Fast Computing Reachability Labelings for Large Graphs with High Compression Rate

Jiefeng Cheng [*], Jeffrey Xu Yu [*], Xuemin Lin [♭], Haixun Wang [†], Philip S. Yu [‡]

[*] *The Chinese University of Hong Kong, China,* {jfcheng,yu}@se.cuhk.edu.hk
[♭] *University of New South Wales, Sydney, Australia,* lxue@cse.unsw.edu.au
[†] *T. J. Watson Research Center, IBM, USA,* haixun@us.ibm.com
[‡] *University of Illinois at Chicago, USA,* psyu@cs.uic.edu

ABSTRACT

There are numerous applications that need to deal with a large graph and need to query reachability between nodes in the graph. A 2-hop cover can compactly represent the whole edge transitive closure of a graph in $O(|V| \cdot |E|^{1/2})$ space, and be used to answer reachability query efficiently. However, it is challenging to compute a 2-hop cover. The existing approaches suffer from either large resource consumption or low compression rate. In this paper, we propose a hierarchical partitioning approach to partition a large graph G into two subgraphs repeatedly in a top-down fashion. The unique feature of our approach is that we compute 2-hop cover while partitioning. In brief, in every iteration of top-down partitioning, we provide techniques to compute the 2-hop cover for connections between the two subgraphs first. A cover is computed to cut the graph into two subgraphs, which results in an overall cover with high compression for the entire graph G. Two approaches are proposed, namely a node-oriented approach and an edge-oriented approach. Our approach can efficiently compute 2-hop cover for a large graph with high compression rate. Our extensive experiment study shows that the 2-hop cover for a graph with 1,700,000 nodes and 169 billion connections can be obtained in less than 30 minutes with a compression rate about 40,000 using a PC.

1. INTRODUCTION

Graph structured data is enjoying an increasing popularity as Web technology and new data management and archiving techniques advance. Numerous emerging applications need to work with graph-like data. Instances include navigation behavior analysis for Web usage mining [3], web site analysis [11], and biological network analysis for life science [22]. In addition, *RDF* allows users to explicitly describe semantical resource in graphs [4]. And querying and analyzing graph structured data becomes important. As a major standard for representing data on the World-Wide-Web, *XML* provides facilities for users to view data as graphs with two different links, the parent-child links (document-internal links) and reference links (cross-document links). *XLink* (*XML* Linking Language) [9] and *XPointer* (*XML* Pointer Language) [10] provide more facilities for users to manage their complex data as graphs and integrate data effectively. The dominance of graphs in real-world applications demands new graph data management so that users can access graph data effectively and efficiently.

Reachability queries, to test whether there is a path from a node v to another node u in a directed graph, have being studied [1, 7, 17, 18, 23, 21] and are deemed to be a very basic type of queries for many applications. Consider a semantic network that consists of people as nodes in the graph and relationships among people as edges in the graph. There are needs to know whether two people are related somehow for security reason [2]. On biological networks, where nodes are either molecules, or reactions, or physical interactions of living cells, and edges are interactions among them, there is an important question to "find all genes whose expressions is directly or indirectly influenced by a given molecule" [22]. All those questions can be mapped into reachability queries. Reachability queries are so common that fast processing for them are required. The needs of such a reachability query can be even found in XML when two types of links are treated the same.

Being introduced in [7], a 2-hop cover can compactly represent the whole edge transitive closure of a graph in $O(|V| \cdot |E|^{1/2})$ space, providing a time- and space-efficient solution to reachability query processing. Despite the importance of the theoretical bound on the time and space complexity for 2-hop covers, the cost for computing the minimum 2-hop cover can be very high in practice. It needs to precompute the transitive closure which requires large memory space. In [19], Schenkel, Theobald and Weikum run Cohen et al's algorithm on a 64 processor Sun Fire-15000 server with 180 gigabyte memory for a subset of *DBLP* which consists of 344 millions of connections. It took 45 hours and 23 minutes using 80 gigabytes of memory to find the 2-hop cover which is in size of 1,289,930 entries.

Contributions of this paper: (1) Different from the existing partitioning in a flat fashion, we propose a hierarchical partitioning in a top-down fashion. (2) We focus on 2-hop cover sensitive partitioning, and compute 2-hop cover while partitioning. (3) We studied two approaches, namely, a node-oriented approach and an edge-oriented approach. The node-oriented approach focuses on selecting a set of nodes to cut a graph while computing 2-hop cover.

EDBT'08, March 25–30, 2008, Nantes, France.

The edge-oriented approach considers edges as nodes, while computing 2-hop cover when partitioning, and it does not need to add additional nodes. (4) Our approaches do not assume the pre-computed transitive closure for the underlying graph. (5) We conducted extensive performance studies. Our extensive experiment studies shows that the 2-hop cover for a graph with 1,700,000 nodes and 169 billion connections can be obtained in less than 30 minutes with a compression rate about 40,000 using a PC.

Organization of this paper: We review the 2-hop cover problem in Section 2 and existing approaches in Section 3. A novel framework, a 2-hop sensitive graph partitioning method is then introduced in Section 4. Section 5 proposes the first solution within the framework based on a node-separator using an R-tree and operations on rectangles. Section 6 discusses the second solution based on an edge-separator using only intervals and linear-time operations on them. We conducted extensive experiment evaluation on our proposed approaches in Section 7. Section 8 gives related work. Finally, the paper is concluded by Section 9.

2. THE 2-HOP COVER PROBLEM

We introduce 2-hop Labelings and 2-hop cover in brief. Let $G = (V, E)$ be a directed graph. The 2-hop reachability labeling for G is proposed by Cohen et al. in [7] to efficiently process *reachability queries* in the form of $u \leadsto v$, where u and v are two nodes in G. $u \leadsto v$ returns true if and only if there is a directed path in G from u to v. In other words, let T_G be the edge transitive closure of G, $u \leadsto v$ is true if $(u, v) \in T_G$. We call such a pair (u, v) a connection. Note: T_G can be very large for a large and dense graph G.

A *2-hop reachability labeling* over graph G assigns every node $v \in V$ a label $L(v) = (L_{in}(v), L_{out}(v))$, where $L_{in}(v), L_{out}(v) \subseteq V$, and $u \leadsto v$ is true if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. A 2-hop reachability labeling for G is derived from a *2-hop cover* of G, which is defined upon a set of *2-hop clusters*. Let $ancs(w)$ and $desc(w)$ be the set of all ancestors of node w and the set of all descendants of w, in G, respectively. And let $A_w \subseteq ancs(w) \cup \{w\}$ and $D_w \subseteq desc(w) \cup \{w\}$. A 2-hop cluster, $S(A_w, w, D_w)$, represents a set of connections from every node a in A_w to every node d in D_w, such as $\{(a, d) \mid a \in A_w \wedge d \in D_w\}$. The 2-hop cover L of G compactly encodes all connections of $S(A_w, w, D_w)$ by adding a *center*, w, into all $L_{out}(a)$ and $L_{in}(d)$, and covers every connection in T_G for graph G. Let all connections covered by L be P_L. P_L consists of all (a, d) such that $L_{out}(a) \cap L_{in}(d) \neq \emptyset$, and $P_L = T_G$ [7]. In this paper, we use the 2-hop cover and the 2-hop reachability labeling interchangeably, because L consists of L_{in} and L_{out} for all nodes in graph G. The size of the 2-hop cover is given as below.

$$|L| = \sum_{v \in V(G)} (|L_{in}(v)| + |L_{out}(v)|) \qquad (1)$$

The optimal 2-hop cover problem is to find the minimum size 2-hop cover for a given graph $G(V, E)$, which is proved to be NP-hard [7]. Based on the greedy algorithm for minimum set cover problem [14], Cohen et al. give an approximation algorithm to get a nearly optimal 2-hop cover which is larger than the optimal one at most $O(\log |V|)$.

The algorithm proceeds in an iterative manner. In each iteration, it examines all different 2-hop clusters, $S(A_w, w, D_w)$, by varying A_w and D_w for different node w as center. 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. (2) below.

$$\frac{|S(A_w, w, D_w) \cap T'|}{|A_w| + |D_w|} \qquad (2)$$

Here, T' is the set of uncovered connections by then. Eq. (2) means to cover as many uncovered connections as the dividend with a cost as small as the divisor. The algorithm completes when all connections in G are covered. We refer to the above iteration processing as the *2-hop cover program* in this paper. In [6] we showed that we can solve the minimum 2-hop cover problem effectively by employing another merit function to replace Eq. (2) in a 2-hop cover program as in Eq. (3).

$$|S(A_w, w, D_w) \cap T'| \qquad (3)$$

Eq. (3) means to cover as many uncovered connections as possible by one $S(A_w, w, D_w)$. Eq. (3) implies that it does not vary A_w and D_w. Note: the the cost of $|A_w| + |D_w|$ is taken into consideration, although it does not explicitly appear in Eq.(3), because the node a and d are added into A_w and D_w, respectively, only if there exists an uncovered connection $(a, d) \in T'$. In [6], we showed that the size of the 2-hop cover computed with Eq. (3) is only slightly larger than that obtained with Eq. (2) but can be computed fast. The quality of a 2-hop cover L is weighted by a *compression rate*, R, which is defined to be the ratio of the number of covered connections to the total size of L, namely $R = \frac{|P_L|}{|L|}$. The higher compression rate, the better quality of a 2-hop cover.

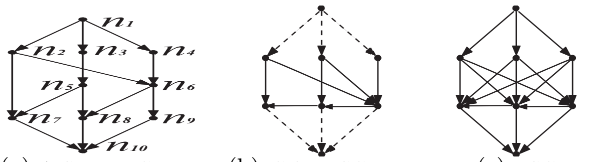
We proposed a fast algorithm in [6] to compute a 2-hop cover for an entire graph G without partitioning the graph. However, when a graph G is very large and dense, it becomes necessary to compute a 2-hop cover by considering the possibility of partitioning a graph into small graphs, which is the main focus of this paper.

3. EXISTING WORK

The necessity of partitioning a graph into small graphs was first pointed out by Schenkel et al. in [18]. Schenkel et al. proposed an approach in [18] to compute 2-hop cover in three steps. First, it partitions graph, G, into k subgraphs $G_1, G_2, \cdots, G_k$. Second, it computes the transitive closure and the 2-hop cover for each subgraph G_i, for $1 \leq i \leq k$, and stores the 2-hop cover on disk. Third, it merges the k 2-hop covers for the k subgraphs, which needs to take the edges that cross subgraphs into consideration. The merging yields a 2-hop cover for the entire graph G.

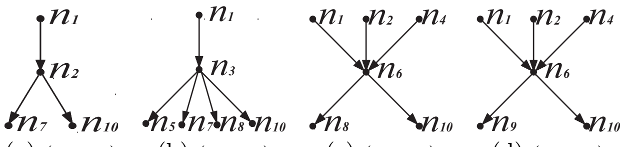
The first and the second step are straightforward. In [18], Cohen's algorithm [7] is used to compute the 2-hop cover for each subgraph G_i. The third step is called the cover joining phase, which combines all 2-hop covers obtained in the second step, and more importantly, adds new covers to ensure the correctness of the 2-hop cover for the entire graph, G. As reported in [18, 19], the third step becomes the bottleneck of the whole processing, and most of the running time is spent on joining the 2-hop covers from subgraphs, $G_1, G_2, \cdots, G_k$. We focus on the third step below.

