support only undirected graphs [11], [13], this technique is applicable to directed graphs. It also overcomes the limitation of an algorithm by Kang et al. [12] that cannot support weighted graphs (both algorithms process unweighted graphs in an identical way). The delivery times of these approximate results are shown in Figure 10 (see the curves labeled "approximate"). In each unweighted graph containing up to 1,000,000 vertices, approximate results were obtained in less than 1.8 minutes. In graphs from the RI data set, the delivery time was longer since it is proportional to the network diameter, which is large in real-world road networks. Given graphs from the Web, Wiki, and DBLP data sets, 73% of entries in the initial top-k answer were correct (i.e., remained in the final answer). When approximate answers were obtained from the RI data set, the answers were less accurate since the variation in the centrality values was small (for further details, refer to our technical report [19]).

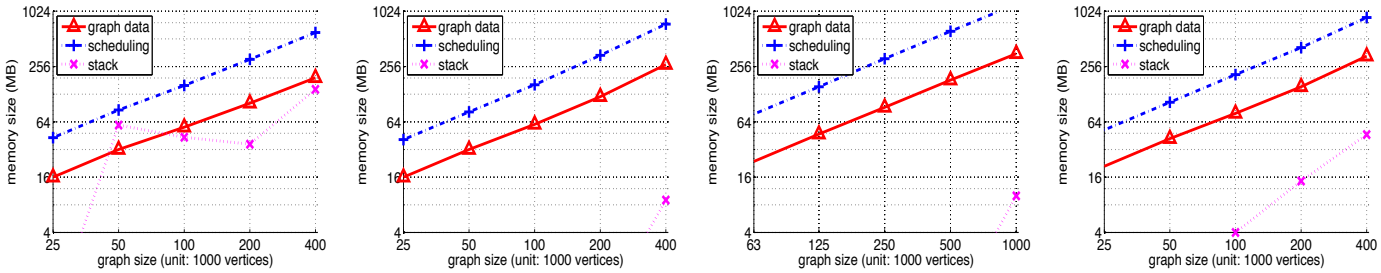
VII. Related Work

In addition to closeness centrality (Definition 1), researchers have developed several types of centrality metrics to capture the influence of real-world entities from a different perspective. For example, the *degree* centrality of a vertex refers to the number of edges incident on that vertex [24], [25]. Another popular centrality metric is *PageRank* which assigns relatively high scores to vertices that have connections to other vertices with a high score [24]. The degree of all vertices can be computed in $O(|V|+|E|)$ time by counting the edges incident on every vertex. In this case, k vertices with the highest degree centrality can be found in $O(|V| \cdot \log k)$ time by inserting each vertex into a priority queue and dequeuing a vertex the lowest degree whenever the queue contains more than k vertices. It is also known that an appropriate PageRank value can be obtained for all vertices usually in $O(|V| + |E|)$ time [26].

The *betweenness* centrality of a vertex [27], [24], [28] is defined as:

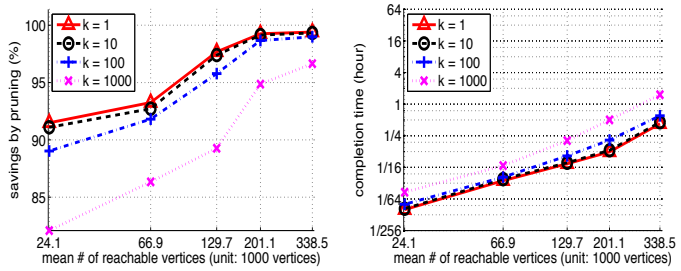
$$\sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{3}$$

where σ_{st} is the number of shortest paths from vertex s to vertex t and $\sigma_{st}(v)$ is the number of shortest paths from s to t which pass through v. While the closeness centrality of a vertex v can be computed using only one PFS from v, the betweenness centrality of v requires examining all shortest paths for all pairs of vertices. Variants of betweenness centrality, including *stress* centrality [27], [28] and *bridging* centrality [29] share this inherent complexity. Brandes proposed an algorithm which uses Dijkstra's algorithm repeatedly



