Fast Computation of Reachability Labeling for Large Graphs

Jiefeng Cheng[1], Jeffrey Xu Yu[1], Xuemin Lin[2], Haixun Wang[3], Philip S. Yu[3]

[1] The Chinese University of Hong Kong, China, {jfcheng,yu}@se.cuhk.edu.hk
[2] University of New South Wales & NICTA, Australia, lxue@cse.unsw.edu.au
[3] T. J. Watson Research Center, IBM, USA, {haixun,psyu}@us.ibm.com

Abstract. The need of processing graph reachability queries stems from many applications that manage complex data as graphs. The applications include transportation network, Internet traffic analyzing, Web navigation, semantic web, chemical informatics and bio-informatics systems, and computer vision. A graph reachability query, as one of the primary tasks, is to find whether two given data objects, u and v, are related in any ways in a large and complex dataset. Formally, the query is about to find if v is reachable from u in a directed graph which is large in size. In this paper, we focus ourselves on building a reachability labeling for a large directed graph, in order to process reachability queries efficiently. Such a labeling needs to be minimized in size for the efficiency of answering the queries, and needs to be computed fast for the efficiency of constructing such a labeling. As such a labeling, 2-hop cover was proposed for arbitrary graphs with theoretical bounds on both the construction cost and the size of the resulting labeling. However, in practice, as reported, the construction cost of 2-hop cover is very high even with super power machines. In this paper, we propose a novel geometry-based algorithm which computes high-quality 2-hop cover fast. Our experimental results verify the effectiveness of our techniques over large real and synthetic graph datasets.

1 Introduction

Consider a reachability query querying whether a node v is reachable from node u in a large directed graph, G. There are several possible yet feasible solutions for efficiently answering such a query, as indicated in [2]. Those solutions include i) maintaining the transitive closure of edges, which results in high storage consumption, and ii) computing the shortest path from u to v over such a large graph on demand, which results high query processing cost. A 2-hop reachability labeling, or 2-hop cover, was proposed by Cohen et al, as a feasible solution, to answer such reachability queries [2]. The key issue is how to minimize such a 2-hop cover, because the minimum 2-hop cover leads to the efficiency of answering reachability queries. The problem is shown to be NP-hard, because minimum 2-hop cover is a minimum set cover problem. Cohen et al proposed an approximation solution. The theoretical bound on the size of 2-hop cover is also provided. Despite the excellence of the theoretical bound on the time complexity, the cost for computing the minimum 2-hop cover is high when dealing with large graphs. 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 344,992,370 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. The long construction time makes it difficult to construct such a 2-hop cover for large graphs.

The main contribution of our work in this paper are summarized below.

- We propose a set cover I solution (*SCI*) instead Cohen et al's set cover II solution (*SCII*) [8], where *SCI* minimizes the number of subsets in a set cover and *SCII* minimizes the overlapping among subsets in a set cover. We show evidences that *SCI* can achieve a similar satisfactory level as *SCII* as to minimize the 2-hop cover for a large graph, and at the same time can compute 2-hop cover efficiently.
- We propose a novel geometry-based algorithm to further improve the efficiency of computing 2-hop cover. The two main features of our solution are given below. First, we do not need to compute transitive closure as required in all algorithms that need to compute 2-hop. Second, we map the 2-hop cover problem onto a two-dimensional grid, and compute 2-hop using operations against rectangles with help of a R-tree.
- We conducted extensive experimental studies using different graph generators, and real datasets, with different parameter settings. Our results support our approach as it can significantly improve the efficiency of finding 2-hop cover for large graphs.

The remainder of this paper is organized as follows. Section 2 gives the definition of the 2-hop cover problem. Section 3 discusses our motivation of solving the 2-hop cover problem using a set cover I solution [8] instead of the set cover II solution used in Cohen et al's study [2]. Our work is motivated by the main requirements of the 2-hop cover problem: minimization of the 2-hop cover and minimization of processing time. Section 4 discusses a new geometry-based approach as a set cover I solution to the 2-hop cover problem. Experimental results are presented in Section 5 followed by related work in Section 6. Finally, Section 7 concludes the paper.

2 Problem Definition

The 2-hop reachability labeling is defined in [2]. We introduce it below in brief. Let $G = (V, E)$ be a directed graph. A 2-hop reachability labeling on graph G assigns every node $v \in V$ a label $L(v) = (L_{in}(v), L_{out}(v))$, where $L_{in}(v), L_{out}(v) \subseteq V$ such as every node x in $L_{in}(v)$ connects to every node y in $L_{out}(v)$ via the node v. A node v is reachable from a node u, denoted $u \rightsquigarrow v$, if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. The size of the 2-hop reachability labeling over a graph $G(V, E)$, is given as L, below.

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

In order to solve the 2-hop reachability labeling, Cohen et al. introduced 2-hop cover, which is given below [2].

Definition 1. *(***2-hop cover***) Given a directed graph $G = (V, E)$. Let $P_{u \rightsquigarrow v}$ be a set of paths from node u to node v in G, and P be a set of all such $P_{u \rightsquigarrow v}$ in G. A hop,*

h_u, is defined as $h_u = (p_u, u)$, where p_u is a path in G and u is one of the endpoints of p_u. A 2-hop cover, denoted H, is a set of hops that covers P, such as, if node v is reachable from node u then there exists a path p in the non-empty $P_{u \rightsquigarrow v}$ where the path p is concatenation of p_u and p_v, denoted $p = p_u p_v$, and $h_u = (p_u, u)$ and $h_v = (p_v, v)$.

The 2-hop reachability labeling can be derived from a 2-hop cover [2]. In addition, the size of the 2-hop cover, $|H|$, for a graph G, is the same as that of 2-hop reachability labeling ($|H| = L$).

The 2-hop cover problem is to find the minimum size of 2-hop cover for a given graph $G(V, E)$, which is proved to be NP-hard [2]. Cohen et al show that a greedy algorithm exists to compute a nearly optimal solution for the 2-hop cover problem. The resulting size of the greedy algorithm is larger than the optimal at most $O(\log n)$. The basic idea is to solve the minimum 2-hop cover problem as a minimum set cover problem [8]. Note: in the corresponding minimum set cover problem, a set is a set of edges.

We illustrate Cohen et al's algorithm in Algorithm 1. We call it *MaxDSCovering*. In Algorithm 1, it initializes the 2-hop cover H (line 1), and computes the transitive closure, T, for the given graph G (line 2). Here, T is treated as the ground set of the minimum set cover problem. The main body of the algorithm is a while loop, which repeatedly finds hops until T becomes empty (line 3-14). The 2-hop cover is returned in line 15. In line 5-10, it finds a densest bipartite graph, B, which has node w as its virtual center, by calling a function *denSubGraph*. In *denSubGraph*, the densest bipartite graph is constructed, based on a node w, in two main steps.

– Construct a bipartite graph $B_C(V_C, E_C)$ where $V_C = V_{C_{in}} \cup V_{C_{out}}$, based on node w, such as

$$V_{C_{in}} = \{u \mid (u, w) \in T\} \cup \{w\} \tag{2}$$
$$V_{C_{out}} = \{v \mid (w, v) \in T\} \cup \{w\} \tag{3}$$

and

$$E_C = V_{C_{in}} \times V_{C_{out}} \tag{4}$$

The sets, $V_{C_{in}}$ and $V_{C_{out}}$, are all connected via the virtual center w, respectively.