Suppose the 2-hop covers of subgraphs, G_i, for $1 \leq i \leq k$, are computed. A cross-partition edge, (x, y), is an edge that does not exist in any single subgraph, and will introduce additional uncovered connections from the ancestors of x to



(a) A Sample Graph (b) G for PSG (c) PSG

Figure 1: Sample Graphs



(a) (n_2, n_7) (b) (n_3, n_5) (c) (n_6, n_8) (d) (n_6, n_9)

Figure 2: 2-Hop Clusters

the descendants of y. In [18], Schenkel et al. constructed a 2-hop cover, L, by encoding all such connections via (x, y) according to the following two operations:

- For all $a \in ancs(x)$, $L_{out}(a) \leftarrow L_{out}(a) \cup \{x\}$, and
- For all $d \in desc(y) \cup \{y\}$, $L_{in}(d) \leftarrow L_{in}(d) \cup \{x\}$.

It means that, for all cross-partition edges, (x, y), 2-hop clusters, $(ancs(x), x, desc(y))$, will be constructed mandatorily to augment L. This, however, determines A_x and D_x disregarding whether a covered connection (a, d) already exists or not for $a \in ancs(x)$ and $d \in desc(y)$. The above two operations consume a lot of time, because the sizes of $ancs(x)$ and $desc(y)$ can be large for a large graph.

Example 3.1: A sample graph is shown in Figure 1(a) as the running example. With the approach in [18], it partitions the graph into two subgraphs, G_1 and G_2. G_1 consists of nodes, $\{n_1, n_2, n_3, n_4, n_6\}$, and G_2 consists of nodes, $\{n_5, n_7, n_8, n_9, n_{10}\}$. The set of cross-partition edges is $\{(n_2, n_7)$ (n_3, n_5),(n_6, n_8),$(n_6, n_9)\}$. Figure 2 shows four 2-hop clusters corresponding to the 4 cross-partition edges, where we illustrate a 2-hop cluster with all its nodes arranged in three layers. The top layer or bottom layer consists of those nodes in A_w or D_w except for the center w, respectively, whereas the middle layer is for the center w. All connections covered by a 2-hop cluster are all the connections in the corresponding three-layer graph. The 2-hop cover for the graph includes Figure 2 (c) and Figure 2 (d), where both consist of many overlapped connections. It will result in additional computing cost and low compression rate. □

In [19], Schenkel et al. proposed another approach. For all cross-partition edges, (x, y), it refines this process within ancestors of x and the descendants of y on partitions instead of the entire graph. It first computes a 2-hop cover L' for a auxiliary graph, called the partition skeleton graph, or PSG for short. The nodes in PSG are all x and y of cross-partition edges (x, y), and the edges in PSG are either cross-partition edges, (x, y), or newly added edges (a, d), where d is a source node for some cross-partition edge and a is a source node for some cross-partition edge, if (1) a and d are in the same partition, and (2) a reaches d. L is augmented by the following two operations for each cross-partition edge (x, y):

- $L_{out}(a) \leftarrow L_{out}(a) \cup L'_{out}(x)$, and
- $L_{in}(d) \leftarrow L_{in}(d) \cup L'_{in}(y)$.

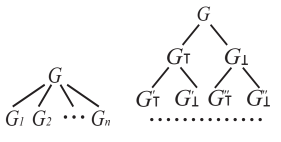
Here, a is in $ancs'(x)$, the set of ancestors of x in the same partition of x, and d is in $desc'(y)$, the set of descendants of y in the same partition of y. It means that a number of 2-hop clusters, $(ancs'(x), w, \{w\})$ and $(\{w\}, w, desc'(y))$, will be constructed mandatorily. Note: the approach in [19] will identify an identical 2-hop cover as that obtained by the approach in [18], when partitioning a graph G into two smaller graphs, G_1 and G_2. When it partitions a graph into k subgraphs, for $k > 2$, it may result in a PSG which is denser than the original graph G. Consider graph G in Figure 1(a). Suppose that it partitions G into three subgraphs: G_1, G_2, and G_3. Here, G_1 is the top node, G_3 is the bottom node, and G_2 is the 6 nodes in the middle plus the edges among the 6 nodes as shown in Figure 1(b). The approach in [19] generates a PSG graph by connecting all the cross-partition edges over all subgraphs, as a 'global' graph. It attempts to find the 2-hop cover by the cross-partitions edges globally. However, as shown in Figure 1(c), the PSG is with more additional edges, and becomes denser than the original graph. This fact implies that it may need even more efforts to compute the 2-hop cover for PSG.

4. 2-HOP SENSITIVE PARTITIONING

In this paper, we propose a new top-down hierarchical partitioning approach. The main difference between our top-down hierarchical partitioning and the existing approaches given [18, 19] is illustrated in Figure 3.

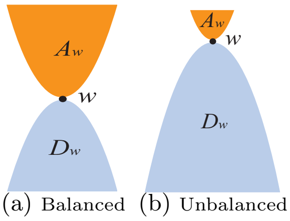
As discussed in the previous section, the approaches in [18, 19] partition a graph G into a set of subgraphs $G_1, G_2, \cdots$ (Figure 3 (a)). We call them a flat approach. The approach in [18] is to work on small subgraphs, because it consumes too much computation and I/O time to deal with a large graph directly. In other words, it considers efficiency as a more important issue than the compression rate. The approach in [18] can possibly compute a 2-hop cover for a subgraph with high compression rate. But, those 2-hop covers may not be the best for the entire graph. When merging these 2-hop covers to be correct for the entire graph, it may generate a final 2-hop cover with low compression rate. The approach in [19] considers the compression rate, and attempts to find a global graph, PSG, to handle the cross-partition edges first. The problem is that PSG can be even denser than the original graph.

On the other hand, our hierarchical approach repeatedly partitions a graph into two subgraphs in a top-down fashion (Figure 3 (b)). In every step, first, for a given graph G, we bisect G into two subgraphs, G_A and G_D. The bisection of G into G_A and G_D is outlined in Figure 6. In addition to the two subgraphs G_A and G_D, we consider an induced subgraph, $G_c(V_c, E_c)$, represented by the small dots in Figure 6, severing as the connection between two subgraphs G_A and G_D. Second, we compute a minimum 2-hop cover, L, for G_c, such that all connections (induced by cross-partition edges) between the two subgraphs, G_A and G_D, are covered by L. It is important to note that L becomes a portion of the 2-hop cover for the graph, G. Third, with the computed 2-hop cover L, we extract an induced subgraph of G_A, denoted $G_\top$, which does not include any centers w computed in L to cover G_c, and extract an induced subgraph of G_D, denoted $G_\perp$, which does not include any centers w computed in L to cover G_c. Therefore, we have three subgraphs G_c, $G_\top$, and $G_\perp$ that they do not have any nodes in common. Afterwords, there is no need to consider any connections being missed due to the partitioning, and only 2-hop covers of the two remaining subgraphs, $G_\top$ and $G_\perp$, need to be com-



(a) Flat (b) Hierarchical

Figure 3: Different Partitioning



(a) Balanced (b) Unbalanced

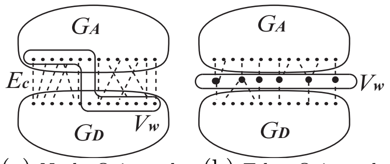
Figure 4: $S(A_w, w, D_w)$

```
Topdown(G, H)

Input:   a DAG, G, and H as a 2-hop cover repository.
Output: the 2-hop cover, H, of G.

 1: bisect $G$ into $G_A$ and $G_D$;
 2: let $G_c$ be the edges between $G_A$ and $G_D$;
 3: CrossCover($G_c$, $G$, $H$);
 4: $G_\top \leftarrow$ Extract($G_A$, $H$);
 5: $G_\perp \leftarrow$ Extract($G_D$, $H$);
 6: if $|V(G_\top)|$ is smaller than a pre-defined threshold $\mathcal{T}$
 7:     Cover($G_\top$, $H$);
 8: else
 9:     Topdown($G_\top$, $H$);
10: if $|V(G_\perp)|$ is smaller than $\mathcal{T}$
11:     Cover($G_\perp$, $H$);
12: else
13:     Topdown($G_\perp$, $H$);
```

Figure 5: The Top-Down Hierarchical Approach

puted. Let the 2-hop cover for the subgraphs, $G_\top$ and $G_\perp$, be $L_\top$ and $L_\perp$, respectively. There is $P_L \cup P_{L_\top} \cup P_{L_\perp} = T_G$. Thus we guarantee the correctness of resulting 2-hop covers covering all connections in the entire graph G. It is worth noting that we do not need to further merge 2-hop covers computed from subgraphs. In addition, because all 2-hop covers are computed to be minimized, the compression rate is high. As a remark, we consider both efficiency and impression rate as important issues.

We outline our approach, denoted *Topdown*(G, H) in Figure 5. *Topdown* takes a graph G as input. The input H (call-by-reference) is used to maintain the 2-hop covers computed in each step in *Topdown*, and is initialized as empty. In the following, we assume a graph G is DAG, because the 2-hop cover for an arbitrary directed graph can be obtained from the 2-hop cover of its corresponding DAG with a simple pre- and post-processing step as discussed in [6]. In *Topdown*(G, H), it first bisects the input graph, G, into two subgraphs, G_A and G_D (line 1). Let $G_c(V_c, E_c)$ be the induced graph for the connections between G_A and G_D (line 2), where the set of edges, E_c, contains all edges (a, d) if $a \in G_A$ and $d \in G_D$, and (a, d) does not appear in G_A and G_D, and the set of nodes, V_c, contains all nodes appearing in E_c. Then, it computes the 2-hop cover for G_c by calling *CrossCover*(G_c, G, H) (line 3). We will discuss it in detail later in this paper. When it returns, H will be updated by including the 2-hop cover for G_c. Next, it extracts $G_\top$ and $G_\perp$ from G_A and G_D, respectively. It is simply to remove nodes from G_A and G_D and the edges connecting to them, respectively, if they appear in the 2-hop cover computed for G_c. In the remaining (line 6-13), it computes a 2-hop cover for $G_\top$ and $G_\perp$, respectively, by calling *Cover*(), if its size is small to be stored in main memory. *Cover*() can be implemented using the fast algorithm given in [6]. Otherwise, it will recursively calling *Topdown* in a top-down fashion. When it completes, H maintains the 2-hop cover for the original graph.



(a) Node-Oriented (b) Edge-Oriented

Figure 6: Bisect G into G_A and G_D

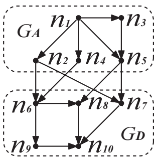


Figure 7: G_A/G_D

Below, we will discuss bisection in Section 4.1, and discuss *CrossCover*() in Section 5 and Section 6.

4.1 Bisection

In this section, we discuss how to bisect a graph G into G_A and G_D, and how to obtain a graph G_c. An important issue here is how to find the best way to bisect a graph G when the compression rate is concerned. The issue is different from the existing graph partitioning approaches in the literature, because the main concern is the compression rate of computing a 2-hop cover.

