Efficiently Answering Reachability Queries on Very Large Directed Graphs

Ruoming Jin† Yang Xiang† Ning Ruan† Haixun Wang‡
† Department of Computer Science
Kent State University, Kent, OH, USA
{jin,yxiang,nruan}@cs.kent.edu
‡ IBM T.J. Watson Research
Hawthorne, NY, USA
haixun@us.ibm.com

ABSTRACT

Efficiently processing queries against very large graphs is an important research topic largely driven by emerging real world applications, as diverse as XML databases, GIS, web mining, social network analysis, ontologies, and bioinformatics. In particular, graph reachability has attracted a lot of research attention as reachability queries are not only common on graph databases, but they also serve as fundamental operations for many other graph queries. The main idea behind answering reachability queries in graphs is to build indices based on reachability labels. Essentially, each vertex in the graph is assigned with certain labels such that the reachability between any two vertices can be determined by their labels. Several approaches have been proposed for building these reachability labels; among them are interval labeling (tree cover) and 2-hop labeling. However, due to the large number of vertices in many real world graphs (some graphs can easily contain millions of vertices), the computational cost and (index) size of the labels using existing methods would prove too expensive to be practical. In this paper, we introduce a novel graph structure, referred to as *path-tree*, to help labeling very large graphs. The path-tree cover is a spanning subgraph of G in a tree shape. We demonstrate both analytically and empirically the effectiveness of our new approaches.

Categories and Subject Descriptors

H.2.8 [**Database management**]: Database Applications—*graph indexing and querying*

General Terms

Performance

Keywords

Graph indexing, Reachability queries, Transitive closure, Path-tree cover, Maximal directed spanning tree

1. INTRODUCTION

Ubiquitous graph data coupled with advances in graph analyzing techniques are pushing the database community to pay more attention to graph databases. Efficiently managing and answering queries against very large graphs is becoming an increasingly important research topic driven by many emerging real world applications: XML databases, GIS, web mining, social network analysis, ontologies, and bioinformatics, to name a few.

Among them, graph reachability queries have attracted a lot of research attention. Given two vertices u and v in a directed graph, a reachability query asks if there is a path from u to v. Graph reachability is one of the most common queries in a graph database. In many other applications where graphs are used as the basic data structure (e.g., XML data management), it is also one of the fundamental operations. Thus, efficient processing of reachability queries is a critical issue in graph databases.

1.1 Applications

Reachability queries are very important for many XML databases. Typical XML documents are tree structures. In such cases, the reachability query simply corresponds to ancestor-descendant search ("//"). However, with the widespread usage of ID and IDREF attributes, which represent relationships unaccounted for by a strict tree structure, it is sometimes more appropriate to represent the XML documents as directed graphs. Queries on such data often invoke a reachability query. For instance, in bibliographic data which contains a paper citation network, such as in Citeseer, we may ask if author A is influenced by paper B, which can be represented as a simple path expression `//B//A`. A typical way of processing this query is to obtain (possibly through some index on elements) elements A and B and then test if author A is reachable from paper B in the XML graph. Clearly, it is crucial to provide efficient support for reachability testing due to its importance for complex XML queries.

Querying ontologies is becoming increasingly important as many large domain ontologies are being constructed. One of the most well-known ontologies is the gene ontology (GO) [1]. GO can be represented as a directed acyclic graph (DAG) in which nodes are concepts (vocabulary terms) and edges are relationships (*is-a* or *part-of*). It provides a controlled vocabulary of terms to describe a gene product, e.g., proteins or RNA, in any organism. For instance, we may query if a certain protein is related to a certain biological process or

[1]http://www.geneontology.org

has a certain molecular function. In the simple case, this can be transformed into a reachability query on two vertices over the GO DAG. As a protein can directly associate with several vertices in the DAG, the entire query process may actually invoke several reachability queries.

Recent advances in system biology have amassed a large amount of graph data, e.g., various kinds of biological networks: gene regulatory, protein-protein interaction, signal transduction, metabolic, etc. Many databases are being constructed to maintain these data. Biology and bioinformatics are actually becoming a key driving force for graph databases. Here again, reachability is one of the fundamental queries frequently used. For instance, we may ask if one gene is (indirectly) regulated by another gene, or if there is a biological pathway between two proteins. Biological networks may soon reach sizes that require improvements to existing reachability query techniques.

1.2 Prior Work

In order to tell whether a vertex u can reach another vertex v in a directed graph $G = (V, E)$, we can use two "extreme" approaches. The first approach traverses the graph (by DFS or BFS), which will take $O(n + m)$ time, where $n = |V|$ (number of vertices) and $m = |E|$ (number of edges). This is apparently too slow for large graphs. The other approach precomputes the transitive closure of G, i.e., it records the reachability between any pair of vertices in advance. While this approach can answer reachability queries in $O(1)$ time, the computation of transitive closure has complexity of $O(mn)$ [14] and the storage cost is $O(n^2)$. Both are unacceptable for large graphs. Existing research has been trying to find good ways to reduce the precomputation time and storage cost with reasonable answering time.

A key idea which has been explored in existing research is to utilize simpler graph structures, such as chains or trees, in the original graph to compute and compress the transitive closure and/or help with reachability answering.

Chain Decomposition Approach. Chains are the first simple graph structure which has been studied in both graph theory and database literature to improve the efficiency of the transitive closure computation [14] and to compress the transitive closure matrix [11]. The basic idea of chain decomposition is as follows: the DAG is partitioned into several pair-wise disjoint chains (one vertex appears in one and only one chain). Each vertex in the graph is assigned a chain number and its sequence number in the chain. For any vertex v and any chain c, we record at most one vertex u such that u is the smallest vertex (in terms of u's sequence number) on chain c that is reachable from v. To tell if any vertex x reaches any vertex y, we only need to check if x reaches any vertex y' in y's chain and y' has a smaller sequence number than y.

Currently, Simon's algorithm [14], which uses chain decomposition to compute the transitive closure, has worst case complexity $O(k \cdot e_{red})$, where k is width (the total number of chains) of the chain decomposition and e_{red} is the number of edges in the transitive reduction of the DAG G (the transitive reduction of G is the smallest subgraph of G which has the same transitive closure as G, $e_{red} \leq e$). Jagadish *et al.* [11] applied chain decomposition to reduce the size of a transitive closure matrix. It finds the minimal number of chains from G by transforming the problem to an equivalent network flow problem, which can be solved in $O(n^3)$, where n is the number of vertices in DAG G. Several heuristic algorithms have been proposed to reduce the computational cost for chain decomposition.

Even though chain decomposition can help with compressing the transitive closure, its compression rate is limited by the fact that each node can have no more than one immediate successor. In many applications, even though the graphs are rather sparse, each node can have multiple immediate successors, and the chain decomposition can consider only at most one of them.

