On-line Exact Shortest Distance Query Processing

Jiefeng Cheng, Jeffrey Xu Yu
The Chinese University of Hong Kong, China
{jfcheng,yu}@se.cuhk.edu.hk

ABSTRACT

Shortest-path query processing not only serves as a long established routine for numerous applications in the past but also is of increasing popularity to support novel graph applications in very large databases nowadays. For a large graph, there is the new scenario to query intensively against arbitrary nodes, asking to quickly return node distance or even shortest paths. And traditional main memory algorithms and shortest paths materialization become inadequate. We are interested in graph labelings to encode the underlying graphs and assign labels to nodes to support efficient query processing. Surprisingly, the existing work of this category mainly emphasizes on reachability query processing, while no sufficient effort has been given to distance labelings to support querying exact shortest distances between nodes. Distance labelings must be developed on the graph in whole to correctly retain node distance information. It makes many existing methods to be inapplicable. We focus on fast computing distance-aware 2-hop covers, which can encode the all-pairs shortest paths of a graph in $O(|V| \cdot |E|^{1/2})$ space. Our approach exploits strongly connected components collapsing and graph partitioning to gain speed, while it can overcome the challenges in correctly retaining node distance information and appropriately encoding all-pairs shortest paths with small overhead. Furthermore, our approach avoids pre-computing all-pairs shortest paths, which can be prohibitive over large graphs. We conducted extensive performance studies, and confirm the efficiency of our proposed new approaches.

1. INTRODUCTION

With the rapid growth of World-Wide-Web, new data archiving and analyzing techniques, graph data becomes more and more important. New graph applications include web mining, biological network analysis, social networks, XML databases and bioinfomatics. Efficiently querying and analyzing graph data draws much attention from database community recently. In this paper, we are interested in a primitive kind of queries, namely, *shortest-path queries* on very large graphs, directed or undirected. It asks for the distance between arbitrary two nodes in the underlying graph, or even a shortest path between the two nodes, to be returned. Shortest-path query processing not only serves as a long-established routin for numerous applications in the past but is of increasing popularity to support novel graph applications in very large databases nowadays. For example, it is quite common to ask for the *Erdös* distance between two authors in a collaboration network, or to request a shortest route between two cities in a domestic road network. In social networks, a commonly used influence measure in sociology is "distance centrality" [40]. And the distance information provides a basis for selecting the influential nodes [26] and features for mining community membership, growth, and evolution [3]. In biological networks, shortest paths and distance information are employed to identify optimal pathways and valid connectivity in metabolic networks [32]. To handle the intensive processing of shortest path queries to support top-k keyword queries [18] and twig matching [17] in graphs, even the whole edge transitive closure with distance information of the underlying graph is employed. Shortest-path queries are so important that fast answering for them is almost mandatory.

Extensive study has been done on shortest-path query process, which can be main memory methods or with index on disks. One of the most well-known algorithms is Dijkstra's algorithm [12]. It is a main memory algorithm and computes shortest paths from scratch, i.e. without preprocessing the underlying graph. One best time bound of it is $O(|E| \lg \lg |V|)$ [10]. For all-pairs shortest path computing, the algorithm in [31] achieves $O(|V| \cdot |E| + |V|^2 \log \log |V|)$ time. In [42], over graphs with small integer weights ranged from $-M$ to M, an algorithm is proposed which runs in $O(M^{0.68}|V|^{2.58})$. However, because the system may preferably need real-time response, computing a shortest path from scratch at querying time can still become unacceptable for very large graphs. We are interested in graph labelings to encode all-pairs shortest paths and assign short labels to nodes in a preprocessing step, in order to support efficient on-line exact distance and shortest path query processing. We do not intend to advance all-pairs shortest path computing like [31, 42], which are orthogonal to our problem and whose output can be used to provide us the all-pairs shortest paths to be encoded. A naive solution to materialize all-pairs shortest paths can be used to answer queries on any two nodes, but the $O(|V|^3)$ space cost can be prohibitive. It is worth noting that much work within this category has been devoted to reachability query processing. For example, [1, 6, 39, 38] all considered assigning intervals to each node to encode the set of reachable nodes from that node in the underlying graph. However, this type of work can hardly be extended to support shortest-path queries, because these intervals are usually based on a spanning tree (called tree-cover in [1, 6, 20]) of the underlying graph, where the distance information is incomplete for the whole graph. The similar setback also exists in the work [19, 20] which decomposes the graph into a number of simple structures, such as chains or trees, to compress the transitive closure. Furthermore, many approaches usually focus on directed acyclic graphs [1, 6, 39, 7, 38, 8, 20] rather than general directed graphs. For shortest-paths queries, however, general directed graphs can impose new challenges. Distance labeling was considered by [30, 37, 14]. But they are for undirected graphs and can not be applied to directed graphs. Some recent work [15, 16, 29, 11] pre-computes distance information for a number of selected nodes, which are called *landmarks*, and considers approximating the distance of any given two nodes based on pre-computed distance information. These approaches can avoid pre-computing all-pairs shortest paths. But they are thus unaware of the number of shortest paths and exact distance information that can be recorded by those landmarks, and can not support exact distance and shortest path answering. [41] preprocesses a graph in $\tilde{O}(|E| \cdot |V|^{\omega})$ time, where $\omega < 2.376$ is the exponent of matrix multiplication. And any distance query afterward can be answered in $O(|V|)$ time. However, the preprocessing along needs $O(|V|^2)$ space for matrix operations, which can still be too large for a large and dense graph.

Cohen et al. in [9] proposed a family of labelings over directed or undirected graphs, to support both reachability and shortest-path queries, based on 2-hop covers. A distance-aware 2-hop cover [9] provides a time- and space-efficient solution to shortest path and distance query processing and it only needs $O(|V| \cdot |E|^{1/2})$ space. Though it is appealing for the theoretical bound on the time and space complexity, unfortunately, computing 2-hop covers is challenging. In practice, it almost takes two days even with a 64-processor and 80G-memory super machine for the *DBLP* data set in [34, 33].

Contributions: In this paper, we concentrate ourselves on fast computation of distance-aware 2-hop covers for large graphs. It imposes new challenges, because the existing methods [7, 8, 5] often focus on reachability 2-hop covers (without distance) and assume the underlying graph to be directed acyclic graphs (or simply DAGs). However, the strongly connected components in a graph can introduce many additional shortest paths in addition to those can be found in its DAG components. Another challenge is how to efficiently obtain the shortest paths already covered and those left to be covered as required in computation, since the all-pairs shortest paths for a large graph can be particularly large. In addition, a divide-and-conquer approach [33, 34] needs to partition the underlying graph and is challenged with incompleteness of node distance information. In our approach, (1) we exploit strongly connected components collapsing and graph partitioning to gain speed; (2) we investigate correctly retaining node distance information and appropriately encoding all-pairs shortest paths with small overhead under graph partitioning; (3) we propose heuristics and strategies to achieve high-compression rate in fast computing distance-aware 2-hop covers; (4) our approach avoids pre-computing all-pairs shortest paths as required by existing approaches; 5) we conducted extensive performance studies, and confirm the efficiency of our proposed new approaches.

Paper organization: We give our problem statement and discuss the hardness of the problem to be studied in Section 2. We then discuss existing work with their limitations in computing the distance labeling. Section 2. We give the outline of our new approach in Section 4, and discuss two main issues, namely, obtaining a DAG from graph G in Section 5, and top-down partitioning strategies, in Section 6, in order to compute the distance labeling. The experimental results are discussed in Section 7 followed by discussions of related work in Section 8. We conclude this paper in Section 9.

2. PROBLEM STATEMENT

Let $G = (V, E)$ be an edge-weighted directed graph, where V is a set of nodes, and E is a set of edges, and every edge weight is a non-negative number. Here the graph G is a *simple* graph which has no loops nor multiple edges. Given any two nodes u and v in G, the distance from u to v, denoted $\delta(u, v)$, is the minimum total weight along a path from u and v. And a shortest path from u to v is a path from u to v with the minimum total weight $\delta(u, v)$. We simply use δ to indicate $\delta(u, v)$ when it is obvious.

We focus on answering the exact shortest distance queries, $Q_d(u, v)$, in this paper, which is to query the shortest distance from u to v, $\delta(u, v)$, and we will also address how to extend the same framework to answer the exact shortest path queries, $Q_p(u, v)$, which is to query the shortest path from u to v, with low overhead.

In order to fast answer shortest distance queries, $Q_d(u, v)$, one approach is to pre-compute all the shortest paths beforehand. In brief, let D_G be a set of pairs, $\langle (u, v)\colon \delta(u, v)\rangle$ (or simply $\langle (u, v)\colon \delta\rangle$), for every (u, v) in the edge transitive closure of G, denoted as T_G. With the pre-computed D_G, the query $Q_d(u, v)$ can be answered as to retrieve $\delta(u, v)$ for the given (u, v) in D_G. With additional information as the predecessor along with (u, v), to be maintained in D_G, the query $Q_p(u, v)$ can be answered efficiently too. However, $|D_G|$ can be very large for a large and dense graph G.

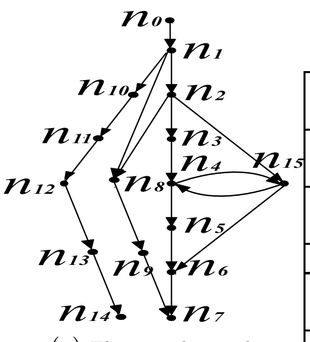
The problem we study in this paper is to efficiently compute a graph distance labeling of minimum size. And, with such a distance labeling, we can quickly answer exact shortest distance, $Q_d(u, v)$, as well as exact shortest path queries, $Q_p(u, v)$.

In the literature, 2-hop distance labeling is such a labeling proposed by Cohen et al. in [9] for G. It assigns every node $v \in V$ a label $L(v) = (L_{in}(v), L_{out}(v))$, where $L_{in}(v)$ and $L_{out}(v)$ are subsets of D_G whose entries are in the form of $\langle (w, v)\colon \delta\rangle$ and $\langle (v, w)\colon \delta\rangle$. Then, a query, $Q_d(u, v)$, querying the shortest distance from u to v, can be answered by

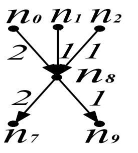
$$\min\{\delta_1 + \delta_2 | \langle (u, w)\colon \delta_1\rangle \in L_{out}(u) \wedge \langle (w, v)\colon \delta_2\rangle \in L_{in}(v)\} \quad (1)$$

It means to find a node, w, called center, in both $L_{out}(u)$ and $L_{in}(v)$ with the minimum $\delta_1 + \delta_2$. $Q_d(u, v)$ will be infinite if there is no such w found using Eq. (1).