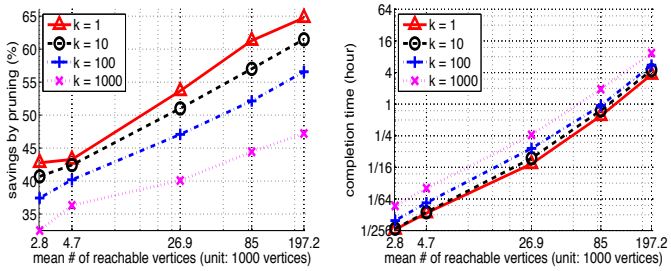
(a) RI (b) Web (c) Wiki (d) DBLP

Fig. 12. Memory Utilization during Different Stages in Top-k Query Processing



(a) savings by pruning (b) completion time

Fig. 13. Impact of k (RI)



(a) savings by pruning (b) completion time

Fig. 14. Impact of k (DBLP)

to compute the betweenness centrality of every vertex in $O(|V||E|+|V|^2\log(|V|))$ time and $O(|V|+|E|)$ space [27].

Eccentricity [15] measures the maximum distance from a vertex to any other vertex. In our preliminary experiments, we observed that many vertices had identical eccentricity values, rendering the metric inappropriate in the context of finding k most central vertices.

Given a graph, k vertices with the highest closeness centrality values can be found by solving the all pairs shortest paths (APSP) problem and then computing the centrality of each vertex. The Floyd-Warshall algorithm [8] solves APSP using dynamic programming in $\Theta(|V|^3)$ time and $\Theta(|V|^2)$ space. In dense graphs, the time complexity of APSP has been reduced to $O(|V|^3\log^3\log|V|/\log^2|V|)$ [7]. Johnson's algorithm for APSP is faster than the Floyd-Warshall algorithm in sparse graphs [30]. This algorithm runs Djikstra for each vertex and uses the results to construct a distance matrix. For this reason, Johnson's algorithm for APSP runs in $O(|V||E|+|V|^2\log(|V|))$ time and $O(|V|^2)$ space. In order to calculate the closeness centrality of a vertex, only the shortest distance from that vertex to every other vertex is needed. Therefore, Johnson's APSP algorithm can be modified so that it does not construct a distance matrix, and instead calculates the closeness centrality of each vertex. This modified algorithm has a space complexity of only $O(|V|+|E|)$. There are methods that construct an index for quickly finding the shortest distance between any pair of vertices [31]. However, computing the centrality of vertex v by repeatedly finding the shortest distance from v to every other vertex is known to be slower than doing the same using Dijkstra's algorithm [31]. In contrast to these algorithms, our technique derives unique performance benefits by sharing data across centrality computations and bypassing vertices that cannot be in the final answer.

There are also techniques for providing approximate answers to top-k centrality queries. Eppstein and Wang [11] developed an algorithm that approximates the closeness centrality of every vertex in a graph. This algorithm first runs Dijkstra on a small number of randomly selected pivot vertices and then uses the results to estimate the closeness centrality of every vertex. This algorithm, however, supports only undirected graphs since it assumes that the shortest distance from each vertex v to any other vertex v' is the same as the shortest distance from v' to v. Okamoto et al. [13] extended Eppstein's algorithm to provide approximate top-k results. Their method strives to strike a balance between result accuracy and query completion time. An approximation technique by Kang et al. [12] is summarized in Section V-B. In contrast to these approximation algorithms, our solution supports both directed and weighted graphs and refines approximate answers until it finds exact answers at a much higher speed than other methods.

Additionally, algorithms have been devised to efficiently update the centrality values of vertices as the structure of a graph changes [32], [33]. In this paper, we focus on the problem of expediting centrality computation given static graphs. We reserve as future work the problem of quickly finding central vertices in the context of dynamic graphs.

The standard approach for processing top-k queries in database systems is to maintain the current top-k set, from which a threshold (the least valued object in the set) and an upper bound on unexamined objects are derived [34]. If the upper bound on the unexamined objects is lower than the threshold, the current set is returned as the final answer. Our approach is similar in that it obtains a threshold from a list of examined vertices. However, its pruning condition exploits the structural properties of graphs (Theorem 1), which has high utility for large graphs (Section VI-A).

VIII. Conclusion

In this paper, we proposed a new solution for efficiently finding k vertices with the highest closeness centrality values in directed graphs with nonnegative edge weights. By efficiently updating an upper bound on the centrality of every vertex, our solution proactively skips vertices that cannot be among the k most central vertices. Our solution also shares intermediate results between centrality computations scheduled in a manner that minimizes the estimated completion time. Evaluations on real-world data sets show our solution to be, in addition to having a small memory footprint, a magnitude of one or two orders faster than other traditional approaches.