Tree Cover Approach. Instead of using chains, Agrawal *et al.* use a (spanning) tree to "cover" the graph and compress the transitive closure matrix. They show that the tree cover can beat the best chain decomposition [1]. The proposed algorithm finds the best tree cover that can maximally compress the transitive closure. The cost of such a procedure, however, is equivalent to computing the transitive closure.

The idea of tree cover is based on interval labeling. Given a tree, we assign each vertex a pair of numbers (an interval). If vertex u can reach vertex v, then the interval of u contains the interval of v. The interval can be obtained by performing a postorder traversal of the tree. Each vertex v is associated with an interval (i, j), where j is the postorder number of vertex v and i is the smallest postorder number among its descendants (each vertex is a descendant of itself).

Assume we have found a tree cover (a spanning tree) of the given DAG G, and vertices of G are indexed by their interval label. Then, for any vertex, we only need to remember those nodes that it can reach, but the reachability is not embodied by the interval labels. Thus, the transitive closure can be compressed. In other words, if u reaches the root of a subtree, then we only need to record the root vertex as the interval of any other vertex in the subtree is contained by that of the root vertex. To answer whether u can reach v, we will check if the interval of v is contained by any interval associated with the vertices we have recorded for u.

Other Variants of Tree Covers (Dual-Labeling, Label+SSPI, and GRIPP). Several recent studies try to address the deficiency of the tree cover approach by Agrawal *et al.* Wang *et al.* [16] develop the Dual-Labeling approach which tries to improve the query time and index size for the sparse graph, as the original tree cover would cost $O(n)$ and $O(n^2)$, respectively. For a very sparse graph, they claim the number of non-tree edges t is much smaller than n ($t << n$). Their approaches can reduce the index size to $O(n+t^2)$ and achieve constant query answering time. Their major idea is to build a transitive link matrix, which can be thought of as the transitive closure for the non-tree edges. Basically, each non-tree edge is represented as a vertex and a pair of them is linked if the starting of one edge v can be reached by the end of another edge u through the interval index (v is u's descendant in the tree cover). They develop approaches to utilize this matrix to answer the reachability query with constant time. In addition, the tree generated in dual-labeling is different from the optimal tree cover, as here the goal is to minimize the number of non-tree edges. This is essentially equivalent to the transitive reduction computation which has proved to be as costly as the transitive closure computation. Thus, their approach (including the transitive reduction) requires an additional $O(nm)$ construction time if non-tree edges should be minimized. Clearly, the major issue of this approach is that it depends heavily on the number of non-tree edges. If $t > n$ or $m_{red} \geq 2n$, this approach will not help with the computation of transitive closure, or compress the index size.

Label+SSPI [2] and GRIPP [15] aim to minimize the index construction time and index size. They achieve $O(m+n)$ index construction time and $O(m+n)$ index size. However, this is at the sacrifice of the query time, which will cost $O(m-n)$. Both algorithms start by extracting a tree cover. Label+SSPI utilizes pre- and post-order labeling for a spanning tree and an additional data structure for storing non-tree edges. GRIPP builds the cover using a depth-first search traversal, and each vertex which has multiple incoming edges will be duplicated accordingly in the tree cover. In some sense, their non-tree edges are recorded as non-tree vertex instances in the tree cover. To answer a query, both of them will deploy an online search over the index to see if u can reach v. GRIPP has developed a couple of heuristics which utilize the interval property to speed up the search process.

2-HOP Labeling. The 2-hop labeling method proposed by Cohen *et al.* [5] represents a quite different approach. Intuitively, it tries to identify a subset of vertices V_s in the graph which best capture the connectivity information of the DAG. Then, for each vertex v in the DAG, we record a list of vertices in V_s which can reach v, denoted as $L_{in}(v)$, and a list of vertices in V_s which v can reach, denoted as $L_{out}(v)$. These two sets record all the necessary information to infer the reachability of any pair of vertices u and v, i.e., if $u \rightarrow v$, then $L_{out}(v) \cap L_{in}(v) \neq \emptyset$, and vice versa. For a given labeling, the index size is $I = \sum_{v \in V} |L_{in}(v)| + |L_{out}(v)|$. They propose an approximate (greedy) algorithm based on set-covering which can produce a 2-hop cover with size no larger than the minimum possible 2-hop cover by a logarithmic factor. The minimum 2-hop cover is conjectured to be $\tilde{O}(nm^{1/2})$. However, their original algorithm will require computing the transitive closure first with an $O(n^4)$ time complexity to find the good 2-hop cover.

Recently, several approaches have been proposed to reduce the construction time of 2-hop. Schenkel *et al.* propose the HOPI algorithm, which applies a divide-and-conquer strategy to compute 2-hop labeling [13]. They reduce the 2-hop labeling complexity from $O(n^4)$ to $O(n^3)$, which is still very expensive for large graphs. Cheng *et al.* [3] propose a geometric-based algorithm to produce a 2-hop labeling. Their algorithm does not require the computation of transitive closure, but it does not produce the approximation bound of the labeling size which is produced by Cohen's approach.

[1] m is the number of edges and $O(n^3)$ if using Floyd-Warshall algorithm [6]

[2] k is the width of chain decomposition; Query time can be improved to $O(\log k)$ (assuming binary search) and construction time becomes $O(mn+n^2 \log n)$, which includes the cost of sorting.

[3] Query time can be improved to $O(\log n)$ and construction time becomes $O(mn + n^2 \log n)$.

[4] The index size is still a conjecture.

[5] It requires an additional $O(nm)$ construction time if the number of non-tree edges should be minimized.

	Query time	Construction time	Index size
Transitive Closure	$O(1)$	$O(nm)$[1]	$O(n^2)$
Opt. Chain Cover[2]	$O(k)$	$O(nm)$	$O(nk)$
Opt. Tree Cover [3]	$O(n)$	$O(nm)$	$O(n^2)$
2-Hop[4]	$\tilde{O}(m^{1/2})$	$O(n^4)$	$\tilde{O}(nm^{1/2})$
HOPI[4]	$\tilde{O}(m^{1/2})$	$O(n^3)$	$\tilde{O}(nm^{1/2})$
Dual Labeling[5]	$O(1)$	$O(n+m+t^3)$	$O(n+t^2)$
Labeling+SSPI	$O(m-n)$	$O(n+m)$	$O(n+m)$
GRIPP	$O(m-n)$	$O(n+m)$	$O(n+m)$

Table 1: Complexity comparison

1.3 Our Contribution

In Table 1 we show the indexing and querying complexity of different reachability approaches. Throughout the above comparison and several existing studies [15, 16, 13], we can see that even though the 2-hop approach is theoretically appealing, it is rather difficult to apply it on very large graphs due to its computational cost. At the same time, as most of the large graph is rather sparse, the tree-based approach seems to provide a good starting point to compress transitive closure and to answer reachability queries. Most of the recent studies try to improve different aspects of the tree-based approach [1, 16, 2, 15]. Since we can effectively transform any directed graph into a DAG by contracting strongly connected components into vertices and utilizing the DAG to answer the reachability query, we will only focus on DAG for the rest of the paper.