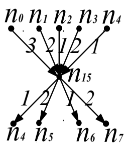
Below, we discuss how to compute the 2-hop distance labeling for G, and the difficulty of minimizing $|D_G|$ in computing the 2-hop distance labeling.



(a) The sample graph



(b) n_8 cluster



(c) n_{15} cluster

$L(n_0)=(\emptyset,\{\langle n_1{:}1\rangle\langle n_2{:}2\rangle\langle n_8{:}2\rangle\langle n_{10}{:}2\rangle\langle n_{11}{:}3\rangle\langle n_{15}{:}3\rangle\})$
$L(n_1)=(\{\langle n_1{:}0\rangle\},\{\langle n_1{:}0\rangle\langle n_2{:}1\rangle\langle n_8{:}1\rangle\langle n_{11}{:}2\rangle\langle n_{15}{:}2\rangle\})$
$L(n_2)=(\{\langle n_1{:}1\rangle\langle n_2{:}0\rangle\},\{\langle n_2{:}0\rangle\langle n_8{:}1\rangle\langle n_{15}{:}1\rangle\})$
$L(n_3)=(\{\langle n_1{:}2\rangle\},\{\langle n_4{:}1\rangle\langle n_{15}{:}2\rangle\})$
$L(n_4)=(\{\langle n_4{:}0\rangle\langle n_{15}{:}1\rangle\},\{\langle n_4{:}0\rangle\langle n_{15}{:}1\rangle\})$
$L(n_5)=(\{\langle n_4{:}1\rangle\langle n_{15}{:}2\rangle\},\{\langle n_6{:}1\rangle\})$
$L(n_6)=(\{\langle n_{15}{:}1\rangle\},\{\langle n_6{:}0\rangle\})$
$L(n_7)=(\{\langle n_6{:}1\rangle\langle n_8{:}2\rangle\langle n_9{:}1\rangle\langle n_{15}{:}2\rangle\},\emptyset)$
$L(n_8)=(\{\langle n_8{:}0\rangle\},\{\langle n_8{:}0\rangle\})\cdots$
$L(n_{15})=(\{\langle n_{15}{:}0\rangle\},\{\langle n_{15}{:}0\rangle\})$

(d) Labels

Figure 1: An example for the distance-aware 2-hop cover

2-hop distance labeling and 2-hop cover: A 2-hop distance labeling for G is derived from a distance-aware 2-hop cover of G, which is defined to be a set of (distance-aware) *2-hop clusters*. A 2-hop cluster is constructed based on a so-called *center* node w, together with a set of ancestor nodes of w and a set of descendant nodes of w, which represent shortest distances from these ancestor nodes to these descendant nodes, via w.

Formally, let $ancs(w)$ and $desc(w)$ be the sets consisting of all entries in the form of $\langle (a, w){:}\delta\rangle$ and $\langle (w, d){:}\delta\rangle$ in D_G, respectively. Let $A_w \subseteq ancs(w)$ and $D_w \subseteq desc(w)$. A 2-hop cluster, denoted $S(A_w, w, D_w)$, compactly represents a subset of D_G, or, in other words, covers the shortest paths from $u \in A_w$ to $v \in D_w$ via w as many as possible. By "cover", it means the following equation (Eq. (2)) must hold.

$$\delta(a, w) + \delta(w, d) = \delta(a, d) \tag{2}$$

where $\langle (a, w){:}\, \delta(a, w)\rangle \in A_w$ and $\langle (w, d){:}\, \delta(w, d)\rangle \in D_w$. It suggests that the shortest path from u to v is the concatenation of the shortest path from u to w and the shortest path from w to v. Note that it is possible that some $a \in A_w$ and $d \in D_w$, are not covered by $S(A_w, w, D_w)$, due to the constraint given in Eq. (2).

The distance-aware 2-hop cover L of G compactly encodes all entries in D_G by computing 2-hop clusters. Given L, the 2-hop distance labeling for G is determined by adding $\langle (a, w){:}\delta\rangle$ into all $L_{out}(a)$ and $\langle (w, d){:}\delta\rangle$ into all $L_{in}(d)$, for every $S(A_w, w, D_w)$ in L. Let P_L be the set of all entries $\langle (a, d){:}\ \delta_1 + \delta_2\rangle$ for any $\langle (a, w){:}\ \delta_1\rangle$ and $\langle (w, d){:}\ \delta_2\rangle$ in L, $P_L \supseteq D_G$ as shown in [9], which implies that all shortest paths can be answered using the 2-hop distance labeling. Below, we use the distance-aware 2-hop cover and the 2-hop distance labeling interchangeably.

Example 1: A running example is shown in Fig. 1. A small directed graph is given in Fig. 1(a) where all the edge weights are 1. Fig. 1(b) and Fig. 1(c) show two 2-hop clusters, namely, $S(A_{n_8}, n_8, D_{n_8})$ and $S(A_{n_{15}}, n_{15}, D_{n_{15}})$, respectively. A 2-hop cluster is illustrated with all its nodes arranged in three layers, where the center w is place in the middle, and A_w and D_w are placed on the top and bottom, respectively. Also, an edge, (u, w) or (w, v), in a 2-hop cluster indicates a shortest path associated with the total weight for the path.

In Fig. 1(c), $S(A_{n_{15}}, n_{15}, D_{n_{15}})$ covers those shortest paths from n_0, n_1, n_2, n_3, n_4 and n_{15} (in $A_{n_{15}}$) to n_4, n_5, n_6, n_7 and n_{15} (in $D_{n_{15}}$), but does not cover the two paths from n_0 and n_1 to n_7. Consider the path from n_0 to n_7. The shortest path from n_0 to n_7 is not the concatenated path of the shortest path from n_0 to n_{15} and the shortest path from n_{15} to n_7 ($\delta(n_0, n_{15}) + \delta(n_{15}, n_7) > \delta(n_0, n_7)$). It is worth of noting that $S(A_{n_8}, n_8, D_{n_8})$ covers the shortest path from n_0 to n_7, as given in Fig. 1(b), for $\delta(n_0, n_8) + \delta(n_8, n_7) = \delta(n_0, n_7)$.

Fig. 1(d) gives the resulting 2-hop distance labels computed. Note that, as shown in Fig. 1(d), v are omitted from $L_{in}(v)$ and L_{out} to reduce the space. We maintain $\langle w{:}\ \delta\rangle$ instead of $\langle (w, v){:}\delta\rangle$ in $L_{in}(v)$ and $\langle w{:}\ \delta\rangle$ instead of $\langle (v, w){:}\delta\rangle$ in L_{out}.

Now consider how to answer the shortest distance query, $Q_d(n_0, n_7)$. Because both n_8 and n_{15} appear in $L_{out}(n_0)$ and $L_{in}(n_7)$, there is at least a path from n_0 to n_7. The distance from n_0 to n_7 via n_{15} is $5 = 3 + 2$, and the distance from n_0 to n_7 via n_8 is $4 = 2 + 2$. Therefore, the shortest distance from n_0 to n_7 is 4, which is the minimum of 5 and 4, as specified in Eq. (1). We call a labeling *redundant* like n_0 to n_7 via n_{15}, the compression rate will be higher if redundancy is less in the 2-hop cover. □

The 2-hop cover program: The size of the 2-hop cover is given as $|L| = \sum(|A_v| + |D_v|)$ for all identified $S(A_v, v, D_v)$. The minimum size 2-hop cover problem is NP-hard [9]. The quality of a 2-hop cover L is weighted by a *compression rate*, which is defined to be the ratio of the number of covered entries to the total size of L. The higher compression rate, the better quality of a 2-hop cover. Cohen et al. give an algorithm to compute 2-hop cover based on the set cover problem [23]. It chooses 2-hop clusters in iterations. In each iteration, it examines all different 2-hop clusters, $S(A_w, w, D_w)$, by varying the center w and all possible A_w and D_w. The algorithm picks the best 2-hop cluster in each iteration, where the best $S(A_w, w, D_w)$ has the maximum ratio as given in Eq. (3) below.

$$\frac{|S(A_w, w, D_w) \cap D'_G|}{|A_w| + |D_w|} \tag{3}$$

Here, D'_G is the set of shortest paths not yet covered and is initially set to be D_G. Eq. (3) means to newly cover as many entries as the dividend with a cost as small as the divisor. However, pre-computing all-pairs shortest paths can be prohibitive for a large graph.

3. THE CHALLENGES AND EXISTING SOLUTIONS

In this section, we reexamine the existing work to compute a 2-hop cover (not distance-aware 2-hop cover) [33, 34, 7, 8], we discuss the main challenges to compute a distance-aware 2-hop cover.

The divide-and-conquer approach: In [34, 33], Schenkel et al. proposed a divide-and-conquer approach to partition a graph into small graphs, in order to compute 2-hop covers for a large graph in three major steps. First, it evenly partitions the graph G into k subgraphs: $G_1, G_2, \cdots, G_k$, where the all-pairs shortest paths of each graph G_i can be read into

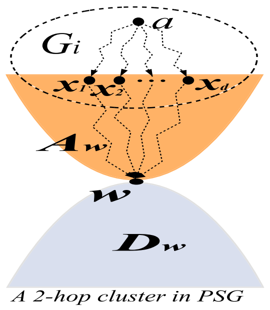


Figure 2: The divide-&-conquer approach

main memory. Second, it computes the 2-hop cover for every subgraph G_i, for $1 \leq i \leq k$, and stores the computed 2-hop covers on disk. [9] can be applied to compute the 2-hop cover for each subgraph G_i. The third step is called the cover joining phase, which builds an overall 2-hop cover by joining the k 2-hop covers for the k subgraphs. In computing the cover joining phase, [34] builds an auxiliary graph, called PSG, including all cross-partition edges and additional edges to specify the connectivity between boundary nodes in all subgraphs as a higher level graph. It then computes the 2-hop cover L' of PSG and obtains the overall cover L for G, by augmenting L' using the 2-hop covers computed for G_i ($1 \leq i \leq k$). As reported in [34, 33], the third step becomes the bottleneck of the whole processing, on which most of the running time is consumed.