We propose a heuristics to bisect a graph G based on compression rate. We define the compression rate of a single 2-hop cluster as the ratio of the number of connections it can cover. It fits our intuition that a set of 2-hop clusters with high compression rate can also result in a high compression rate for the 2-hop cover. Let $x = |A_w|$ and $y = |D_w|$. The compression rate of the 2-hop cluster is calculated as: $r(x, y) = \frac{xy-1}{x+y}$, where we have $x \geq 1$ and $y \geq 1$. Given a 2-hop cluster $S(A_w, w, D_w)$ to cover n connections, consider the maximum and the minimum of its compression rate. Since $xy - 1 = n$, there is $r(x, y) = \frac{xn}{x^2+n+1}$ or $\frac{yn}{y^2+n+1}$ We examine the derivative of r:

$$\frac{dr}{dx} = \frac{n^2 + n - nx^2}{(x^2 + n + 1)^2} \text{ or } \frac{n^2 + n - ny^2}{(y^2 + n + 1)^2}$$

When $\frac{dr}{dx} = 0$, r becomes the maximum where we have $x^2 = y^2 = n + 1$. In other words, for a 2-hop cluster $S(A_w, w, D_w)$ to cover n connections, its compression rate is maximized when $|A_w|$ equals to $|D_w|$. On the other hand, when $x^2 < n + 1$, r increases in terms of x, whereas when $x^2 > n + 1$, r decreases in terms of x. It also applies to y. When $x = 1$ and $y = n + 1$ or $y = 1$ and $x = n + 1$, r will be minimized, and when x and y become closer, r becomes larger. Therefore, we shall select a center w where A_w and D_w of $S(A_w, w, D_w)$ tend to have similar size, and avoid to select a center w where the sizes of A_w and D_w of $S(A_w, w, D_w)$ differ significantly. Figure 4 (a) depicts the desired balanced 2-hop cluster, whereas Figure 4 (b) depicts the unbalanced. Hence, we bisect a graph to cut the graph in the *middle*.

The procedure of bisection of G is given below. We sort all nodes in G using topological sort [8]. Note: all the directed edges (ordered pairs) are toward one direction, because G is a DAG. The bisection is to cut G in the middle. One half of nodes go to G_A, and the other half go to G_D, and G_A and G_D are two induced subgraphs of G, respectively. Next, we can obtain an induced subgraph, $G_c(V_c, E_c)$, with a set of cross-partition edges $E_c = \{(a, d) | a \in G_A, d \in G_D\}$.

Example 4.1: Consider Example 3.1. We bisect G (Figure 1(a)) into G_A and G_D as shown in Figure 7, where $V(G_A) = \{n_1, n_2, n_3, n_4, n_5\}$, and $V(G_D) = \{n_6, n_7, n_8, n_9, n_{10}\}$. The graph G_c contains the following edges: (n_2, n_6), (n_2, n_7), (n_4, n_6), (n_5, n_7), and (n_5, n_8). □

5. CROSSCOVER: A NODE-ORIENTED APPROACH

The *CrossCover*(G_c, G, H) is to find a set of centers, w, (appearing in the 2-hop cover) by which the two subgraphs, $G_\top$ $(\subseteq G_A)$ and $G_\perp$ $(\subseteq G_D)$ can be completely disconnected. In addition, any connections from $G_\top$ to $G_\perp$ must go via some centers w. We propose two approaches in this paper to compute the 2-hop cover for graph G_c. One is a node-oriented approach, which we will discuss in this section, and the other is an edge-oriented approach, which we will discuss in the next section.

We explain our node-oriented approach using an example. Consider Figure 6 (a). The graph $G_c(V_c, E_c)$ is illustrated as the small dots and the edges among them. We want to compute a 2-hop cover for G_c that can also disconnect two subgraphs, G_A and G_D in Figure 6 (a). Regarding the function of disconnecting two subgraphs, for any edge $(a, d) \in E_c$, we only need to consider one of the two nodes, a and d, to be selected as a center to cut.

Therefore, we first select a subset of nodes, V_w of $V(G_c)$ as a node-separator by which G are disconnected. To obtain such a node-separator V_w from E_c is to find a set of nodes V_w such that every edge in E_c is incident with some node in V_w. We use a simple algorithm as follows to do so. Let $V_w = \emptyset$ initially. We iteratively add to V_w a node v with the largest degree in E_c, that is the node with the 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. The resulting V_w is a node-separator to disconnect G into two subgraphs.

Let $V_w = \{w_1, w_2, \cdots, w_k\}$. The 2-hop cover, L, can be computed under a restriction that all centers must be selected in V_w using Eq. (3), which can be done with a simple modification of algorithm in [6]. Note: the algorithm will select $S(A_w, w, D_w)$ with the maximized value of Eq. (3). It will results in a set of 2-hop clusters, $S(A_{w_1}, w_1, D_{w_1})$, $S(A_{w_2}, w_2, D_{w_2})$, $\cdots$, $S(A_{w_k}, w_k, D_{w_k})$. $G_\top$ and $G_\perp$ are two induced subgraphs that contain sets of nodes, $V(G_\top) = V(G_A) \setminus V_w$ and $V(G_\perp) = V(G_D) \setminus V_w$, respectively.

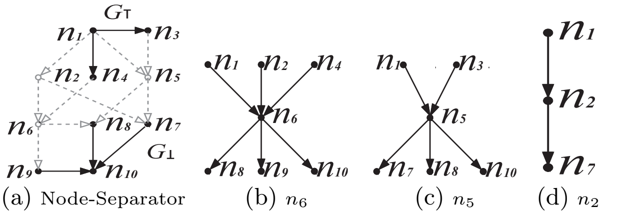
Theorem 5.1: *All connections (a, d) (for $a \leadsto d$), where $a \in G_\top$ and $d \in G_\perp$, are covered by L.* □

Proof Sketch: Suppose there exists an uncovered connection (a, d) by L, where $a \in G_\top$ and $d \in G_\perp$. Because removing all k centers, w_1, w_2, $\cdots$, w_k, will disconnect $G_\top$ and $G_\perp$ in G, there must be a center w_i, where $1 \leq i \leq k$, on the path from a to d. In other words, there must exist two connections (a, w_i) and (w_i, d) in G. Therefore, if there is connection, (a, d), it must be covered by L. □

Example 5.1: To continue Example 4.1. Figure 7 shows G with G_A and G_D and G_c. The node-separator $V_w = \{n_2, n_5, n_6\}$ is shown in Figure 8(a). There are three 2-hop clusters shown in Figure 8 based on the node-separator. Then, we obtain two induced subgraphs, $G_\top$ and $G_\perp$ where $V(G_\top) = \{n_1, n_3, n_4\}$ and $V(G_\perp) = \{n_7, n_8, n_9, n_{10}\}$. Note that all connections between $G_\top$ and $G_\perp$ are covered by the three 2-hop clusters shown in Figure 8. □

R-Tree Based Implementation

With $V_w = \{w_1, w_2, \cdots, w_k\}$, we discuss how to compute the 2-hop cover, L, that is $S(A_{w_1}, w_1, D_{w_1})$, $S(A_{w_2}, w_2, D_{w_2})$, $\cdots$, $S(A_{w_k}, w_k, D_{w_k})$. This can be done using our R-tree based approach given in [6]. We review main techniques used in [6], which compute the 2-hop cover for a DAG. (Recall: the 2-hop cover for an arbitrary directed graph can be obtained from the 2-hop cover of its corresponding DAG with a simple pre- and post-processing step as discussed in [6].)



(a) Node-Separator (b) n_6 (c) n_5 (d) n_2

Figure 8: 2-hop clusters Based on a Node-Separator

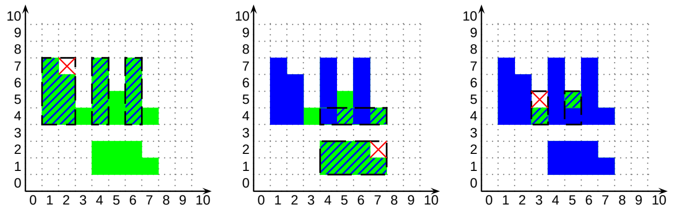
It uses a 2-dimensional *reachability map* that maps a connection (a, d) onto a grid point in a 2-dimensional grid and any 2-hop cluster $S(A_w, w, D_w)$ into a number of rectangles in the 2-dimensional space. The reachability map operates on a *reachability table* which is developed from a technique for transitive closure management in [1].

For the given DAG G, the transitive closure management technique in [1] assigns an interval label to each node v in G. The interval label on each node v comprises a set of non-overlapping intervals $\{[s_1, e_1], [s_2, e_2], \cdots, [s_n, e_n]\}$, denoted by $I_v^\downarrow$, and a unique postorder number, $po_v^\downarrow$. We omit the detail of how to generate such interval labeling for G due to the limit of space. Then, $u \leadsto v$ is true iff $po_v^\downarrow$ is contained in some interval $[s, e]$ in $I_u^\downarrow$, that is, $s \leq po_v^\downarrow \leq e$. This is due to the fact that each number in $I_u^\downarrow$ corresponds to a postorder number of a node v such that v is in *desc*(u) in G, and those postorder numbers of all nodes in *desc*(u) are contained in $I_u^\downarrow$. For reachability table, we construct the interval labeling for G as well as the interval labeling for another auxiliary graph $G^\uparrow$. In brief, $G^\uparrow$ is obtained from G by reversing the direction of each edge in G. We denote the set of intervals in $G^\uparrow$ assigned to v as $I_v^\uparrow$, and its unique postorder number as $po_v^\uparrow$. In term of G, each number in $I_v^\uparrow$ corresponds to a postorder number of a node u such that u is in *ancs*(v), and those postorder numbers of all nodes in *ancs*(v) are contained in $I_v^\uparrow$. For the sample graph in Figure 1(a), the reachability table is in Table 1.

w	G		$G^\uparrow$	
	$po_w^\downarrow$	$I_w^\downarrow$	$po_w^\uparrow$	$I_w^\uparrow$
n_1	10	[1,10]	4	[4,4]
n_2	3	[1,6]	5	[4,5]
n_3	8	[4,8]	1	[1,1][4,4]
n_4	9	[1,2][4,4][6,6][9,9]	6	[4,4],[6,6]
n_5	7	[4,7]	2	[1,2][4,4]
n_6	2	[1,2][4,4][6,6]	7	[4,7]
n_7	5	[4,5]	3	[1,5]
n_8	6	[4,4][6,6]	8	[1,2][4,8]
n_9	1	[1,1][4,4]	9	[4,7][9,9]
n_{10}	4	[4,4]	10	[1,10]

Table 1: The Sample Reachability Table

With the reachability map, for all nodes v, we consider $po_v^\downarrow$ to be a number being mapped on x-axis and $po_v^\uparrow$ to be a number being mapped on y-axis. Any grid point (x, y) in the two dimensional space will uniquely represent a node pair (u, v) s.t. there are $po_v^\downarrow = x$ and $po_u^\uparrow = y$. Therefore, any 2-hop cluster $S(A_w, w, D_w)$ will cover a set of connections that are represented as a number of rectangles in the two dimensional space, for D_w and A_w to be represented as intervals on x-axis and y-axis, respectively. A covered area



(a) $w = n_6$ (b) $w = n_5$ (c) $w = n_2$

Figure 9: The Reachability Map

in the two dimensional space is maintained and represent the set of connections covered. The 2-hop cover program identifies the best 2-hop cluster in each iteration as the correspondent rectangles with the largest uncovered area, based on Eq. (3). R-tree is used in this process.