Our study is motivated by a list of challenging issues which tree-based approaches do not adequately address. First of all, the computational cost of finding a good tree cover can be expensive. For instance, it costs $O(mn)$ to extract a tree cover with Agrawal's optimal tree cover [1] and Wang's Dual-labeling tree [16]. Second, the tree cover cannot represent some common types of DAGs, for instance, the Grid type of DAG [13], where each vertex in the graph links to its right and upper corners. For a $k \times k$ grid, the tree cover can maximally cover half of the edges and the compressed transitive closure is almost as big as the original one. We believe the difficulty here is that the strict tree structures are too limited to express many different types of DAGs even when they are very sparse. From another perspective, most of the existing methods which utilize the tree cover are greatly affected by how many edges are left uncovered.

Driven by these challenges, in this paper, we propose a novel graph structure, referred to as *path-tree*, to cover a DAG. It creates a tree structure where each node in the tree represents a path in the original graph. This potentially doubles our capability to cover DAGs. Given that many real world graphs are very sparse, e.g., the number of edges is no more than 2 times of the number of vertices, the path-tree provides us a better tool to cover the DAG. In addition, we develop a labeling scheme where each label has only 3 elements in the path-tree to answer a reachability query. We show that a good path-tree cover can be constructed in $O(m + n \log n)$ time. Theoretically, we prove that the path-tree cover can always perform the compression of transitive closure better than or equal to the optimal tree cover approaches and chain decomposition approaches. Finally, we note that our approach can be combined with existing methods to handle non-path-tree edges. We have performed a detailed experimental evaluation on both real and synthetic datasets. Our results show that the path-tree cover can significantly reduce the transitive closure size and improve query answering time.

The rest of the paper is organized as follows. In Section 2, we introduce the path-tree concept and an algorithm to construct a path-tree from the DAG. In Section 3, we investigate several optimality questions related to path-tree cover. In Section 4, we present the experimental results. We conclude in Section 5.

2. PATH-TREE COVER FOR REACHABILITY QUERY

We propose to use a novel graph structure, *Path-Tree*, to cover a DAG G. The path-tree cover is a spanning subgraph of G in a tree shape. Under a labeling scheme we devise for the path-tree cover wherein each vertex is labeled with a 3-tuple, we can answer reachability queries in $O(1)$ time. We also show that a good path-tree cover can be extracted from G to help reduce the size of G's transitive closure.

Below, Section 2.1 defines notations used in this paper. Section 2.2 describes how to partition a DAG into paths. Using this partitioning, we define the *path-pair subgraph* of G and reveal a nice structure of this subgraph (Section 2.3). We then discuss how to extract a good *path-tree cover* from G (Section 2.4). We present the labeling schema for the path-tree cover in Section 2.5. Finally, we show how the path-tree cover can be applied to compress the transitive closure of G in Section 2.6.

2.1 Notations

Let $G = (V, E)$ be a directed acyclic graph (DAG), where $V = \{1, 2, \cdots, n\}$ is the vertex set, and $E \subseteq V \times V$ is the edge set. We use (v, w) to denote the edge from vertex v to vertex w, and we use $(v_0, v_1, \cdots, v_p)$ to denote a *path* from vertex v_0 to vertex v_p, where (v_i, v_{i+1}) is an edge $(0 \leq i \leq p-1)$. Because G is acyclic, all vertices in a path must be pairwise distinct. We say vertex v is reachable from vertex u (denoted as $u \rightarrow v$) if there is a path starting from u and ending at v.

For a vertex v, we refer to all edges that start from v as *outgoing edges* of v, and all edges ending at v as *incoming edges* of v. The *predecessor set* of vertex v, denoted as $S(v)$, is the set of all vertices that can reach v, and the *successor set* of vertex v, denoted as $R(v)$, is the set of all vertices that v can reach. The successor set of v is also called the *transitive closure* of v. The transitive closure of DAG G is the directed graph where there is a direct edge from each vertex v to any vertex in its successor set.

In addition, we say $G_s = (V_s, E_s)$ is a *subgraph* of $G = (V, E)$ if $V_s \subseteq V$ and $E_s \subseteq E \cap (V_s \times V_s)$. We denote G_s as a *spanning subgraph* of G if it covers all the vertices of G, i.e., $V_s = V$. A *tree* T is a special DAG where each vertex has only one incoming edge (except for the root vertex, which does not have any incoming edge). A *forest* (or *branching*) is a union of multiple trees. A forest can be converted into a tree by simply adding a virtual vertex with an edge to the roots of each individual tree. To simplify our discussion, we will use trees to refer to both trees and forests.

In this paper, we introduce a novel graph structure called *path-tree cover* (or simply *path-tree*). A path-tree cover for G, denoted as $G[T] = (V, E', T)$, is a spanning subgraph

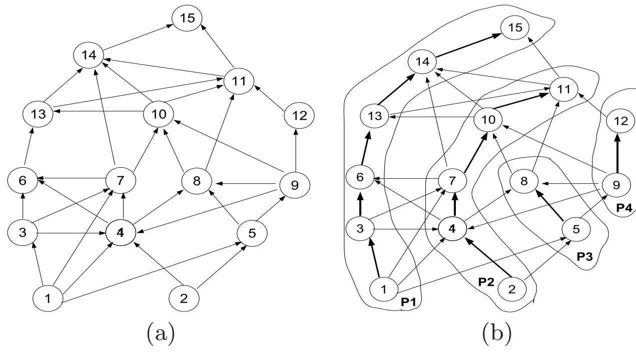

Figure 1: Path-Decomposition for a DAG

of G and has a tree-like shape which is described by tree $T = (V_T, E_T)$: Each vertex v of G is uniquely mapped to a single vertex in T, denoted as $f(v) \in V_T$, and each edge (u, v) in E' is uniquely mapped to either a single edge in T, $(f(u), f(v)) \in E_T$, or a single vertex in T.

2.2 Path-Decomposition of DAG

Let P_1, P_2 be two paths of G. We use $P_1 \cap P_2$ to denote the set of vertices that appear in both paths, and we use $P_1 \cup P_2$ to denote the set of vertices that appear in at least one of the two paths. We define graph partitions based on the above terminology.

DEFINITION 1. *Let $G = (V, E)$ be a DAG. We say a partition $P_1, \cdots, P_k$ of V is a path-decomposition of G if and only if $P_1 \cup \cdots \cup P_k = V$, and $P_i \cap P_j = \emptyset$ for any $i \neq j$. We also refer to k as the width of the decomposition.*

As an example, Figure 1(b) represents a partition of graph G in Figure 1(a). The path decomposition contains four paths $P_1 = \{1, 3, 6, 13, 14, 15\}$, $P_2 = \{2, 4, 7, 10, 11\}$, $P_3 = \{5, 8\}$ and $P_4 = \{9, 12\}$.

