On Incremental Maintenance of 2-hop Labeling of Graphs

Ramadhana Bramandia, Byron Choi, and Wee Keong Ng
School of Computer Engineering, Nanyang Technological University
Singapore
bramandia@pmail.ntu.edu.sg, kkchoi@ntu.edu.sg, askng@ntu.edu.sg

ABSTRACT

Recent interests on XML, Semantic Web, and Web ontology, among other topics, have sparked a renewed interest on graph-structured databases. A fundamental query on graphs is the reachability test of nodes. Recently, *2-hop* labeling has been proposed to index large collections of XML and/or graphs for efficient reachability tests. However, there has been few work on updates of *2-hop* labeling. This is compounded by the fact that Web data changes over time. In response to these, this paper studies the incremental maintenance of *2-hop* labeling. We identify the main reason for the inefficiency of updates of existing *2-hop* labels. We propose two updatable *2-hop* labelings, hybrids of *2-hop* labeling, and their incremental maintenance algorithms. The proposed *2-hop* labeling is derived from graph connectivities, as opposed to SET COVER which is used by *all* previous work. Our experimental evaluation illustrates the space efficiency and update performance of various kinds of *2-hop* labeling. The main conclusion is that there is a natural way to spare some index size for update performance in *2-hop* labeling.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; E.1 [**Data**]: Data Structure—*Graphs and Networks*

General Terms

Algorithms, Performance

Keywords

Reachability Test, Graph Indexing, 2-hop, Incremental Maintenance

1. INTRODUCTION

Recent interests on XML, Semantic Web, and Web ontology, among other topics, have sparked a renewed interest on graph-structured databases. There has been some work on large XML repositories [18], ontology data on the Web [11], graph networks [3], and classical graph databases with recursive query language support. A fundamental query on graphs is the reachability test. Specifically, given two nodes u and v of a graph, the test returns true if and only if v is reachable from u. This query evidently cannot be expressed by first order languages, *e.g.,* SQL. For all the reasons that reachability tests are important in classical graph databases, it is also useful to XML and the Semantic Web. In particular, the descendant axis, "//", in XPATH can be considered a special application of reachability tests. The descendant axis in XPATH determines a set of the nodes that are reachable from a set of input nodes (the context nodes). This can be implemented by simply extending the reachability test to support sets of nodes.

Another example of graphs is the Semantic Web. Resources [23] on the web can be naturally represented as a graph. We sketch an example of resources on the Web in Figure 1. One may want to ask: What resources/services are related/reachable to Resource A? In addition, reachability test can also be used in implementing OWL queries, a W3C recommendation for Semantic Web [24].

Various techniques have been proposed to implement reachability tests efficiently. On the one hand, reachability tests on a graph can be evaluated using a traversal of the entire graph. However, this method cannot handle data at Web-scale. On the another hand, one may precompute and materialize the transitive closure of the graph. Then, the reachability test becomes a simple selection on the transitive closure. However, the size of the transitive closure may be large, $O(|G|^2)$ in the worst case. Previous work on indexes for reachability tests has mainly focused on optimizing query performance and the size of the transitive closure, *e.g.,* [1].

Recently, a number of indexes for reachability tests have been proposed for optimizing the query performance and/or index size on trees (*e.g.,* [29]), DAGs (*e.g.,* [25]) or arbitrary graphs (*e.g.,* [21]). Web data is often cyclic. Thus, we focus on methods that support arbitrary graphs. This paper studies a popular indexing technique for reachability tests on arbitrary graphs called *2-hop labeling*, originally proposed by [9] and later studied in [20, 21, 7], among others. When data evolves, there is a need for maintenance of *2-hop* labeling. We study the incremental maintenance of such labeling, which receives little attention. For ease of presentation, we may use *2-hop* labeling and *2-hop* interchangeably.

Previous work on *2-hop* labeling has mainly focused on time-efficient index construction and optimization of the index size. However, determining the *2-hop* labeling with the minimum size is an NP-hard optimization problem [9]. To minimize the index size, *all* previous work used SET COVER as a heuristics for computing a minimal *2-hop* labeling of an input graph [9, 20, 21, 7]. Unfortunately, it is also known that the heuristic construction of *2-hop* labeling is computationally intensive. For example, [21] reported that the original algorithm [9] spends almost two days to construct the *2-hop* labels for a subset of the DBLP XML document – a bibliography repository for Computer Science publications. A divide-and-conquer approach [21] and a geometric approach [7] have been proposed to improve the performance of *2-hop* construction with a small tradeoff in index size.

Since the construction of *2-hop* labeling is costly, it is not feasible to rebuild the labels in response to each single update of the

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

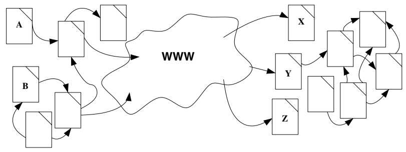


Figure 1: Semantic Web example

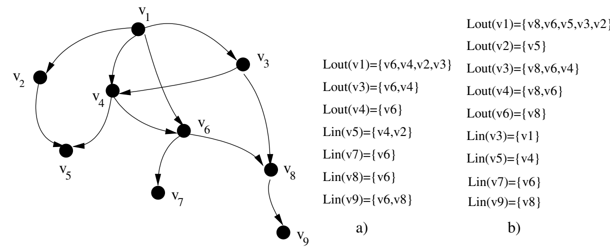


Figure 2: An example graph G_0 and two possible 2-hop covers

graph. What is desirable is an efficient *2-hop* label maintenance algorithm. To the best of our knowledge, the only work that studied incremental maintenance of *2-hop* labeling is [21]. [21] determines the elements in the transitive closure that are affected by an update (deletions or insertions). Then, a *2-hop* construction is applied to the affected elements. Since a single deletion (or insertion) of a graph may affect many elements in the transitive closure, the corresponding updates of its *2-hop* labeling is not trivial. In Section 4, we perform a case analysis of the affected elements of the deletion of a node and determine the bottleneck of deletions in *2-hop* labeling. (We skip the analysis on insertions since it is simple). Since the heuristics for the construction does not take update into considerations, the incremental maintenance of *2-hop* labeling can be inefficient.

Based on a case analysis on updates, we define a *node-separation* property of *2-hop* labels. When *2-hop* labeling satisfies this property, the deletion of *2-hop* labeling can be simplified, as the inefficient cases in deletion are no longer necessary. (In any case, insertions are simple.) In this paper, we propose a few heuristic functions, derived from cut vertex or minimum graph bisection, that produce *2-hop* labeling that satisfies the node-separation property.

The drawback of such heuristics is that it has a relatively remote relationship with the index size, when compared to the heuristics using SET COVER. As a consequence, the size of our *2-hop* labelings is relatively larger than those in [21, 7]. We derive some properties of our heuristics that facilitate many hybrids *2-hop* labeling of our and previously proposed *2-hop* labeling. We yield a family of updatable *2-hop* labeling, called *u2-hop* labeling, and the hybrids of *2-hop* labeling from these heuristics.

The main contributions of this paper are the followings:

- We illustrate inefficient cases in deletions of *2-hop* labeling and propose a simple algorithm for processing them;
- We present the node-separation property that leads to efficient *2-hop* maintenance. Based on this, we present two heuristic algorithms derived from graph connectivity for *2-hop* labeling construction and analyze their complexities. The size of our *2-hop* labels, however, is often larger than the previous proposed *2-hop* labels;
- We propose hybrids of *2-hop* labels from different heuristics;
- We propose a novel incremental maintenance algorithm for deletion that works very efficiently on our updatable *2-hop*s. In addition, the algorithm is extended to work on arbitrary *2-hop* labels. We also present an insertion algorithm that works on arbitrary *2-hop* labels;
- We conduct extensive experiments on updates of various versions of our *2-hop* labeling to verify the effectiveness of our heuristics and illustrate their performance characteristics.

Organization. The structure of the paper is as follows. Related work is discussed in Section 2. Section 3 briefly reviews *2-hop* labeling and other preliminaries of this work. Section 4 analyzes deletions of *2-hop* labeling. In Section 5, we present the definition of two updatable *2-hop* labeling due to graph connectivities and the hybrid of *2-hop* labeling. The construction and incremental maintenance algorithms of the updatable *2-hop* labeling are presented in Section 6. Section 7 presents an experimental study of the updatable *2-hop* labeling to illustrate its characteristics. We conclude and present future work in Section 8.