– Find the densest bipartite graph, denoted $B(V, E)$, where $V = V_{in} \cup V_{out}$, from B_C, such as

$$\max_{\substack{V_{in} \subseteq V_{C_{in}} \\ V_{out} \subseteq V_{C_{out}} \\ E \subseteq E_C}} \frac{|E \cap T'|}{|V_{in}| + |V_{out}|} \tag{5}$$

where T' is the set of uncovered edges. Note: finding the minimum set over is equivalent to find the densest subgraph [2], as illustrated in Eq. (5). As a densest subgraph problem, it can be solved in polynomial time [5].

The candidate bipartite graph $\mathcal{B}$ with the highest score (Eq. (5)), after checking every node in G, is identified after line 10. In line 11-12, new hops are added into the 2-hop cover, based on $\mathcal{B}$. After it, the set of edges of $\mathcal{B}$, denoted $\mathcal{E}(\mathcal{B})$ will be removed from T'.

Algorithm 1 *MaxDSCovering*(G)

Input: graph, $G(V, E)$.
Output: 2-hop cover, H.

1: $H \leftarrow \emptyset$;
2: $T' \leftarrow T \leftarrow \{(u, v) \mid P_{u \rightsquigarrow v} \neq \emptyset\}$;
3: **while** $T' \neq \emptyset$ **do**
4: $\tau \leftarrow 0$;
5: **for all** $w \in V$ **do**
6: $B \leftarrow$ *denSubGraph*(w, T, T') with a score c_w; $\{B(V, E)$ is a bipartite graph with $V = V_{in} \cup V_{out}$.$\}$
7: **if** $c_w > \tau$ **then**
8: $v_b \leftarrow w$; $\tau \leftarrow c_w$; $\mathcal{B} \leftarrow B$;
9: **end if**
10: **end for**
11: **for all** $u \in \mathcal{V}_{in}$ of $\mathcal{B}$ **do** $H \leftarrow H \cup \{(u \rightsquigarrow v_b, u)\}$;
12: **for all** $v \in \mathcal{V}_{out}$ of $\mathcal{B}$ **do** $H \leftarrow H \cup \{(v_b \rightsquigarrow v, v)\}$;
13: $T' \leftarrow T' \setminus \mathcal{E}(\mathcal{B})$;
14: **end while**
15: **return** H;

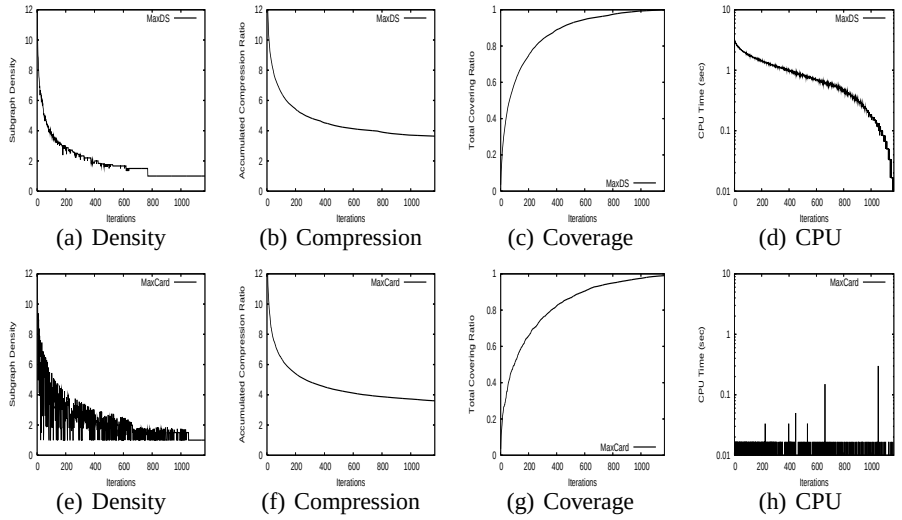
3 A Set Cover I Solution

Cohen et al. solve (approximate) the minimum 2-hop cover problem as a minimum set cover problem with a theoretical bound on the time complexity, $O(n^4)$ where n is the number of nodes in the graph G. However, it is challenging to compute such a 2-hop cover for very large graphs because the algorithm is CPU intensive as reported in [18, 19]. Also, it needs to precompute the transitive closure which requires large memory space. Recall, 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 344,992,370 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.

The minimum set cover problem used in *MaxDSCovering* is called a minimum set cover II problem, denoted *SCII*, in [8]. *SCII* is to find a set cover which has the least overlapping, and shares the same goal as 2-hop cover problem's. In [8], Johnson also gave a set cover I problem, denoted *SCI*, which is to minimize the cardinality. Here, consider a set of subsets $S_1, S_2, \cdots, S_m$, over a finite set S $(= \cup_i S_i)$. The minimum set cover I (*SCI*) is to find a smallest set of sets, denoted $\mathcal{S}$, such as $\cup_j \mathcal{S}_j = S$ for $\mathcal{S}_j \in \mathcal{S}$. The two set cover problems, *SCI* and *SCII*, are different. The optimal solution for one may not be the optimal solution for the other.

In this paper, we show that the minimum 2-hop cover problem can be solved using *SCI* effectively. By effectiveness, we mean that the size of 2-hop set identified by *SCI* is very similar to the size of 2-hop set identified by *SCII*, in practice, using greedy algorithms, when handling large graphs.

We propose an algorithm called *MaxCardinality*. The algorithm is the same as *MaxDSCovering* after replacing Eq. (5) with the following equation Eq. (6), for finding a bipartite subgraph $B(V, E)$ from a bipartite graph $B_C(V_C, E_C)$ where $V =$



(a) Density (b) Compression (c) Coverage (d) CPU

(e) Density (f) Compression (g) Coverage (h) CPU

Fig. 1. *SCII* vs *SCI* over a Directed Acyclic Graph ($|V| = 2,000$, $|E| = 4,000$)

$V_{in} \cup V_{out}$. As the name of the algorithm indicates, it is to maximize the cardinality of the edges (uncovered paths) in each bipartite graph.

$$\max_{\substack{V_{in} \subseteq V_{C_{in}} \\ V_{out} \subseteq V_{C_{out}} \\ E \subseteq E_C}} |E \cap T'| \tag{6}$$

where T' is the uncovered set.

We show the similarities and differences between the two solutions, namely, *SCI* and *SCII*, in Fig. 1, using a random graph, $G(V, E)$ where $|V| = 2,000$ and $|E| = 4,000$, generated by a graph generator [9]. In Fig. 1, it compares *MaxCardinality* (a *SCI* solution) with *MaxDSCovering* (a *SCII* solution). The figures (a)-(d) are for *MaxDSCovering*, and the figures (e)-(h) are for *MaxCardinality*. Note: both algorithms are the same as shown in Algorithm 1 except that *MaxDSCovering* uses Eq. (5) and *MaxCardinality* uses Eq. (6) for identifying a subgraph.

– In Fig. 1 (a) and (e), we show the density of the subgraph found in *MaxCardinality* and *MaxDSCovering* ($\mathcal{B}(\mathcal{V}, \mathcal{E})$ in Algorithm 1). The density (y-axis) is $|\mathcal{E}|/(|\mathcal{V}_{in}| + |\mathcal{V}_{out}|)$ where $\mathcal{V} = \mathcal{V}_{in} \cup \mathcal{V}_{out}$. It is important to note that *MaxCardinality* does not use it to compute, but uses it to measure its density per iteration, in comparison with *MaxDSCovering*.
MaxDSCovering decreases monotonically, because it always finds the best densest subgraph per iteration (Fig. 1 (a)). *MaxCardinality* decreases globally, but shows fluctuation patterns (Fig. 1 (e)), because it cannot find the best densest subgraph per iteration. However, one very important observation is that *MaxCardinality* can find a more dense graph than former selected ones in one of the following iterations after it misses a densest graph in some iteration. If we compare the two subfigures,

MaxCardinality sometime outperforms *MaxDSCovering* in many iterations for this reason.