Based on the partition, we can identify each vertex v by a pair of IDs: (pid, sid), where pid is the ID of the path vertex v belongs to, and sid is v's relative order on that path. For instance, vertex 3 in G shown in Figure 1(b) is identified by (1, 2). For two vertices u and v in path P_i, we use $u \preceq v$ to denote u precedes v (or $u = v$) in path P_i:

$$u \preceq v \Longleftrightarrow u.sid \leq v.sid \text{ and } u, v \in P_i$$

NOTE: A simple path-decomposition algorithm is given by [14]. It can be described briefly as follows: first, we perform a topological sort of the DAG. Then, we extract paths from the DAG as follows. We find v, the smallest vertex (in the ascending order of the topological sort) in the graph and add it to the path. We then find v', such that v' is the smallest vertex in the graph such that there is an edge from v to v'. In other words, we repeatedly add the smallest nodes to the latest extracted vertex until the path could not be extended (the vertex added last has no out-going edges). Then, we remove the entire path (including the edges connecting to it) from the DAG and extract another path. The decomposition is complete when the DAG is empty.

2.3 Path Subgraph and Minimal Equivalent Edge Set

Let us consider the relationships between two paths. We use $P_i \rightarrow P_j$ to denote the subgraph of G consisting of i) path

P_i, ii) path P_j, and iii) $E_{P_i \to P_j}$, which is the set of edges from vertices on path P_i to vertices on path P_j. For instance, $E_{P_1 \to P_2} = \{(1,4),(1,7),(3,4),(3,7),(13,11)\}$ is the set of edges from vertices in P_1 to vertices in P_2. We say subgraph $P_i \to P_j$ is *connected* if $E_{P_i \to P_j}$ is not empty.

Given a vertex u in path P_i, we want to find all vertices in path P_j that are reachable from u (through paths in subgraph $P_i \to P_j$ only). It turns out that we only need to know one vertex – the smallest (with regard to sequence id) vertex on path P_j reachable from u. We denote its sid as $r_j(u)$.

$$r_j(u) = min\{v.sid | u \to v \text{ and } v.pid = j\}$$

Clearly, for any vertex $v' \in P_j$,

$$u \to v' \Longleftrightarrow v'.sid \geq r_j(u)$$

Certain edges in $E_{P_i \to P_j}$ can be removed without changing the reachability between any two vertices in subgraph $P_i \to P_j$. This is characterized by the following definition.

Definition 2. *A set of edges $E^R_{P_i \to P_j} \subseteq E_{P_i \to P_j}$ is called the* minimal equivalent edge set *of $E_{P_i \to P_j}$ if removing any edge from $E^R_{P_i \to P_j}$ changes the reachability of vertices in $P_i \to P_j$.*

As shown in Figure 2(a), $\{(3,4),(13,11)\}$ is the minimal equivalent edge set for subgraph $P_1 \to P_2$. In Figure 2(b), $\{(7,6),(10,13),(11,14)\}$ is the minimal equivalent edge set of $E_{P_2 \to P_1}$=$\{(4,6)$, $(7,6)$, $(7,14)$, $(10,13)$, $(10,14)$, $(11,14)$, $(11,15)\}$. In Figure 2, edges belonging to the minimal equivalent edge set for subgraphs $P_i \to P_j$ in G are marked in bold.

In the following, we introduce a property of the minimal equivalent edge set that is important to our reachability algorithm.

Definition 3. *Let (u,v) and (w,z) be two edges in $E_{P_i \to P_j}$, where $u,w \in P_i$ and $v,z \in P_j$. We say the two edges are* crossing *if $u \preceq w$ (i.e., $u.sid \leq w.sid$) and $v \succeq z$ (i.e., $v.sid \leq z.sid$). For instance, $(1,7)$ and $(3,4)$ are crossing in $E_{P_i \to P_j}$. Given a set of edges, if no two edges in the set are crossing, then we say they are* parallel.

Lemma 1. *No two edges in any minimal equivalent edge set of $E_{P_i \to P_j}$ are crossing, or equivalently, edges in $E^R_{P_i \to P_j}$ are parallel.*

Proof Sketch: This can easily be proved by contradiction. Suppose (u,v) and (w,z) are crossing in $E^R_{P_i \to P_j}$. Without loss of generality, let us assume $u \preceq w (u \to w)$ and $v \succeq z (v \leftarrow z)$. Thus, we have $u \to w \to z \to v$. Therefore (u,v) is simply a short cut of $u \to v$, and dropping (u,v) will not affect the reachability for $P_i \to P_j$ as it can still be inferred through edge (w,z). □

After extra edges in $E_{P_i \to P_j}$ are removed, the subgraph $P_i \to P_j$ becomes a simple grid-like planar graph where each node has at most 2 outgoing edges and at most 2 incoming edges. This nice structure, as we will show later, allows us to map its vertices to a two-dimensional space and enables answering reachability queries in constant time.

Lemma 2. *The minimal equivalent edge set of $E_{P_i \to P_j}$ is unique for subbgraph $P_i \to P_j$.*

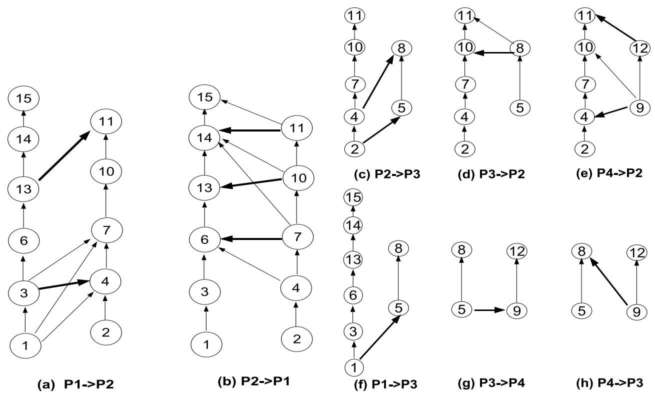

Figure 2: Path-Relationship of a DAG

Proof Sketch: We can prove this by contradiction. Assuming the lemma is not true, then there are two different minimal equivalent edge sets of $E_{P_i \to P_j}$, which we call $E^R_{P_i \to P_j}$ and $E^{R'}_{P_i \to P_j}$. We sort edges in each set from low to high, using vertex *sid* in P_i and vertex *sid* in P_j as primary and secondary keys, respectively. We compare edges in these two sets in sorted order. Assume $uv \in E^R_{P_i \to P_j}$ and $u'v' \in E^{R'}_{P_i \to P_j}$ are the first pair of different edges such that $u \neq u'$ or $v \neq v'$. This is a contradiction because it means either these two sets have different reachability information or one of the sets is not a minimal equivalent edge set. □

A simple algorithm that extracts the minimal equivalent edge set of $E_{P_i \to P_j}$ is sketched in Algorithm 1. We order all the edges from P_i to P_j ($E_{P_i \to P_j}$) by their end vertex in P_j. Let v' be the first vertex in P_j which is reachable from P_i. Let u' be the last vertex in P_i can reach v'. Then, we add (u',v') into $E^R_{P_i \to P_j}$ and remove all the edges in $E_{P_i \to P_j}$ which start from a vertex in P_i which proceed u' (or equivalently, which cross edge (u',v')). We repeat this procedure until the edge set $E_{P_i \to P_j}$ becomes empty.