2. RELATED WORK

There has been a host of works on *2-hop* label construction with heuristics derived from SET COVER and its simplifications [7, 8, 20]. In Section 4, we illustrate that such constructions generate *2-hop* labels with small sizes but not optimized for update. As a consequence, previous maintenance algorithms [4, 21] for the *2-hop* labels required isolating the elements of the transitive closure affected by an update and applying a *2-hop* construction algorithm on the affected elements, which can often be large. In comparison, our heuristics are based on node-separation property that is optimized for update. Hence, incremental maintenance algorithms can be simpler than the previous ones.

Incremental maintenance of *2-hop* labeling has only been discussed in [21]. The labeling was constructed with graph partitioning. Each partition and its transitive closure fit into main memory and a heuristics, based on SET COVER, is re-used for the construction of *2-hop* labels for each partition. An objective of [21] is to scale the construction of *2-hop* labels. Since SET COVER is part of the heuristics proposed, the *2-hop* labels generated by this method have the same problems as the ones discussed above. In comparison, we study heuristics that produce update-efficient *2-hop* labels.

A number of techniques have been proposed to support reachability tests on trees, *e.g.,* [26, 29, 12]. While there have been studies on updates of the index proposed in [29], *e.g.,* [10], there is a lack of its extension on the support of arbitrary graphs. Recently, [26] has been extended to support DAGs [25, 22, 6]. However, there is no discussion on the extension of the update algorithm of [26] to DAGs. [22, 6] propose very efficient index construction algorithms. When there are a lot of updates, rebuilding the index in response to all of the updates may be more efficient than our incremental maintenance approach

It is worth-mentioning that there has been work in matching patterns in graphs [27, 28, 5]. The queries considered subsume reachability tests. Reachability tests can be considered as a primitive operation of pattern matchings.

There is another stream of work, *e.g.,* [16], on mining structures from Web graphs, where Web pages and hyperlinks are nodes and edges of a Web graph. Research on Internet computing has proposed methods to detect authorities (nodes with a large number of incident edges) and hubs (nodes with a large number of outgoing edges) from Web graphs. While authorities and hubs may imply a reasonable *2-hop* labels, it remains open whether there is a direct relationship between these structures and space-/update-efficient *2-hop* covers. There has also been work on graph clustering, in particular, clustering/mining evolving graphs [17]. However, there is a lack of a study on the cluster properties and reachability tests.

3. BACKGROUND ON 2-HOP LABELING

In this section, we provide some background on *2-hop* labeling and show how reachability test is efficiently supported.

Since the reachability information of the nodes in a strongly connected component in a graph is trivial, we assume that each strongly connected component in the graph is reduced to a node. This can be efficiently done by Tarjan algorithm in one scan of the graph. The reduced graph is a directed acyclic graph (DAG). Our subsequent discussions always assume the reduced graph.

We denote a directed graph as $G(V, E)$. Each node v in V is associated with a *label* L, which are two lists of nodes $L_{in}(v)$ and $L_{out}(v)$. The two lists are called *2-hop labels*. The nodes that are stored in $L_{in}(v)$ (resp. $L_{out}(v)$) are some nodes that can reach (resp. are reachable from) v. We often refer to the nodes in either $L_{in}(v)$ or $L_{out}(v)$ as *center nodes*. Given two nodes u and v, v is reachable from u, denoted as $u \rightsquigarrow v$, if and only if $L_{in}(v) \cap L_{out}(u)$ is non-empty. To ensure that the *2-hop* labels contain all reachability information of G, the *2-hop* labels must cover all elements in the transitive closure $T(G)$ of G. The reflexive closure is implicitly encoded by $L_{in}(v)$ and $L_{out}(v)$. The *2-hop* labels that cover all elements in $T(G)$ is called *2-hop cover* $H(G)$ of G. Obviously, there are many correct *2-hop* covers of a graph. We may omit v from $L_{in}(v)$ and $L_{out}(v)$, and G from $T(G)$ and $H(G)$ when they are clear from the context.

Next, we illustrate how *2-hop* labeling works with an example. Consider the graph G_0 in Figure 2 and the nodes v_1 and v_9. We show one possible *2-hop* labels of v_1 and v_9 in Figure 2 (a): $L_{out}(v_1) = \{v_2, v_3, v_4, v_6\}$ and $L_{in}(v_9) = \{v_6, v_8\}$. The labels can be interpreted as follows: v_2, v_3, v_4 and v_6 are reachable from v_1; and v_9 is reachable from v_6 and v_8. $L_{out}(v_1) \cap L_{in}(v_9) = \{v_6\}$ means that there is a path from v_1 to v_9 via the center node v_6.

Previous work has mainly focused on minimizing the size of a *2-hop* cover, defined as $\sum_{v \in V} |L_{in}(v)| + |L_{out}(v)|$. In the original proposal of *2-hop* labeling, Cohen *et al.* [9] proved that finding the *2-hop* cover with the minimum size is an NP-hard problem. Various heuristics have been proposed to determine space-efficient *2-hop* cover iteratively. In particular, we briefly describe [9, 21, 7], which are essential to our discussion on updates.

In [9, 21, 7], a variable T' stores the uncovered elements in T. Initially, $T' = T$. Elements are iteratively removed from T' and heuristic algorithms terminate when T' is empty.

In [9], an undirected bipartite graph $G_w(A_w, D_w, E_w)$ is constructed for each node w. $u \in A_w$ and $v \in D_w$ and $(u, v) \in E_w$ if and only if (u, v) is in T and v is reachable from u via w. Then, the SET COVER heuristics finds an induced subgraph $G_i(A_i, D_i, E_i)$ of G_w with $r = \frac{|E_i \cap T'|}{|A_i \cup D_i|}$ maximized. This is exactly the problem of finding the *densest* subgraph of G_w. At each iteration of the algorithm, a node w having the largest r is picked as a center node. Then, we add w to L_{out} of nodes in A_i and L_{in} of nodes in D_i and remove (a, w) and (w, d), where $a \in A_i$ and $d \in D_i$, from T'.

While $\frac{|E_i \cap T'|}{|A_w \cup D_w|}$ returned space-efficient *2-hop* cover, the time and memory requirements for computing G_w are prohibitive. One of the results in [7] showed that the division in this heuristics has minor impact on the size of *2-hop* cover. Hence, [7] proposed a simpler heuristics where $|E_i \cap T'|$ is maximized, which leads to more efficient *2-hop* construction.

[21] proposed to (recursively) *partition* a graph into partitions, where each of the partition fits into main memory. A *2-hop* cover H^i of the intra-partition edges is constructed by using [9]. A supplement cover $\hat{H}$ is constructed for the interconnections between partitions – the skeleton graph. The *2-hop* cover proposed in [21] is the union of H^i's and $\hat{H}$.

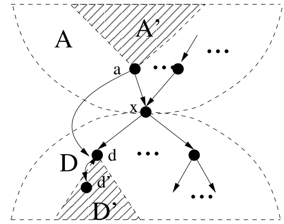


Figure 3: Illustration of deletion of x

Despite the first effort on incremental maintenance of *2-hop* labels by [21], to date, there has not been heuristics that considers updates of the *2-hop* labels *when they are constructed*. In the next section, we shall analyze updates of *2-hop* labeling and illustrate how maintenance of *2-hop* labels becomes *inherently* complicated if special efforts are not spent on the construction of *2-hop* labels.

4. ANALYSIS OF UPDATES OF 2-HOP LABELING

In this section, we perform a case analysis on the steps required to update *2-hop* labels after the deletion of a node of a graph. The aim is to highlight the inefficient steps among them. For *2-hop* labeling, insertions are simpler than deletions (see Section 6.2). Therefore, in this section, we focus on deletions.

Consider a deletion of a node $x \in V$. The nodes in G can be partitioned into three disjoint sets with respect to x (see Figure 3): (1) $A(x) = \{a \mid (a, x) \in T\}$; (2) $D(x) = \{d \mid (x, d) \in T\}$; and (3) $R(x) = V - A(x) - D(x)$. We omit x from $A(x)$, $D(x)$ and $R(x)$ when it is clear from the context. Since the updates of *2-hop* labels of D are symmetric to those of A, we shall discuss the updates of *2-hop* labels of nodes in A only, unless otherwise specified. An element (a, d) in T, where $d \in L_{out}(a)$, belongs to one of these four disjoint sets: (1) $E_1 = \{(a, d) \mid a, d \in A\}$; (2) $E_2 = \{(a, d) \mid a \in A, d \in R\}$; (3) $E_3 = \{(a, d) \mid a \in A, d \in D\}$; and (4) $E_4 = \{(a, x) \mid a \in A\}$.