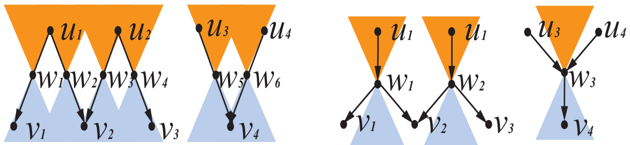
Let's consider $V_w = \{w_1, w_2, \cdots, w_k\}$. We search for the best $S(A_{w_i}, w_i, D_{w_i})$, which corresponds to some rectangles with the largest total area among all 2-hop clusters that can be obtained from V_w. We add those rectangles into the R-tree, and remove w_i from V_w. Then, we search for the next best $S(A_{w_j}, w_j, D_{w_j})$, whose rectangles have no overlap with those rectangles maintained in the R-tree. The R-tree is updated by adding those rectangles and V_w is updated by removing w_j from itself. We repeat the above operation till V_w becomes empty. Those identified 2-hop clusters are included into L.

Example 5.2: Reconsider Example 5.1. Here, the node-separator is $V_w = \{n_2, n_5, n_6\}$. We search for 2-hop clusters using V_w. Figure 9 shows the steps. In each figure, the dark area represents the set of connections covered; the striped area represents connections that can be newly covered by the identified 2-hop cluster; the shadowed area represents the set of connections remained to be covered. The crossed points are self-loop covered by some 2-hop cluster. Initially, rectangles of $S(A_{n_6}, n_6, D_{n_6})$ is identified to have the largest area in the two-dimensional space, as shown in Figure **??**. $S(A_{n_6}, n_6, D_{n_6})$ is shown in Figure 8(b). Next, rectangles of $S(A_{n_5}, n_5, D_{n_5})$ is found to be the second large area, with dark area not counted (Figure 8(c)). Finally, $S(A_{n_2}, n_2, D_{n_2})$ is found. □

A potential problem still exists for the R-tree based implementation for *CrossCover*(G_c, G, H). When G is large and dense, V_w becomes large. In order to construct 2-hop clusters using nodes in V_w, we need to maintain a great number of rectangles in the R-tree to represent the covered connections. It is required that no two rectangles can be overlapped [6] to correctly compute 2-hop clusters based on Eq. (3). And it is difficult to minimize the number of rectangles in the R-tree [13]. The R-tree needs to be used to operate on a great number of rectangles. This makes it difficult to use the R-tree based approach to support very large graphs efficiently.

6. CROSSCOVER: AN EDGE-ORIENTED APPROACH

In the previous section, we discuss a node-oriented approach that uses a node-separator to disconnect two subgraphs. The node-oriented approach first identifies a set of centers V_w, which is a subset of $V(G_c)$. In other words, the set of centers, V_w, is fixed. It then concentrates on selecting the best $S(A_{w_i}, w_i, D_{w_i})$, for $w_i \in V_w$, in each iteration in the 2-hop cover program. It needs to take the entire graph G_c into consideration, when constructing the best $S(A_{w_i}, w_i, D_{w_i})$.



(a) Edges (b) Shared Edges

Figure 10: Edge-Separator

In this section, we discuss an edge-oriented approach. We explain it below. Recall the induced graph G_c which serves as connections between G_A and G_D, and we now consider it to be a bipartite graph $B(U \cup V, E_c)$, where $U \in V(G_c)$ consists of all nodes from G_A and $V \in V(G_c)$ consists of all nodes from G_D, and E_c consists of the set of edges (u, v) s.t. $u \in U$ and $v \in V$. Figure 10 (a) illustrates a bipartite graph $B(U \cup V, E_c)$ where $U = \{u_1, u_2, u_3, u_4\}$ and $V = \{v_1, v_2, v_3, v_4\}$. E_c contains 6 edges. We conceptually consider an edge, for example (u_1, v_1), as a node by assigning an additional virtual node w_1 on the edge, and replace (u_1, v_1) by (u_1, w_1) and (w_1, v_1). Here, the additional virtual node, say w_1, represents an edge (u_1, v_1). In Figure 10 (a), there are possibly 6 additional virtual nodes, w_i, for $1 \leq i \leq 6$. When computing 2-hop cover L based on G_c, we concentrate on these additional virtual nodes, w_i. The two main advantages are discussed below. (1) We do not need to fix V_w as a set of centers first and then compute 2-hop cover as the node-oriented approach does. We can possibly compute 2-hop cover with high compression rate, because we select centers and compute 2-hop cover at the same time. (2) We can compute fast, because those additional virtual nodes do not belong to other subgraphs, G_A and G_D. As we discuss later, we do not need to use R-tree and can manipulate the interval labeling.

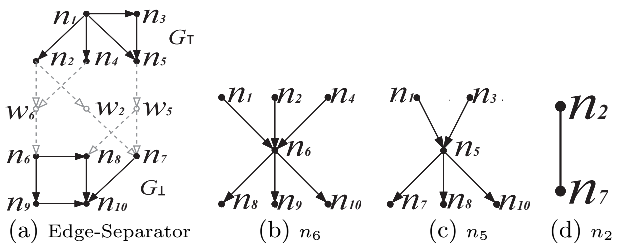
A question is whether we need to add additional nodes that do not appear in the original graph, G, but are needed as a part of the 2-hop cover computed. The answer is: there is no need to add any virtual nodes into the 2-hop cover computed. In our algorithm, we dynamically cluster nodes in U (V) and make them connect to a single node in the other U (V). Consider Figure 10 (b), where v_1 and v_2 are clustered together to connect to u_1 via w_1. The node w_1 represents the edge, but it has a node u_1 on the upper side. The function of w_1 can be replaced by u_1.

Example 6.1: Reconsider Example 4.1. Its G_A, G_D and B can be found in Figure 7. The edge-separator $\{w_6, w_2, w_5\}$ is computed (Figure 11(a)). There are three 2-hop clusters shown in Figure 11 based on the edge-separator, which is slightly better than the result computed by the node-oriented approach (Figure 8). As can be seen in Figure 11, there are no additional nodes. n_6, n_2, and n_5 can serve the same role for w_6, w_2, and w_5, respectively. □

6.1 Two Properties

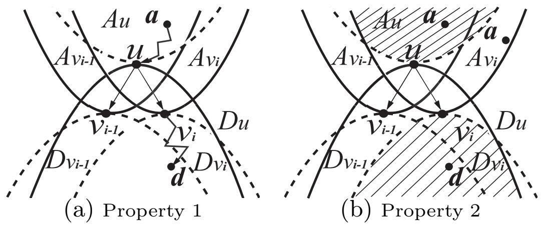
Given the edge-separator, we compute a 2-hop cover L for all cross-partition connections (u, v) where $u \in G_A$ and $v \in G_D$. Consider a bipartite graph $B(U \cup V, E_c)$. Let all neighbors of $u \in U$ in B be $\mathsf{neighbor}(u) = \{v_1, v_2, \cdots, v_l\} \subseteq V$.

We discuss two properties on the set of cross-partition connections covered by $S(A_{v_1}, v_1, D_{v_1})$, $S(A_{v_2}, v_2, D_{v_2})$, $\cdots$,



(a) Edge-Separator (b) n_6 (c) n_5 (d) n_2

Figure 11: 2-hop clusters Based on an Edge-Separator



(a) Property 1 (b) Property 2

Figure 12: 2-hop clusters on B

$S(A_{v_l}, v_l, D_{v_l})$ and $S(A_u, u, D_u)$, where neighbor$(u) = \{v_1, v_2, \cdots, v_l\}$. The properties enable us to dynamically maintain just one node set D_u, for $u \in U$, and one node set A_v, for $v \in V$, in order to correctly compute $S(A_w, w, D_w)$, where $w \in U \cup V$. And only A_{v_1}, A_{v_2}, $\cdots$, A_{v_l} need to be updated when $S(A_u, u, D_u)$ is picked out. As will be seen latter, it can be done efficiently based on a set of intervals assigned to each node in B.

Property 6.1: *Given $S(A_{v_1}, v_1, D_{v_1})$, $S(A_{v_2}, v_2, D_{v_2})$, $\cdots$, $S(A_{v_l}, v_l, D_{v_l})$ and $S(A_u, u, D_u)$ to cover connections (a, d) where $a \in V(G_A)$ and $d \in V(G_D)$ and* neighbor$(u) = \{v_1, v_2, \cdots, v_l\}$*, then $A_u = (A_{v_1} \cup A_{v_2} \cup \cdots \cup A_{v_l}) \cap ancs(u)$.* □

Proof Sketch: For any node $a \in A_u$, there must be an uncovered cross-partition connection (a, d), where $d \in D_u$. As depicted in Figure 12(a), a is in A_u, the area circled by dotted line on u; and d is in D_u, the area circled by solid line on u. Because all the neighbors of u on the partition G_D are $v_1, v_2, \cdots, v_l$, there must exist some v_i, where $1 \leq i \leq l$, and $d \in desc(v_i)$. Since there is an edge (u, v_i), there should be $a \in ancs(v_i)$. Also, because (a, d) is not covered, it should be covered by $S(A_{v_i}, v_i, D_{v_i})$. Therefore, we have $a \in A_{v_i}$ and $d \in D_{v_i}$. As depicted in Figure 12(a), a should be also in A_{v_i}, the area circled by solid line on v_i; and d is in D_{v_i}, the area circled by dotted line on v_i. Thus, we have $A_u \subseteq A_{v_i} \cap ancs(u)$ for $1 \leq i \leq l$, that is, $A_u \subseteq (A_{v_1} \cup A_{v_2} \cup \cdots \cup A_{v_l}) \cap ancs(u)$. On the other hand, for any node $a \in (A_{v_1} \cup A_{v_2} \cup \cdots \cup A_{v_l}) \cap ancs(u)$, it means there must exist v_i, where $1 \leq i \leq l$, and $a \in A_{v_i}$. And, there must be an uncovered cross-partition connection (a, d), where $d \in D_{v_i}$. Note: there is an edge (u, v_i), there should be $d \in desc(u)$. In addition, we have $a \in ancs(u)$. Because (a, d) is not covered, so it should be covered by $S(A_u, u, D_u)$. Therefore, we have $a \in A_u$ and $d \in D_u$. This case is also indicated by Figure 12(a). So we also have $(A_{v_1} \cup A_{v_2} \cup \cdots \cup A_{v_l}) \cap ancs(u) \subseteq A_u$. □

Similarly, let $u_1, u_2, \cdots, u_l \in U$ be all neighbors of $v \in V$ in B, such that neighbor$(v) = \{u_1, u_2, \cdots, u_l\}$. Property 6.1 can be also applied and we have $D_v = (D_{u_1} \cup D_{u_2} \cup \cdots \cup D_{u_l}) \cap desc(u)$. By maintaining D_u or A_v where $u \in U$ or $v \in V$, Property 6.1 allows us to compute $S(A_u, u, D_u)$ or $S(A_v, v, D_v)$ on the fly based on either A_{v_i} or D_{u_i} of its neighbors in B, respectively.