Algorithm 1 MinimalEquivalentEdgeSet(P_i,P_j,$E_{P_i \to P_j}$)

```
1: $E^R_{P_i \to P_j} = \emptyset$
2: while $E_{P_i \to P_j} \neq \emptyset$ do
3:     $v' \to min(\{v | (u,v) \in E_{P_i \to P_j}\})$ {the first vertex in $P_j$ that $P_i$ can reach}
4:     $u' \leftarrow max(\{u | (u,v') \in E_{P_i \to P_j}\})$
5:     $E^R_{P_i \to P_j} \leftarrow E^R_{P_i \to P_j} \cup \{(u',v')\}$
6:     $E_{P_i \to P_j} \leftarrow E_{P_i \to P_j} \backslash \{(u,v) \in E_{P_i \to P_j} | u \preceq u'\}$ {Remove all edges which cross $(u',v')$}
7: end while
8: return $E^R_{P_i \to P_j}$
```

2.4 Path-Graph and its Spanning Tree (Path-Tree)

We create a directed *path-graph* for DAG G as follows. Each vertex i in the path-graph correponds to a path P_i in G. If path P_i connects to P_j in G, we create an edge (i,j) in the path graph. Let T be a directed spanning tree (or a forest) of the path-graph. Let $G[T]$ be the subgraph of G that contains i) all the paths of G, and ii) the minimal edge sets, $E^R_{P_i \to P_j}$, for every i, j if edge $(i,j) \in T$. We will show

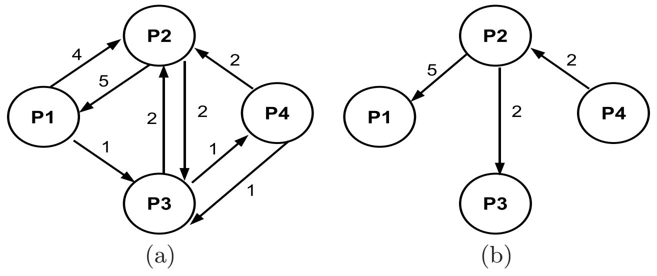

Figure 3: (a) Weighted Directed Path-Graph & (b) Maximal Directed Spanning Tree

that there is a vector labeling for $G[T]$ which can answer the reachability query for $G[T]$ in constant time. We refer to $G[T]$ as the *path-tree cover* for DAG G.

Just like Agrawal *et al.*'s tree cover [1], in order to utilize the path-tree cover, we need to "remember" those edges that are not covered by the path-tree. Ideally, we would like to minimize the index size, which means we need to minimize the number of the non-covered edges. Meanwhile, unlike the tree cover, we want to avoid computing the predecessor set (computing the predecessor set of each vertex is equivalent to computing the transitive closure). In the next subsection, we will investigate how to find the optimal path-tree cover if the knowledge of predecessor set is available. Here, we introduce a couple of alternative criteria which can help reduce the index size without such knowledge.

The first criterion is referred to as *MaxEdgeCover*. The main idea is to use the path-tree to cover as many edges in DAG G as possible. Let t be the remaining edges in DAG G (edges not covered by the path-tree). As we will show later in this subsection, t provides an upper-bound for the compression of transitive closure for G, i.e., each vertex needs to record at most t vertices for answering a reachability query. Given this, we can simply assign $|E_{P_i \to P_j}|$ as the cost for edge (i, j) in the directed path-graph.

The second criterion, referred to as *MinPathIndex*, is more involved. As we discussed in the path-decomposition, each vertex needs to remember at most one vertex on any other path to answer a reachability query. Given two paths P_i, P_j, and their link set $E_{P_i \to P_j}$, we can quickly compute the index cost as follows if $E_{P_i \to P_j}$ does not include the tree-cover. Let u be the last vertex in path P_i that can reach path P_j. Let $P_i[\to u] = \{v | v \in P_i, v \preceq u\}$ be the subsequence of P_i that ends with vertex u. For instance, in our running example, vertex 13 is the last vertex in path P_1 which can reach path P_2, and $P_1[\to 13] = \{1, 3, 6, 13\}$ (Figure 2). We assign a weight $w_{P_i \to P_j}$ to be the size of $P_j[\to u]$. In our example, the weight $w_{P_1 \to P_2} = 4$. Basically, this weight is the labeling cost if we have to materialize the reachability information for path P_i about path P_j. Considering path P_1 and P_2, we only need to record vertex 4 in path P_2 for vertices 1 and 3 in path P_1 and vertex 11 for vertices 6 and 13. Then, we can answer if any vertex in P_2 can be reached from any vertex in P_1. Thus, finding the maximum spanning tree in such a weighted directed path-graph corresponds to minimizing the index size by using path labeling schema. Figure 3(a) is the weighted directed path-graph using the *MinPathIndex* criteria.

To reduce the index size for the path-tree cover, we would like to extract the maximum directed spanning tree (or forest). As an example, Figure 3(b) is the maximum directed spanning tree extracted from the weighted directed path-graph of Figure 3(a). The Chu-Liu/Edmonds algorithm can be directly applied to this problem [4, 8]. The fast implementation that uses the Fibonacci heap requires $O(m' + k \log k)$ time complexity, where k is the width of path-decomposition and m' is the number of directed edges in the weighted directed path-graph [9]. Clearly, $k \leq n$ and $m' \leq m$, m is the number edges in the original DAG.

2.5 Reachability Labeling for Path-Tree Cover

The path-tree is formed after the minimal equivalent edge sets and maximal directed spanning tree are established. In this section, we introduce a vector labeling scheme for vertices in the path-tree. The labeling scheme enables us to answer reachability queries in constant time. We partition the path-tree into paths, and we call the result a path-path cover, denoting it as $G[P]$.