Cases 1 and 2. When x is deleted, E_1 and E_2 are not affected.

Case 3. To illustrate the updates on E_3, we describe a procedure for processing the deletion of x. Consider $(a, d) \in E_3$ and $d \in L_{out}(a)$. We need to check if d should still be in $L_{out}(a)$ after the deletion of x. We check whether or not some of the children of a can reach d via some path(s) that do not pass through x. For all edges in E_3, this can be efficiently checked in a topological order of nodes in A, starting from x. If a can no longer reach d after the deletion of x, then d would be removed from $L_{out}(a)$ and we need to perform some additional check to the descendants of d. We do this by Procedure `check_All_Lin`: Consider a descendant d' of d where $d \in L_{in}(d')$, *i.e.*, d' uses d as a center node. For each such d', we perform `check_Lin`(a, d') as follows: We need to check if there are some paths from a to d' that do not pass through x. If there is such a path, then d' should be added into $L_{in}(a)$ to maintain the connectivity. Note that (a, d') may have been removed from the *2-hop* cover due to the removal of d from $L_{out}(a)$. In the worst case, we need to consider $|A| \times |D|$ `check_Lin` cases.

Case 4. We consider (a, x) and (x, d) for all $a \in A$ and $d \in D$ *together*. We define two sets: P: $\{p \mid x \in L_{out}(p), p \in A\}$ and Q: $\{q \mid x \in L_{in}(q), q \in D\}$, where x is the node to be deleted. For each $p \in P$ and $q \in Q$, we need to use `check_Lin` (in Case 3) to check if (p, q) is still in T after x is deleted. Hence, Case 4 requires at most $|P| \times |Q|$ `check_Lin`s. In addition, we would remove x from $L_{out}(p)$ and $L_{in}(q)$.

The bottleneck of deletion of *2-hop* labels is Case 3. Case 4 requires $|P|\times|Q|$ `check_Lins` in Case 3. Hence, simplifications on Case 3 have a significant impact on the overall performance.

5. UPDATABLE 2-HOP LABELING

Based on the analysis in Section 4, we present the definition of a family of updatable *2-hop* labeling (or simply *u2-hops*) that are derived from graph connectivities, in particular, cut vertex and minimum bisection. We present the node-separation property and merging property that lead to simplified deletions, specifically, for Case 3 and 4.

The heuristics of *u2-hops* are derived from the *node-separation* property: We say that a set of nodes X *separate* u and v if and only if u can reach v and the removal of all nodes in X disconnects u and v. We define the *center nodes of* u *and* v to be $\{x \mid x \in L_{out}(u) \cap L_{in}(v)\}$. A *2-hop* cover satisfies the node-separation property if and only if for each element (u, v) in T, the center nodes of u and v separate u and v.

When a *2-hop* cover satisfies the *node-separation* property, the processing of E_3 and E_4 can be simplified as follows: (1) `check_All_Lin` is no longer required for E_3; (2) x can simply be removed from $L_{out}(a)$ and $L_{in}(d)$ for E_4. There is no insertion of nodes into the *2-hop* labels required. These can be easily derived from the definition of the node-separation property.

Example 5.1: Consider the example graph G_0 shown in Figure 2 (a). The *2-hop* cover, as shown, does not exhibit the node-separation property because the centers node of v_1 and v_8 is $\{v_6\}$, which does not separate v_1 and v_8. After the removal of v_6, v_1 can still reach v_8 through v_3. Similarly, the center nodes of v_1 and v_9, and the center nodes of v_3 and v_8 do not satisfy the node-separation property.

After the deletion of v_6, the *2-hop* labels need to be updated by deleting v_6 from L_{in} and L_{out} of all nodes. In addition, it is necessary to insert v_3 into $L_{in}(v_8)$ and $L_{in}(v_9)$ to cover the paths from v_1 and v_3 to v_8 and v_9.

In comparison, suppose that v_8 is added to $L_{out}(v_1)$ and $L_{out}(v_3)$ and v_3 is added to $L_{in}(v_8)$. The resulting *2-hop* cover satisfies the node-separation property. For example, $\{v_3, v_6\}$ separates v_1 and v_8. In this case, the deletion of v_6 could be processed by simply removing v_6 from L_{in} and L_{out} of all nodes. □

Next, we discuss the merging property that is used in the construction of *u2-hops* that satisfies the node-separation property. Consider a possibly overlapping subsets of $T(G)$: $T_1, T_2, ..., T_m$, where each T_i represents partial connectivity of a graph G and $\bigcup_{i=1..m} T_i = T$. Each T_i is covered by the 2-hop labels H_i. Reachability query can be done by independently querying H_is. The *merging* property states that if T is covered by H_1, H_2..., H_m and each H_i satisfies the node-separation property, then we can merge H_i for $i = 1..m$ into a single *2-hop* H_{all} and H_{all} also satisfies the node-separation property. The merging is defined as follows: $L_{out}(a) = \bigcup_{i=1..m} L_{out}(a)$ of H_i. We can defined $L_{in}(a)$ in a similar manner. It is immediate that H_{all} is still a correct *2-hop* cover of T.

The correctness of this property can be easily derived from the fact that the center nodes of a and b in some H_is, $(a, b) \in T$, already separate a and b. Adding more nodes into the *2-hop* labels does not violate the node-separation property.

In the next subsection, we describe two heuristic functions that satisfy the node-separation property. First, we consider X as a singleton set – a cut vertex of a subgraph. Second, we consider X as a bisection cut in G.

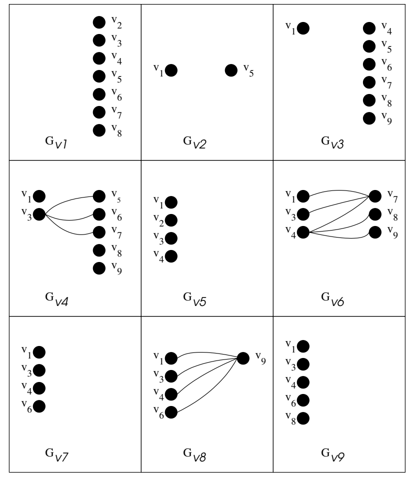


Figure 4: Bipartite graph G_x for each node $x \in G_0$

5.1 2-hop Cover with Cut Vertex

We first introduce the definition of a new *2-hop* cover, namely *u2-hop-A*, that is based on cut vertex. For each node $x \in V$, we construct a bipartite graph $G_x(A, D, E_x)$, where A and D are $A(x)$ and $D(x)$ respectively and the edges E_x are $\{(a, d) \mid x$ separates a and d, $a \in A$, $d \in D\}$.

Given a bipartite graph G_x, we are interested in finding $A' \subseteq A$ and $D' \subseteq D$ that x separates. From the definition of G_x, this is equivalent to finding A' and D' in which there is an edge (a, d), for all $a \in A'$ and $d \in D'$. Note that the induced subgraph of A' and D' is a biclique in G_x. Hence, our problem is equivalent to finding a biclique in G_x.

Consider a biclique $B_x(A', D')$. If we add x to $L_{out}(a)$ and $L_{in}(d)$, for all $a \in A'$ and $d \in D'$, to cover B_x, then x covers $|A'| + |D'| + |A'| \times |D'|$ elements of T.

u2-hop-A is constructed by iteratively finding biclique $B_x(A', D')$ and augmenting *u2-hop-A* to cover B_x's until T is fully covered. Similar to other heuristic algorithms, to minimize the index size, we find the node x whose biclique $B_x(A', D')$ maximizes $|A'| + |D'| + |A'| \times |D'|$ in each iteration. That is, we greedily maximize the number of elements of T that are covered. We remark that a node v can be chosen as a center node multiple number of times. This does not cause any problem due to the merging property discussed previously. More importantly, this guarantees that the *u2-hop-A* construction terminates and covers all elements of T.