- In Fig. 1 (b) and (f), we show the accumulated compression ratio, $|T_i|/H_i$, where T_i is the transitive closure that has been covered already at the i-th iteration, and H_i is the 2-hop cover for T_i. Both figures are almost the same. It shows that the 2-hop cover can be solved effectively using a *SCI* solution. For this graph, the size of the transitive closure, T, is $|T| = 24,888$. The sizes of the 2-hop covers found by *MaxDSCovering* and *MaxCardinality* are, 6,840 and 7,089. The difference is 249. The compression rate of the 2-hop covers by *MaxDSCovering* and *MaxCardinality* are, 0.27 ($6,800/248,88$) and 0.28 ($7,089/24,888$).
- In Fig. 1 (c) and (g), we show the coverage of the graph up to the i-th iteration, $|T_i|/|T_i'|$, where T_i is the transitive closure being covered at the i-th iteration, and T' is the transitive closure that has not been covered up to the i-th iteration. Both share the similar trend.
- In Fig. 1 (d) and (h), we show the CPU time spent in every iteration. Due to the difference between *SCI* (Eq. (5)) and *SCII* (Eq. (6)), *MaxCardinality* spends much less time than *MaxDSCovering*.

The above discussions show that a *SCI* solution can effectively and efficiently solve the minimum 2-hop cover problem.

4 A Fast Geometry-Based Algorithm for the Set Cover I Solution

In this section, we show that we can significantly improve the efficiency of *MaxCardinality* (a *SCI* solution) by solving it over a 2-dimensional space using simple operations against rectangles.

The outline of our approach is given below. First, for a given directed graph G, we construct a directed acyclic graph, denoted $G_\downarrow$. Second, we compute the 2-hop cover for the directed acyclic graph $G_\downarrow$. Third, we compute the 2-hop cover for the directed graph G using the 2-hop cover obtained for $G_\downarrow$, in a simple post-processing step. Below, we discuss the first step, and the third step and will discuss the second step in the following subsections.

Directed acyclic graph construction: Given a directed graph $G(V, E)$, we identify its strongly connected components, $C_1, C_2, \cdots$ efficiently, in the order of $O(|V| + |E|)$ [4]. Note: any two nodes are reachable if they are in the same strongly connected component, C_i. The directed acyclic graph $G_\downarrow(V_\downarrow, E_\downarrow)$ is constructed as follows. A node $v \in V_\downarrow$ represents either a strongly connected component, C_i or a node in G. If v represents a strongly connected component C_i, we randomly select one of the nodes in C_i, denoted v', as the representative in $V_\downarrow$. All other nodes in C_i will not appear in $V_\downarrow$. All the edges between the nodes in the strongly connected component C_i will not appear in $E_\downarrow$; all edges going into/from the strongly component, C_i, will be represented as edges going into/from the node v' in $E_\downarrow$. If v represents a node in G, which is not involved in any strongly connected component, the node will be added into $V_\downarrow$, and the corresponding edges going into/from v appear in $E_\downarrow$. The conversion of G to $G_\downarrow$ can be done as the same time when finding strongly connected components as a by-product.

Generation of 2-hop cover for G upon the 2-hop cover for $G_\downarrow$: Recall in a strongly component C_i, any two nodes, u and v, are reachable such as $u \rightsquigarrow v$ and $v \rightsquigarrow u$. Therefore, they share the same 2-hop. Suppose that we know the 2-hop for a node u in a strongly connected component, C_i, all the nodes in C_i should have the same 2-hop. The 2-hops can be simply added for connecting nodes in a single strongly component.

4.1 Computing 2-hop Cover for a Directed Acyclic Graph

In the following subsections, we explain how to compute 2-hop cover for a directed acyclic graph. The main techniques are: 1) to map a reachability between $u \rightsquigarrow v$ onto a grid point in a 2-dimensional grid, 2) map a bipartite graph with a virtual center into rectangles, and 3) compute the densest bipartite graph, based on Eq. (6), as to compute the largest area of rectangles. Note: R-tree can be used to assist the last step.

Below, in Section 4.2, we introduce an efficient approach [1] which computes an interval labeling for reachability over a directed acyclic graph. Note: there is no need to compute transitive closure. We will discuss space complexity between the interval labeling and 2-hop labeling in our experimental studies. In Section 4.3, we discuss a 2-dimensional reachability map, which is constructed using the interval labeling [1]. The reachability information is preserved completely in the map. In Section 4.4, we give our algorithm, and explain it using an example.

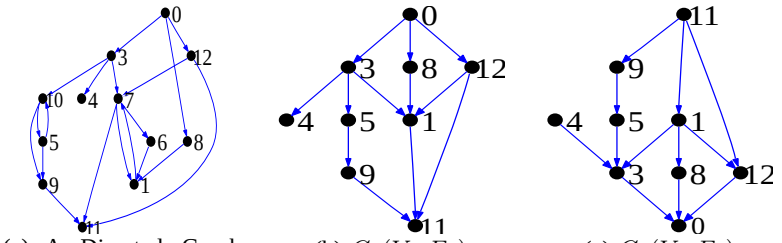
4.2 An Interval Based Reachability Labeling for Directed Acyclic Graphs

Agrawal et al [1] proposed a method for labeling directed acyclic graphs using intervals. The labeling is done in three steps for a directed acyclic graph, G_D. 1) Construct an optimum tree-cover $\mathcal{T}$. An optimum tree-cover is defined as to minimize the number of intervals. 2) Every node, v, in $\mathcal{T}$ is labeled using an internal $[s, e]$. A node v has a unique *postorder number*, denoted *po*, which is the number assigned following a postorder traversal of the tree starting from 1. The e value in $[s, e]$ for a node v is the postorder number of the node v, and the s value in the interval is the smallest postorder number of its descendants, where $s = e$ if v is a leaf node. 3) After $\mathcal{T}$ is labeled, it examines all nodes of $G_D$in the reverse topological order. During the traversal, for each node u, add all the intervals associated with v, if there exists an edge (u, v), into the interval associated with u. An interval can be eliminated if it is contained in another. Let I_u be a list of intervals assigned to a node u. Suppose there are two nodes u and v where $I_u = \{[s_1, e_1], [s_2, e_2], \cdots, [s_n, e_n]\}$, and $I_v = \{[s'_1, e'_1], [s'_2, e'_2], \cdots, [s'_m, e'_m]\}$. There exists a path from u to v iff the postorder of v is in an interval, $[s_j, e_j]$, of u.

4.3 A 2-Dimensional Reachability Map

First, we show how to construct a 2-dimensional reachability map, M. With the help of M, we want to check $u \rightsquigarrow v$ in a directed acyclic graph, $G_\downarrow$, quickly, using a function $f(u, v)$, such as $f(u, v) = 1$ iff $u \rightsquigarrow v$, and $f(u, v) = 0$ iff $u \not\rightsquigarrow v$.