We plan to extend our technique to a wider range of graph problems including the computation of network diameter and other centrality metrics such as betweenness centrality and stress centrality. Another future research plan is to develop a parallel processing framework that will facilitate centrality computations on large graphs. We also intend to investigate the adaptability of our algorithms to an environment in which graph updates are allowed.

References

[1] S. Porta, "Street Centrality and Densities of Retail and Services in Bologna, Italy," *Environment and Planning B: Planning and Design*, vol. 36, no. 3, pp. 450–465, 2009.

[2] C. Kiss and M. Bichler, "Identification of Influencers–Measuring Influence in Customer Networks," *Decision Support Systems*, vol. 46, no. 1, pp. 233–253, 2008.

[3] D. Bell, J. Atkinson, and J. Carlson, "Centrality Measures for Disease Transmission Networks," *Social Networks*, vol. 21, no. 1, pp. 1–21, 1999.

[4] E. Elmacioglu and D. Lee, "On Six Degrees of Separation in DBLP-DB and More," *ACM SIGMOD Record*, vol. 34, no. 2, pp. 33–40, 2005.

[5] S. A. Macskassy, "Using Graph-Based Metrics with Empirical Risk Minimization to Speed up Active Learning on Networked Data," in *Proc. of SIGKDD*, 2009, pp. 597–606.

[6] Z. Zhuang, E. Elmacioglu, D. Lee, and C. L. Giles, "Measuring Conference Quality by Mining Program Committee Characteristics," in *Proc. of JCDL*, 2007, pp. 225–234.

[7] T. Chan, "More Algorithms for All-Pairs Shortest Paths in Weighted Graphs," *SIAM Journal on Computing*, vol. 39, no. 5, pp. 2075–2089, 2010.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT press, 2001.

[9] E. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[10] M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," in *Proc. of FOCS*, 1984, pp. 338–346.

[11] D. Eppstein and J. Wang, "Fast Approximation of Centrality," in *Proc. of SODA*, 2001, pp. 228–229.

[12] U. Kang, S. Papadimitriou, J. Sun, and H. Tong, "Centralities in Large Networks: Algorithms and Observations," in *Proc. of ICDM*, 2011, pp. 119–130.

[13] K. Okamoto, W. Chen, and X.-Y. Li, "Ranking of Closeness Centrality for Large-Scale Social Networks," in *Proc. of FAW*, 2008, pp. 186–195.

[14] N. Lin, *Foundations of Social Research*. Mcgraw-Hill, 1976.

[15] W. Stanley and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[16] P. Narváez, K.-Y. Siu, and H.-Y. Tzeng, "New Dynamic Algorithms for Shortest Path Tree Computation," *IEEE/ACM Transactions on Networking*, vol. 8, no. 6, pp. 734–746, 2000.

[17] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, "Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs," *Combinatorica*, vol. 6, no. 2, pp. 109–122, 1986.

[18] P. Flajolet and G. N. Martin, "Probabilistic Counting Algorithms for Data Base Applications," *Journal of Computer and System Sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[19] P. Olsen Jr., A. Labouseur, and J.-H. Hwang, "Efficient Top-k Closeness Centrality Search," University at Albany – State University of New York, Technical Report SUNYA-CS-13-01, 2013.

[20] J. Leskovec, J. M. Kleinberg, and C. Faloutsos, "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations," in *Proc. of SIGKDD*, 2005, pp. 177–187.

[21] OpenStreetMap, http://www.openstreetmap.org/.

[22] SNAP, http://snap.stanford.edu/data/.

[23] DBLP, http://dblp.uni-trier.de/xml/.

[24] S. Borgatti, "Centrality and Network Flow," *Social Networks*, vol. 27, no. 1, pp. 55–71, 2005.

[25] L. Freeman, "Centrality in Social Networks: Conceptual Clarification," *Social Networks*, vol. 1, no. 3, pp. 215–239, 1979.

[26] T. Haveliwala, "Efficient Computation of PageRank," Stanford InfoLab, Technical Report 1999-31, 1999.

[27] U. Brandes, "A Faster Algorithm for Betweenness Centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[28] L. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.