It is straightforward that the *2-hop* cover constructed by this heuristics satisfies the node-separation property.

Example 5.2: Figure 4 shows the bipartite graph G_x constructed from each $x \in V_0$. Consider G_{v_4} in Figure 4. It shows the bipartite graph G_{v_4}. From the graph G_0 depicted in Figure 2, v_4 separates v_3 from v_5, v_6 and v_7. Hence, there is an edge from v_3 to v_5, v_6 and v_7 in G_{v_4}. Note that it is possible that G_x does not have any edge, *e.g.,* G_{v_2} in Figure 4. It is also possible that one side of the graph is empty, *e.g.,* G_{v_1} in Figure 4. The biclique $B_{v_8}(\{v_1, v_3, v_4, v_6\}, \{v_9\})$ covers $4 + 1 + 4 \times 1 = 9$ elements. This is the maximum among all possible bicliques in G_x for all x. Another biclique $B_{v_1}(\{\}, \{v_2, v_3, v_4, v_5, v_6, v_7, v_8\})$ only covers $7 + 0 + 7 \times 0 = 7$ elements. □

We find that given a bipartite graph G, finding a biclique B in G that covers the maximum number of elements in the transitive closure T is intractable. Specifically, we proved the following theorem. (Note that it is neither the maximal independent set problem of a bipartite graph nor the maximum edge biclique problem.)

Theorem 5.1: *[MSEBP] Given a bipartite graph $G(A, D, E)$, finding a biclique $B(B_X, B_Y)$ that maximizes $f(B) = |B_X| + |B_Y| + |B_X| \times |B_Y|$ is* NP*-complete, where $B_X \subseteq A$ and $B_Y \subseteq D$.* □

PROOF. *(Sketch)* Our reduction is established from an NP-complete problem, namely, the *maximum edge bipartite* problem (MEBP) [19]: Given a bipartite graph $G(A, D, E)$, MEBP finds a biclique $B(B_X, B_Y)$ having $g(B) = |B_X| \times |B_Y| \geq k$.

Note that for both problems, we only need to consider maximal biclique B, otherwise the biclique can be extended and produce larger value of $f(B)$ and $g(B)$.

Given an instance of MEBP on an input graph $G(A, D, E)$, we generate $|A| \times |D|$ instances of MSEBP. Specifically, for each pair of nodes $a \in A$ and $d \in D$, we generate an instance of MSEBP $\mathcal{G}_{a,d}$ as follows: We remove (1) a and d; (2) $d' \in D$ where d' is not adjacent to a, *e.g.,* there is no edge (a, d'); and (3) $a' \in A$ where a' is not adjacent to d, *e.g.,* there is no edge (a', d). The graph induced by the remaining nodes is an instance of MSEBP.

Consider any maximal biclique $B(A', D')$ in G having $g(A', D') = |A'| \times |D'| = k$. In every instance $\mathcal{G}_{a,d}$, there exists some nodes in B but not in $\mathcal{G}_{a,d}$. We refer to the subgraph of B in $\mathcal{G}_{a,d}$ as the reduced biclique, denoted as $B_{red}(A'_{red}, D'_{red})$. We show that 1) in all $\mathcal{G}_{a,d}$ generated, $f(B_{red}) \leq k - 1$ and 2) there are some $\mathcal{G}_{a,d}$ having $f(B_{red}) = k - 1$.

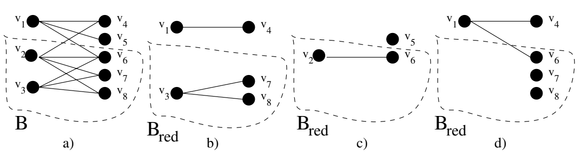


Figure 5: (a) The input graph for MEBP (b) An instance of MSEBP generated from v_2 and v_6 (c) An instance of MSEBP generated from v_1 and v_4 (d) An instance of MSEBP generated from v_2 and v_5

We first give a proof of 2). Consider an instance $\mathcal{G}_{a,d}$, where $a \in A'$ and $d \in D'$. In this instance, there is *exactly* one node in A' and D', respectively, that is not in $\mathcal{G}_{a,d}$, *e.g.,* $A'_{red} = A' \setminus \{a\}$ and $D'_{red} = D' \setminus \{d\}$. Hence, $f(B_{red}) = (|A'| - 1) + (|D'| - 1) + (|A'| - 1) \times (|D'| - 1) = |A'| \times |D'| - 1 = k$-1. This is illustrated in Figure 5. Consider the biclique B and the reduced biclique B_{red} in Figures 5 (a) and 5 (b). The values of $g(B)$ and $f(B_{red})$ are 6 and 5, respectively.

We then prove 1). In any $\mathcal{G}_{a,d}$, there is *at least* one node in A' and D', respectively, that is not in $\mathcal{G}_{a,d}$, *e.g.,* $|A'_{red}| \leq |A'| - 1$, $|D'_{red}| \leq |D'| - 1$. Hence, $f(B_{red}) \leq (|A'| - 1) + (|D'| - 1) + (|A'| - 1) \times (|D'| - 1) \leq |A'| \times |D'| - 1 \leq k$-1. This is illustrated in Figures 5 (c) and 5 (d), where the values of $f(B_{red})$ are 3 and 1.

Hence, the answer of MEBP is true if and only if there is at least one instance of MSEBP that contains a biclique B_{red} having $f(B_{red}) \geq k - 1$.

□

In response to this, we reuse an approximation algorithm [2] for MSEBP as the heuristics for *u2-hop-A* construction (Section 6).

5.2 2-hop Cover with Minimum Bisection

Next, we generalize *u2-hop-A* to *u2-hop-B* in this subsection. Specifically, as opposed to choosing a single-node separation, we use a node separation which may be a set of nodes. We were tempted to use min-cuts for construction. However, the construction algorithm may be guided by numerous small cuts and the resulting *2-hop* cover can be large. To reduce the number of cuts, we opt to use the minimum graph bisection. This leads to relatively smaller number of iterations and tend to produce smaller *2-hop* covers. While finding the minimum graph bisection is also a classical NP-hard optimization problem, there has been a number of heuristics for solving this problem [15]. In particular, we used [14] to determine a small bisection.

Suppose the bisection B, $B \subseteq E$, divides the graph G into $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, where $|G_1| \approx |G_2|$. We construct *2-hop* labels as follows. We cast B into an undirected bipartite graph. We determine the minimum vertex cover C of B. Since we are dealing with bipartite graph, the minimum vertex cover can be computed in PTIME using the network flow technique. Consider a node c in C. For each ancestor a of c, we insert c into $L_{out}(a)$. Similarly, for each descendant d of c, we add c into $L_{in}(d)$.

Next, we construct *u2-hop-B* recursively on G_1 and G_2, respectively, until the transitive closure is entirely covered. Due to the merging property, the *2-hop* labels obtained can be merged into a single *2-hop* cover.

Discussions. It is immediately true that the previous two heuristics generate *u2-hop* covers that satisfy the node-separation property. It should also be remarked that all center nodes in the *2-hop* cover are selected in a special way such that simple deletions for *u2-hop-A* and *u2-hop-B* become possible: *u2-hop-A* ensures that a node w that separates many (uncovered) node pairs are selected earlier than the others. In contrast, when there are multiple alternative paths P from u to v, *u2-hop-B* enforces (at least) one node on each alternative path in P is included in both $L_{out}(u)$ and $L_{in}(v)$. A deletion becomes simply the maintenance of the node separation between pairs of nodes.

5.3 Hybrid Updatable 2-hop Cover

The two *u2-hop*s introduced in the previous subsections have different properties. *u2-hop-A* requires the construction of a large number of bipartite graphs as in [9] that may be memory-bound and computationally intensive. In contrast, *u2-hop-B* may result in large *2-hop* covers as the bisection can often be relatively large when the input subgraphs is relatively small. The reason is that the minimum bisection of a small graph may often be a large subset of the edges of the graph. Therefore, we propose a hybrid approach of *u2-hop*s that takes advantages of both *u2-hop*s.

Recall the merging property discussed earlier. We can mix *u2-hop-A* with *u2-hop-B*. The hybrid of these *2-hop* covers still satisfies the node-separation property.