We start with a simple scenario. For example, in Figure 4, we have the path-path: $(P4, P2, P1)$. We map each vertex in the path-path to a two-dimensional space as follows. First, all the vertices in each single path have the same path ID, which we define to be Y. For instance, vertices $P4, P2$ and $P1$ have path ID 1, 2, and 3, respectively.

```
Algorithm 2 DFSLabel($G[P](V, E), P_1 \cup \cdots \cup P_k$)
Parameter: $P_1 \cup \cdots \cup P_k$ is the path-decomposition of $G$
Parameter: $G[P]$ is represented as linked lists: $\forall v \in V$ :
    $linkedlist(v)$ records all the immediate neighbors of $v$.
    Let $v \in P_i$. If $v$ is not the last vertex in path $P_i$, the
    first vertex in the linked list is the next vertex of $v$ in the
    path
Parameter: $P_i \preceq P_j \iff i \leq j$
1: $N \leftarrow |V|$
2: for $i = 1$ to $k$ do
3:    $v \leftarrow P_i[1]$ {$P_i[1]$ is the first vertex in the path}
4:    if $v$ is not visited then
5:       DFS($v$)
6:    end if
7: end for
Procedure DFS($v$)
1: $visited(v) \leftarrow TRUE$
2: for each $v' \in linkedlist(v)$ do
3:    if $v'$ is not visited then
4:       DFS($v'$)
5:    end if
6: end for
7: $X(v) \leftarrow N$ {Label vertex $v$ with $N$}
8: $N \leftarrow N - 1$
```

Then, we perform a depth-first search (DFS) to assign the X label for each vertex (The procedure sketch is in Algorithm 2). For labeling purposes, in the linked list for each vertex, we always put its neighbor of the same path (the right neighbor, if it exists) ahead of its neighbors of other paths (the upper neighbor, if it exists). This allows us to visit the vertices in the same path before any other vertices. In the DFS search, we will maintain a counter N which records the number of all vertices in the graph initially (In our running example, $N = 13$, see Figure 4). To start the DFS search, we begin with the first vertex v_0 in the first path. In our example, it is the vertex 9 in path $P4$. Starting from this vertex, our DFS always tries to visit its right neighbor and then tries to visit its upper neighbor. For a given vertex, when we finish visiting all of its neighbors, we assign this vertex the label N and reduce N by one. Once we visit all the vertices which can be reached from v_0, we start from the first vertex in the second path if it has not been visited. We continue this process until all the vertices have been visited. Note that our labeling procedure bears some similarity to [12]. However, their procedure can handle only a specific type of planar graph, while our labeling procedure handles path-tree graphs which can be non-planar.

Figure 4(a) shows the X label based on the DFS procedure. Figure 4(b) shows the embedding of the path-path in the two dimensional space based on their X and Y labels.

LEMMA 3. *Given two vertices u and v in the path-path, u can reach v if and only if $u.X \leq v.X$ and $u.Y \leq v.Y$ (this is also referred to as u dominates v).*

Proof Sketch: First, we prove $u \rightarrow v \implies u.X \leq v.X \wedge u.Y \leq v.Y$. Clearly if u can reach v, then $u.Y \leq v.Y$ (path-path property), and DFS traversal will visit u earlier than v, and only after visiting all v's neighbor will it return to u. So, $u.X \leq v.X$ based on DFS. Second, we prove $u.X \leq v.X \wedge u.Y \leq v.Y \implies u \rightarrow v$. This can be proved by way of contradiction. Let us assume u can not reach v. Then, (Case 1:) if u and v are on the same path ($u.Y = v.Y$), then $u.X > v.X$, i.e., we will visit v before we visit u. In other words, we complete u's visit before we complete v's visit. Thus, $u.X > v.X$ contradicts our assumption. (Case 2:) if u and v are on different paths ($u.Y < v.Y$), similar to case 1, we will complete the visit of u before we complete the visit of v as u can not reach v. So we have $u.X > v.X$, a contradiction. Combining both cases 1 and 2, we prove our result. $\Box$

For the general case, the paths form a tree instead of a single path. In this case, each vertex will have an additional interval labeling based on the tree structure. Figure 5(c) shows the interval labeling of the path-tree in Figure 5(a). All the vertices on the path share the same interval label for this path. Besides, the Y label for each vertex is generalized to be the level of its corresponding path in the tree path, i.e., the distance from the path to the root (we assume there is a virtual root connecting all the roots of each tree in the branching/forest). The X label is similar to the simple path-path labeling. The only difference is that each vertex can have more than one upper-neighbor. Besides, we note that we will traverse the first vertex in each path based on the path's level in the path-tree and any of the traversal orders of the paths in the same level will work for the Y labeling. Figure 5(a) shows the X label of all the vertices in the path-tree and Figure 5(b) shows the two dimensional embedding.

LEMMA 4. *Given two vertices u and v in the path-tree, u can reach v if and only if 1) u dominates v, i.e., $u.X \leq v.X$ and $u.Y \leq v.Y$; and 2) $v.I \subseteq u.I$, where $u.I$ and $v.I$ are the interval labels of u and v's corresponding paths.*

Proof Sketch: First, we note that the procedure will maintain this fact that if u can reach v, then $u.X \leq v.X$. This is based on the DFS procedure. Assuming u can reach v, then, there is a path in the tree from u's path to v's path. So we have $v.I \subseteq u.I$ (based on the tree labeling) and $u.Y \leq v.Y$. In addition, if we have $v.I \subseteq u.I$, then there is a path from u's path to v's path. Then, using the similar argument from

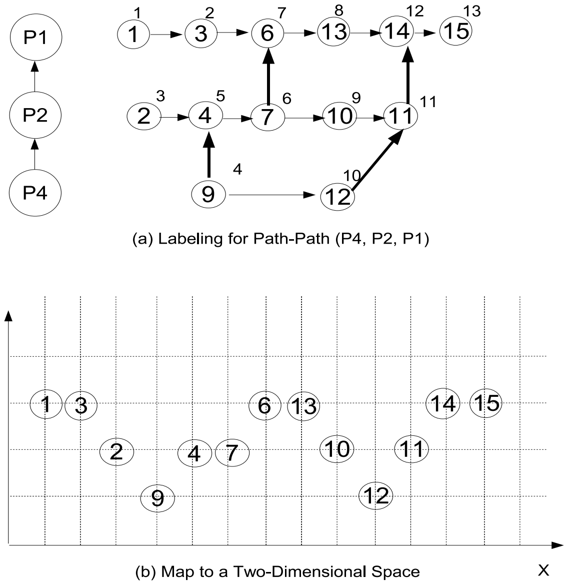
P1
P2
P4
(a) Labeling for Path-Path (P4, P2, P1)
Y
X
(b) Map to a Two-Dimensional Space

Figure 4: Labeling for Path-Path (A simple case of Path-Tree)

Lemma 3 we can see that if $u.X \leq v.X$ then u can reach v. $\Box$

Assuming any interval I has the format $[I.begin, I.end]$, we have the following theorem:

THEOREM 1. *A three dimensional vector labeling $(X, I.begin, I.end)$ is sufficient for answering the reachability query for any path-tree.*

Proof Sketch: Note that if $v.I \subseteq u.I$, then $v.Y \geq u.Y$. Thus, we can drop Y's label without losing any information. Thus, for any vertex v, we have $v.X$ (the first dimension) and $v.I$ (the interval for the last two dimensions). $\Box$

Our labeling algorithm for path-tree is very similar to the labeling algorithm for path-path. It has two steps:

1. Create tree labeling for the Maximal Directed Spanning Tree obtained from weighed directed path-graph (by Edmonds' algorithm), as shown in Figure 5(c)

2. Let $P^L = P_1^L \cup \cdots \cup P_{k'}^L$, where P_i^L is the set of vertices (i.e. paths) in level i of the Maximal Directed Spanning Tree, which has k' levels. Call Algorithm 2 with $G[P^L](V, E), P_1^L \cup \cdots \cup P_{k'}^L$.