We discuss another property that guarantees the correctness of maintaining all D_u or A_v dynamically. Note: We need to update B while the 2-hop cover program proceeds. Each time when $S(A_u, u, D_u)$ is picked out, we remove u together with all edges being incident to u from B. When the time comes to pick $S(A_u, u, D_u)$, let $v_1, v_2, \cdots, v_l \in V$ be all neighbors of $u \in U$ in B before the update and $S(A_{v_1}, v_1, D_{v_1}), S(A_{v_2}, v_2, D_{v_2}), \cdots, S(A_{v_l}, v_l, D_{v_l})$ be those 2-hop clusters before we pick out $S(A_u, u, D_u)$. By then, there are connections of $S(A_u, u, D_u)$ not covered .

Property 6.2: *After all connections of $S(A_u, u, D_u)$ are covered, new 2-hop clusters $S(A'_{v_1}, v_1, D'_{v_1})$, $S(A'_{v_2}, v_2, D'_{v_2})$, $\cdots$, $S(A'_{v_l}, v_l, D'_{v_l})$ can be obtained by computing A'_{v_i} based on $A'_{v_i} = A_{v_i} \setminus A_u$ and D'_{v_i} with Property 6.1 on the updated B, where $1 \leq i \leq l$.* □

Proof Sketch: Note that for all new 2-hop clusters obtained above, there is $S(A'_{v_i}, v_i, D'_{v_i}) \subset S(A_{v_i}, v_i, D_{v_i})$. We only need to prove (1) all connections of $S(A'_{v_i}, v_i, D'_{v_i})$ are not covered by $S(A_u, u, D_u)$, and (2) all connections covered by $S(A_{v_i}, v_i, D_{v_i})$ but not by $S(A'_{v_i}, v_i, D'_{v_i})$ are covered by $S(A_u, u, D_u)$.

Since for any connection (a, d) covered by $S(A'_{v_i}, v_i, D'_{v_i})$, we have $a \notin A_u$ due to $a \in A_{v_i} \setminus A_u$, hence (a, d) can not be covered by $S(A_u, u, D_u)$. So (1) holds. This is illustrated by Figure 12(b) when a is outside the shaded area in A_{v_i}. Assume (a, d) is a connection covered by $S(A_{v_i}, v_i, D_{v_i})$ but not by $S(A'_{v_i}, v_i, D'_{v_i})$. Then, for a, there should be $a \in A_{v_i} \cap A_u$, hence we have $a \in A_u$. a for this case is depicted as in the shadowed area in Figure 12(b). For d, since we have $d \in desc(v_i)$ and there exists an edge (u, v_i), hence $d \in D_u$. Therefore, $S(A_u, u, D_u)$ covers all such connections (a, d). Those (a, d) are seen as those contained in the shadow area of Figure 12(b). Then, (2) holds. □

Similarly, if $S(A_v, v, D_v)$ is to be picked out, let u_1, u_2, $\cdots$, $u_l \in U$ be all neighbors of $v \in V$ in B before the update. After we newly cover uncovered connections with $S(A_v, v, D_v)$, Property 6.2 can be also applied and new 2-hop clusters $S(A'_{u_1}, u_1, D'_{u_1})$, $S(A'_{u_2}, u_2, D'_{u_2})$, $\cdots$, $S(A'_{u_l}, u_l, D'_{u_l})$ can be obtained by computing D'_{u_i} based on $D'_{u_i} = D_{u_i} \setminus D_v$ and A'_{u_i} with Property 6.1 on updated B, where $1 \leq i \leq l$.

In our algorithm, initially, let $D_u = desc(u) \cap \mathcal{D}$, for $u \in U$, and $A_v = ancs(u) \cap \mathcal{A}$, for $v \in V$, where $\mathcal{A}$ and $\mathcal{D}$ are the set of nodes in G_A and G_D respectively. Then, by the two properties, we can compute every $S(A_w, w, D_w)$, where $w \in U \cup V$, with D_u, for $u \in U$, and A_v, for $v \in V$, as well as correctly maintain those A_v and D_u during a 2-hop cover program to compute L to cover all (a, d) where $a \in G_A$ and $d \in G_D$. To avoid costly set operations, we process it using interval labelings.

6.2 Interval Based Computing

We give some definitions and notations. Assume I is a set of non-overlapping intervals and it can be $I_v^{\downarrow}$ or $I_v^{\uparrow}$ for a node v as obtained in Section 5. Let $|I|$ to be the size of $I = \{[s_1, e_1], [s_2, e_2], \cdots, [s_n, e_n]\}$ and $|I| = \sum_{i=1}^{n}(e_i - s_i + 1)$. Given a set of nodes $A \in V$, we use $I^{\uparrow}(A)$ and $I^{\downarrow}(A)$ to denote a set of non-overlapping intervals which includes those $po_v^{\uparrow}$ and $po_v^{\downarrow}$ as described in Section 5 for each $v \in A$, respectively. We also define the *set difference* of two sets of intervals I_1 and I_2, denoted by $I_1 \ominus I_2$, which is the set of non-overlapping intervals including all numbers included in I_1 but I_2. We further define the *union* of two sets of intervals I_1 and I_2, denoted by $I_1 \sqcup I_2$, to be the set of non-overlapping intervals including those numbers either included in I_1 or in I_2. The *overlap* of two sets of intervals I_1 and I_2, denoted by $I_1 \sqcap I_2$, is defined to be the set of non-overlapping intervals including those numbers both included in I_1 and I_2 in the same time. We implemente those interval sets as sorted lists in the order of start value s_i for each interval $[s_i, e_i]$. The last three operations can be done efficiently in a similar manner to the sort-merge join.

Compute $S(A_w, w, D_w)$ Based on Intervals: We maintain a set of interval set for each node in B, denoted as $\hat{I}_u^{\downarrow}$ for $u \in U$, and $\hat{I}_v^{\uparrow}$ for $v \in V$, respectively. Specifically, $\hat{I}_u^{\downarrow}$ includes all $po_d^{\downarrow}$ for $d \in D_u$ and $\hat{I}_v^{\uparrow}$ includes all $po_a^{\uparrow}$ for $a \in A_v$. Based on Property 6.1 and Property 6.2, we can compute every $S(A_w, w, D_w)$ for $w \in U \cup V$ based on $\hat{I}_u^{\downarrow}$ and $\hat{I}_v^{\uparrow}$ with the reachability table described in Section 5, and we can correctly maintain those $\hat{I}_u^{\downarrow}$ and $\hat{I}_v^{\uparrow}$ during a 2-hop cover program to compute L to cover all (a, d) where $a \in G_A$ and $d \in G_D$.

Let all neighbors of $u \in U$ in B be $v_1, v_2, \cdots, v_l \in V$. Property 6.1 can be easily extended to the set of intervals $\hat{I}_u^{\uparrow}$ to include all $po_a^{\uparrow}$ such that $a \in A_u$. It can be obtained by

$$\hat{I}_u^{\uparrow} = \hat{I}_{v_1}^{\uparrow} \sqcup \hat{I}_{v_2}^{\uparrow} \sqcup \cdots \sqcup \hat{I}_{v_l}^{\uparrow} \sqcap I_u^{\uparrow} \tag{4}$$

In addition, Property 6.2 can also be extended to work with intervals. When $S(A_u, u, D_u)$ is newly picked out to cover connections, we only need to update $\hat{I}_{v_i}^{\uparrow}$ by

$$\hat{I}_{v_i}^{\uparrow} = \hat{I}_{v_i}^{\uparrow} \ominus \hat{I}_u^{\uparrow} \tag{5}$$

Let $u_1, u_2, \cdots, u_l \in U$ be all neighbors of $v \in V$ in B. Similarly, for Property 6.1 we have

$$\hat{I}_v^{\downarrow} = \hat{I}_{u_1}^{\downarrow} \sqcup \hat{I}_{u_2}^{\downarrow} \sqcup \cdots \sqcup \hat{I}_{u_l}^{\downarrow} \sqcap I_v^{\downarrow} \tag{6}$$

And for Property 6.2 we have

$$\hat{I}_{u_i}^{\downarrow} = \hat{I}_{u_i}^{\downarrow} \ominus \hat{I}_v^{\downarrow} \tag{7}$$

With the computed $\hat{I}_u^{\uparrow}$ and $\hat{I}_u^{\downarrow}$, A_u and D_u can be obtained by mapping all $po_d^{\downarrow} \in \hat{I}_u^{\uparrow}$ and $po_a^{\uparrow} \in \hat{I}_u^{\downarrow}$ to the corresponding a and d to be added into A_u and D_u, respectively, using the reachability table. Initially, since there is no connection covered yet, we let $\hat{I}_u^{\downarrow} = I_u^{\downarrow} \sqcap I^{\downarrow}(V(G_D))$ for each $u \in U$ and $\hat{I}_v^{\uparrow} = I_v^{\uparrow} \sqcap I^{\uparrow}(V(G_A))$ for each $v \in V$.

Calculate Eq. (3) Based on Intervals: The value of Eq. (3) for $S(A_w, w, D_w)$ is the number of uncovered connections that can be covered by it. Note that the total number of connections that $S(A_w, w, D_w)$ can cover is simplified as $|\hat{I}_w^{\downarrow}| \times |\hat{I}_w^{\uparrow}|$. Among them, we still need to know how many connections are already covered, that is, the *overlapped connections* of $S(A_w, w, D_w)$. This can be obtained with the reachability map and a covered area as rectangles maintained in the R-tree [6] or with set operations and a set of covered connections so far [7, 18, 19]. However, for graphs with millions of nodes, those operations are all too expensive. We use a simple method based on interval to calculate this value approximately. Though this is not the exact value of Eq. (3), it provides a proper estimate under most circumstances.

```
Algorithm MaxCardinality-I

Input:   a DAG, $G$, the bipartite graph $B(U, V, E_c)$ built
         from an edge-separator $E_c$ of $G$.
Output:  the cover $L$ that covers all cross-partition connections
         between $G_A$ and $G_D$.

 1: compute the reachability table of $G$;
 2: assign a set of intervals to each $u \in U$ as $\hat{I}_u^{\downarrow} = I_u^{\downarrow} \cap I^{\downarrow}(\mathcal{D})$
    and to each $v \in V$ as $\hat{I}_v^{\uparrow} = I_v^{\uparrow} \cap I^{\uparrow}(\mathcal{A})$;
 3: $L \leftarrow \emptyset$; $I_{\mathcal{D}'}^{\downarrow} \leftarrow \emptyset$; $I_{\mathcal{A}'}^{\uparrow} \leftarrow \emptyset$;
 4: while $U \cup V \neq \emptyset$ do
 5:    let $w \in U \cup V$ be the node with the max value of
       Eq. (8); {we compute $|\hat{I}_w^{\uparrow}|$ for $w \in U$ based on
       Eq. (4) or $|\hat{I}_w^{\downarrow}|$ for $w \in V$ based on Eq.( 6).}
 6:    for all $a$ s.t. $po_a^{\uparrow}$ is included in $\hat{I}_w^{\uparrow}$ do
 7:       update $L$ by $L_{out}(a) \leftarrow L_{out}(a) \cup \{w\}$;
 8:    for all $d$ s.t. $po_d^{\downarrow}$ is included in $\hat{I}_w^{\downarrow}$ do
 9:       update $L$ by $L_{in}(d) \leftarrow L_{in}(d) \cup \{w\}$;
10:    $I_{\mathcal{D}'}^{\downarrow} \leftarrow I_{\mathcal{D}'}^{\downarrow} \sqcup \hat{I}_w^{\downarrow}$; $I_{\mathcal{A}'}^{\uparrow} \leftarrow I_{\mathcal{A}'}^{\uparrow} \sqcup \hat{I}_w^{\uparrow}$;
11:    let $x_1, x_2, \cdots, x_l$ be the nodes adjacent to $w$ in $B$;
       update the interval sets assigned to $x_1, x_2, \cdots, x_l$
       by Eq. (5) or Eq. (7);
12:    update $B$ by removing from $B$ the node $w$ together
       with all edges being incident to $w$ and any node
       with its degree to be zero in $B$;
13: end while
14: return $L$;
```