The algorithms in [34, 33] can be used to compute the distance-aware 2-hop cover. However, in addition to the bottleneck in the third step, it needs high overhead to compute the shortest paths, and the resulting 2-hop cover can be unnecessarily large. Consider the case shown in Fig. 2. Here, we assume that there is a subgraph G_i in which node a is an ancestor of the nodes $x_1, x_2, \cdots, x_d$ in G_i, that are involved in the cross-partition edges. Consequently, all nodes, $x_1, x_2, \cdots, x_d$, also appear in PSG. Furthermore, suppose that is a 2-hop cluster, $S(A_w, w, D_w)$, in PSG, that contains all $x_1, x_2, \cdots, x_d$ in A_w. In computing the 2-hop cover L for G by augmenting the 2-hop cover L' obtained for PSG, it needs to identify the shortest path from a to w (Fig. 2). It must examine all entries such as $\langle(a, x_i)\colon \delta(a, x_i)\rangle$ and $\langle(x_i, w)\colon \delta(x_i, w)\rangle$, for $1 \leq i \leq d$. There can be many unwanted entries in the resulting 2-hop cover, $\langle(a, x)\colon\delta(a, x)\rangle$, such that $\delta(a, x) + \delta(x, w) > \delta(a, w)$. (Recall n_0 to n_7 via n_{15} in Example 1.)

DAG-based approach: In [7, 8], we studied DAG-based approaches to compute the 2-hop cover for G. Given a directed graph G, we first identify all strongly connected components (or simply SCCs) in G, and obtain a DAG, G', by representing every SCC as a node in G' where all in/out edges to/from a SCC are represented as in/out edges to/from the representative node in G'. It is because, for answering a non distance-aware path query from node u to v, it is easy to see that if v is reachable from u via a certain node, w in a SCC, then v is reachable from u via any node in the SCC. Second, we compute the 2-hop cover for G' by utilizing the multi-interval labeling [1] and R-tree to reduce the high cost of selecting the best 2-hop cluster in every iteration, as indicated in the 2-hop cover program. Third, we determine the 2-hop cover for G based on the 2-hop cover computed for G'. All nodes in a SCC will share the same graph labels. The DAG-based approaches outperform the divide-&-conquer approaches in terms of computational cost and compression rate [7, 8]. But, there are two main difficulties to compute the distance-aware 2-hop cover based on the DAG-based approach.

- **Issue-1**: It cannot take the same approach to obtain a DAG graph, G', by condensing SCCs in G as representative nodes in G', and then obtain the distance-aware 2-hop cover for G based on the distance-aware 2-hop cover for G'. It is because that a node w in a SCC on the shortest path from u to v does not necessarily mean that every node in the SCC is on the shortest path from u to v.

- **Issue-2**: The cost of dynamically selecting the best 2-hop cluster, based on Eq. (3) and Eq. (2), in an iteration, in the 2-hop cover program, cannot be reduced using the multi-interval labeling [1] and R-tree, because such techniques cannot handle distance information. In consequence, in order to be able to select the best 2-hop cluster in an iteration, it needs to either maintain all the uncovered distance-aware 2-hop clusters (refer to D'_G in Eq. (3)), or all the already covered distance-aware 2-hop clusters ($D_G - D'_G$), which consumes very large space and high computational cost.

In this paper, we will study how to overcome the two difficulties and compute the distance-aware 2-hop cover.

4. A NEW DAG-BASED APPROACH

In this section, we discuss a new DAG-based approach. Our approach is based on an important observation: if a 2-hop cluster, $S(A_w, w, D_w)$, is found that covers all shortest paths containing the center node w, we can remove w from the underneath graph, because we do not need to consider again any shortest paths via w any more. Motivated by the observation, in our new algorithm, to deal with Issue-1, instead of condensing a SCC into a representative node in a modified graph as done in [7, 8], we collapse every SCC into DAG by removing a small number of nodes from the SCC repeatedly until we obtain a DAG graph. To deal with Issue-2, when constructing 2-hop clusters, we propose a new technique to reduce the redundancy[1] in 2-hop clusters by taking the already identified 2-hop clusters into consideration, and therefore avoid storing all-pairs shortest paths (D'_G in Eq. (3)) to select the next best 2-hop cluster. In addition, we also propose heuristics to reduce the cost of selecting the next best 2-hop cluster.

Our new algorithm, called *DAPar* for Distance-Aware Partitioning, is outlined in Algorithm 1. The control flow of Algorithm 1 is sketched in Fig. 4. There are two main phases in *DAPar*. In the first phase (line 1-7), it attempts to obtain a DAG $G^{\downarrow}$ for a given graph G by removing small number of nodes, $\hat{V}_{C_i}$, from every SCC, $C_i(V_{C_i}, E_{C_i})$. In computing a SCC $C_i(V_{C_i}, E_{C_i})$, every node, $w \in \hat{V}_{C_i}$ is taken as a center, and $S(A_w, w, D_w)$ is computed to cover shortest paths for graph G (line 4). Then, all nodes in $\hat{V}_{C_i}$ will be removed, and a modified graph G is constructed as an induced subgraph of $G(V, E)$, denoted as $G[V \setminus \hat{V}_{C_i}]$, with the set of nodes $V \setminus \hat{V}_{C_i}$. It is important to note that nodes in $\hat{V}_{C_i}$ are

[1]refer to Example 1.

Algorithm 1 *DAPar*(G)

Input: a directed graph G
Output: distance-aware 2-hop cover labeling

```
1: identify all SCCs, $C_1(V_{C_1}, E_{C_1}), C_2(V_{C_2}, E_{C_2}), \cdots$, for $G$;
2: for every $C_i(V_{C_i}, E_{C_i})$ do
3:     $\hat{V}_{C_i} \leftarrow getDAG(C_i(V_{C_i}, E_{C_i}))$;
4:     construct $S(A_w, w, D_w)$ for every $w \in \hat{V}_{C_i}$;
5:     $G \leftarrow G[V \setminus \hat{V}_{C_i}]$;
6: end for
7: $G^{\downarrow} \leftarrow G$; {$G^{\downarrow}$ is a DAG}
8: find a node-separator, $V_w$, for $G^{\downarrow}$;
9: partition $G^{\downarrow}$ into two disconnected subgraphs $G_{\top}$ and $G_{\bot}$ by
   removing $V_w$ from $G^{\downarrow}$;
10: construct $S(A_w, w, D_w)$ for every $w \in V_w$;
11: if $G_{\top}$ ($G_{\bot}$) is small then
12:     compute the 2-hop cover for $G_{\top}$ ($G_{\bot}$);
13: else
14:     let $G^{\downarrow}$ be $G_{\top}$ ($G_{\bot}$); goto line 8;
15: end if
```

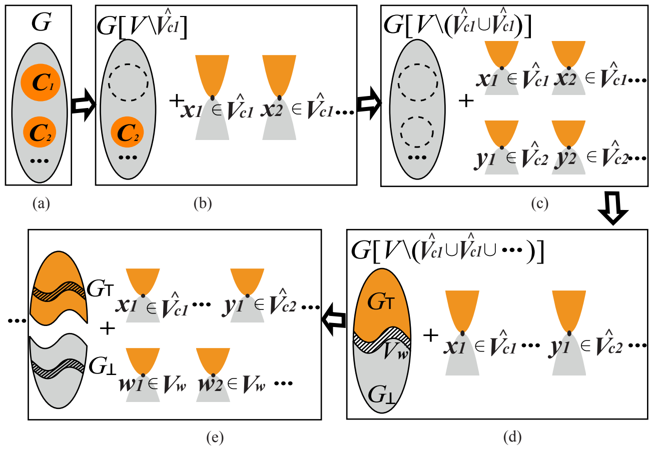
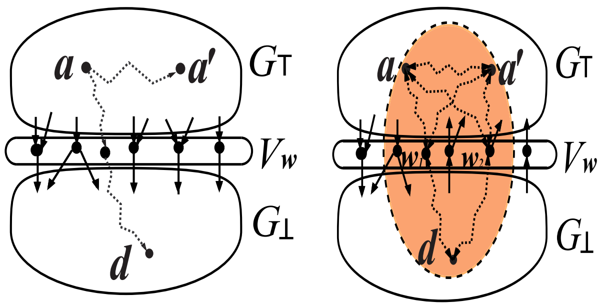


Figure 3: The Algorithm Steps

considered as effective since they are in SCC and may cover many shortest paths, and, in addition, those nodes do not need to be reconsidered again to cover shortest paths based on our observation. Here, Fig. 4 (a) shows a graph G with several SCCs. Fig. 4 (b)-(d) illustrate the main idea of collapsing SCCs while computing 2-hop clusters. All shortest paths covered are the union of the shortest paths covered in 2-hop clusters, $S(A_w, w, D_w)$, for every node, $w \in \hat{V}_{C_i}$. Such $S(A_w, w, D_w)$ is computed to cover the shortest paths in G globally.

In the second phase, for the obtained DAG $G^{\downarrow}$, we take a top-down partitioning approach to partition the DAG $G^{\downarrow}$ (line 8-15), based on our early work in [8]. if a $G^{\downarrow}$ is large, we partition $G^{\downarrow}$ evenly by finding a set of node-separator V_w. With removal of V_w from $G^{\downarrow}$, two disjoint subgraph $G_{\top}$ and $G_{\bot}$ are obtained. We compute 2-hop clusters for nodes $w \in V_w$, and repeat the same procedure if either $G_{\top}$ or $G_{\bot}$ is not small until we compute the 2-hop cover for the original graph G. Fig. 4 (d) and (e) tell that the graph can be partitioned hierarchically. Note that a key issue here is how to obtain $\hat{V}_C$ and V_w to give high-quality 2-hop covers. Note that, in a similar fashion, all the nodes in V_w are not needed further for covering shortest paths in the following steps.



(a) The Bisection of A DAG (b) In Presence of SCCs

Figure 4: The Setback of Graph Partitioning in Presence of SCCs

4.1 More on Partitioning

Graph partitioning is essential to speed the computing of 2-hop cover for a large and dense graph. In this section, we discuss the issues of partitioning for a directed graph without SCCs and with SCCs.