The construction of the reachability map is done using two interval labelings obtained on the directed acyclic graph, $G_\downarrow(V_\downarrow, E_\downarrow)$, on which we are going to compute



(a) A Directed Graph $G_1(V_1, E_1)$ (b) $G_\downarrow(V_\downarrow, E_\downarrow)$ (c) $G_\uparrow(V_\uparrow, E_\uparrow)$

Fig. 2. A directed graph, and its two directed acyclic graphs, $G_\downarrow$ and $G_\uparrow$

w	$G_\downarrow$		$G_\uparrow$	
	$po_\downarrow(w)$	$I_\downarrow(w)$	$po_\uparrow(w)$	$I_\uparrow(w)$
0	9	[1,9]	4	[4,4]
1	1	[1,1],[3,3]	3	[1,5]
3	6	[1,6]	5	[4,5]
4	2	[2,2]	9	[4,5],[9,9]
5	5	[3,5]	6	[4,6]
8	7	[1,1],[3,3],[7,7]	1	[1,1],[4,4]
9	4	[3,4]	7	[4,7]
11	3	[3,3]	8	[1,8]
12	8	[1,1],[3,3],[8,8]	2	[2,2],[4,4]

Table 1. A Reachability Table for $G_\downarrow$ and $G_\uparrow$

its 2-hop cover, and another auxiliary directed acyclic graph, $G_\uparrow(V_\uparrow, E_\uparrow)$, respectively. Note: $G_\uparrow(V_\uparrow, E_\uparrow)$ can be easily obtained from $G_\downarrow(V_\downarrow, E_\downarrow)$, such as $V_\uparrow = V_\downarrow$, and a corresponding edge $(v, u) \in E_\uparrow$ if $(u, v) \in E_\downarrow$. In brief, for a node, u, the former can tell which nodes u can reach, and the latter can tell which nodes can reach u, fast. For the pair of graphs, $G_\downarrow$ and $G_\uparrow$, we compute the postorder numbers ($op_\downarrow$ and $op_\uparrow$) and interval labels ($I_\downarrow$ and $I_\uparrow$), using Agrawal et al's algorithm efficiently [1]. We store them in a table, called a *reachability table*.

Example 1 *As a running example, a random directed graph, $G_1(V_1, E_1)$, with 12 nodes and 19 edges, is shown in Fig. 2 (a). There are two strongly connected components. One is among nodes* 10 *and* 5*, the other is among nodes* 1, 6 *and* 7*.*

Consider the example graph G_1 (Fig. 2 (a)). Its two directed acyclic graphs, $G_\downarrow$ and $G_\uparrow$, are shown in Fig. 2 (b) and (c), respectively. In $G_\downarrow$, there are only 9 nodes out of 12 nodes in G_1, because there are two strongly connected components. One is among nodes 5 and 10, and the other is among 1, 6 and 7. We select 5 and 1 as the representatives for the former and latter strongly connected components in $G_\downarrow$. The corresponding reachability table is shown in Table 1. In Table 1, the first column is the node identifiers in G_1 (Fig. 2 (a)). The second and third columns are the postorder number and the intervals for $G_\downarrow$, and the fourth and fifth columns are the postorder number and the intervals for $G_\uparrow$.

We can virtually represent the reachability table, as an $n \times n$-grid reachability map M, where $n = |V_\downarrow| = |V_\uparrow|$. The x-axis represents the postorder numbers of the nodes

in the graph $G_{\downarrow}$, and the y-axis represents the postorder numbers of the same nodes in the graph $G_{\uparrow}$. Note, the postorder numbers are in the range of $[1, n]$. Given a pair of nodes, u and v in $G_{\downarrow}$, a function $f(u, v)$ maps it onto a grid $(x(v), y(u))$ in M, where $x(w) = op_{\downarrow}(w)$ and $y(w) = op_{\uparrow}(w)$. Here, $op_{\downarrow}(w)$ represents the postorder number of w in $G_{\downarrow}$ and $op_{\uparrow}(w)$ represents the postorder number of w in $G_{\uparrow}$. The grid value of $f(u, v)$ is 1, if $u \rightsquigarrow v$, otherwise 0.

The reachability map M for $G_{\downarrow}$ (Fig. 2 (b)) is shown in Fig 3, where a shaded grid shows a reachability $u \rightsquigarrow v$. The details for all possible $u \rightsquigarrow v$, such as $u \neq v$, $G_{\downarrow}$, are given in Table 2. For example, $3 \rightsquigarrow 9$, is mapped onto $(4, 5)$ in M, and $(4, 5)$ represents $3 \rightsquigarrow 9$, because it is shaded.

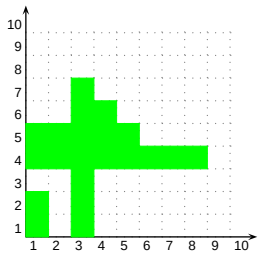


Fig. 3. Reachability Map

p	$f(p)$	p	$f(p)$	p	$f(p)$
$0 \rightsquigarrow 1$	$(1, 4)$	$0 \rightsquigarrow 3$	$(6, 4)$	$0 \rightsquigarrow 4$	$(2, 4)$
$0 \rightsquigarrow 5$	$(5, 4)$	$0 \rightsquigarrow 8$	$(7, 4)$	$0 \rightsquigarrow 9$	$(4, 4)$
$0 \rightsquigarrow 11$	$(3, 4)$	$0 \rightsquigarrow 12$	$(8, 4)$	$1 \rightsquigarrow 11$	$(3, 3)$
$3 \rightsquigarrow 1$	$(1, 5)$	$3 \rightsquigarrow 4$	$(2, 5)$	$3 \rightsquigarrow 5$	$(5, 5)$
$3 \rightsquigarrow 9$	$(4, 5)$	$3 \rightsquigarrow 11$	$(3, 5)$	$5 \rightsquigarrow 9$	$(4, 6)$
$5 \rightsquigarrow 11$	$(3, 6)$	$8 \rightsquigarrow 1$	$(1, 1)$	$8 \rightsquigarrow 11$	$(3, 1)$
$9 \rightsquigarrow 11$	$(3, 7)$	$12 \rightsquigarrow 1$	$(1, 2)$	$12 \rightsquigarrow 11$	$(3, 2)$

Table 2. All $u \rightsquigarrow v$ in $G_{\downarrow}$

Second, we show that, for a node w as a virtual center, all the the nodes that w can reach and the nodes that can reach w, can be represented as rectangles in the reachability map, M. We explain it below. Given a node $w \in G_{\downarrow}$. Suppose that $I_{\downarrow}(w) = ([s_1, e_1], [s_2, e_2], \cdots, [s_n, e_n])$ and $I_{\uparrow}(w) = ([s'_1, e'_1], [s'_2, e'_2], \cdots, [s'_m, e'_m])$. It is important to note that a pair $[s_i, e_i]$ in $I_{\downarrow}(w)$ indicates that the corresponding nodes in $[s_i, e_i]$ can be reached from w and a pair $[s'_j, e'_j]$ in $I_{\uparrow}(w)$ indicates that the corresponding nodes in $[s_i, e_i]$ can reach w. Therefore, all the possible pairs of $[s_i, e_j]$ and $[s'_j, e'_j]$ represent the reachability with w as the center.

We define a function $Rect(w)$ which maps the all reachability, with w as the virtual center, onto $n \times m$ rectangles in M, such as $((s_i, s'_j), (e_i, e'_j))$ for every $1 \leq i \leq n$ and $1 \leq j \leq m$. Note: a rectangle being contained in another can be eliminated. Two adjacent rectangles can be merged into a single rectangle.

The rectangular representation of the reachability of the nine nodes in $G_{\downarrow}$ (Fig. 4 (b)) are shown in Fig. 4. For example, consider node $w = 1$ in $G_{\downarrow}$. The cross in Fig. 4 (b) represents node $w = 1$ as $1 \rightsquigarrow 1$ at the grid $(x, y) = (1, 3)$ in M. Here, $I_{\downarrow}(1)$ has two intervals, $[s_1, e_1] = [1, 1]$ and $[s_2, e_2] = [3, 3]$, and $I_{\uparrow}(1)$ has an interval $[s'_1, e'_1] = [1, 5]$. The two rectangular representations become $((s_1, s'_1), (e_1, e'_1)) = ((1, 1), (1, 5))$ and $((s_2, s'_1), (e_2, e'_1)) = ((3, 1), (3, 5))$.