Figure 13: An Interval Based Algorithm

Assume $\mathcal{A}'$ and $\mathcal{D}'$ to be respectively the union of all A_v and D_v of previous 2-hop clusters that are decided in a 2-hop cover program. Among all connections covered by $S(A_w, w, D_w)$, we estimate the overlapped connections to be all (a, d) s.t. $a \in \mathcal{A}' \cap A_w$ and $d \in \mathcal{D}' \cap D_w$. And it is likely that those connections are covered by previous 2-hop clusters. Based on intervals, the number of overlapped connections is $|\hat{I}_u^{\downarrow} \sqcap I^{\downarrow}(\mathcal{D}')| \times |\hat{I}_u^{\uparrow} \sqcap I^{\uparrow}(\mathcal{A}')|$. We maintain two set of intervals $I_{\mathcal{D}'}^{\downarrow}$ and $I_{\mathcal{A}'}^{\uparrow}$ which are the union of all $I_v^{\downarrow}$ or $I_v^{\uparrow}$ of identified 2-hop clusters in the 2-hop cover program, respectively. It is clear to see that $I^{\downarrow}(\mathcal{D}')$ and $I^{\uparrow}(\mathcal{A}')$ are identical to $I_{\mathcal{D}'}^{\downarrow}$ and $I_{\mathcal{A}'}^{\uparrow}$, respectively. Therefore, Eq. (3) can be calculated by the following equation.

$$|\hat{I}_w^{\downarrow}| \times |\hat{I}_w^{\uparrow}| - |\hat{I}_w^{\downarrow} \sqcap I_{\mathcal{D}'}^{\downarrow}| \times |\hat{I}_w^{\uparrow} \sqcap I_{\mathcal{A}'}^{\uparrow}| \tag{8}$$

6.3 Algorithm

We give our algorithm to perform the detachment step based on intervals in Figure 13, called *MaxCardinality-I*, because it chooses 2-hop clusters based on the number of covered connections according to Eq. (3). The efficiency of *MaxCardinality-I* is improved over the R-tree approach. We don't need to use and maintain the R-tree and we use the operations over intervals, such as Eq. (4), Eq. (5), Eq. (6) and Eq. (7). We do not have to perform set operations to compute 2-hop clusters as in [7, 18, 19].

MaxCardinality-I takes graph G and a bipartite graph $B(U, V, E_c)$, which is graph G_c as described in Section 4 as input. In line 1, it computes the reachability table of G in order to obtain $I^{\uparrow}$ and $I^{\downarrow}$ for each node in B. Then we set the initial value of $\hat{I}^{\uparrow}$ or $\hat{I}^{\downarrow}$ for each node in line 2. Line 3 also initializes L to be empty. Line 4 to line 13 is the main body to compute L: we pick a 2-hop cluster at line 5, for $\hat{I}_w^{\downarrow}$ or $\hat{I}_w^{\uparrow}$ represents D_w or A_w. Line 6 to line 9 add w into all $L_{out}(a)$ and $L_{in}(d)$. Then we update $I_{\mathcal{A}'}^{\uparrow}$, $I_{\mathcal{D}'}^{\downarrow}$, $\hat{I}^{\uparrow}$, $\hat{I}^{\downarrow}$ and B from Line 10 to line 12. To update B, we also remove nodes with zero degree, since the $\hat{I}_u^{\downarrow}$ or $\hat{I}_v^{\uparrow}$ is then zero. Note that

u	D_u
n_2	$\{n_2, n_6, n_7, n_8, n_9, n_{10}\}$
n_4	$\{n_4, n_6, n_8, n_9, n_{10}\}$
n_5	$\{n_5, n_7, n_8, n_{10}\}$

(a) D_u

v	A_v
n_6	$\{n_1, n_2, n_4, n_6\}$
n_7	$\{n_1, n_2, n_3, n_5, n_7\}$
n_8	$\{n_1, n_2, n_3, n_4, n_5, n_6, n_8\}$

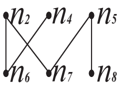
(b) A_v

u	$\hat{I}^{\downarrow}_u$
n_2	[1,6]
n_4	[1,2][4,4][6,6][9,9]
n_5	[4,7]

(c) $\hat{I}^{\downarrow}_u$

v	$\hat{I}^{\uparrow}_v$
n_6	[4,7]
n_7	[1,5]
n_8	[1,2][4,8]

(d) $\hat{I}^{\uparrow}_v$



(e) B

Figure 14: The Begining

u	D_u
n_2	$\{n_2, n_7\}$
n_5	$\{n_5, n_7, n_8, n_{10}\}$

(a) D_u

v	A_v
n_7	$\{n_1, n_2, n_3, n_5, n_7\}$
n_8	$\{n_1, n_2, n_3, n_4, n_5, n_6, n_8\}$

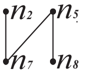
(b) A_v

u	$\hat{I}^{\downarrow}_u$
n_2	[3,3][5,5]
n_5	[4,7]

(c) $\hat{I}^{\downarrow}_u$

v	$\hat{I}^{\uparrow}_v$
n_7	[1,5]
n_8	[1,2][4,8]

(d) $\hat{I}^{\uparrow}_v$



(e) B

$I^{\uparrow}_{\mathcal{A}'}$	[4,7]
$I^{\downarrow}_{\mathcal{D}'}$	[1,2][4,4] [6,6]

(f) $I^{\uparrow}_{\mathcal{A}'}$ and $I^{\downarrow}_{\mathcal{D}'}$

Figure 15: Step 1

u	D_u
n_2	$\{n_2, n_7\}$

(a) D_u

v	A_v
n_7	$\{n_2, n_7\}$

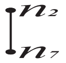
(b) A_v

u	$\hat{I}^{\downarrow}_u$
n_2	[3,3][5,5]

(c) $\hat{I}^{\downarrow}_u$

v	$\hat{I}^{\uparrow}_v$
n_7	$[3, 3][5, 5]$

(d) $\hat{I}^{\uparrow}_v$



(e) B

$I^{\uparrow}_{\mathcal{A}'}$	[1,2][4,7]
$I^{\downarrow}_{\mathcal{D}'}$	[1,2][4,7]

(f) $I^{\uparrow}_{\mathcal{A}'}$ and $I^{\downarrow}_{\mathcal{D}'}$

Figure 16: Step 2

there are $V_{\top} = \mathcal{A}$ and $V_{\perp} = \mathcal{D}$ for the remaining processing, since we compute the cover L based on an edge-separator.

Example 6.2: Major steps of the edge-oriented approach of *CrossCover()* for our running example are shown in figures from Figure 14 to Figure 16. Each figure shows those D_u and $\hat{I}^{\downarrow}_u$ for $u \in U$, A_v and $\hat{I}^{\uparrow}_v$ for $v \in V$, the bipartite graph B, and the covered intervals $I^{\uparrow}_{\mathcal{A}'}$ and $I^{\downarrow}_{\mathcal{D}'}$, if they are not empty. All nodes u (or v) in every entry D_u (or A_v) of Figure 14(a) (or (b)) are represented as numbers $po^{\downarrow}_u$ (or $po^{\uparrow}_v$) included in the intervals in the corresponding entry of Figure 14(c) (or (d)), which can be referred to Table 1. Note that the D_u or A_v of in Figure 14(a) or 14(b) do not need to be maintained explicitly, they are shown only for explanation. We only maintain $I^{\uparrow}_{\mathcal{A}'}$ and $I^{\downarrow}_{\mathcal{D}'}$ as shown in Figure 14(a) and Figure 14(b) when computing the *CrossCover()*. Above description also applies to those in Figure 15 and Figure 16. We give detailed description for each step until we finished computing the *CrossCover()* as below. □

The Begining: As shown in Figure 14, for $u \in U$ and $v \in V$, D_u and A_v consists of all of its descendants or ancestors in G_D and G_A, respectively. They are shown in Figure 14(a)(b). Those $\hat{I}^{\downarrow}_u$ and $\hat{I}^{\uparrow}_v$ are initially computed based on $\hat{I}^{\downarrow}_u = I^{\downarrow}_u \sqcap I^{\downarrow}(\mathcal{D})$, which are shown in Figure 14(c)(d). $I^{\uparrow}_{\mathcal{A}'}$ and $I^{\downarrow}_{\mathcal{D}'}$ are both empty.

Step 1: Consider the six candidates of centers in Figure 14(e). We want to get the 2-hop clusters of all nodes from Figure 14(c)(d). That is, to obtain $S(A_w, w, D_w)$ with only A_w or D_w for each node, which is possible based on Property 6.1. With the computed $\hat{I}^{\uparrow}_w$ and $\hat{I}^{\downarrow}_w$, A_w and D_w are known for $S(A_w, w, D_w)$. We could get all nodes in $\hat{I}^{\uparrow}_w$ and $\hat{I}^{\downarrow}_w$ with the reachability table. For example, we have $\hat{I}^{\downarrow}_{n_6} = \{[1, 2][4, 4][6, 6]\}$ and there are 1, 2, 4 and 6—four $po^{\downarrow}$ numbers for some nodes. From Table 1, n_9, n_6, n_{10} and n_8 belong to those $po^{\downarrow}$ numbers. On the other hand, $\hat{I}^{\uparrow}_{n_6} = \{[4, 7]\}$ and there are 4, 5, 6 and 7—four $po^{\uparrow}$ numbers for some nodes. From Table 1, n_1, n_2, n_4 and n_6 belong to those $po^{\uparrow}$. To find a best $S(A_w, w, D_w)$, Eq.(8) is used here. It only involves the known $\hat{I}^{\uparrow}_w$ and $\hat{I}^{\downarrow}_w$ for each node, and $I^{\uparrow}_{\mathcal{A}'}$ and $I^{\downarrow}_{\mathcal{D}'}$ which we explicitly maintain. After we pick the best w and the associated $\hat{I}^{\uparrow}_w$ and $\hat{I}^{\downarrow}_w$, we restore a corresponding $S(A_w, w, D_w)$ with reachability table. In this step,n_6 is picked out. We identify $S(A_{n_6}, n_6, D_{n_6}) = S(\{n_1, n_2, n_4, n_6\}, n_6, \{n_6, n_7, n_8, n_{10}\})$ (Figure 11(b)).