[29] W. Hwang, T. Kim, M. Ramanathan, and A. Zhang, "Bridging Centrality: Graph Mining from Element Level to Group Level," in *Proc. of SIGKDD*, 2008, pp. 336–344.

[30] D. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM*, vol. 24, no. 1, pp. 1–13, 1977.

[31] R. Jin, N. Ruan, Y. Xiang, and V. Lee, "A Highway-Centric Labeling Approach for Answering Distance Queries on Large Sparse Graphs," in *Proc. of SIGMOD*, 2012, pp. 445–456.

[32] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung, "QUBE: a Quick algorithm for Updating BEtweenness centrality," in *Proc. of WWW*, 2012, pp. 351–360.

[33] A. E. Sariyuce, K. Kaya, E. Saule, and U. V. Catalyurek, "Incremental Algorithms for Network Management and Analysis based on Closeness Centrality," *CoRR, abs/1303.0422*, 2013.

[34] I. Ilyas, G. Beskales, and M. Soliman, "A Survey of Top-k Query Processing Techniques in Relational Database Systems," *ACM Computing Surveys*, vol. 40, no. 4, p. 11, 2008.

Appendix A
PFS and Δ-PFS Algorithms

Algorithm 4 shows the PFS method mentioned in Section III-B. It extends Dijkstra's algorithm [9] (lines 4-6 and 9-16) with the addition of updating variables s for centrality computation (lines 2 and 7) and δ_v for pruning (lines 3 and 8).

Algorithm 5 describes the Δ-PFS method explained in Section III-A. It is similar to Algorithm 4 except that it places vertex v at a level higher than p by $w(v,p)$ (lines 1-2), skips vertices that are already placed at the target level (lines 15-19), updates variable s in a more complex way in response to new vertex visits (lines 10-11) and vertex promotions (lines 13-14), and logs operations that may need to be undone (lines 3 and 23).

Appendix B
Correctness of Δ-PFS

Lemma 2: Given vertices v and p, Algorithm 5 places every vertex $v' \in \mathcal{V}_{v \setminus p}$ at level $\alpha_v + d(v,v')$ where

$$\mathcal{V}_{v \setminus p} = \{v\} \cup \{v' \in \mathcal{V}_v : d(v,v') < w(v,p) + d(p,v')\} \quad (4)$$

Algorithm 4: $PFS(v)$

input : source vertex v
output : vertex-level map L, sum s of geodesic distances from p, maximum geodesic distance δ_v

1 $L[v] \leftarrow 0$; *// place vertex v at level 0*
2 $s \leftarrow 0$; *// set the sum of distances to 0*
3 $\delta_v \leftarrow 0$; *// set the maximum distance to 0*
4 insert $(v, 0)$ into priority queue Q (priority: 0);
5 **while** $|Q| > 0$ **do**
6 $\quad (n, l) \leftarrow remove\_min(Q)$; *// dequeue* $\arg\min_{(n',l') \in Q} l'$
7 $\quad s \leftarrow s + l$; *// add $d(v, n)$ to s*
8 $\quad \delta_v \leftarrow max(\delta_v, l)$; *// update δ_v if $d(v, n) > \delta_v$*
9 $\quad$ **for** *each vertex v' with an edge from vertex n* **do**
10 $\quad\quad l' \leftarrow l + w(n, v')$; *// level where v' can be placed*
11 $\quad\quad$ **if** $(L[v'] =$ **null** $)$ **then** *// visiting v' first time*
12 $\quad\quad\quad L[v'] \leftarrow l'$; *// place vertex v' at level l'*
13 $\quad\quad\quad$ insert (v', l') into priority queue Q (priority: l');
14 $\quad\quad$ **else if** $(l' < L[v'])$ **then** *// shorter path to v'*
15 $\quad\quad\quad L[v'] \leftarrow l'$; *// place vertex v' at level l'*
16 $\quad\quad\quad$ decrease the priority of v' to l' in Q;
17 **return** (L, s, δ_v);

Algorithm 5: $\Delta PFS(v, p, L, s, \delta_p)$

input : source vertex v, previous source vertex p, map L, sum of distances s, distance upper bound δ_p
output : vertex-level map L, sum of distances s, previous levels of vertices Λ, distance upper bound δ_v