There are two simple alternatives for combining the *u2-hop*s. Firstly, we propose to first use *u2-hop-B* to build *2-hop* recursively until *u2-hop-B* becomes inefficient in terms of space. Then, we use *u2-hop-A* to cover the remaining elements in T. The inefficiency of *u2-hop-B* can be estimated as follows: In the worst case, the size of T is $|V|^2$. Suppose C is the vertex cover of the bisection cut of G_1 and G_2. The size of *u2-hop-B* of a given C, denoted as |*u2-hop-B*(C)|, can be estimated as $|V_1| \times |C| + |V_2| \times |C| + |V_1|^2 + |V_2|^2$. Through experimental studies on the size of T and *u2-hop-A* of random graphs, we can obtain the average size of *u2-hop-A* when compared to T, say |*u2-hop-A*| $\approx X\% \times |T|$. Hence, we use *u2-hop-B* recursively until |*u2-hop-B*(C)| $\geq X\% \times |T|$.

Secondly, we can use *u2-hop-B* recursively until the size of the graph is small enough that the graph together with its transitive closure and bipartite graphs can be stored in the main memory. Then, *u2-hop-A* is used.

Hybrid of updatable and arbitrary 2-hop labeling. Similarly, a hybrid of updatable and arbitrary *2-hop* labeling, not necessarily updatable, can be easily defined.

One scenario is to apply the *u2-hop-A* construction algorithm until the estimated compression ratio of the remaining uncovered elements in T is smaller than certain threshold. Then, the remaining elements are covered by any *2-hop* construction technique.

Since this hybrid *2-hop* labeling does not entirely satisfy the node-separation property, the merging property is not applicable here. However, we can store/maintain the two *2-hop*s separately.

As verified by our experiment, this hybrid *2-hop* labeling yielded a better index size when compared to *u2-hop*s. However, the construction of the smallest hybrid of *2-hop* is no easier than that of *u2-hop*s. For example, consider the hybrid of *u2-hop-A* and [9].

To avoid having numerous "small" center nodes in the resulting *2-hop* cover, we ignore *u2-hop-A* center nodes that connect only two nodes, where the node-separation property does not offer any advantage on deletions of E_3 and E_4. Then, we have the following.

Theorem 5.2: *Finding the hybrid of u2-hop-A and [9] with the minimum size is NP-hard.* □

Theorem 5.2 can be obtained by using the reduction from 3-SAT due to Cohen *et al.* [9]. The graph obtained from a 3-SAT instance is dense where an empty *u2-hop-A* is obtained. This graph is not modified and we need to find its minimum *2-hop* cover [9], which is NP-hard.

6. ALGORITHMS FOR U2-HOP LABELING

We have discussed the definition of a family of *u2-hop*s in the previous section. In this section, we describe the construction and maintenance algorithms for these *u2-hop*s.

6.1 Constructions for u2-hop Labeling

Construction of u2-hop-A. The key issue in constructing a reasonable *u2-hop-A* with a small size is to find a reasonable approximation for MSEBP for each bipartite graph of each node.

We associate a weight to an element of T. Weight 0 (resp. 1) means that the associated element in T has been covered (resp. has not been covered). Initially, all elements of T have not been covered and thus have a weight of 1. In addition, we also associate a weight to all nodes and edges in the bipartite graph $G_x(A, D, E)$. The weight of a node $a \in A$ in G_x is the weight of (a, x) in T. Similarly, the weight of a node $d \in D$ in $G_x(A, D, E)$ is the weight of (x, d) in T. Whereas, the weight of an edge (a, b) in E is the weight of (a, b) in T. The weights are updated in each iteration of the algorithm as T is updated.

Our heuristic function for each bipartite graph is to find a biclique with the maximum sum of weights. This problem can be solved by a 2-approximation algorithm given in [2], namely (2,2)-deletion problem. Note that this problem is a more general problem than MSEBP.

Putting these together, we present a greedy algorithm for *u2-hop-A* construction in Figure 6. The algorithm operates as follows. At Line 01, we construct the bipartite graph G_v for each $v \in G$. T' is used to record the uncovered elements in T. Initially, T' = T (Line 02). max stores the center node of B_{max} that would cover T the most. Initially, B_{max} is initialized to an empty biclique with weight 0 (Line 03). As long as T' is not fully covered, the iteration repeats. At each iteration, we compute an approximation of the biclique with the maximal weight of the bipartite graph for each node, as discussed above and in Section 5.1. Then, we determine the biclique B_{max} and the center node max with the largest weight among other bicliques (Lines 05-08). At the end of each iteration, we select max as a center node to cover connections between the nodes in V_{max_1} and V_{max_2} in B_{max} (Lines 09-10). We update T' and the weights of all bipartite graphs before the next iteration (Lines 11 and 12).

Procedure u2-hop-A-construction
Input: a directed graph G: (V, E)
Output: *u2-hop-A* of G: (L_{in}, L_{out})

01 for each $v \in V$ construct bipartite graph G_v
02 T' = T
03 $B_{max}(V_{max_1}, V_{max_2})$ = empty biclique; max = null
04 while T' is not empty
05 for each $v \in V$
06 construct max weight biclique $B_v(V_1, V_2)$ of G_v
07 if $\texttt{totalweight}(B_v) > \texttt{totalweight}(B_{max})$
08 $B_{max} = B_v$; $max = v$
09 $L_{out}(a)$ = $\{max\} \cup L_{out}(a)$, where $a \in V_{max_1}$
10 $L_{in}(d)$ = $\{max\} \cup L_{in}(d)$, where $d \in V_{max_2}$
11 update T' according to B_{max}
12 update the weights of G_v for $v \in V$ according to T'

Figure 6: A greedy algorithm for *u2-hop-A* construction

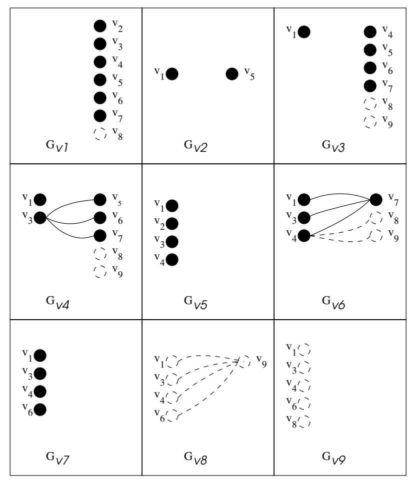


Figure 7: Updated weights of bipartite graph G_x for each node $x \in G_0$. Dotted edges and no-fill nodes are with weight 0. Solid edges and filled nodes are with weight 1.

Example 6.1: Consider G_0, as shown in Figure 2. The bipartite graphs G_v constructed in Line 1 is shown in Figure 4. All the nodes and edges have weight 1. As described in Example 5.2, in the first iteration, we obtain $max = v_8$, V_{max_1}=$\{v_1, v_3, v_4, v_6\}$ and V_{max_2}=$\{v_9\}$. Then, in Line 9 and 10, v_8 is added to L_{out} of v_1, v_3, v_4 and v_6 and L_{in} of v_9. Then, the weights of T' is updated. Subsequently, the weights of the bipartite graphs are updated. Figure 7 shows the updated weights after the first iteration. At the next iteration, the biclique having the maximal weight is ($\{v_1, v_3, v_4\}$,$\{v_7\}$) with center node v_6. Subsequent iterations would choose the following bicliques and their center nodes in this order: ($\{v_3\}, \{v_5\}$) with center v_4, ($\{v_1, v_2\}, \{\}$) with center v_5, ($\{v_1\}, \{\}$) with center v_3, ($\{v_1\}, \{\}$) with center v_2 and, finally, ($\{\}, \{v_3\}$) with center v_1. The resulting *2-hop* cover is shown in Figure 2 (b). □

Complexity. The initial construction of a bipartite graph is as costly as computing the transitive closure, $O(|V|\times(|V| + |E|))$. To compute all bipartite graphs, it takes $O(|V|^2\times(|V| + |E|))$. The approximation algorithm that we used takes $O(|V+|E|)$. We adopted the priority queue implementation to optimize the for-loop (Lines

Procedure `u2-hop-B-construction`
Input: a directed graph G: (V, E),
T': the uncovered elements of T
Output: *u2-hop-B* of G: (L_{in}, L_{out})

01 B = `min_bisection`(G)
02 denote the subgraph separated by B as G_1 and G_2
03 C = `min_vertex_cover`(B)
04 for each c in C
05 for v in V
06 if $v \rightsquigarrow c$ then $L_{out}(v) = L_{out}(v) \cup \{c\}$
07 else if $c \rightsquigarrow v$ then $L_{in}(v) = L_{in}(v) \cup \{c\}$
08 update T' according to Lines 06-07
09 if T' is not empty
10 `u2-hop-B-construction`(G_1, T')
11 `u2-hop-B-construction`(G_2, T')

Figure 8: A greedy algorithm for *u2-hop-B* construction

05-08) in *u2-hop-A* construction [20]. Overall, the dominating step in the construction is Line 01.