$$
\begin{aligned}
\hat{I}^{\downarrow}_{n_6} &= \hat{I}^{\downarrow}_{n_2} \sqcup \hat{I}^{\downarrow}_{n_4} \sqcap \{[1, 2][4, 4][6, 6]\} = \{[1, 2][4, 4][6, 6]\} \\
\hat{I}^{\downarrow}_{n_7} &= \hat{I}^{\downarrow}_{n_2} \sqcup \hat{I}^{\downarrow}_{n_5} \sqcap \{[4, 5]\} = \{[4, 5]\} \\
\hat{I}^{\downarrow}_{n_8} &= \hat{I}^{\downarrow}_{n_5} \sqcap \{[4, 4][6, 6]\} = \{[4, 4][6, 6]\} \\
\hat{I}^{\uparrow}_{n_2} &= \hat{I}^{\uparrow}_{n_6} \sqcup \hat{I}^{\uparrow}_{n_7} \sqcap \{[4, 5]\} = \{[4, 5]\} \\
\hat{I}^{\uparrow}_{n_4} &= \hat{I}^{\uparrow}_{n_6} \sqcap \{[4, 4][6, 6]\} = \{[4, 4][6, 6]\} \\
\hat{I}^{\uparrow}_{n_5} &= \hat{I}^{\uparrow}_{n_7} \sqcup \hat{I}^{\uparrow}_{n_8} \sqcap \{[1, 2][4, 4]\} = \{[1, 2][4, 4]\}
\end{aligned}
$$

Table 2: Step 1: Compute $I^{\uparrow}_u$ and $I^{\downarrow}_v$

Then, we update B and intervals of its adjacent nodes with the set of intervals computed based on Property 6.1. In this step, $\hat{I}^{\downarrow}_{n_6} = \{[1, 2][4, 4][6, 6]\}$. We do updates based on Property 6.2 for all adjacent nodes of the center w on B. In this step, they are n_2 and n_4. We further find that n_4 needs no update. Because n_4 will be disconnected in B after B is update by removing n_6, which means n_4 can acquire no intervals based on Property 6.1. We do update on n_2 by $\hat{I}^{\downarrow}_{n_2} = \hat{I}^{\downarrow}_{n_2} - \hat{I}^{\downarrow}_{n_6} = \{[1, 6]\} \ominus \{[1, 2][4, 4][6, 6]\} = \{[3, 3][5, 5]\}$. This is equivalent as $D_{n_2} = D_{n_2} - D_{n_6} = \{n_2, n_6, n_7, n_8, n_9, n_{10}\} - \{n_6, n_7, n_8, n_{10}\} = \{n_2, n_7\}$. B is updated as in Figure 15(e). And $I^{\uparrow}_{\mathcal{A}'} = I^{\uparrow}_{\mathcal{A}'} \sqcup \hat{I}^{\uparrow}_{n_6} = \{[4, 7]\}$ and $I^{\downarrow}_{\mathcal{D}'} = I^{\downarrow}_{\mathcal{D}'} \sqcup \hat{I}^{\downarrow}_{n_6} = \{[1, 2][4, 4][6, 6]\}$.

Step 2 Based on Property 6.1, we compute A_w or D_w for each node on Figure 15(e).

$$
\begin{aligned}
\hat{I}^{\downarrow}_{n_7} &= \hat{I}^{\downarrow}_{n_2} \sqcup \hat{I}^{\downarrow}_{n_5} \sqcap \{[4, 5]\} = \{[4, 5]\} \\
\hat{I}^{\downarrow}_{n_8} &= \hat{I}^{\downarrow}_{n_5} \sqcap \{[4, 4][6, 6]\} = \{[4, 4][6, 6]\} \\
\hat{I}^{\uparrow}_{n_2} &= \hat{I}^{\uparrow}_{n_7} \sqcap \{[4, 5]\} = \{[4, 5]\} \\
\hat{I}^{\uparrow}_{n_5} &= \hat{I}^{\uparrow}_{n_7} \sqcap \hat{I}^{\uparrow}_{n_8} \{[1, 2][4, 4]\} = \{[1, 2][4, 4]\}
\end{aligned}
$$

Table 3: Step 2: Compute $I^{\uparrow}_u$ and $I^{\downarrow}_v$

With the computed $\hat{I}^{\uparrow}_w$ and $\hat{I}^{\downarrow}_w$ for each node, we consider the value of Eq. (8). Particularly, for $S(A_{n_8}, n_8, D_{n_8})$, the value of Eq. (8) is computed as $|\hat{I}^{\downarrow}_8| \times |\hat{I}^{\uparrow}_8| - |\hat{I}^{\downarrow}_8 \sqcap I^{\downarrow}_{\mathcal{D}'}| \times |\hat{I}^{\uparrow}_8 \sqcap I^{\uparrow}_{\mathcal{A}'}| = 7 \times 4 - |\{[4, 4][6, 6] \sqcap \{[1, 2][4, 4][6, 6]\}| \times |\{[1, 2][4, 8]\} \sqcap \{[4, 7]\}| = 7 \times 4 - |\{[4, 4][6, 6]\}| \times |\{[4, 7]\}| = 5$. This is because $S(A_{n_8}, n_8, D_{n_8})$ can cover total 14 connections , however, 8 connections are already covered by some existing 2-hop clusters. $S(A_{n_5}, n_5, D_{n_5})$ can cover total 12 connections , however, only 2 connections are already covered, where the number of newly covered connections can be obtained by Eq. (8) to be 2. So $S(A_{n_5}, n_5, D_{n_5})$ is restored with $\hat{I}^{\uparrow}_5$ and $\hat{I}^{\downarrow}_5$. And $S(A_{n_5}, n_5, D_{n_5}) = (\{n_1, n_3, n_5\}, n_5, \{n_5, n_7, n_8, n_{10}\})$. It is shown in Figure 11(c). We then update intervals of its adjacent nodes. $\hat{I}^{\uparrow}_{n_7} = \{[1, 5]\} \ominus \{[1, 2][4, 4]\} = \{[3, 3][5, 5]\}$. $\hat{I}^{\uparrow}_{n_8}$ is isolated and needs no update. B is updated(Figure 16(e)). $I^{\uparrow}_{\mathcal{A}'}$ and $I^{\downarrow}_{\mathcal{D}'}$ are updated(Figure 16(f)).

Step 3: Consider the four candidates of centers in Figure 16(e). $\hat{I}^{\downarrow}_{n_7} = \hat{I}^{\downarrow}_{n_2} \sqcap \{[4,5]\} = \{[5,5]\}$ and $\hat{I}^{\uparrow}_{n_2} = \hat{I}^{\uparrow}_{n_7} \sqcap \{[4,5]\} = \{[4,5]\}$ We have $D_{n_7} = \{n_2\}$ and $A_{n_2} = \{n_7\}$. So $S(A_{n_2}, n_2, D_{n_2})$ and $S(A_{n_7}, n_7, D_{n_7})$ cover the same number of connection, to break the tie arbitrarily, n_2 is picked out, and B is updated to be empty. Figure 11(d) shows the $S(A_{n_2}, n_2, D_{n_2})$ identified. All connections between $G_{\top}$ and $G_{\perp}$ are now covered.

7. PERFORMANCE EVALUATION

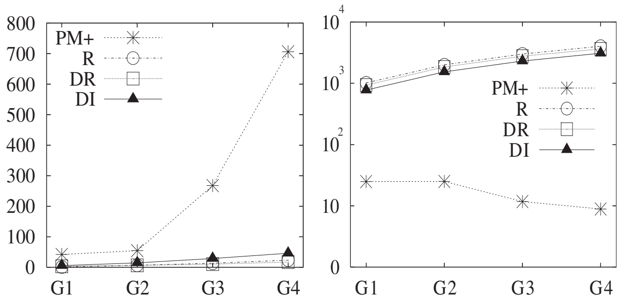
We conducted extensive experiment studies to evaluate the performance of different approaches for 2-hop cover construction, including divide-and-conquer approaches and the fast R-tree approach with no graph partitioning [6]. Specifically, we implemented algorithms of the existing bottom-up strategies [18, 19]. We use PM to illustrate the performance of the approach in [18] and PM+ the approach in [19], for they both employ a bottom-up strategy as partitioning and merging, while the latter further uses an auxiliary 2-hop cover of PSG. For our top-down strategy, we use DR to denote the R-tree method and DI the interval method for *CrossCover()* computation . The R-tree approach [6] will be showed as R for performance compare. All those algorithms are implemented using C++.

We generated various synthetic data using two graph generators, namely, the random directed graph generator named *GraphBase* developed by Knuth [16] and the random directed acyclic graph generator *DAG-Graph* developed by Johnson baugh [15]. We vary two parameters, $|V|$ and $|E|$, in the two generators, and use the default values for the other parameters. We also tested several large XML graphs as the real datasets, including XMark [20] and DBLP[1].

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

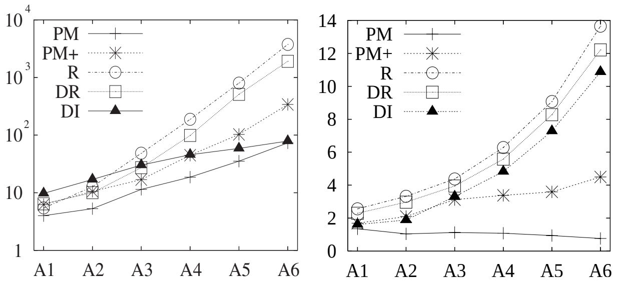
7.1 Exp-1: General Directed Graphs

To compare the performance of the five algorithms to compute 2-hop covers, we conduct experiments on random directed graphs generated by *GraphBase*. 4 sets of graphs are generated for this purpose and graphs in each sets contain the same number of nodes and edges but with various seeds for the random graph generator. The graphs in the first set have $20,000$ nodes and $30,000$ edges, denoted by G1; those in the second set have $40,000$ nodes and $60,000$ edges, denoted by G2; those in the third set have $60,000$ nodes and $90,000$ edges, denoted by G3; while those in the last set have $80,000$ nodes and $120,000$ edges, denoted by G4. We show the performance on one graph for each set. The node number in each partition of original graph is limited to be under $10,000$. For PM+, it needs to partition PSG for G3 and G4. Particularly, we found it is hard to compare the performance of PM, for it needs a lot of time in the cover joining phase but results in low compression rate. It needs to construct 2-hop clusters based on all ancestors and descendants in the whole graph for all end nodes of cross-partition edges. Those 2-hop clusters can be very large, since those sets of ancestors and descendants in G can be very large. For example, the smallest given graphs G1 has total $24,172,802$ connections, PM spends 237.74 seconds and obtains a cover size of $29,233,309$, which is larger than the total number of connections in the graph. We illustrate performance of its updated version PM+ only in Figure 17.



(a) Elapsed Time (sec.) (b) Compression Rate

Figure 17: General Graph Testing



(a) Elapsed Time (sec.) (b) Compression Rate

Figure 18: Dense DAG Testing

In Figure 17(a), as the scale of the underlying graph increases, the 2-hop cover construction time required by the top-down strategy increases sharply. There are some reasons: All processing are based on the transitive closure of the graph with set operations; it also needs to mandatorily construct a number of 2-hop clusters based on a large number of nodes as those ancestors and descendants for all end nodes on cross-partition edges in some partition, as well as to further partition and compute the 2-hop covers for PSGs recursively for large graphs. In addition, due to the recursively partitioning and repeatedly cover joining for PSGs, the extra space consumption for correlated 2-hop labeling can be multiplied. In Figure 17(b) for PM+, there is even a decline regarding the compression ratio for G3 and G4. For other approaches, Figure 17 can not produce a clear difference between them. Take G4 as an example, PM+ consumes as much as 706 second, while R, DR and DI only spend 24, 17 and 48 seconds to achieve the compression rate as 4026, 3643 and 3100. This means the R-tree approach is still a good choice for random general directed graphs with less than $100,000$ nodes. We will test them with more challenging settings.