The overall construction time of the path-tree cover is as follows. The first step of path-decomposition is $O(n + m)$, which includes the cost of the topological sort. The second step of building the weighted directed path-graph is $O(m)$. The third step of extracting the maximum spanning tree is $O(m' + k \log k)$, where $m' \leq m$ and $k \leq n$. The fourth step of labeling basically utilizes a DFS proceduce which costs $O(m'' + n)$, where m'' is the number of edges in the path-tree and $m'' \leq m$. Thus, the total construction time of path-tree cover is $O(m + n \log n)$.

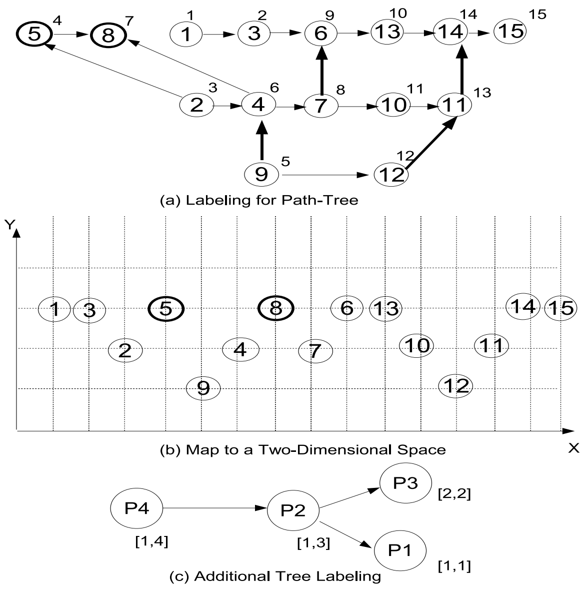
(a) Labeling for Path-Tree
(b) Map to a Two-Dimensional Space
(c) Additional Tree Labeling
Y
X
P4
P2
P3
P1
[1,4]
[1,3]
[2,2]
[1,1]

Figure 5: Complete Labeling for the Path-Tree

2.6 Transitive Closure Compression and Reachability Query Answering

Edges not included in the path-tree cover can result in extra reachability which will not be covered by the path-tree structure. The same problem appears in the tree cover related approaches.

For example, Dual Labeling and GRIPP utilize a tree as their first steps; they then try to find novel ways to handle non-tree edges. Their ideas are in general applicable to dealing with non-path-tree edges as well. From this perspective, our path-tree cover approach can be looked as being orthogonal to these approaches.

To answer a reachability query for the entire DAG, a simple strategy is to actually construct the transitive closure for non-path-tree edges in the DAG. The construction time is $O(n+m+t'^3)$ and the index size is $O(n+t'^2)$ according to Dual Labeling [16], where t' is the number of non-path-tree edges. However, as we will see later in theorem 5, if the path-tree cover approach utilizes the same tree cover as Dual Labeling for a graph, t' is guaranteed to be smaller than t (non-tree edges).

Moreover, if a maximally compressed transitive closure is desired, the path-tree structure can help us significantly reduce the transitive size (index size) and its construction time as well. Let $R^c(u)$ be the compressed set of vertices we record for u's transitive closure utilizing the path-tree. Assume u is a vertex in P_i. To answer a reachability query for u and v (i.e. if v is reachable from u), we need to test 1) if v is reachable from u based on the path-tree labeling and if not 2) for each x in $R^c(u)$, whether v is reachable from x based on the path-tree labeling. We note that $R^c(u)$ includes at most one vertex from any path and in the worst case, $|R^c(u)| = k$, where k is number of paths in the path-tree. Thus, a query would cost $O(k)$. In Subsection 3.4, we will introduce a procedure which costs $O(\log^2 k)$.

Algorithm 3 constructs a maximally compressed transitive closure for each vertex. The construction time is $O(mk)$ because for any vertex u, $|R^c(u)| \le k$.

Algorithm 3 CompressTransitiveClosure $(G,G[T])$

```
1: $V_R \leftarrow Reversed\ Topological\ Order\ of\ G$ {Perform topological sort of $G$}
2: $N \leftarrow |V|$
3: for $i = 1$ to $N$ do
4:    $R^c(V_R[i]) \leftarrow \emptyset$;
5:    Let $S$ be the set of immediate successors of $V_R[i]$ in $G$;
6:    for each $v \in S$ do
7:       for each $v' \in R^c(v) \cup \{v\}$ do
8:          if $V_R[i]$ cannot reach $v'$ in $G[T]$ then
9:             Add $v'$ into $R^c(V_R[i])$ ;
10:         end if
11:      end for
12:   end for
13: end for
```

3. THEORETICAL ANALYSIS OF OPTIMAL PATH-TREE COVER CONSTRUCTION

In this section, we investigate several theoretical questions related to path-tree cover construction. We show that given the path-decomposition of DAG G, finding the optimal path-tree cover of G is equivalent to the problem of finding the *maximum spanning tree* of a directed graph. We demonstrate that the optimal tree cover by Agrawal *et al.* [1] is a special case of our problem. In addition, we show that our path-tree cover can always achieve better compression than any chain cover or tree cover.

To achieve this, we utilize the predecessor set of each vertex. But first we note that the computational cost for computing all of the predecessor sets of a given DAG G is equivalent to the cost of the transitive closure of G, with $O(nm)$ time complexity. Thus it may not be applicable to very large graphs as its computational cost would be prohibitive. It can, however, still be utilized as a starting point for understanding the potential of path-tree cover, and its study may help to develop better heuristics to efficiently extract a good path-tree cover. In addition, these algorithms might be better suited for other applications if the knowledge of predecessor sets is readily available. Thus, they can be applied to compress the transitive closure.

We will introduce an optimal query procedure for reachability queries which can achieve $O(\log^2 k)$ time complexity in the worst case, where k is the number of paths in the path-decomposition.

3.1 Optimal Path-Tree Cover with Path-Decomposition

We first consider the following restricted version of the optimal path-tree cover problem.