Third, we show that $Rect(w)$ represents a bipartite graph $B_C(V_C, E_C) \subseteq G_{\downarrow}$, which has w as its virtually center, in the reachability map, M. Recall: $V_C = V_{C_{in}} \cup V_{C_{out}}$, V_{in} (Eq. (2)) and V_{out} (Eq. (3)) can be computed as follows.

$$V_{C_{in}} = g_{\uparrow}(Rect(w)) \tag{7}$$

$$V_{C_{out}} = g_{\downarrow}(Rect(w)) \tag{8}$$

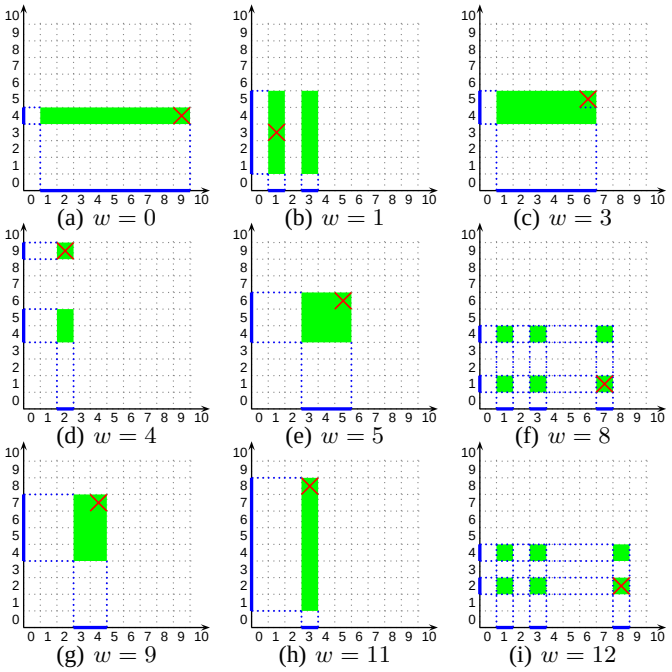


Fig. 4. Rectangular representations of bipartite graphs for nodes $w \in G_{\downarrow}$.

where $g_{\downarrow}$ and $g_{\uparrow}$ are functions that return a set of node identifiers represented as postorder numbers in the y-axis and x-axis. We explain the two functions, $g_{\downarrow}$ and $g_{\uparrow}$, using an example.

Reconsider node $w = 1$ again in $G_{\downarrow}$ (Fig. 2 (b)). $Rect(w)$ represents two rectangles, $((1,1),(1,5))$ and $((3,1),(3,5))$. $Rect(w)$ covers x-values in $X = \{1,3\}$ and y-values in $Y = \{1,2,3,4,5\}$. As shown in Table 1, $V_{out} = \{1,11\}$, because $1 = op_{\downarrow}^{-1}(1)$ and $11 = op_{\downarrow}^{-1}(3)$. In a similar fashion, $V_{in} = \{8,12,1,0,3\}$, because every value, $k \in V_{in}$, is obtained by a value $l \in Y$, such as $k = op_{\uparrow}(l)$. The corresponding bipartite graph is shown in Fig. 5.

Fourth, we show that we can compute densest bipartite graphs using rectangles. Let B_{C_1} and B_{C_2} be two bipartite graphs for nodes w_1 and w_2. We have the following three equations.

$$Rect(B_{C_1} \cap B_{C_2}) = Rect(B_{C_1}) \cap Rect(B_{C_2}) \tag{9}$$

$$Rect(B_{C_1} \cup B_{C_2}) = Rect(B_{C_1}) \cup Rect(B_{C_2}) \tag{10}$$

$$Rect(B_{C_1} - B_{C_2}) = Rect(B_{C_1}) - Rect(B_{C_2}) \tag{11}$$

The above equations state that the rectangle of union/intersection/difference of two bipartite graphs is the union/intersection/difference of the rectangles of the two bipartite graphs. Based on them, we can fast compute *SCI* using rectangles. We omit the proof, because it is trivial. An example is shown in Fig. 6. Here, B_{C_1} is mapped onto $((x_1,y_1),(x_2,y_2))$ by $Rect(B_{C_1})$, and B_{C_2} is mapped onto $((x_3,y_3),(x_4,y_4))$

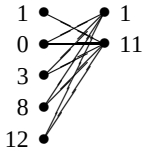


Fig. 5. A bipartite graph for $w = 1$ in $G_{\downarrow}$

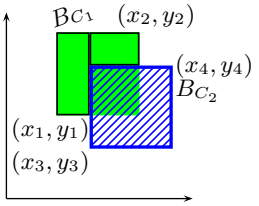


Fig. 6. $B_{C_1} - B_{C_2}$

by $Rect(B_{C_2})$. $Rec(B_{C_1} - B_{C_2})$ is the two rectangles: $((x_1, y_1), (x_3 - 1, y_2))$ and $((x_3, y_4 + 1), (x_2, y_2))$.

4.4 The Algorithm

We discuss our new fast 2-hop algorithm, called *MaxCardinality-G*, because it can result in the same set of 2-hop cover as *MaxCardinality*. The efficiency of *MaxCardinality-G* is achieved due to the introduction of reachability map and the operations over rectangles (Eq. (9), (Eq. (10) and (Eq. (11)). We do not need to compute bipartite graphs, B_C, with a node w as its virtual center, and we do not need to compute sets. Instead, we use $I_{\downarrow}$ and $I_{\uparrow}$ to obtain B_C, and use rectangles to determine the densest subgraph based on *SCI*.

In *MaxCardinality-G* (Algorithm 2), it takes G as an input directed graph. It constructs a directed acyclic graph $G_{\downarrow}$ for G (line 1), and computes its reachability table and its reachability map (line 2). The 2-hop cover, $H_{\downarrow}$, will be obtained after line 12. In line 13, it computes a 2-hop cover for the given graph G based on the 2-hop cover, $H_{\downarrow}$, for $G_{\downarrow}$. The 2-hop cover H is returned in line 14. The main body of *MaxCardinality-G* is to compute the 2-hop cover $H_{\downarrow}$ for the directed acyclic graph $G_{\downarrow}$. For computing $H_{\downarrow}$, it initializes $H_{\downarrow}$ in line 3. Also, in line 4, it initializes Δ as empty which is used to maintain all the rectangles covered by the algorithm. A rectangle represents a bipartite subgraph in $G_{\downarrow}$. In line 6, it finds the densest bipartite subgraph, with node w as its center in $G_{\downarrow}$, in terms of Eq. (5), using operations (Eq. (9), (Eq. (10) and (Eq. (11)) upon its corresponding rectangles, $Rect(w)$, over the reachability map M. In line 6, it finds the largest area of $Rect(w) - \Delta$. Suppose the largest rectangle is for node w, in line 7-9, it add hops into $H_{\downarrow}$. Afterward, it adds the covered rectangles into Δ (line 10), and removes node w from the set of nodes $V_{\downarrow}$ (line 11).

We explain *MaxCardinality-G* using the directed acyclic graph example $G_{\downarrow}$ (Fig. 2) (b). Below, we show the details of the algorithm *MaxCardinality-G*, in comparison with its counterpart algorithm *MaxCardinality*. The 4 bipartite graphs, generated in the 4 iterations of the algorithm *MaxCardinality* are shown in Fig. 8, using Eq. (6). In the 1st iteration, it finds a bipartite graph with $w = 3$ as its virtual center (Fig. 8 (a)); in the 2nd iteration, it finds a bipartite graph with $w = 1$ as its virtual center (Fig. 8 (b)); in the 3rd iteration, it finds a bipartite graph with $w = 9$ as its virtual center (Fig. 8 (c)); and in the 4th iteration, it finds a bipartite graph with $w = 0$ as its virtual center (Fig. 8 (d)).