Consider 2-hop cover computing when partitioning a directed graph without SCCs (DAG). As shown in Fig. 4(a), a DAG, $G^{\downarrow}$, can be evenly partitioned with a set of node-separator V_w, resulting in two subgraphs $G_{\top}$ and $G_{\bot}$. We can obtain high quality distance-aware 2-hop covers using bisection for a DAG. There are two main issues. First, the shortest paths covered using nodes in the node-separator V_w, as the centers 2-hop clusters, are global, in the graph $G^{\downarrow}$. In other words, they cover the shortest paths across $G_{\top}$ and $G_{\bot}$. Let the shortest paths covered using V_w be P_w. Furthermore, after the bisection, let the two sets of all shortest paths covered in resulting subgraphs $G_{\top}$ and $G_{\bot}$ are denoted as $D_{G_{\top}}$ and $D_{G_{\top}}$, respectively. Then, due to the fact $P_w \cap D_{G_{\top}} = \emptyset$ and $P_w \cap D_{G_{\bot}} = \emptyset$, we will never cover the same shortest paths in $G_{\top}$ or $G_{\bot}$ more than necessary. It can be easily verified because $G^{\downarrow}$ is a DAG. However, this does not hold any longer in G in the presence of SCCs.

Consider a SCC as shown in Fig. 4(b). Here, a, a' and some nodes in V_w, such as w_1 and w_2, are in the same SCC. In other words, both the shortest paths from a to a' and from a' to a can possibly contain w_1 and w_2. Therefore, P_w, if we use the same approach to evenly partition $G^{\downarrow}$ using the node-separator V_w, can contain shortest paths between nodes in $G_{\top}$ and/or between nodes in $G_{\bot}$ respectively. Consequently, those paths will be covered multiple times when computing 2-hop clusters in $G^{\downarrow}$, $G_{\top}$, as well as $G_{\bot}$. It reduces the compression rate for computing the 2-hop cover. As a remark, it becomes important to collapse SCCs in a general directed graph G, to obtain a DAG $G^{\downarrow}$ from G.

We collapse a SCC, $C_i(V_{C_i}, E_{C_i})$, into a DAG, by selecting a small number of nodes such as a set $\hat{V}_{C_i} \subseteq V_{C_i}$. Then, we construct 2-hop clusters based on the nodes in $\hat{V}_{C_i}$ followed by removing $\hat{V}_{C_i}$ from G. It is important that selection of a small number of nodes in $\hat{V}$ leads to reduction of unnecessary redundancy in the 2-hop cover. It is because a node, w, in a SCC can be connected by many nodes in a graph, because any node that connect to a node w' in the same SCC connect to w. It also means that we will have large A_w and D_w of 2-hop clusters, and we will include many redundancy in our resulting 2-hop cover if we do not select the nodes wisely.

5. FROM SCC TO DAG

In this section, we discuss two issues: collpasing SCCs from graph G to obtain a DAG, and how to compute 2-hop clusters along with the process of collapsing SCCs.

5.1 Collapsing SCCs

We discuss how to collapse SCCs into a DAG (refer to line 3 in Algorithm 1). We can find out all SCCs in graph G, denoted $C_1(V_{C_1}, E_{C_1}), C_2(V_{C_2}, E_{C_2}), \cdots)$, in $O(|V|+|E|)$ time [10].

Consider a SCC, $C(V_C, E_C)$, in G, we find a minimum number of nodes, $\hat{V}_C \subseteq V_C$, such that all cycles in the SCC C are disconnected by removing $\hat{V}_C$ from C. We consider all simple cycles in $C(V_C, E_C)$ and compute $\hat{V}_C$ in two steps, where a simple cycle contains no other cycle in the graph. In the first step, we find out all simple cycles in C, $\sigma_1, \sigma_2, \cdots \sigma_m$, and determine all member nodes on each cycle σ. In the second step, we choose a minimum number of nodes $\hat{V}_C$ such that any cycle σ contains at least one node in $\hat{V}_C$. The removal of all nodes in $\hat{V}_C$ will disconnect all cycles, σ, in C. This problem has been studied in [36, 22] and it requires $O((V_C+E_C)(m+1))$ time. In our problem setting, however, the number of simple cycles m can be very large and listing all simple cycles is not practical. Therefore, we adapt an *early stop* strategy in the depth-first traversal [36] when enumerating all simple cycles in C. That is, in the first step, every time when the number of cycles identified reaches a given limit, we stop and start selecting a number of nodes to disconnect those already identified cycles (the second step task). After removing those nodes from C, we then continue the first step to search the remaining cycles in C. We repeat the two steps until no cycles in C can be identified.

The second step is a set cover problem [10]. The ground set Δ to be covered consists of all cycles in C, namely, $\Delta = \{\sigma_1, \sigma_2, \cdots, \sigma_m\}$. Consider all nodes $v_1, v_2, \cdots$, on all simple cycles in G. We associate a subset of Δ, Δ_i, for each member node v_i, s.t. Δ_i consists of all simple cycles σ_j that contains v_i. The problem is to find a set, $\mathbb{S}$, consisting of a minimum number of sets from $\Delta_1, \Delta_2, \cdots$, such that any $\sigma_i \in \Delta$ can be found in at least one set $\Delta_j \in \mathbb{S}$. Then, $\hat{V}_C$ is immediately known from $\mathbb{S}$. The set cover problem is NP-hard and we use the greedy algorithm [10] to obtain $\mathbb{S}$. Note: $C[V_C \setminus \hat{V}_C]$ contains no cycle and the SCC C will be collapsed by deleting all nodes of $\hat{V}_C$ from G. It is easy to see that after removing all nodes in $\hat{V} = \hat{V}_{C_1} \cup \hat{V}_{C_2} \cup \cdots \cup \hat{V}_{C_l}$, all cycles in G will be disconnected thus G contains no SCCs. We obtain a DAG $G^{\downarrow}$ as the induced graph $G[V \setminus \hat{V}]$.

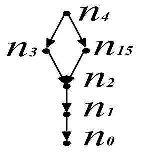
5.2 Reducing 2-Hop Clusters

We discuss a technique to reduce the redundancy in 2-hop clusters by already identified 2-hop cluster (refer to line 4 in Algorithm 1 for example). It can be used to construct any 2-hop clusters in a 2-hop cover program.

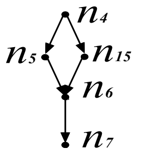
In order to remove all nodes in $\hat{V}$ from G, we need to construct $|\hat{V}|$ 2-hop clusters. And each node $w \in \hat{V}$ becomes a center for one of those 2-hop clusters. Since it is not practical to precompute all shortest paths in G and maintain all already computed 2-hop clusters for G, we do not want to compute all 2-hop clusters based on Eq. (3). A naive solution is to construct $S(A_w, w, D_w)$ by including all nodes in *ancs*(w) into A_w and *desc*(w) into D_w, for all $w \in \hat{V}$.

Algorithm 2 *ReduceCluster*

```
1: For each node $w \in \hat{V}$, two shortest-paths graphs, $\mathcal{A}_w$, and $\mathcal{D}_w$, are constructed;
2: Choose the node $w \in \hat{V}$ such that the number of descendants of $w$ in all $\mathcal{A}$ and $\mathcal{D}$ is the largest among all nodes in $\hat{V}$; construct $S(A_w, w, D_w)$ based on $\mathcal{A}_w$ and $\mathcal{D}_w$; remove $w$ from $\hat{V}$; if $\hat{V}$ becomes empty then terminate;
3: Update all $\mathcal{A}$ and $\mathcal{D}$ for nodes in $\hat{V}$ to remove all descendants of $w$ from $\mathcal{A}$ and $\mathcal{D}$, and goto 2;
```



(a) $\mathcal{A}_{n_4}$



(b) $\mathcal{D}_{n_4}$

Figure 5: Shortest-Paths Graphs

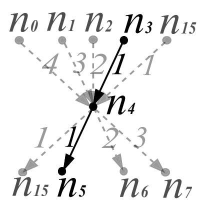


Figure 6: $S(A_{n_4}, n_4, D_{n_4})$

Though this naive solution can obviously cover all shortest paths containing any $w \in \hat{V}$, it in fact introduces many redundant entries in A_w and D_w of $S(A_w, w, D_w)$. Therefore, we propose a heuristics to construct all required 2-hop clusters and reduce the redundancy in $S(A_w, w, D_w)$ by already identified $S(A_{w'}, w', D_{w'})$. Its overall steps is described in Algorithm *ReduceCluster* (Algorithm 2). To understand the three steps, we first explain two important components in step 1 (line 1) and step 3 (line 3) in Algorithm 2.

Shortest-paths graphs: [10] describes a shortest-paths tree constructed in single source shortest path algorithms. When the shortest path from the source s to some node t is found, the immediate predecessor on the path is also recorded for t. Thus, after the single source shortest path algorithm finds all shortest paths from s to its reachable nodes, a shortest-paths tree can be formed, where a tree edge is added for each node from its recorded immediate predecessor to it. Such a shortest-paths tree contains one shortest path from the source s to every reachable node t from s. However, in order to obtain a *shortest-paths graph* for the source s, we make a slight modification. We record all immediate predecessors for t with which the same minimum weight can be derived for a shortest path from s to t. This is easy with Dijkstra's algorithm [10]. Due to the lack of space, we omit the details. The shortest-paths graph is a DAG, because it only contains edges from nearer nodes t to relatively far nodes t' such as $\delta(s,t) < \delta(s,t')$. The shortest-paths tree can be easily extended to the *shortest-paths graph*. The shortest-paths graph for s contains all alternative shortest paths from s to every reachable node t from s.

For each node $w \in \hat{V}$, two shortest-paths graphs, $\mathcal{A}_w$ and $\mathcal{D}_w$, are constructed. $\mathcal{D}_w$ is constructed upon G to include all shortest paths starting from w; while $\mathcal{A}_w$ is constructed upon $G^{\uparrow}$ to include all shortest paths ending at w in G. $G^{\uparrow}$ is an auxiliary graph obtained from G which contains the same set of nodes in $G^{\downarrow}$, but edges in the reversed direction as in $G^{\downarrow}$. That is, we add an edge (v,u) to $G^{\uparrow}$ only if (u,v) is in $G^{\downarrow}$. For our running example, $\mathcal{A}_{n_4}$ and $\mathcal{D}_{n_4}$ is shown in Fig. 5. To construct $S(A_w, w, D_w)$, nodes in D_w and A_w can be directly obtained from $\mathcal{A}_w$ and $\mathcal{D}_w$, respectively.