1 $\alpha_p \leftarrow L[p]$; *// previous level of p (previous top level)*
2 $\alpha_v \leftarrow \alpha_p - w(v, p)$; *// new level for v (current top level)*
3 insert $(v, L[v])$ into Λ (i.e., log the previous level of v);
4 insert (v, α_v) into priority queue Q (priority: α_v);
5 $s \leftarrow s + w(v, p)|L|$; *// add $w(v, p)$ to each distance*
6 $L[v] \leftarrow \alpha_v$; *// place v at current top level α_v*
7 $\delta_v \leftarrow \delta_p + w(v, p)$; *// ensure that $\delta_v \geq d(v, v')$ for $v' \in \mathcal{V}_p$*
8 **while** $|Q| > 0$ **do**
9 $\quad (n, l) \leftarrow remove\_min(Q)$; *// dequeue* $\arg\min_{(n',l') \in Q} l'$
10 $\quad$ **if** $\Lambda[n] =$ **null then** *// visiting n for the first time*
11 $\quad\quad s \leftarrow s + (l - \alpha_v)$; *// add $d(v, n)$ to s*
12 $\quad\quad \delta_v \leftarrow max(\delta_v, l - \alpha_v)$; *// update δ_v if $d(v, n) > \delta_v$*
13 $\quad$ **else** *// if n is promoted from $\Lambda[n]$ to l*
14 $\quad\quad s \leftarrow s - (\Lambda[n] - l)$; *// subtract promotion distance*
15 $\quad$ **for** *each vertex v' with an edge from vertex n* **do**
16 $\quad\quad l' \leftarrow l + w(n, v')$; *// level where v' can be placed*
17 $\quad\quad l'' \leftarrow L[v']$; *// previous level of v'*
18 $\quad\quad$ **if** $(l'' =$ **null** *// if visiting v' for the first time*
19 $\quad\quad\quad$ **or** $l' < l'')$ **then** *// if v' can be promoted*
20 $\quad\quad\quad L[v'] \leftarrow l'$; *// place vertex v' at level l'*
21 $\quad\quad\quad$ set the priority of v' to l' in priority queue Q;
22 $\quad\quad\quad$ **if** $v' \notin \Lambda$ **then**
23 $\quad\quad\quad\quad$ add (v', l'') to Λ; *// log the prev. level of v'*
24 **return** $(L, s, \Lambda, \delta_v)$;

Proof: Since Algorithm 5 places the source vertex v at level $\alpha_v = \alpha_v + d(v, v)$, we prove the above for any vertex v' such that (i) $d(v, v') < w(v, p) + d(p, v')$ (i.e., $v' \in \mathcal{V}_{v/p} - \{v\}$).

Given the aforementioned v', consider a shortest path $v \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$ from v to $v' = v_k$ (i.e., $\forall i \in [1, k], d(v, v') = d(v, v_i) + d(v_i, v')$). If $d(v, v_i) = w(v, p) + d(p, v_i)$, then $d(v, v') = d(v, v_i) + d(v_i, v') = \big(w(v, p) + d(p, v_i)\big) + d(v_i, v') = w(v, p) + \big(d(p, v_i) + d(v_i, v')\big) \geq w(v, p) + d(p, v')$, which contradicts (i). Therefore, (ii) any shortest path $v \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$ from v to $v' = v_k$ has the property that $\forall i \in [1, k], d(v, v_i) < w(v, p) + d(p, v_i)$.

Let us prove by induction that Algorithm 5 places v' at level $\alpha_v + d(v, v')$. First, when v is dequeued (line 9) and then v_1 is examined (lines 15-23), v_1's target level value ($l' = \alpha_v + w(v, v_1) = \alpha_v + d(v, v_1)$; line 16) must be smaller than its previous level value ($l'' = \alpha_p + d(p, v_1)$; line 17) since $\alpha_v + d(v, v_1) = \big(\alpha_p - w(v, p)\big) + d(v, v_1) = \alpha_p + \big(d(v, v_1) - w(v, p)\big) < \alpha_p + d(p, v_1)$ by (ii). Thus, v_1 must be promoted to level $\alpha_v + d(v, v_1)$ (lines 19 and 20) and then enqueued (line 21). Second, when v_i $(i \in [1, k-1])$ is dequeued and then v_{i+1} is examined, v_{i+1} must be promoted, as in the case of v_1, from its previous level $(\alpha_p + d(p, v_{i+1}))$ to a new level $(\alpha_v + d(v, v_{i+1}))$ and then enqueued, unless the same was done earlier along a different shortest path from v to v_{i+1}. For this reason, $v' = v_k$ is always placed at level $\alpha_v + d(v, v')$. ■

Lemma 3: Algorithm 5 skips (i.e., does not follow the outgoing edges of) every vertex $v' \neq v$ such that $d(v, v') = w(v, p) + d(p, v')$.