Recall the reachability map, which preserves the complete reachability information is given in Fig. 3. Therefore, the algorithm *MaxCardinality-G* needs to find all rectangles $Rect(w)$, for node w, that cover all the valid points in the reachability map. We

Algorithm 2 *MaxCardinality-G*

Input: a graph, $G(V, E)$
Output: a 2-hop cover, H

Construct a directed acyclic graph $G_\downarrow(V_\downarrow, E_\downarrow)$;
Compute the reachability table, and consider it as a virtual reachability map;
$H_\downarrow \leftarrow \emptyset$ {2-hop cover for $G_\downarrow$}
$\Delta \leftarrow \emptyset$; {covered rectangles}
while $V_\downarrow \neq \emptyset$ **do**
  let w be the node with the max area of $Rect(w) - \Delta$; {Densest subgraph in terms of *SCI*}
  let u and v be two nodes in $G_\downarrow$;
  for all $(x(w), y(u)) \in Rect(w)$ **do** $H_\downarrow \leftarrow H_\downarrow \cup \{(u \rightsquigarrow w, u)\}$;
  for all $(x(v), y(w)) \in Rect(w)$ **do** $H_\downarrow \leftarrow H_\downarrow \cup \{(w \rightsquigarrow v, v)\}$;
  $\Delta \leftarrow \Delta \cup (Rect(w) - \Delta)$;
  $V_\downarrow \leftarrow V_\downarrow \setminus \{w\}$;
end while
Compute H over $H_\downarrow$ for G;
return H;

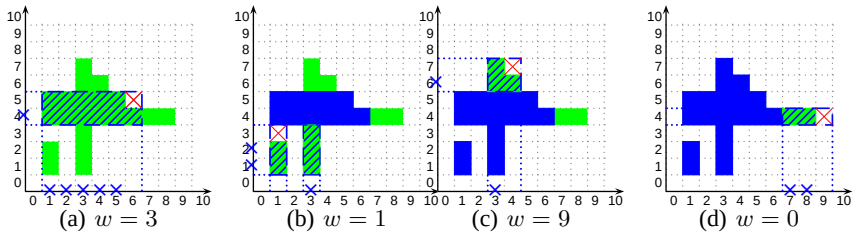


Fig. 7. *MaxCardinality-G* Steps for $G_\downarrow$

show how it is done using Fig. 7. In Fig. 7 (a), all the shaded points are the valid points; the cross point show the node $w = 3$, which is the same node selected in the 1st iteration of *MaxCardinality* (Fig. 8 (a)); and the striped points shows the largest area of $Rect(w)$, for $w = 3$, among all the other nodes. The $Rect(3)$ corresponds to Fig. 8 (a). After this step, the covered area, Δ, is shown as dark points in Fig. 8 (b)-(d)). In the second iteration, the algorithm *MaxCardinality-G* will select a node $w = 1$ which has largest area of $Rect(w) - \Delta$. As shown above, the algorithm *MaxCardinality-G* finds the exact bipartite graphs as the algorithm *MaxCardinality* but performs more efficiently, because it only needs to use operations against rectangles. The hops found in every iteration are given in Table 3.

We give implementation details for the algorithm *MaxCardinality-G*. The reachability table for the directed acyclic graph $G_\downarrow$ is maintained in memory. The rectangles for the covered areas, Δ, are maintained in a R-tree [6]. The area of a node w with $Rect(w) - \Delta$ is done as follows. 1) use $Rect(w)$ to retrieve all the areas in Δ that overlap with $Rect(w)$. 2) Suppose there are n rectangles, $R_1, \cdots, R_n$, returned. It does $Rect(w) - R_i$ for all $1 \leq i \leq n$. 3) The area of the $Rect(w) - \Delta$ can be computed.

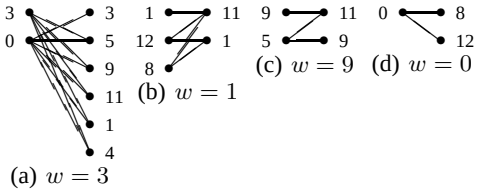


Fig. 8. *MaxCardinality* Steps for $G_{\downarrow}$

w	**2-hops**
3	$(0 \rightsquigarrow 3, 0)$,$(3 \rightsquigarrow 5, 5)$,$(3 \rightsquigarrow 9, 9)$ $(3 \rightsquigarrow 11, 11)$,$(3 \rightsquigarrow 1, 1)$,$(3 \rightsquigarrow 4, 4)$
1	$(12 \rightsquigarrow 1, 12)$,$(8 \rightsquigarrow 1, 8)$,$(1 \rightsquigarrow 11, 11)$
9	$((5 \rightsquigarrow 9), 5)$,$(9 \rightsquigarrow 11, 11)$
0	$(0 \rightsquigarrow 8, 8)$,$(0 \rightsquigarrow 12, 12)$

Table 3. 2-hops

5 Experimental Studies

We conducted extensive experimental studies to study the performance of the three algorithms, namely, the algorithm *MaxDSCovering*, *MaxCardinality*, and *MaxCardinality-G*. We have implemented all the algorithms using C++. In the following, denote them as D, C and C-G, respectively.

Both D and C compute set cover, for a graph $G(V, E)$, upon its transitive closure, T, whose size can be very large, in the worst case, $O(|V|^2)$. We compute the transitive closure using the algorithm [7], and precompute all bipartite graphs, $B_C(V_C, E_C)$ which has w as its center. All those precomputed bipartite graphs are stored in a B-tree on disk. For a given node $w \in G$, we can efficiently retrieve its corresponding bipartite graph B_C from disk through a simple buffering mechanism from the B-tree. For D and C, all the other data, except the transitive closure T, are maintained in main memory. We also implemented a variation for D and C by the procedure of DAG conversion, that is: 1) converting a directed graph into a directed acyclic graph, 2) finding 2-hop cover for the directed acyclic graph using D and C respectively, and 3) generating 2-hop cover for the directed graph using a simple post-processing step, based on the 2-hop obtained in step 2). We denote them as D* and C*, respectively. For C-G, we maintain data structures in main memory where possible including the reachability table and R-tree. We use Antonin Guttman's R-tree code[4]. We also implemented a ranking adopted from [19], which is used to reduce the cost for computing densest bipartite graphs in every iteration. Table 4 summarizes the processing involved in each algorithm.