Construction of u2-hop-B. The construction of *u2-hop-B* also uses T' to keep track of the uncovered elements in T. We assume that $T' = T$ initially. We use [14] to compute the minimum bisection B of G (Line 01). Then, we use a classical algorithm to determine the minimum vertex cover C of B (Line 03). Next, we add *2-hop* labels as discussed in Section 5.2 (Lines 04-07). We updated T' according to the elements of T covered by C (Line 08). If T is not entirely covered, the construction procedure is called recursively with the subgraphs defined by B, G_1 and G_2 (Lines 09-11).

Complexity. In Lines 01-08, the dominating step is the approximation algorithm for finding the minimum bisection of a graph [14]. Denote M to be the time complexity for the algorithm. The construction is called at most $|V|$ times in the worst-case. The overall complexity is $O(M \times |V|)$. However, note that M depends on the size of the input graph, which decreases as the recursion proceeds.

6.2 Incremental Maintenance Algorithms for u2-hop Labeling

Next, we describe our incremental maintenance algorithms for *u2-hop* labeling.

Deletions for u2-hops. The deletion algorithm of *u2-hop*s is presented in Algorithm `delete` in Figure 9. The inputs of Algorithm `delete` are a directed graph G, its *u2-hop* cover H and a node to-be-deleted x. The output of the algorithm is the updated *2-hop* cover that still satisfies the node-separation property. We remove x from $L_{in}(v)$ and $L_{out}(v)$ for all $v \in V$ (Line 01). This deals with Case 4 in Section 4. We obtain the ancestors and descendants of x with the help of the input graph G (Line 02). We sort the ancestors and descendants based on the topological ordering (Line 03-04). Next, we gather the edges that belong to Case 3 in E_3 (Line 05-06). We perform deletions of edges in Case 3 in topological order. We sort the edges of E_3 by their indexes in A' and D' (Line 07). Hence, when we process an edge (a, d) in E_3, the relevant *2-hop* labels have been correctly updated. Then, we scan through the edges in E_3', the sorted E_3 (Line 08). For each edge (a,d) in E_3', there are only two possible cases: (i) d belongs to $L_{out}(a)$ or (ii) a belongs to $L_{in}(d)$ (Lines 09 and 12). (i) For the first case, we check if a can still reach d after the deletion of x. The checking is done by using the *2-hop* cover to test if any child c_a of a can reach d. Specifically, $L_{out}(c_a) \cap L_{in}(d) \neq \{\}$. If no child of a can reach d, we remove d from $L_{out}(a)$ (Lines 10-11). (ii) For the second case, we remove a from $L_{in}(d)$ if a cannot reach d after the deletion of x (Lines 13-14). The checking is done by utilizing the parents of d in a similar manner. We emphasize that the *2-hop* cover can be used for reachability tests in Lines 10 and 13 because the *2-hop* cover has been correctly updated in previous steps.

Procedure `delete`
Input: a directed graph G, an *u2-hop* cover of G and
x a vertex to-be-deleted
Output: the updated *u2-hop* cover

01 remove x from L_{in} and L_{out} of all nodes
02 compute A: $A(x)$ and D: $D(x)$
03 A' := sort A in reverse topological order ("bottom up")
04 D' := sort D in topological order ("top down")
05 E_3 := (a, d_a), $d_a \in L_{out}(a)$, $d_a \in D'$ and $a \in A'$
06 $E_3 \cup$ = (a_d, d), $a_d \in L_{in}(d)$, $a_d \in A'$ and $d \in D'$
07 E_3' := sort $E_3(a,d)$ by i then by j in ascending order
where $a = A'[i]$ and $d = D'[j]$
08 for each (a, d) in E_3' in order
09 if $d \in L_{out}(a)$
10 if $c_a \not\rightsquigarrow d$ for all children c_a of a (after deletion)
11 remove d from $L_{out}(a)$
12 if $a \in L_{in}(d)$
13 if $a \not\rightsquigarrow p_d$ for all parent p_d of d (after deletion)
14 remove a from $L_{in}(d)$

Figure 9: Deletion of *u2-hop* labeling

Example 6.2: Consider a deletion of v_6 in the graph G_0 presented in Figure 2. We use the *2-hop* cover depicted in Figure 2 (b). Note that the *2-hop* cover satisfies the node-separation property. The deletion algorithm removes v_6 from L_{out} of v_1, v_3 and v_4 as well as $L_{in}(v_7)$ (Line 01). The set of ancestors and descendants are obtained: A=$\{v_1, v_3, v_4\}$ and D=$\{v_7, v_8, v_9\}$. The topologically sorted representation is as follows: A'=$\{v_4, v_3, v_1\}$ and D'=$\{v_7, v_8, v_9\}$. The set of edges in Case 3, E_3 is $\{(v_1, v_8), (v_3, v_8), (v_4, v_8)\}$. The sorted E_3, E_3', is $\{(v_4, v_8), (v_3, v_8), (v_1, v_8)\}$. Note that all of these edges will be processed by Lines 10-11. Next, we process E_3' in sequence: 1) process (v_4, v_8): we check if v_4 can still reach v_8 after deleting v_6. The only remaining child of v_4 is v_5. Since $L_{out}(v_5) \cap L_{in}(v_8)$ is empty, v_5 can not reach v_8, thus we remove v_8 from $L_{out}(v_4)$. 2) process (v_3, v_8): since v_8 is a child of v_3, then we keep v_8 in $L_{out}(v_3)$ (a node is implied in the L_{in} and L_{out} of the node itself). 3) process (v_1, v_8): v_1 has three children, v_2, v_3 and v_4. v_2 cannot reach v_8. But, as we have processed in Step 2), v_3 can reach v_8. Thus, we keep v_8 in $L_{out}(v_1)$. □

We also remark that Algorithm `delete` can be extended to handle deletions of arbitrary *2-hop* covers, including those proposed by [21, 7]. This can be implemented with `check_All_Lin`, as described in Section 4. Specifically, if d is to be removed from $L_{out}(a)$ (Lines 11 and 14), then we check all $d' \in D$ having $d \in L_{in}(d')$. Thus, if a can still reach d' after the deletion of x (again, this checking is done through *2-hop* reachability test), then we need to add d' to $L_{out}(a)$ to restore back the reachability information. These additional operations must be performed consistently to the edge order in E_3'. As an optimization, if (a,d') is already covered by the current *2-hop* labels after the removal of d from $L_{out}(a)$, the previous steps can be skipped. Similarly, if a is to be removed from $L_{in}(d)$ (Line 14), a symmetric processing is needed.

From the discussion above, it is clear that the deletion algorithm can also be extended to work on the hybrid of *u2-hop* and non-*u2-hop*, *e.g.,* the hybrids of *2-hop*s described in Section 5.3, and has

Procedure `insert`
Input: a directed graph G, an *u2-hop* cover,
an edge to-be-inserted (x, y)
Output: the updated *u2-hop* cover
01 case 1: $x \in V$ and $y \notin V$
02 $\quad L_{in}(y) := L_{in}(x) \cup \{x\}$
03 case 2: $x \notin V$ and $y \in V$
04 $\quad L_{out}(x) := L_{out}(y) \cup \{y\}$
05 case 3: $x, y \in V$ and $(x, y) \notin E$
06 $\quad$ compute A: $A(x)$ and D: $D(y)$
07 $\quad c_x := |\{a | x \notin L_{out}(a), a \in A\} \cup \{d | x \notin L_{in}(d), d \in D\}|$
08 $\quad c_y := |\{a | y \notin L_{out}(a), a \in A\} \cup \{d | y \notin L_{in}(d), d \in D\}|$
09 $\quad$ if $c_x < c_y$ then $t = x$, else $t := y$
10 $\quad$ for a in A
11 $\quad\quad L_{out}(a) := L_{out}(a) \cup \{t\}$
12 $\quad$ for d in D
13 $\quad\quad L_{in}(d) = L_{in}(d) \cup \{t\}$

Figure 10: Insertion of *u2-hop* labeling

been used in our experiment. To extend the algorithm to work on hybrids of *2-hop* that do not satisfy the node-separation property, the two *2-hop* covers are to be updated in parallel.