Proof: The level l' of vertex v' on line 16 is determined along a path from v to v' after v is placed at level α_v. Therefore, $l' \geq \alpha_v + d(v, v')$. On line 17, the initial value of l'' (i.e., the previous level of $v' \in \mathcal{V}_p$) is $\alpha_p + d(p, v') = \big(\alpha_v + w(v, p)\big) + d(p, v') = \alpha_v + \big(w(v, p)) + d(p, v')\big) = \alpha_v + d(v, v') \leq l'$. In this case, Algorithm 5 skips v' since the conditions on lines 18 and 19 do not hold. ■

Theorem 2: (Correctness of Δ-PFS) Given vertices v and p, a map $L : \mathcal{V}_p \rightarrow \mathbb{R}$ such that $L[v'] = L[p] + d(p, v)$ for each $v' \in \mathcal{V}_p$, and $s = \sum_{v' \in \mathcal{V}_p} d(p, v')$, Algorithm 5 updates L and s so that $L[v'] = L[v] + d(v, v')$ for all $v' \in \mathcal{V}_v$ and $s = \sum_{v' \in \mathcal{V}_v} d(v, v')$.

Proof: By Lemma 2, for all $v' \in \mathcal{V}_{v/p}$, $L[v'] = \alpha_v + d(v, v') = L[v] + d(v, v')$. If $v' \in \mathcal{V}_v - \mathcal{V}_{v/p}$, since Algorithm 5 skips v' (Lemma 3), $L[v']$ keeps the previous level of v' (i.e., $\alpha_p + d(p, v')$). Thus, $L[v'] = \alpha_p + d(p, v') = \big(\alpha_v + w(v, p)\big) + d(p, v') = \alpha_v + \big(w(v, p) + d(p, v')\big)$. Furthermore, since $v' \notin \mathcal{V}_{v/p}$, $d(v, v') = w(v, p) + d(p, v')$. Therefore, $L[v'] = L[v] + d(v, v')$. The reason for preserving the above property of s is explained in Sections III-A and III-B. ■

Appendix C
Time Complexity of Δ-PFS

Algorithms 4 and 5 have the following computational complexity:

Theorem 3: (Computational Overhead) Algorithm 4 takes $O(|\mathcal{E}_v| + |\mathcal{V}_v| \log |\mathcal{V}_v|)$ time and Algorithm 5 takes $O(|\mathcal{E}_{v/p}| + |\mathcal{V}_{v/p}| \log |\mathcal{V}_{v/p}|)$ time, where $\mathcal{V}_v$, $\mathcal{E}_v$, $\mathcal{V}_{v/p}$ and $\mathcal{E}_{v/p}$ are defined as in Table I. For unweighted graphs, the running times of Algorithms 4 and 5 can be reduced to $O(|\mathcal{E}_v| + |\mathcal{V}_v|)$ and $O(|\mathcal{E}_{v/p}| + |\mathcal{V}_{v/p}|)$, respectively.

Proof: Algorithm 4 performs lines 1-4 once, lines 5-8 $|\mathcal{V}_v|$ times, and lines 9-16 at most $|\mathcal{E}_v|$ times. If Q is implemented as a Fibonacci heap [10], lines 4, 13, 16 complete in constant time whereas line 6 takes $O(\log |\mathcal{V}_v|)$ time. Given an unweighted

graph, a FIFO queue implementation of Q suffices and can perform line 6 in constant time. The other lines in Algorithms 4 take constant time.

Algorithm 5 runs lines 1-7 once, lines 8-14 up to $|\mathcal{V}_{v\imath p}|$ times (Lemma 2), and lines 15-23 at most $|\mathcal{E}_{v\imath p}|$ times. Lines 4 and 21 complete in constant time whereas line 9 takes $O(\log|\mathcal{V}_{v\imath p}|)$ time. The rest of the proof is the same as that for Algorithm 4. ■