Processing Involved	*MaxDSCovering*		*MaxCardinality*		
	D	**D***	**C**	**C***	**C-G**
Transitive Closure Computation	√	√	√	√	×
DAG Conversion	×	√	×	√	√
Geometry-based Approach	×	×	×	×	√

Table 4. Algorithms in Testing

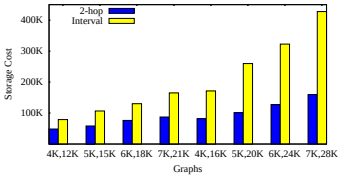
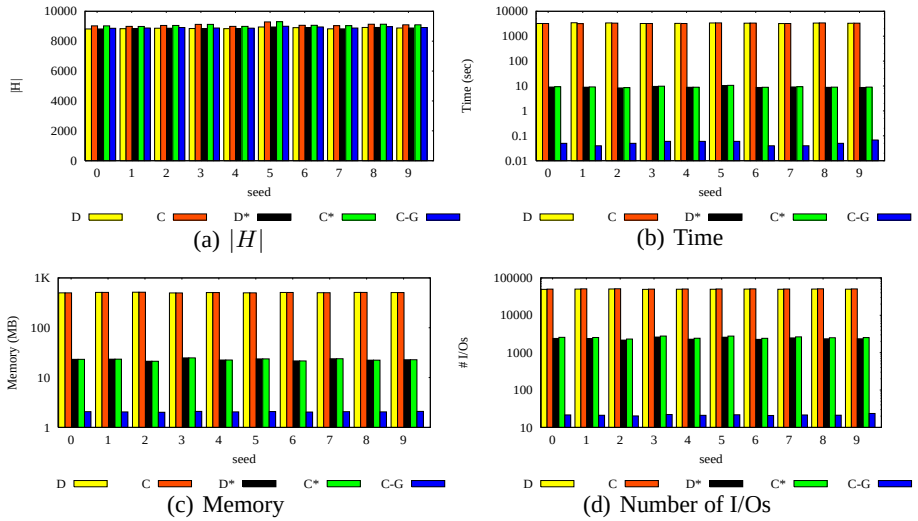


Fig. 9. Interval-Code vs 2-Hop Cover

We generated various synthetic data using two graph generator, namely, the random directed graph generator *GraphBase* developed by Knuth [14] and the random directed acyclic graph generator *DAG-Graph* developed by Johnsonbaugh [9]. We vary two parameters, $|V|$ and $E|$, used in the two generators, while the default values for the other parameters. We also tested several large real graph datasets.

[4] `http://web.archive.org/web/20020802233653/es.ucsc.edu/~tonig/rtrees`

We conducted all the experiments on a PC with a 3.4GHz processor and 2GB memory running Windows XP.



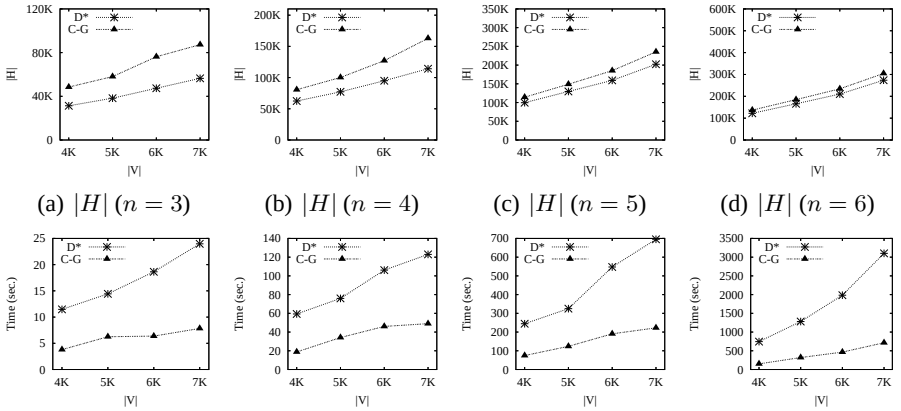
(a) $|H|$ (b) Time (c) Memory (d) Number of I/Os

Fig. 10. Compare of 5 Algorithms over Directed Graphs

5.1 Exp-1: Comparison of the Five Algorithms over Directed Graphs

Because the focus of this paper is to compute 2-hop cover for general directed graphs, we first generate 10 random directed graphs using *GraphBase*, where $|V| = 5,000$ and $|E| = 10,000$, with different seeds. We compare five algorithms, namely, two *SCII* algorithms and three *SCI* algorithms. Note: D and D* are a *SCII* solution, and C, C* and C-G are a *SCI* solution. We report the size of 2-hop cover, H, processing time (sec), memory consumption (MB), and the number of I/O accesses. Figure 10 shows the details for D, C, D*, C*, and C-G in that order, using 10 random directed graphs. In terms of quality, they all performed in a similar way. All algorithms achieved the similar size of 2-hop cover and hence the similar compression ratio. In terms of efficiency (CPU, Memory, I/O), D and C performed worst because they compute 2-hop cover for a directed graph by first computing transitive closure. D* and C* performed better because they compute 2-hop cover by first converting a directed graph into a smaller directed acyclic graph. The cost can be reduced because the cost of computing transitive closure is reduced, and less computational cost is needed for the 2-hop cover. C-G performed the best, and significantly outperformed the others, because it does not need to compute transitive closure and it computes the bipartite graphs using rectangles. Averagely, D uses as much time as 364 times of C*'s and 70,065 times of C-G's.

As expected, as shown in Fig. 10, the strategy of converting a directed graph onto a directed acyclic graph is beneficial. As a *SCII* solution, D* performed the best, and as a *SCI* solution, C-G performed the best. In the following, we focus on D* and C-G, and report our testing results using D* and C-G.



(a) $|H|$ ($n = 3$) (b) $|H|$ ($n = 4$) (c) $|H|$ ($n = 5$) (d) $|H|$ ($n = 6$)

(e) Time ($n = 3$) (f) Time ($n = 4$) (g) Time ($n = 5$) (h) Time ($n = 6$)

Fig. 11. Scalability Testing: Increase $|V|$ with Various $n = |E|/|V|$

5.2 Exp-2: Scalability Testing on Directed Acyclic Graphs

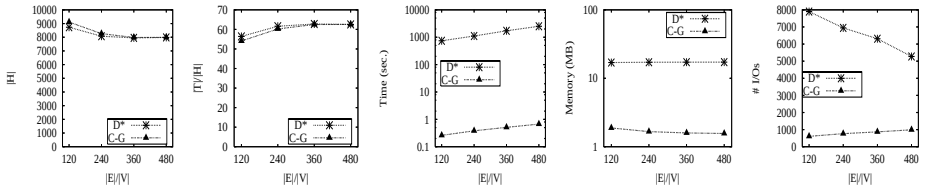
As discussed above, for increasing efficiency, a directed graph can be first converted onto a directed acyclic graph to compute 2-hop cover. In this testing, we focus on scalability testing, for D* and C-G, over directed acyclic graphs. We use the *DAG-Graph* generator to generate directed acyclic graphs, using various $|V|$ and $|E|$. We fix $n = |E|/|V|$ to be 3, 4, 5 and 6, and increase $|V|$ from 4,000 to 6,000. Such a setting is due to the fact that D* consumes much time to complete for larger graphs.

The results are shown in Fig. 11. In terms of quality (the size of 2-hop cover, H), D* marginally outperforms C-G. As shown in Fig. 11 (a-d), when $n = |E|/|V|$ increases from 3 to 6, the difference between C-G and D* becomes smaller in terms of the size of the 2-hop cover. As also confirmed in other testing, C-G and D* becomes very similar when the density of directed acyclic graphs becomes higher. In terms of efficiency, C-G significantly outperforms D*, in particular, when the density of a directed acyclic graph is high, e.g. $n = 6$ in this testing. It is worth noting that D* consumes more 2,387 sec. than C-G to gain a compression ratio larger than C-G by 2.12, about 0.539% of T.

In Fig. 9, we also compared the code size between the 2-hop labeling and the interval labeling [1] over directed acyclic graphs. The 8 directed acyclic graphs are labeled $|V|, |E|$ on the x-axis. Let $n = |E|/|V|$, the first four pairs are with $n = 3$, and the remaining pairs are with $n = 4$. We compare the size by the number of units where a unit can be an integer. Note for the interval code, 2 units for start and end numbers and 1 unit for postnumber. The 2-hop labeling outperforms interval labeling in all the 8 graphs. As the $n = |E|/|V|$ and $|V|$ increase, the size of the interval code increases significantly, while the size of 2-hop cover remains similar.