Update for $\mathcal{A}$ and $\mathcal{D}$: Consider $S(A_{w'}, w', D_{w'})$ to be an

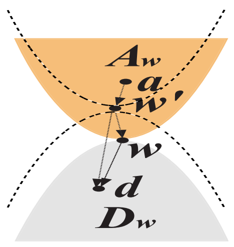


Figure 7: $S(A_w, w, D_w)$ and $S(A_{w'}, w', D_{w'})$

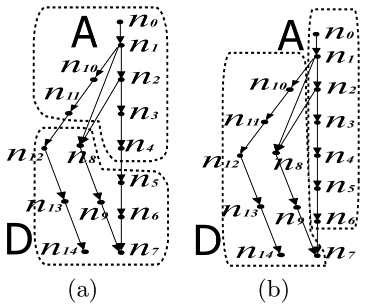


Figure 8: Two Bisections for the Fixed Strategy

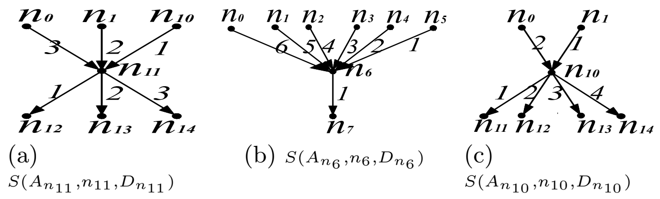


Figure 9: 2-hop clusters Constructed During Top-Down Partitioning

identified 2-hop cluster in the 2-hop cover program. A update operation on $\mathcal{A}_w$ and $\mathcal{D}_w$ can be used to reduce redundancy in $S(A_w, w, D_w)$, based on the correlation between $S(A_{w'}, w', D_{w'})$ and $S(A_w, w, D_w)$. The following theorem provides the basis for our update operation.

Theorem 1: *Given two 2-hop cluster $S(A_{w'}, w', D_{w'})$ and $S(A_w, w, D_w)$. If $\mathcal{A}_w$ (or $\mathcal{D}_w$) contains w' and a node $a \in A_w$ (or $d \in D_w$) is also a descendant of w' in $\mathcal{A}_w$ (or $\mathcal{D}_w$), then a (or d) and w' is redundant in A_w (or D_w) and can be removed from A_w (or D_w), respectively.* □

Proof Sketch: As shown in Fig. 7, we consider that a is a descendant of w' in $\mathcal{A}_w$. For any node d of the descendant of w, it is also the descendant of w'. And $S(A_{w'}, w', D_{w'})$ can cover the shortest path from a to d before w is considered as the center, that is, $\delta(a, d) = \delta(a, w') + \delta(w', d)$. However, if $S(A_{w'}, w', D_{w'})$ cannot cover this way, namely $\delta(a, d) < \delta(a, w') + \delta(w', d)$, then $S(A_w, w, D_w)$ cannot either. We explain it as follows with Fig. 7. There are $\delta(a, w) = \delta(a, w') + \delta(w', w)$ (the subpath property of a shortest path) and $\delta(w', d) \leq \delta(w', w) + \delta(w, d)$ (the triangle inequality). Based on them, we can have $\delta(a, w') + \delta(w', d)\rangle \leq \delta(a, w) + \delta(w, d)\rangle$. So there exists $\delta(a, d) < \delta(a, w) + \delta(w, d)$. Thus in any case, we can remove $\langle (a, w), \delta(a, w)\rangle$ from A_w. On the other hand, we can obtain the similar argument for $\langle (w, d), \delta(w, d)\rangle$ when d is a descendant of w' in $\mathcal{D}_w$. □

Therefore, in step 2 (line 2) of *ReduceCluster*, it tries to pick the 2-hop cluster which can maximally reduce remaining $\mathcal{A}$ and $\mathcal{D}$. And only nodes in the updated $\mathcal{A}_w$ and $\mathcal{D}_w$ needs to be included in A_w and D_w based on Theorem 1, as given in step 3 (line 3) in *ReduceCluster*. We only need to compute two shortest-paths graphs for a small number of nodes in $\hat{v}$ and update them by centers of identified 2-hop clusters effectively.

Example 2: For our running example, $S(A_{n_4}, n_4, D_{n_4})$ can be reduced using $S(A_{n_{15}}, n_{15}, D_{n_{15}})$. From Fig. 5, we can find that n_0, n_1 and n_2 are descendants of n_{15} in $\mathcal{A}_{n_{15}}$ while n_6 and n_7 are descendants of n_{15} in $\mathcal{D}_{n_{15}}$. So, together with n_{15}, they will be removed from A_{n_4} and D_{n_4} respectively as shown in Fig. 6. □

6. TOP-DOWN PARTITIONING

In this section, we focus on top-down partitioning (line 8-15) in Algorithm 1. Given a DAG $G^{\downarrow}$, with a set of node-separator V_w, it is partitioned into two subgraphs $G_{\top}$ and $G_{\perp}$. We discuss two strategies to identify 2-hop clusters, based on V_w. and cover all shortest paths between $G_{\top}$ and $G_{\perp}$. The first one is the fixed strategy, which fixes the centers in the subsequent 2-hop cover program. We further propose a flexible strategy as an enhancement to identify quality center candidates to form V_w. The flexible strategy is fast and robust in producing high quality 2-hop covers.

6.1 The Fixed Strategy

It is intuitive that a number of high quality 2-hop clusters can result a high compression rate 2-hop cover. In [8], we prove that with the same space cost, $S(A_w, w, D_w)$ covers more paths when the cardinality of A_w and D_w are more similar. That is, *balanced* 2-hop cluster are preferred. We in [8] proposed to bisect a DAG $G^{\downarrow}$ in the middle before the 2-hop cover program begin: it sorts all nodes in the DAG $G^{\downarrow}$ using topological sort [10]. All directed edges (ordered pairs) are arranged to direct toward one direction. Then, the bisection is to cut $G^{\downarrow}$ in the middle. One half of nodes go to $\mathbf{A}$, and the other half go to $\mathbf{D}$, and $\mathbf{A}$ and $\mathbf{D}$ are two subsets of $V(G)$, respectively. Next, we obtain an induced subgraph, $G_c(V_c, E_c)$, with the set of edges $E_c = \{(a, d) | a \in \mathbf{A}, d \in \mathbf{D}\}$. From G_c, we find a set of nodes $V_w \subset V_c$ such that every edge in E_c is incident to some node $v_w \in V_w$. Here, $G_{\top}$ and $G_{\perp}$ are two induced subgraphs that contain sets of nodes, $V(G_{\top}) = \mathbf{A} \setminus V_w$ and $V(G_{\perp}) = \mathbf{D} \setminus V_w$, respectively.

Our first strategy finds a node-separator V_w with the same process as above. Then, we construct 2-hop clusters based on all $w \in V_w$ in the same way as discussed in Section 5.2. All shortest paths $\langle (a, d), \delta\rangle$, where $a \in G_{\top}$ and $d \in G_{\perp}$, can thus be covered, because V_w is a node-separator and any shortest path from a to d must hence contain some node $w \in V_w$. Also, all $S(A_w, w, D_w)$ covers all shortest paths containing any $w \in V_w$.

The fixed strategy can already compute high-compression-rate distance-aware 2-hop covers and gracefully scale to large graphs. However, the following example suggests that it is still inadequate to handle graphs which are sparse or have structures as long chains and tree-like subgraphs. To refine the graph partitioning to be adaptable to the structure of the underlying graph, we propose another strategy.

Example 3: Fig. 8 shows two possible ways of bisection on the DAG using the fixed strategy. First, we obtain $V_w = \{n_4, n_8, n_{11}\}$, and construct three 2-hop clusters, namely, $S(A_{n_4}, n_4, D_{n_4})$ (Fig. 6), $S(A_{n_8}, n_8, D_{n_8})$ (Fig. 1(b)), and $S(A_{n_{11}}, n_{11}, D_{n_{11}})$ (Fig. 9(a)). Second, we have $V_w = \{n_6, n_8, n_{10}\}$, which results in three 2-hop clusters: $S(A_{n_6}, n_6, D_{n_6})$ (Fig. 9(b)), $S(A_{n_8}, n_8, D_{n_8})$ (Fig. 1(b)), $S(A_{n_{10}}, n_{10}, D_{n_{10}})$ (Fig. 9(c)). We will select the first way of bisection in Fig. 8(a) over the second way in Fig. 8(b), because 2-hop clusters on n_6 and n_{10} are not so balanced as those on n_4 and n_{11}. □

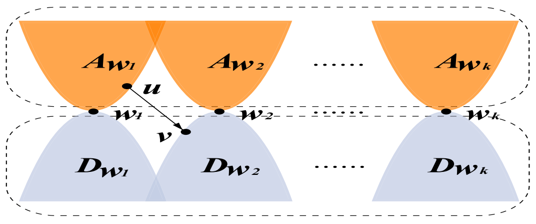
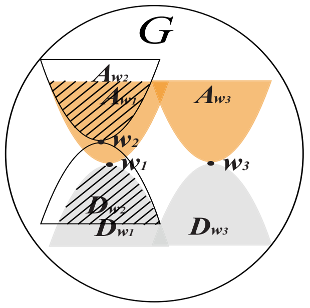
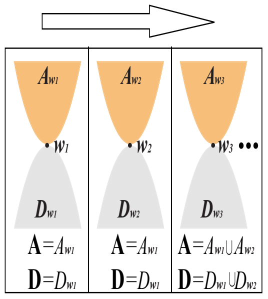


Figure 10: A Bisection Based on 2-Hop Clusters



(a) $S(A_{w_2}, w_2, D_{w_2})$ not Good



(b) $S(A_{w_2}, w_2, D_{w_2})$ is Skipped

Figure 11: Our Flexible Strategy

6.2 The Flexible Strategy

The hierarchical partitioning of [8] resembles a *fixed* strategy in that it first find V_w and then construct 2-hop clusters by restricting centers in V_w, and thus V_w is predetermined before a 2-hop cover program starts. In this paper, we further propose a flexible strategy for top-down partitioning. It features a bisection on the underlying graph resulted from a 2-hop cover program where centers are not predetermined to better adapt to local unusual structure such as long chains and tree-like subgraphs.