Complexity. Sorting the nodes or edges in some topological order can be implemented efficiently. The dominating steps in Algorithm `delete` are Lines 8-14. For each edge in E_3, we performed at most $O(|V|)$ *2-hop* lookups, Lines 10 and 13. Hence, the complexity of the deletion is $O(|E_3| \times |V|)$ *2-hop* lookups. However, in practice, the number of lookups required is much fewer than this.

Insertions for u2-hops. Algorithm `insert`, as shown in Figure 10, handles insertions of *u2-hop*s. We aim at an insertion algorithm that preserves the node-separation property. For simplicity, we assume that the insertion would not introduce cycles to the graph. Consider a single-edge insertion (x, y). Suppose x already exists in the graph and y is new (Line 01). All nodes that can reach x can also reach y. Hence, we put $L_{in}(x)$ together with x in $L_{in}(y)$ (Line 02). Case 2 (Lines 03-04) is symmetric to Case 1.

Case 3 deals with insertions of an edge between two existing nodes. The insertion can be processed by adding either x or y to i) L_{out} of of ancestors of x and 2) L_{in} of descendants of y. It is easy to see that this procedure would preserve the node-separation property. Among the two choices (x and y), we pick the one that minimizes the increase in the size of the updated *2-hop* labels. Based on these, the algorithm proceeds as follows. A and D are the ancestors of x and the descendants of y respectively (Line 06). Next, the increase of index size with respect to x and y is computed (Lines 07-08). The smaller of the two is chosen (Line 09) and finally L_{out} of A and L_{in} of D are updated accordingly (Lines 10-13).

Complexity. The dominating step is Lines 07-08 and Lines 10-13. In the worst case, A and D comprise of all nodes in the graph. The complexity is $O(|V|)$ *2-hop* lookup and update.

Insertion of a subgraph. We end this section with a discussion on the insertion of a subgraph g into an existing graph G. This can be implemented by using Algorithm `insert`. First, we build the *2-hop* cover of the induced subgraph of the new nodes in g. Second, we handle the insertion of crossing edges between the *2-hop* covers of g and G which can be handled by Case 3 of Algorithm `insert`.

7. EXPERIMENTAL RESULTS

Our experimental evaluation focused on the effects of graph size ($|G|$) and edge to vertex ratio ($|E|/|V|$), as a measure of graph density, on the index size and update performance of various versions of *2-hop* labeling and the effectiveness of the proposed updatable *2-hop* labeling.

We used the *2-hop* labeling of [21] (denoted as SC-II) and [7] (denoted as SC-I) as well as their respective deletion algorithm implemented by [4]. *u2-hop-A*, *u2-hop-B* and the hybrid of the two are denoted as UH-A, UH-B and UH-B-A, respectively. We tried UH-B-A on a large number of random graphs and selected a constant X for switching between UH-B and UH-A. The hybrid of *u2-hop-A* and SC is H-A-SC. For H-A-SC, we switched from UH-A to SC-I when UH-A did not offer more than 10% compression. All these labelings have been implemented in C++. The experiments were run on a system with a 3.4GHz Pentium processor with 3G bytes of RAM running Windows XP operating system.

We use both synthetic directed acyclic graphs randomly generated by [13] as well as real-world graphs obtained from [3].

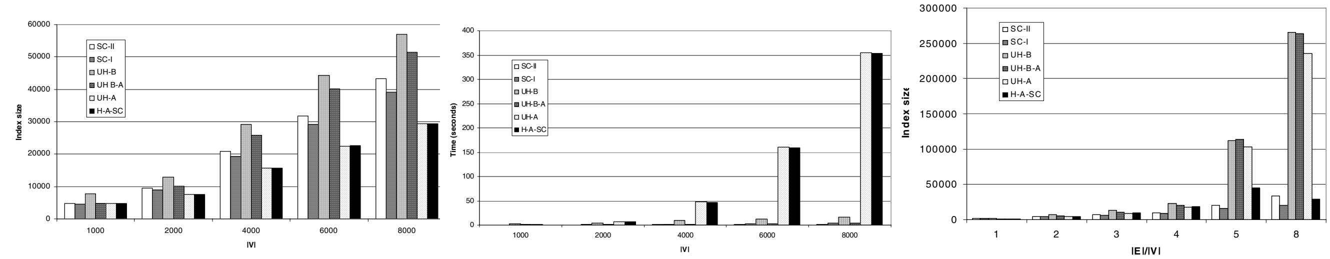
2-hop labeling construction. The first experiment studied the characteristics of the *2-hop* labeling. We set the edge to vertex ratio of the graphs to be 2 and varied the size of the synthetic graphs. The size of the *2-hop* covers of the graphs are reported in Figure 11 (a). It shows that UH-B and UH-B-A were consistently larger than SC-II, SC-I and H-A-SC. The reason is that the latter three uses SET COVER for index construction, which is optimized for index size. UH-B produced *2-hop* covers with the largest size. As expected, UH-B-A returned *2-hop* covers that were smaller than UH-B but larger than UH-A. UH-A and UH-B-A sometimes produced smaller indexes than SC-II, SC-I and H-A-SC. This depends on the structure of the graph, most notably the number of cut vertices. UH-A is small for these particular random data graphs. As verified by Table 2 and 4, UH-A is often larger than SC-II, SC-I and H-A-SC.

The runtime of the *2-hop* label constructions are reported in Figure 11 (b). Our construction algorithms were not as scalable as SC-II and SC-I. The construction increased more rapidly when compared to SC-II and SC-I. This is due to the computation of the initial bipartite graphs for UH-A and H-A-SC. UH-B and UH-B-A were comparable to SC-II and SC-I since they did not require building a large number of bipartite graphs.

Next, we set $|V|$ = 1000 and varied the density of the graphs. We observed that UH-A, UH-B and UH-B-A were more sensitive to graph density than the others. The reason for UH-A is that there were few cut vertices in a dense graph; for UH-B and UH-B-A, we noted that the bisections were large. H-A-SC remained efficient because, when only few cut vertex were found, it switched to SC-I.

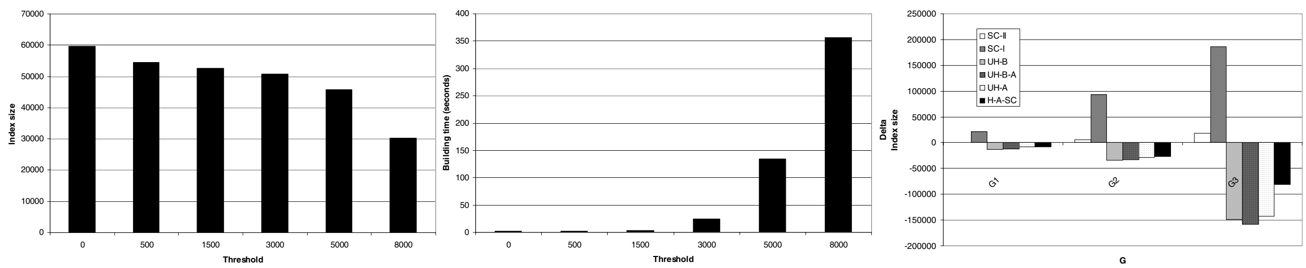
In the next experiment, we studied the impact of the threshold for switching from UH-B to UH-A in index construction. We set $|V|$ as a constant: $|V|$ = 8000. We present X as the number of nodes for switching. When X = 0 (*resp. 8000*) , the index is UH-B (*resp.* UH-A). The result is reported in Figure 12 (a). It shows that the index size decreased gradually as we increased X. Figure 12 (b) shows the construction time for UH-B-A as we varied X. As X increased, we computed more (and possibly large) bipartite graphs and the time increased rapidly.