5.3 Exp-3: Test Dense Graphs

We test dense directed acyclic graphs using the *DAG-Graph* generator. We fix $|V| = 1,000$, and vary $|E|$, based on $|E| = n \cdot |V|$, where n is in range from 120 to 480.

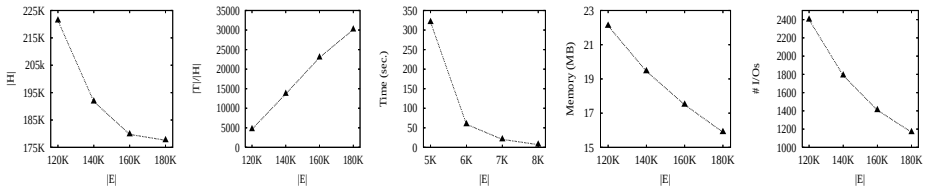


(a) $|H|$ (b) $|T|/|H|$ (c) Time (d) Memory (e) # of I/Os

Fig. 12. Dense Directed Acyclic Graph Testing: Vary $|E|$ while $|V| = 1,000$ is fixed

The results are shown in Fig. 12 where $n = |E|/|V|$ is shown in the x-axis. Note, let $|V| = 1,000$, $|E| = 480,000$ when $n = 480$. C-G significantly outperforms D* in terms of efficiency, and achieves the similar quality as D* does.

We also conducted experiments on C-G using large directed graphs. We fix $|V| = 100,000$ and vary $|E|$ from $120,000$ to $180,000$. The graphs are randomly generated by the *Graph-Base* generator [14]. The processing time decreases while $|E|$ increases, because the number of strongly connected components increases. When the number of strongly connected components is larger, the generated directed acyclic graph becomes smaller. Therefore, the processing time becomes smaller. For the fast one, we only use 6.99 sec. to compute the 2-hop cover for a directed graph with $10,000$ nodes and $180,000$ edges.



(a) $|H|$ (b) $|T|/|H|$ (c) Time (d) Memory (e) # of I/Os

Fig. 13. Large Dense Directed Graph Testing (C-G): $|V| = 100,000$

5.4 Exp-4: Real Graph Datasets Testing

We tested several real datasets including Ecoo157 used [1], a subset of DBLP[5], which consists of all the records for 5 international conferences, SIGMOD, VLDB, ICDE, EDBT and ICDT, until 2004, and two XMark benchmark datasets [20] using factor 0.1 and 0.2. We only show the results of C-G in Table 5, because the others consume too much resources to compute. For example, for XMark dataset with factor 0.2, denoted XMK.02, the number of nodes is is 336K, and the number of edges is 398K. It is a sparse graph, the compression rate achieves up to 3,565. The processing time is 3,600 seconds, and the memory consumption is 223MB at most, because C-G does not need to compute transitive closure, and uses rectangles. For the small real dataset Ecoo157 with 12,620 nodes and 17,308 edges, C-G only takes 0.36 seconds, and consumes 10MB memory.

[5] A snapshot of `http://dblp.uni-trier.de/xml/dblp.xml` in Mar/2004

Data Set	$\|V\|$	$\|E\|$	Time(sec.)	Mem.(MB)	# of I/Os	$\|H\|$	$\|T\|$	$\|T\|/\|H\|$
Ecoo157	12,620	17,308	0.36	9.83	237	23,913	2,402,260	100.46
DBLP	140,005	157,358	737.05	99.67	11,628	653,184	198,008,864	303.14
XMK.01	167,865	198,412	831.87	114.66	4,866	583,706	2,009,963,198	3,443.45
XMK.02	336,244	397,713	3,598.69	222.52	9,418	1,165,683	4,156,191,411	3,565.46

Table 5. Performance on real graphs

6 Related Work

Agrawal et al studied efficient management of transitive relationships in large databases [1]. The interval based labeling in [1] for directed acyclic graphs are reexamined for accessing graph, semistructured and XML data. Kameda [10] proposed a labeling scheme for reachability in planar directed graphs with one source and one sink. Cohen et al studied reachability labeling using 2-hop labels [2]. Schenkel et al [18, 19] studied 2-hop cover problem and proposed a divide-conquer approach. They attempted to divide a large graph into a set of even-partitioned smaller graphs, and solve the 2-hop cover problem for the large graph by post-processing the 2-hop covers for the small graphs. The work presented in this paper suggests that we can compute a large entire graph efficiently without the need to divide a graph into a large number of smaller graphs. Also, when there is a need to compute a large graph using the divide-conquer approach [18, 19], using our approach, it only needs to divide a graph into a rather small number of large graphs. In [22], we proposed a dual labeling scheme, in order to answer reachability queries in constant time for large sparse graphs. The work in [22] is different from the work presented in this paper. In this paper, we focus on computing 2-hops for arbitrary graphs which can be either sparse or dense. Several numbering schema were proposed for processing structural joins over tree structured data (XML data) including region-based [25, 24, 17, 12], prefix-based [3, 16, 11, 13, 21], and k-ary complete-tree-based [15, 23].

7 Conclusion

In this paper, we studied a novel geometry-based algorithm, called *MaxCardinality-G*, as a set cover I solution, to solve the 2-hop cover problem. Our algorithm utilizes an efficient interval based labeling for directed acyclic graphs, and builds up a reachability map which preserves all the reachability information in the directed graph. With the reachability map, our algorithm uses operations against rectangles to solve the 2-hop cover efficiently. As reported in our extensive experimental studies using synthetic datasets and large real datasets, our algorithm can compute 2-hop cover for large directed graphs, and achieve the similar 2-hop cover size as Cohen's algorithm can do.

Acknowledgment: The work described in this paper was supported by grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK418205).

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. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of SODA'02*, 2002.
3. E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *Proc. of PODS'02*, 2002.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2001.
5. G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.*, 18(1):30–55, 1989.
6. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of SIGMOD'84*, 1984.
7. Y. E. Ioannidis. On the computation of the transitive closure of relational operators. In *Proc. of VLDB'86*, 1986.
8. D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proc. of STOC'73*, 1973.
9. R. Johnsonbaugh and M. Kalin. A graph generation software package. In *Proc. of SIGCSE'91*, *(*`http://condor.depaul.edu/rjohnson/algorithm`*)*, 1991.
10. K. Kameda. On the vector representation of the reachability in planar directed graphs. *Information Processing Letters*, 3(3), 1975.
11. H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proc. of SODA'02*, 2002.
12. D. D. Kha, M. Yoshikawa, and S. Uemura. An XML indexing structure with relative region coordinate. In *Proc. of ICDE'01*, 2001.
13. W. E. Kimber. HyTime and SGML: Understanding the HyTime HYQ query language 1.1. Technical report, IBM Corporation, 1993.
14. D. E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing*. ACM Press, 1993.
15. Y. K. Lee, S. J. Yoo, and K. Yoon. Index structures for structured documents. In *Proc. of ACM First International Conference on Digital Libraries*, 1996.
16. S. Lei and G. . a. Z. M. ãzsoyoglu. A graph query language and its query processing. In *Proc. of ICDE'99*, 1999.
17. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of VLDB'01*, 2001.
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. I. Tatarnov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and quering ordered XML using a relational database system. In *Proc. of SIGMOD'02*, 2002.
22. 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.
23. W. Wang, H. Jiang, H. Lu, and J. Yu. Pbitree coding and efficient processing of containment join. In *Proc. of ICDE'03*, 2003.
24. M. YoshiKawa and T. Amagasa. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1), 2001.
25. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proc. of SIGMOD'01*, 2001.