In this strategy, a node-separator V_w, where $V_w = V_{w_1} \cup V_{w_2}$, is obtained in two steps. The first step performs a bisection on $G^{\downarrow}$ and computes V_{w_1}. The purpose is to fast roughly partition graph G. The second step is to compute V_{w_2} based on the bisection in the previous step. The purpose is to ensure that $G_\top$ and $G_\perp$ are surely disjoint, in other words, there is no any path cross $G_\top$ and $G_\perp$ when V_w is removed. Fig. 10 illustrates the idea. In the first step, we attempt to identify a set of 2-hop clusters, $S(A_{w_1}, w_1, D_{w_1})$, $S(A_{w_2}, w_2, D_{w_2})$, $\cdots$, to roughly partition G. All nodes in A_{w_i} will be included in the graph $G_\top$, whereas all nodes in D_{w_i} will be included in graph $G_\perp$. After the first step, there exist links, for example, a path from u in A_{w_2} to v in D_{w_2}. The second step is to cut those cross paths.

Below, we discuss the first step called a bisection based on 2-hop clusters followed the discussion on the second step called Sweeping remaining cross-partition edges.

A bisection based on 2-hop clusters: The main idea of this step is described as follows. Initially, we compute all 2-hop clusters, $S(A_{w_1}, w_1, D_{w_1})$, $S(A_{w_2}, w_2, D_{w_2})$, $\cdots$, for all nodes in G, and select the best $S(A_{w_i}, w_i, D_{w_i})$, among all 2-hop clusters, based on Eq. (3). In a 2-hop cover program, in every iteration, it needs to find the next best 2-hop cluster, which requires reranking. We want to find high-quality rough node-separator V_{w_1}, and we want to avoid reranking which comes with high cost. As shown in Fig. 11, suppose we select w_2 as the center for the first $S(A_{w_1}, w_1, D_{w_1})$, we want to select $S(A_{w_3}, w_3, D_{w_3})$ as the next best rather than $S(A_{w_2}, w_2, D_{w_2})$. It is because the overlapping between the two 2-hop clusters, $S(A_{w_1}, w_1, D_{w_1})$ and $S(A_{w_2}, w_2, D_{w_2})$ is very large, whereas there is no overlapping between the two 2-hop clusters, $S(A_{w_1}, w_1, D_{w_1})$ and $S(A_{w_3}, w_3, D_{w_3})$. This step results in two sets of nodes, $\mathbf{A}$ and $\mathbf{D}$, based on 2-hop clusters identified in the 2-hop cover program, where $\mathbf{A}$ contains all A_{w_i} computed and $\mathbf{D}$ contains all D_{w_i} (Fig. 10). $\mathbf{A}$ and $\mathbf{D}$ will be used to derive the $G_\top$ and $G_\perp$.

The above goal can be achieved as below. Compute and sort all 2-hop clusters, $S(A_{w_1}, w_1, D_{w_1})$, $S(A_{w_2}, w_2, D_{w_2})$, $\cdots$, for all nodes in G, and select $S(A_{w_i}, w_i, D_{w_i})$ along the precomputed order. Suppose 2-hop cluster $S(A_{w_i}, w_i, D_{w_i})$ is picked, $\mathbf{A}$ and $\mathbf{D}$ will be updated immediately to include A_{w_i} and D_{w_i} excluding w_i, respectively. In the next $(i+1)$-th iteration, we only consider $S(A_{w_{i+1}}, w_{i+1}, D_{w_{i+1}})$ satisfying $w_{i+1} \notin \mathbf{A} \cup \mathbf{D}$, as depicted in Fig. 12(a). It is clear for each a in $A_{w_{i+1}}$ (or (d in $D_{w_{i+1}}$), we have $a \notin \mathbf{D}$ (or $d \notin \mathbf{A}$), respectively. Otherwise, we must have $w_{i+1} \notin \mathbf{D}$ (or $\mathbf{A}$). Therefore, except for the centers w_1, w_2, $\ldots$, w_k, any node in $G^{\downarrow}$ can only be included in either $\mathbf{A}$ or $\mathbf{D}$ exclusively. We let $V_{w_1} = \{w_1, w_2, \cdots, w_k\}$.

In order to construct $S(A_w, w, D_w)$, we just include all nodes in $\mathcal{A}(w)$ (or $\mathcal{D}(w)$) into A_w (or D_w), respectively. Because each shortest path from a node a in $\mathcal{A}(w)$ to w (or from w to a node d in $\mathcal{D}(w)$) can only be covered by an corresponding node a in A_w (or d in A_w) and such a in A_w (or d in A_w) cannot be redundant. Therefore, we only need to obtain a fixed ranking for all nodes w by the score of Eq. (3) for $S(A_w, w, D_w)$. And we do not need to consider updating the ranking during the 2-hop cover program, because all A_w or D_w do not change. So the 2-hop cover program examines every center and chooses 2-hop clusters according to such an ranking. It scans the entire ranking list of centers to finish the first step. To avoid computing out all-pairs shortest paths covered by a 2-hop cluster, we use $\frac{|A_w| \times |D_w|}{|A_w| + |D_w|}$ to replace Eq. (3), where the number of shortest paths covered by $S(A_w, w, D_w)$ is roughly calculated as $|A_w| \times |D_w|$. We illustrate this step in Example 4 using our running example.

Example 4: For our running example in Fig. 8, the 2-hop cover program forms the initial ranking as n_4, n_3, n_5, n_8,n_6, n_9, n_2, n_{10}, n_{11}, n_7, n_1, and first picks the 2-hop cluster based on n_4 (Fig. 6). We have $\mathbf{A} = \{n_1, n_2, n_3, n_4\}$ and $\mathbf{D} = \{n_4, n_5, n_6, n_7\}$. Then, 2-hop cluster of centers in $\mathbf{A} \cup \mathbf{D}$ will not be considered any more. Only n_8, n_9, n_{10} and n_{11} can be centers. The next 2-hop cluster picked is constructed from n_8 (Fig. 1(b)). Then only n_{10} and n_{11} are left for consideration. We construct another 2-hop cluster from n_{11} (Fig. 9(a)) and no more nodes left can be centers. This generates our preferred way of bisection in Fig. 8(a). □

Sweeping remaining cross-partition edges: Note that V_{w_1} is not necessarily to be a separator of $G^{\downarrow}$, as showed in Fig. 10, because there can be an edge (u, v) which can not be disconnected by removing V_{w_1}. The second step is to sweep cross-partition edges to obtain a node separator of G. We obtain V_{w_2} as follows. We first identify the set of cross-partition edges E_c to be all edges (a, d) such that both are false for $a \in \mathbf{A} \wedge d \in \mathbf{A}$ and $a \in \mathbf{D} \wedge d \in \mathbf{D}$. We initialize V_{w_2} to be $\emptyset$, and iteratively add to V_{w_2} a node v with the largest degree in E_c, that is the node with the

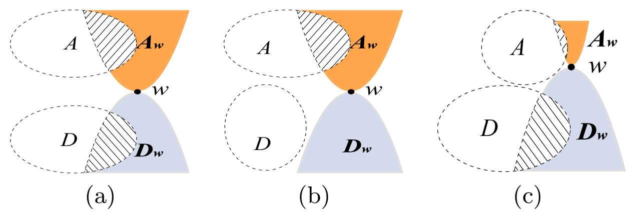


Figure 12: Dealing with Different 2-Hop Clusters

largest number of edges in E_c being incident with v, and remove those edges incident to v from E_c. We repeat it until E_c becomes empty and obtain V_{w_2}. Obviously, $V_{w_1} \cup V_{w_2}$ is a node-separator of $G^{\downarrow}$. We continue to augment L with 2-hop clusters constructed based on V_{w_2} in the same way as in our fixed strategy. Then $G_\top$ and $G_\bot$ are two induced subgraphs that contain sets of nodes, $V(G_\top) = \mathbf{A} \setminus V_{w_2}$ and $V(G_\bot) = \mathbf{D} \setminus V_{w_2}$, respectively.

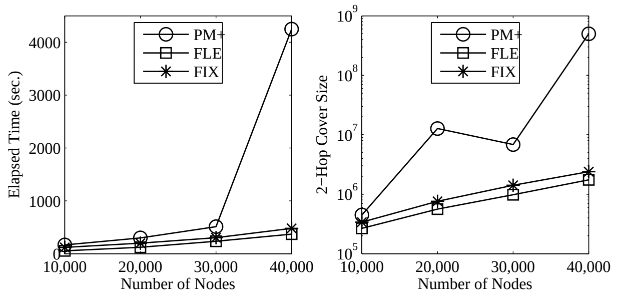
The enhanced scheme: The first enhancement over the above two step basic scheme aims at using less 2-hop clusters to complete the bisection. Consider $S(A_w, w, D_w)$ which is to be picked in the above 2-hop cover program. As illustrated by Fig. 12(b), when A_w or D_w does not overlap with $\mathbf{A}$ or $\mathbf{D}$, we can skip such $S(A_w, w, D_w)$ by directly merging A_w and D_w to $\mathbf{D}$ or $\mathbf{A}$, respectively. That is, if $A_w \cap \mathbf{A} = \emptyset$, all nodes in A_w and D_w are added into $\mathbf{D}$; if $D_w \cap \mathbf{D} = \emptyset$, all nodes in A_w and D_w are added into $\mathbf{A}$. This improvement tries to quickly "tear off" two subgraphs, $G_\top$ and $G_\bot$, from each other.

A further step is to skip $S(A_w, w, D_w)$ which does not have $A_w \cap \mathbf{A} = \emptyset$ (or $D_w \cap \mathbf{D} = \emptyset$), which is illustrated by Fig. 12(c). We still add all nodes in A_w and D_w into $\mathbf{D}$ (or $\mathbf{A}$). It is useful when A_w is very small but D_w is very large (or the reverse), because such a 2-hop cluster confers low compression rate when $|A_w|$ and $|D_w|$ differs largely, as we prefer balanced 2-hop clusters. This enhancement can introduce duplicated subgraphs in $G_\top$ and $G_\bot$, both containing overlapped nodes in $A_w \cap \mathbf{A}$ (or $D_w \cap \mathbf{D}$). It is only triggered when two thresholds meet: We set two thresholds on $|A_w| < \tau_1$ and $|D_w| > \tau_2$, or $|D_w| < \tau_1$ and $|A_w| > \tau_2$.