Deletion performance. The next experiment verified the efficiency of Algorithm `delete`. The deletion algorithm for H-A-SC is the extended version of Algorithm `delete` as described previously. We generated three graphs G_1, G_2 and G_3 where $|V|$ was set to 4000 and their edge to vertex ratios were roughly 3, 4 and 6, respectively. The statistics of the *2-hop* covers constructed by different techniques is presented in Table 2. In order to observe the performance difference, we generated a long deletion sequence consisting of 100 random deletions and applied this workload to the three graphs. The total deletion times are reported in Table 1. The result shows that UH-B, UH-B-A, UH-A and H-A-SC outperformed SC-II and SC-I. For the large graph G_3, deletions on these *2-hop* covers



(a) Index size by varying the graph size (b) Construction time by varying the graph size (c) Index size by varying $|E|/|V|$

Figure 11: (a) and (b) Index performances with $|E|/|V|$=2; (c) Index construction with $|V|$=1000



(a) Index sizes by varying X (b) Index construction times by varying X (c) Change in index sizes after deletions

Figure 12: (a) and (b) Index performances by varying X; (c) Change in index size

Table 1: Total deletion time in seconds

Graph	SC-II	SC-I	UH-B	UH-B-A	UH-A	H-A-SC
G_1	71	51	37	35	28	33
G_2	367	197	104	101	94	109
G_3	8522	6703	406	404	380	146

Table 2: Initial Index size

Graph	SC-II	SC-I	UH-B	UH-B-A	UH-A	H-A-SC
G_1	52K	45K	111K	101K	69K	73K
G_2	92K	78K	404K	393K	359K	338K
G_3	208K	136K	1754K	1755K	1640K	457K

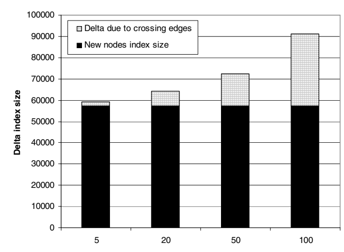


Figure 13: Insertion performance varying crossing edges

Table 3: Real graphs used in the experiment

Graph	$\|V\|$	$\|E\|$	$\|TC\|$
SMAGRI	1022	4892	73479
SCIMET	2188	6527	93637
YEAST	2361	6646	489039

could be more than one order of magnitude faster as no (partial) *2-hop* construction is needed. We noted that H-A-SC is the most efficient for G_3, although the extended algorithm performed more computation. The reason is that the sizes of UH-A, UH-B and UH-B-A are larger than that of H-A-SC. Even though H-A-SC required more steps for deletions, it operated on a small index.

A counter-intuitive fact about deletions is that previous deletion algorithms did not always reduce the size of the index. We reported the change of index size due to the deletion workload in Figure 12 (c). Note that UH-A, UH-B, UH-B-A and H-A-SC always return a smaller *2-hop* cover after deletions. In comparison, since *2-hop* construction was called in the deletions of SC-I and SC-II, the index size may increase after deletions. The decrease in H-A-SC is smaller than UH-A, UH-B and UH-B-A since only part of H-A-SC satisfied the node-separation property.

Insertion performance. The next experiment verified the insertion performance of UH-A, UH-B, UH-B-A and H-A-SC. Insertion of SC-II was briefly discussed in [21] and that of SC-I was not discussed in [7]. Thus, in this experiment, we skipped SC-II and SC-I. We used G_2 for this experiment. The newly inserted graph has 1000 nodes and 4000 edges. Then, we ranged the number of crossing edges, that connect the new graph to G_2, from 5 to 100. The insertion times is roughly 4 seconds among all workload and all *2-hop* labeling. This is because these *2-hop* labels used the same insertion algorithm. We then studied the impact of the number of crossing edges in the insertion workload. We presented the results in Figure 13. Since the number of nodes in the new graph is fixed, we reported the index size of the new graph and the index size due to the crossing edges separately. Figure 13 shows that the index size increased gradually with the number of crossing edges.

Experiment with real-world graphs. We have tested the deletion performance on three real-world graphs. The sizes of the graphs

Table 4: **Initial index size**

Graph	SC-II	SC-I	UH-A	H-A-SC
SMAGRI	7868	6450	45800	20166
SCIMET	13752	11696	41939	30849
YEAST	33452	16208	195934	115475

Table 5: **Total deletion time in seconds**

Graph	SC-II	SC-I	UH-A	H-A-SC
SMAGRI	58	37	13	13
SCIMET	32	23	16	19
YEAST	785	69	54	41

and their indexes are described in Table 3 and Table 4, respectively.

As before, we have randomly chosen 100 nodes and sequentially delete these nodes from the graph. The total time taken to perform this workload is presented in Table 5. The result shows the effectiveness of UH-A and H-A-SC. Both UH-A and H-A-SC were consistently faster than SC-II and SC-I. The index sizes of SC-II and SC-I were larger after deletion. For UH-A and H-A-SC, the updated index size was smaller. H-A-SC required more operations but operated on a smaller index.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed two heuristics based on cut vertex and minimum bisection for *2-hop* label construction. The *2-hop* covers constructed by such heuristics exhibit the node-separation property that lead to a simple incremental maintenance algorithm. We analyze deletions of existing *2-hop* labeling and proposed a simple deletion algorithm for handling such deletions. We have presented incremental maintenance algorithms for our *2-hop* labeling. Extensive experiments have been conducted to show the characteristics of various versions of *2-hop* labeling. The results showed that the incremental maintenance algorithms are efficient and the hybrid of our and existing *2-hop* can achieve both good update performance and small index size.

u2-hop-A construction is computationally intensive, for a similar reason presented in [9]. We have submitted a follow-up work on scalable *u2-hops* for publications.

Acknowledgements. We are grateful to Jiefeng Cheng for the implementation of *2-hop* labelings [21, 7] used in [4]. We thank anonymous referees for their invaluable comments.

9. REFERENCES

[1] R. Agrawal and H. V. Jagadish. Hybrid transitive closure algorithms. In *The VLDB Journal*, pages 326–334, 1990.

[2] R. Bar-Yehuda and D. Rawitz. Approximating element-weighted vertex deletion problems for the complete k-partite property. *J. Algorithms*, 42(1):20–40, 2002.

[3] V. Batagelj and A. Mrvar. Pajek datasets, 2006. http://vlado.fmf.uni-lj.si/pub/networks/data/.

[4] R. Bramandia, J. Cheng, B. Choi, and J. X. Yu. Optimizing updates of recursive xml views of relations. (Technical report at http://www.ntu.edu.sg/home/kkchoi/techreport.pdf).

[5] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.

[6] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, 2008.

[7] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[8] J. Cheng, J. X. Yu, and N. Tang. Fast reachability query processing. In *DASFAA*, pages 674–688, 2006.

[9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[10] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic xml trees. In *PODS*, pages 271–281, 2002.

[11] European Bioinformatics Institute. QuickGo: GO browser. Web interface. http://www.ebi.ac.uk/ego/.

[12] H. Jiang, H. Lu, W. Wang, and B. Ooi. Xr-tree: Indexing xml data for efficient structural join. In *ICDE*, 2003.

[13] R. Johnsonbaugh and M. Kalin. A graph generation software package. In *SIGCSE*, pages 151–154, 1991.

[14] Karypis lab. *Family of Multilevel Partitioning Algorithms*. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[15] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. In *Bell Systems Technical Journal*, pages 291–308, 1970.

[16] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[17] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[18] G. Miklau. UW XML repository. http://www.cs.washington.edu/research/xmldatasets/.

[19] R. Peeters. The maximum edge biclique problem is np-complete. *Discrete Appl. Math.*, 131(3):651–654, 2003.

[20] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *EDBT*, pages 237–255, 2004.

[21] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371, 2005.

[22] S. Triβl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.

[23] W3C. Resource description framework (rdf). http://www.w3.org/RDF.

[24] W3C. Semantic web activity. http://www.w3.org/2001/sw/.

[25] G. Wu, K. Zhang, C. Liu, and J.-Z. Li. Adapting prime number labeling scheme for directed acyclic graphs. In *DASFAA*, pages 787–796, 2006.

[26] X. Wu, M. L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *ICDE*, pages 66–78, 2004.

[27] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.*, 30(4):960–993, 2005.

[28] X. Yan, F. Zhu, P. S. Yu, and J. Han. Feature-based similarity search in graph structures. *ACM Trans. Database Syst.*, 31(4):1418–1453, 2006.

[29] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.