7.2 Exp-2: Dense DAGs

We tested dense DAGs. There is a high ratio of edge number to node number and the graph can not be reduced by collapsing strong connected components in it. We now experiment on 6 sets of random directed acyclic graphs generated by *DAG-Graph*. We will follow the practice in the previous experiment for the similarity of performance on all graphs in a same set. Those DAGs are denoted by A1 to A6. We fix the number of nodes in all graphs to be $20,000$ and increase the number of edges in each set of graphs. The numbers of edges for for A1 to A6 are $30,000$, $40,000$, $50,000$, $60,000$, $70,000$ and $80,000$. We compare performance be-

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

tween the five algorithms in Figure 18.

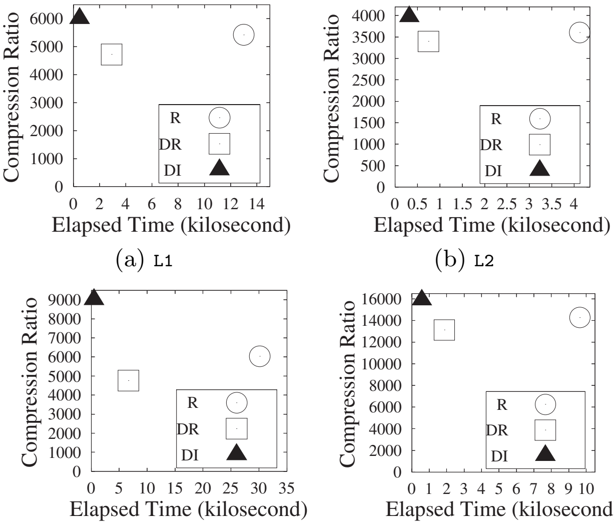
For R and DR, Figure 18(a) shows that there is a sharp rise in elapsed time when the number of edges in DAGs increases. Though R and DR need less time than DI needs on A1 and A2, they spend much more time than that of DI for the remaining graphs. In Figure 18(b), the R, DR and DI all tend to achieve large compression rate when the graph becomes dense, and they perform similarly. Specifically, for A6, R and DR spend $3,752$ and $1,907$ seconds to achieve the compression rate as 14 and 12, respectively. But DI only spends 79 seconds to gain the compression ratio as 10. On the other hand, PM requires the least time, for there are a relatively small number of nodes thus the transitive closure can be relatively small. However, its compression rate begins to decline under 1 from A5. Compared to PM, PM+ displays better compression rate but needs more time for computation. It takes 341 seconds to obtain 5 as the compression rate on A6. Obviously, DI is a good choice for large dense DAGs among all.

7.3 Exp-3: Large Graphs

To test large general graphs with nodes more than $100,000$, we use *GraphBase* to generate 4 random general directed graphs, denoted by L1 to L4. The sizes for L1 to L4 are as follows: $400,000$ nodes, $500,000$ edges and $4,657,410,000$ connections for L1; $400,000$ nodes, $600,000$ edges and $2,669,670,000$ connections for L2; $500,000$ nodes, $600,000$ edges and $6,867,920,000$ connections for L3; $500,000$ nodes, $700,000$ edges and 13,737,200,000 connections for L4. Since R, DR and DI outperform PM and PM+ largely on both general directed graphs and DAGs, we will focus on R, DR and DI below. Figure 19 shows the efficiency of the three algorithms on large graphs. In each figure of Figure 19, the x-axis indicates time efficiency and the y-axis indicates space efficiency (the compression rate). A nice algorithm can result in points to reside close to the top-left corner, in terms of both of the two measurements. We can see from Figure 19 that DI is such a nice algorithm in terms of time and space efficiency. DI can be faster than R up to 2 orders of magnitudes. It is also noticeably faster than DR. On the other hand, the compression ratio of DI is the largest among those of the others, while R outperforms DR slightly. This fact supports the efficiency of our heuristics described in Section 4 for large graphs. DI can possibly achieve a smaller 2-hop cover than R(the algorithm without graph partitioning). For DI, there are more centers to be searched than DR. And DI can obtain more balanced bisection on graphs. This can explain there are a loss of quality for DR when compared to DI. For the 4 graph, DI only uses 479.54, 317.66, 483.65 and 574.70 seconds, while R, which does not partition the graph, requires $12,999.76$, $4,122.77$, $30,121.69$ and $9,634.87$ seconds. The compression rate of DI can be as high as $14,284.54$ for L4.

7.4 Exp-4: XML and Real Datasets

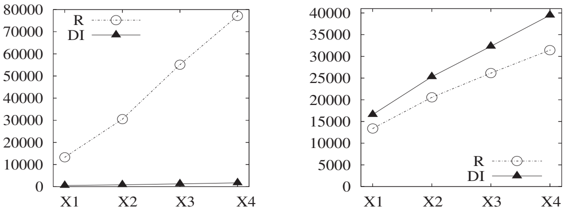
We generated four XML datasets based on *XMark* benchmark [20] using four factors, 0.4, 0.6, 0.8, and 1.0, and name them as X1, X2, X3, and X4, respectively. Then, for each dataset, we generate a large graph by treating both parent-child and ID/IDREF relationships as edges with no difference. We compare the performance of DI, the best algorithm with graph partition, to R, the algorithm to compute 2-hop covers without graph partitioning. The performance of R is given as the one without graph partitioning. We can see in Figure 20 that DI is not only faster than R but also achieves a higher compression rate than R does. The time consumption of DI scales linearly in terms of the size of the datasets, ranging from 530.56 to $1,723.95$ seconds. But for R, it ranges from $13,298.55$ to $77,170.60$ seconds with a fast increasing rate. For X4, which contains 1.67 million nodes and 169 billion connections, DI achieves a compression rate as high as $39,492.28$, while R is $31,455.11$ for that.



(a) L1 (b) L2

(c) L3 (d) L4

Figure 19: Large Graph Testing



(a) Elapsed Time (sec.) (b) Compression Rate

Figure 20: XMark Benchmark Testing

For the DBLP dataset, we get all nodes from elements in the XML data and edges from all parent-child relationships and bibliographic links. This DBLP graph contains $3,170,917$ nodes with more than 4.352 billion connections. R has spent a long time as $26,592.53$ seconds achieving a compression rate of $1,099.99$. DI only requires 566 seconds with a smaller compression rate as 751.72, which is still satisfactory. The lower quality of DI can be explained that the edges in DBLP graph are distributed very irregularly. Because bibliographic links can be concentrated to a certain small number nodes, while there are also many nodes with very few edges. Even when we perform a balanced bisection, those 2-hop clusters are still not so good as those exhaustively found by R. For such cases, a 2-hop cover program with a large search space for 2-hop clusters like R can be more space efficient.

8. RELATED WORK

There can be two simple solutions to process reachability queries in a directed graph. They are (i) maintaining the transitive closure of edges, which results in high storage consumption, and (ii) the breadth- or depth-first search to explore a connection from u to v in the given graph on demand, which may incur high query-time processing cost. Broadly speaking, almost all of proposed approaches contain ingredients similar to either (i) or (ii) and may be a tree-component method or not. We first discuss approaches of (i) and then those of (ii).

Since the space requirement to explicitly store the edge transitive closure is $O(|V|^2)$ and it can be prohibiting for large graphs. So the method using an explicit storage of the transitive closure to process reachability queries is impractical. Several works tried to compress the edge transitive closure of a graph to explore a space- and time-efficient approach for graph reachability queries. For example, Agrawal et al. studied efficient management of transitive relationships in large databases [1] based on intervals. However, its space requirement is $O(|V|^2)$, the same as that of the transitive closure for a graph. Cohen et. al studied reachability labeling using 2-hop labels [7] based on 2-hop covers, a compressed form of the edge transitive closure of a graph with $O(|V| \cdot |E|^{1/2})$ space requirement. Unfortunately, computing 2-hop covers in [7] requires the edge transitive closure of a graph to be computed first, which again can be too costly to be practical for large graphs. Then, Schenkel et al. [18, 19] studied 2-hop cover problem and proposed a divide-and-conquer approach. There are works with the tree-component method. [12] and [23] use a tree labeling based on intervals to encode all reachability relationships for a spanning tree of the graph. For those remaining reachability relationships, [12] computes a 2-hop cover to encode all connections that can not be found in the tree, while [23] explicitly stores those remaining reachability relationships in a transitive link table to achieve constant time querying, at the cost of space. Both of the two works in [12, 23] are motivated by the fact that graphs being often used are large sparse graphs. While in our work, we focus on computing reachability labelings for arbitrary graphs which can be either sparse or dense.

For the prototype solution (ii) mentioned above to process reachability queries in a directed graph, the querying time can be in $O(|V|)$ by the breadth- or depth-first search. So works in [5, 21] first encodes reachability relationships over a spanning tree generated by depth-first traversal of a directed graph based on intervals. Second, for the reachability relationship that may exist over DAG but not in the spanning tree, [5] and [21] employ search strategies triny to explore a path at query time, which requires $O(|E| - |V|)$ time.

9. CONCLUSION

The 2-hop cover can compactly represent the whole edge transitive closure of a graph, and be effectively used to answer reachability queries between nodes in the graph. However, it is challenging to compute such a 2-hop cover for a large graph. In this paper, we proposed a new top-down hierarchical partitioning approach. We partition a graph into two subgraphs repeatedly. We focus on 2-hop cover sensitive partitioning, and proposed two approaches, namely, a node-oriented approach and an edge-oriented approach that use the computed 2-hop covers (precisely centers) to partition graph. We conducted extensive performance studies, and confirmed that our approach significantly outperform the existing approaches when a graph is large and dense.

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] K. Anyanwu and A. Sheth. ρ-queries: enabling querying for semantic associations on the semantic web. In *WWW '03*, 2003.

[3] B. Berendt and M. Spiliopoulou. Analysis of navigation behaviour in web sites integrating multiple information systems. *The VLDB Journal*, 9(1), 2000.

[4] D. Brickley and R. V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. W3C Candidate Recommendation, 2000.

[5] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. of VLDB'05*, 2005.

[6] 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.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of SODA'02*, 2002.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, 2001.

[9] S. DeRose, E. Maler, and D. Orchard. XML linking language (XLink) version 1.0. 2001.

[10] S. DeRose, E. Maler, and D. Orchard. XML pointer language (XPointer) version 1.0. 2001.

[11] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Rec.*, 26(3), 1997.

[12] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *CIKM '05*, 2005.

[13] H. Imai and T. Asano. Efficient algorithms for geometric graph search problems. *SIAM J. Comput.*, 15(2):478–494, 1986.

[14] D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proc. of STOC'73*, 1973.

[15] R. Johnsonbaugh and M. Kalin. A graph generation software package. In *Prof. of SIGCSE'91*, 1991.

[16] D. E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing.* ACM Press, 1993.

[17] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC '04*, 2004.

[18] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex XML document collections. In *Proc. of EDBT'04*, 2004.

[19] 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.

[20] 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.

[21] S. TrißI and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD '07*, 2007.

[22] J. van Helden, A. Naim, R. Mancuso, , M. Eldridge, L. Wernisch, D. Gilbert, and S. Wodak. Reresenting and analysing molecular and cellular function using the computer. *Journal of Biological Chemistry*, 381(9-10), 2000.

[23] 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.