Example 5: Consider our running example in Fig. 8, after the 2-hop clusters of n_4 and n_8 are picked, we are to construct the 2-hop cluster on n_{11}. Note that all nodes in $D_{n_{11}}$ do not appear in $\mathbf{D}$. So all nodes in $S(A_{n_{11}}, n_{11}, D_{n_{11}})$ (Fig. 9(a)) can be merged into $\mathbf{A}$ and $S(A_{n_{11}}, n_{11}, D_{n_{11}})$ can be skipped. Now, $V(G_\top) = \{n_1, n_2, n_3, n_{10}, n_{11}\}$ and $V(G_\bot) = \{n_5, n_6, n_7, n_9\}$ and the cross cover only includes $S(A_{n_4}, n_4, D_{n_4})$ and $S(A_{n_8}, n_8, D_{n_8})$. □

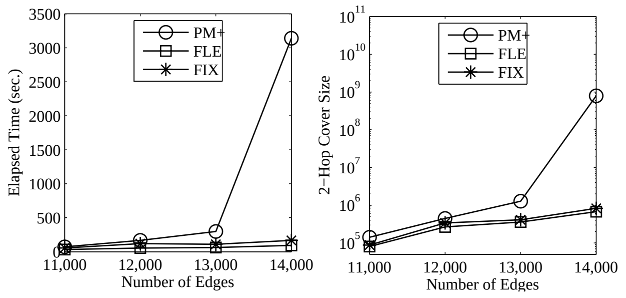
7. PERFORMANCE EVALUATION

We conducted extensive experiment studies to evaluate the performance of different approaches to compute distance-aware 2-hop covers. Specifically, We use PM+ to illustrate the performance of the flat partitioning as in [34]. For our proposed approach, we use FIX to denote the fixed strategy method and FLE the flexible strategy. For all of them, we both use the same center graph approach to compute 2-hop covers for small graphs, when the number of nodes are below the same threshold, in order to appreciate the performance improvement by different strategies. All those algorithms



(a) Elapsed Time (sec.) (b) Cover Size

Figure 13: Performance on Directed Graphs of Increasing Size



(a) Elapsed Time (sec.) (b) Cover Size

Figure 14: Performance on Directed Graphs of Increasing Density

are implemented using C++.

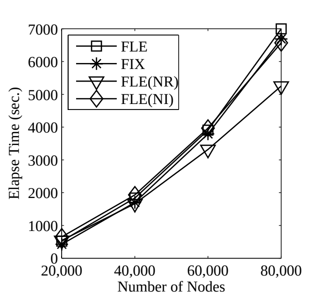
We generated various synthetic data using the random directed graph generator named *GraphBase* developed by Knuth [28]. We vary two parameters, $|V|$ and $|E|$, in the two generators, and use the default values for the other parameters. Extensive tests are also on real datasets, including EcoCyc [27], XMark [35], DBLP[2] and metabolic networks. More detailed specification can be found in Section 7.3. It is important to note that our work focuses on encoding the graph transitive closure or even all-pairs shortest paths in the graph. Even if the graphs to be used during testing can be read into main memory, it is not easy to keep the graph transitive closure or even all-pairs shortest paths in the graph in main memory, to support on-line exact distance queries for any two nodes.

We conducted all the experiments on a PC with a 3.4GHz processor, 180G hard disk and 2GB main memory running Windows XP.

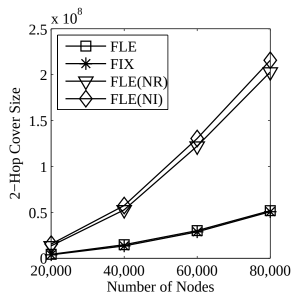
7.1 Exp-1: Bottom-Up Partitioning v.s. Top-Down Partitioning

We conduct experiments to compare PM+, FIX and FLE. Four sets of random directed graphs are generated for this purpose by *GraphBase*. Graphs in each sets contain the same number of nodes and edges but with various seeds for the random graph generator, with a fixed graph density of 1.2 as the ratio of the number edges to the number of nodes. Because this is common among real world data graphs. The 4 sets of graphs contain 10K, 20K, 30K, and 40K nodes respectively, where K is for kilo. We show the performance on one graph for each set, because all graphs in the same

[2]http://dblp.uni-trier.de/xml/

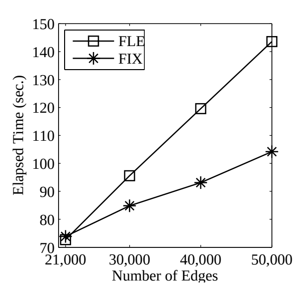


(a) Elapsed Time (sec.)

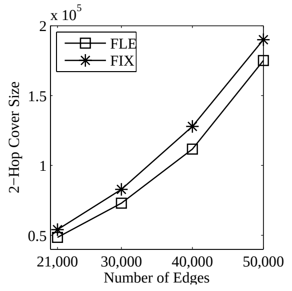


(b) Cover Size

Figure 15: Our Approaches on Graphs with Increasing Size



(a) Elapsed Time (sec.)



(b) Cover Size

Figure 16: Our Approaches on DAGs with Increasing Density

set have similar performance. The total number of shortest paths in the 4 graphs are 12.94M, 43.19M, 100.03M and 165.71M, respectively, where M is for mega. To compare the performance on graphs with various density, we also tested on 4 graphs with 10K nodes. They have 11K, 12K, 13K, and 14K edges, while 3.49M, 12.94M, 18.85M and 26.99M shortest paths, respectively. We illustrate the performance of the three algorithms in Fig. 13 and Fig. 14. The figures show that when the graph is large and dense, FIX and FLE can be significantly faster than PM+, and smaller 2-hop covers can be obtained with FIX and FLE than those from PM+. The main reason is that as the number of nodes or edges in the underlying graph increases, the size of the PSG graph generated by PM+ increases also. So PM+ may need to recursively partition the PSG and perform the cover joining phase repeatedly. And the cover join phase is costly. This is the case on the graph of 40K nodes in Fig. 13 and 14K edges in Fig. 14. The cover join phase is costly. For example, when processing the graph of 14K edges in Fig. 14(a), PM+ spent 3,140 seconds in total. But 2,605 of them are for cover joining. Further more, repeated cover join phases can generate more inaccurate entries. On the graph of 14K edges in Fig. 14(b), total entries generated is around 794M. However, only about 4M entries are with correct distance information and should be kept. In contrast, for the same graph, FLE and FIX only spent 166 and 92 seconds; and the resulted 2-hop cover size is 0.8M and 0.7M, respectively.

7.2 Exp-2: Performance of Different Strategies

For our proposed algorithms, We show their performance on 4 large graphs with a fixed density of 1.5. The 4 graphs contain 20K, 40K, 60K, and 80K nodes respectively. They contain 139.47M, 543.61M, 1.21 billion and 2.16 billion number of shortest paths, respectively. To appreciate the power of our proposed technique to reduce the redundancy in 2-hop clusters (Section 5), we further illustrate the one without reducing the redundancy as FLE(NR). We also illustrate one version without the enhancement Section 6.2 as FLE(NI). Fig. 15 shows the performance of them. In Fig. 15(a) and Fig. 15(b), FLE(NR) and FLE(NI) performs similarly: they all need slightly shorter time but result in larger cover than FLE and FIX. Particularly, the size by FLE and FIX can be smaller than that by FLE(NR) and FLE(NI) up to an order of magnitude. For example, on the 80K nodes graph, FLE(NR) and FLE(NI) spend 5195 and 6572 seconds, respectively. And FLE and FIX spend 6689 and 6998 seconds. But cover size resulted from FLE(NR) and FLE(NI) are as high as 198.48M and 215.67M, respectively. While FLE and FIX can obtain the covers with the size of 51.81M and 50.70M, respectively. Therefore, the proposed optimization techniques such as reduce the redundancy in 2-hop clusters and enhanced flexible partitioning are very effective to get hight-quality 2-hop covers.

To get the difference of FLE and FIX, we tested them on 4 large DAGs with 20,000 nodes by the random DAG generator developed by Johnson baugh [24]. The 4 DAGs have various number of edges as shown in Fig. 16. FLE can be faster than FIX when the graph is sparse, as indicated with the graph with 21K edges. However, when the graph is dense, FLE can obtain a cover with smaller size than can FIX. For example, with graph with 21K edges. The size of the over by FLE is 16.27K smaller than that by FIX, and it spends 26 more seconds than FIX does. FIX can be fast because it can result in small subgraphs $G_\top$ and $G_\perp$. Because FIX fixes a number of centers with most incident edges among all nodes. And all such centers and incident edges will removed from $G_\top$ and $G_\perp$.

7.3 Exp-3: Tests on XML and Real Datasets

We tested several real datasets: Ecoo157, AgroCyc, Anthra, HpyCyc, Human, Mtbra, and VchoCyc are from EcoCyc [27]; Reactome, aMaze, and KEGG are metabolic networks from [38]. DBLP, XMK10M and XMK20M, are XML documents; We selected a subset of DBLP , which consists of all the records for 5 conferences, SIGMOD, VLDB, ICDE, EDBT and ICDT; The DBLP graph contains edges from all parent-child relationships and bibliographic links. Then, for XMK10M and XMK20M, the graph contains edges from parent-child and ID/IDREF relationships. Reactome, aMaze, and KEGG are metabolic networks [38]. Table 1 lists parameters of these graphs including total number of shortest paths and the 2-hop cover size and the elapsed time resulted by FIX, FLE and PM+. For example, on XMK20M, it is a sparse graph. The size of the over by FLE is 3.41M smaller than that by FIX, and FLE uses 5836 seconds less time than FIX does also. For many graphs, PM+ also achieves similar small 2-hop cover size to ours. It is due to those graphs are not large and dense. But it worths noting the time it spends is usually an order of magnitude more than that used by FIX or FLE. Some graphs are too dense or too large for PM+ to perform the cover joining phase (the third step) with our limited main memory capacity. PM+ has to partition the graph recursively and repeatedly performs the joining phase more than once for the same graph. The partial 2-hop cover during the

#	Data Set	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert D_G\rvert$		Time	$\lvert H\rvert$		Time	$\lvert H\rvert$		Time	$\lvert H\rvert$
1	Ecoo157	12,620	17,308	2.4M	FLE	23.07	58,279	FIX	28.00	60,157	PM+	200.25	54,841
2	AgroCyc	13,969	17,694	2.73M	FLE	22.90	60,794	FIX	30.99	61,913	PM+	229.66	91,052
3	aMaze	11,512	28,700	83.2M	FLE	642.31	411,911	FIX	636.91	387,762	PM+	-	-
4	Anthra	13,733	17,307	2.44M	FLE	22.91	66,154	FIX	31.67	65,262	PM+	215.10	57,217
5	HpyCys	5,565	8,474	1.11M	FLE	14.04	39,898	FIX	18.13	39,570	PM+	102.68	33,266
6	Human	40,051	43,879	2.8M	FLE	28.5	84,791	FIX	32.90	84,754	PM+	231.95	133,865
7	Kegg	14,269	35,170	145.65M	FLE	1934.66	824,244	FIX	1934.6	822,891	PM+	-	-
8	Mtbrv	10,697	13,922	2.03M	FLE	20.29	55,577	FIX	10.02	54,989	PM+	173.97	48,430
9	Reactome	3,647	14,447	6.42M	FLE	54.67	100,057	FIX	53.01	90,105	PM+	870.11	52,869
10	Vchocyc	10,694	14,207	2.34M	FLE	21.62	68,362	FIX	28.18	65,222	PM+	195.66	51,054
11	DBLP	52,682	59,395	409,467	FLE	20	88,070	FIX	19	91,076	PM+	173.81	266,834
12	XMK10M	167,866	198,412	2.01G	FLE	2,642	26.68M	FIX	3,435	23.80M	PM+	-	-
13	XMK20M	336,243	397,713	8.45G	FLE	10,727	94.29M	FIX	16,563	97.71M	PM+	-	-

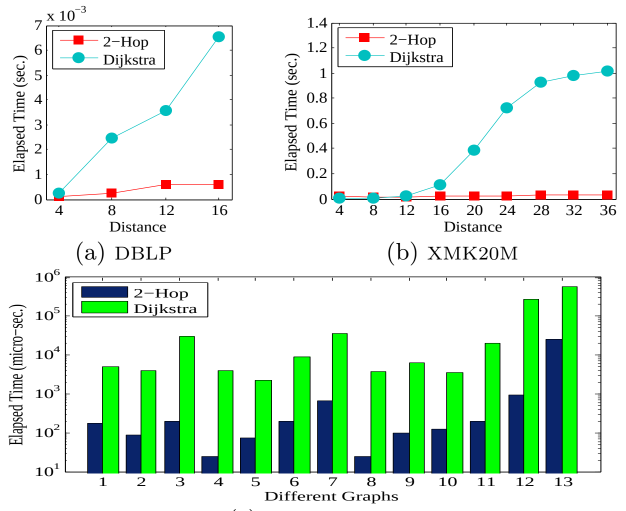
Table 1: Performance on real graphs

joining phase can contains an overwhelming number of total entries before removing inaccurate ones in it.

7.4 Exp-4: Querying Performance

In terms of performance of distance queries for 2-hop covers, we compare the 2-hop cover method FLE with the Dijkstra algorithm. We tested 100 randomly generated queries for the average querying time to be considered for each one. Fig. 17(a) and (b) show their performance regarding different distance value. While Fig. 17(c) shows all random queries for graphs in Table 1. The 2-hop cover can outperform the Dijkstra algorithm up to two orders of magnitudes. For XMK20M graph, the 2-hop cover spends 32 mini-seconds, while Dijkstra needs 1.015 seconds.

The quality of 2-hop covers is important for efficient query processing. For example, 2-hop covers of AgroCyc, Human and DBLP obtained with PM+ are larger than those covers obtained with FLE. Then, the querying performance of larger covers degrades accordingly. In this tests, for the three graphs mentioned above, we can obtain the elapsed time as 0.17, 0.15 and 0.13 mini-seconds with PM+ covers. However, such numbers for the smaller covers are 0.087, 0.098 and 0.049. The smaller 2-hop covers by FLE outperform the larger 2-hop covers by PM+ noticeably.



(a) DBLP (b) XMK20M

(c) Real Datasets

Figure 17: Distance Querying Time

8. RELATED WORK

Extensive work has been done on shortest path query process. Main memory methods includes the Dijkstra's algorithm [12], Floyd [13] and Bellman-Ford [4]. [10] provides a comprehensive introduction to them. [42, 31] are two recent results. Main memory methods are orthogonal to our problem and the output can be used to provide us the all-pairs shortest paths to be encoded. There is also work to only materialize shortest paths in a number of subgraphs of the underlying graphs, which are further organized to attain pruning of nodes for exploration at querying time. Instances are [2], HEPV [21] and HiTi [25].

The more related work to ours follows the paradigm to encode the underlying graphs and assign short labels to nodes, in order to support efficiently query processing. [1, 6, 39, 38] all considered assigning intervals to nodes which encodes the set of reachable nodes from that node in the underlying graph to support reachability query processing. However, they can hardly be extended to support shortest path queries, because these intervals are usually based on a spanning tree (called tree-cover in [1, 6, 20]) of the underlying graph, where distance information are incomplete for the whole graph. Similar problem also exists in some work [19, 20] which decomposes the graph into a number of simple structures, such as chains or trees, to compute and compress the transitive closure. Distance labels were considered by [30, 37, 14]. But they can not be applied to directed graphs. [15, 16, 29, 11] considered materializing complete distance information for a subset of nodes in the graph, called landmarks, for approximating the distance of any given two nodes. Similar to them, we also need to compute the complete distance information for a certain number of centers. However, the major difference is that they can not answer exact distance and shortest path queries for two nodes. While we focus on compressing the whole distance information and shortest paths in the graphs with those centers. [41] preprocesses a graph first to answer any one subsequent distance query in $O(|V|)$ time. However, the preprocessing itself needs $O(|V|^2)$ space for matrix operations. There has been research work focus on efficient computing of 2-hop covers [33, 34, 7, 8, 5]. Surprisingly, existing work emphasizes on the reachability of the 2-hop cover problem, while no sufficient effort has been devoted for distance-aware 2-hop cover problem. Only [33, 34] considered distance-aware 2-hop covers, which is examined in this paper. Our approach significantly outperforms the existing approach when the underlying graph is either large or dense.

9. CONCLUSION

In this paper, we concentrate ourselves on fast computation of distance-aware 2-hop covers for large graphs. We exploit strongly connected components collapsing and graph partitioning to gain speed. We have investigated how to correctly retain node distance information and appropriately encode all-pairs shortest paths with small overhead under graph partitioning. We propose heuristics and strategies to achieve high-compression rate in fast computing distance-aware 2-hop covers. We suggest using the fixed strategy if the graph is dense; while if the graph is sparse, the flexible strategy is recommended.

Acknowledgment

The work described in this paper was supported by grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (No. 418206).

10. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. of SIGMOD'89*, 1989.
[2] R. Agrawal and H. V. Jagadish. Algorithms for searching massive graphs. *IEEE Trans. on Knowl. and Data Eng.*, 06(2), 1994.
[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proc. of KDD '06*, 2006.
[4] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
[5] R. Bramandia, B. Choi, and W. K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *Proc. of WWW '08*, 2008.
[6] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. of VLDB'05*, 2005.
[7] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *Proc. of EDBT'06*, 2006.
[8] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *Proc. of EDBT '08*, 2008.
[9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of SODA'02*, 2002.
[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, 2001.
[11] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Predicting internet network distance with coordinates-based approaches. In *Proc. of SIGCOMM '04*, 2004.
[12] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Math.*, 1:269–271, 1959.
[13] R. W. Floyd. Shortest path. *Communications of the ACM*, 5:345, 1962.
[14] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. *J. Algorithms*, 53(1):85–112, 2004.
[15] A. V. Goldberg and R. F. Werneck. Computing point-to-point shortest paths from external memory. In *Proc. of ALENEX '05*, 2005.
[16] A. V. Goldberg and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In *Proc. of ALENEX '06*, 2006.
[17] G. Gou and R. Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *Proc. of SIGMOD '08*, 2008.
[18] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *Proc. of SIGMOD '07*, 2007.
[19] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.
[20] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *Proc. of SIGMOD '08*, 2008.
[21] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Trans. on Knowl. and Data Eng.*, 10(3), 1998.
[22] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.
[23] D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proc. of STOC'73*, 1973.
[24] R. Johnsonbaugh and M. Kalin. A graph generation software package. In *Prof. of SIGCSE'91*, 1991.
[25] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Trans. on Knowl. and Data Eng.*, 14(5), 2002.
[26] D. Kempe, J. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proc. of KDD '03*, 2003.
[27] I. M. Keseler, J. Collado-Vides, S. Gama-Castro, J. Ingraham, S. Paley, I. T. Paulsen, M. Peralta-Gil, and P. D. Karp. Ecocyc: a comprehensive database resource for escherichia coli. *Nucleic Acids Research*, 33(D334-7), 2005.
[28] D. E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing.* ACM Press, 1993.
[29] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordiantes-based approaches. In *Proc. of INFOCOM '01*, 2001.
[30] D. Peleg. Proximity-preserving labeling schemes. *J. Graph Theory*, 33:167–176, 2000.
[31] S. Pettie. On the shortest path and minimum spanning tree problems. PH.D Dissertation, The University of Texas at Austin, 2003.
[32] S. A. Rahman, P. Advani, R. Schunk, R. Schrader, and D. Schomburg. Metabolic pathway analysis web service (Pathway Hunter Tool at CUBIC). *Bioinformatics*, 21(7):1189–1193.
[33] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex XML document collections. In *Proc. of EDBT'04*, 2004.
[34] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In *Proc. of ICDE'05*, 2005.
[35] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *Proc. of VLDB'02*, 2002.
[36] R. E. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.*, 2(3):211–216, 1973.
[37] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. of STOC '01*, 2001.
[38] S. TrißI and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. of SIGMOD '07*, 2007.
[39] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proc. of ICDE'06*, 2006.
[40] S. Wasserman and K. Faust. *Social Network Analysis.* Cambridge University Press, 1994.
[41] R. Yuster and U. Zwick. Answering distance queries in directed graphs using fast matrix multiplication. In *Proc. of FOCS '05*, 2005.
[42] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.