Optimal Path-Tree Cover (OPTC) Problem: *Let $P = (P_1, \cdots, P_k)$ be a path-decomposition of DAG G, and let $\mathcal{G}_s(P)$ be the family including all the path tree covers of G which are based on P. The OPTC problem tries to find the optimal tree cover $G[T] \in \mathcal{G}_s(P)$, such that it requires minimal index size to compress the transitive closure of G.*

To solve this problem, let us first analyze the index size which will be needed to compress the transitive closure utilizing a path-tree $G[T]$. Note that $R^c(u)$ is the compressed set of vertices which vertex u can reach and for compression purposes, $R^c(u)$ does not include v if 1) u can reach

v through the path-tree $G[T]$ and 2) there is an vertex $x \in R^c(u)$, such that x can reach v through the path-tree $G[T]$. Given this, we can define the compressed index size as

$$Index\_cost = \sum_{u \in V(G)} |R^c(u)|$$

(We omit the labeling cost for each vertex as it is the same for any path-tree.) To optimize the index size, we utilize the following equivalence.

Lemma 5. *For any vertex v, let $T_{pre}(v)$ be the* ***immediate*** *predecessor of v on the path-tree $G[T]$. Then, we have*

$$Index\_cost = \sum_{v \in V(G)} |S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))|$$

where $S(v)$ includes all the vertices which can reach vertex v in DAG G.

Proof Sketch: For vertex v, if $v \in R^c(u)$, then $u \in S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$. This can be proved by contradiction. Assume $u \notin S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$. Then there must exist a vertex w such that u can reach w in DAG G and w can reach v in the path tree. Then v should be replaced by w in $R^c(u)$, a contradiction.

For vertex u, if $u \in S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$, then $v \in R^c(u)$. This can also be proved by contradiction. Assume $v \notin R^c(u)$. Then because $u \in S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$ we conclude u cannot reach v no matter what other vertices are in $R^c(u)$, a contradiction. Thus, for each vertex

$$v \in R^c(u) \Longleftrightarrow u \in S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$$

□

Given this, we can solve the OPTC problem by utilizing the predecessor sets to assign weights to the edges of the weighted directed path-graph in Subsection 2.4. Thus, the path-tree which corresponds to the maximum spanning tree of the weighted directed path-graph optimizes the index size for the transitive closure. Consider two paths P_i, P_j and the minimal equivalent edge set $E^R_{P_i,P_j}$. For each edge $(u, v) \in E^R_{P_i,P_j}$, let v' be the vertex which is the immediate predecessor of v in path P_j. Then, we define the predecessor set of v with respect to u as

$$S_u(v) = (S(u) \cup \{u\}) \backslash (S(v') \cup \{v'\})$$

If v is the first vertex in the path P_j, then we define $S(v') = \emptyset$. Given this, we define the weight from path P_i to path P_j as

$$w_{P_i \to P_j} = \sum_{(u,v) \in E^R_{P_i \to P_j}} |S_u(v)|$$

We refer to such criteria as *OptIndex*.

Theorem 2. *The path-tree cover corresponding to the maximum spanning tree from the weighted directed path-graph defined by OptIndex achieves the minimal index size for the compressed transitive closure among all the path-trees in $\mathcal{G}_s(P)$.*

Proof Sketch: We decompose $Index\_cost$ utilizing the path-decomposition $P = P_1 \cup \cdots \cup \cdots P_k$ as follows:

$$Index\_cost = \sum_{1 \le i \le k} \sum_{v \in P_i} |S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))|$$

Note that $S(v) \supseteq (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$. Then, minimizing the $Index\_cost$ is equivalent to maximizing

$$\sum_{1 \le i \le k} \sum_{v \in P_i} | \bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\})|$$

We can further rewrite it as (v_l being the vertex with largest *sid* in the path P_i)

$$\sum_{1 \le i \le k} (\sum_{v \in P_i \backslash \{v_l\}} |S(v) \cup \{v\}| + \sum_{(u,v) \in E^R_{P_j \to P_i}} |S_u(v)|)$$

where P_j is the parent path in the path-tree of path P_i. Since the first half of the sum is the same for the given path decomposition, we essentially need to maximize

$$\sum_{1 \le i \le k} \sum_{(u,v) \in E^R_{P_i \to P_j}} |S_u(v)| = \sum_{1 \le i \le k} w_{P_i \to P_j}$$

□

Recall that in Agrawal's optimal tree cover algorithm [1], to build the tree, for each vertex v in DAG G, essentially they choose its immediate predecessor u with the maximal number of predecessors as its parent vertex, i.e.,

$$|S(u)| \ge |S(x)|, \forall x \in in(v), u \in in(v)$$

Given this, we can easily see that if the path decomposition treats each vertex in G as an individual path, then we have the optimal tree cover algorithm from Agrawal *et al.* [1].

Theorem 3. *The optimal tree cover algorithm [1] is a special case of path-tree construction when each vertex corresponds to an individual path and the weighted directed path-graph utilizes the* OptIndex *criteria.*

Proof Sketch: Note that for any vertices u and v such that $(u, v) \in E(G)$, then the weight on the edge (u, v) in the weighted directed path-graph (each path is a single vertex) is $w_{u,v} = |S(u) \cup \{u\}|$. □

3.2 Optimal Path-Decomposition

Theorem 2 shows the optimal path-tree cover for the given path-decomposition. A follow-up question is then how to choose the path-decomposition which can achieve overall optimality. This problem, however, remains open at this point (undecided between P and NP). Instead, we ask the following question.

Optimal Path-Decomposition (OPD) Problem: *Assuming we utilize only the path-decomposition to compress the transitive closure (in other words, no cross-path edges), the OPD problem is to find the optimal path-decomposition which can maximally compress the transitive closure.*

There are clearly cases where the optimal path-decomposition does not lead to the perfect path-tree that *alone* can answer all the reachability queries. This nevertheless provides a good heuristic to choose a good path-decomposition in the case where the predecessor sets are available. Note that the OPD problem is different from the chain decomposition problem in [11], where the goal is to find the minimal width of the chain decomposition.

We map this problem to the *minimal-cost flow* problem [10]. We transform the given DAG G into a network G_N (referred to as the flow-network for G) as follows. First, each vertex v in G is split into two vertices s_v and e_v, and we insert a single edge connecting s_v to e_v. We assign the cost of such an edge $F(s_v, e_v)$ to be 0. Then, for an edge (u, v) in G, we map it to (e_u, s_v) in G_N. The cost of such an edge is $F(e_u, s_v) = -|S(u) \cup \{u\}|$, where $S(u)$ is the predecessor set of u. Finally, we add a virtual source vertex and a virtual sink vertex. The virtual source vertex is connected to any vertex s_v in G_N with cost 0. Similarly, each vertex e_v is connected to the sink vertex with cost being zero. The capacity of each edge in G_N is one ($C(x, y) = 1$). Thus, each edge can take maximally one unit of flow, and correspondingly each vertex can belong to one and only one path.

Let $c(x, y)$ be the amount (0 or 1) of flow over edge (x, y) in G_N. The cost of the flow over the edge is $c(x, y) \cdot F(x, y)$, where $c(x, y) \leq C(x, y)$. We would like to find a set of flows which go through all the vertex-edges (s_v, e_v) and whose overall cost is minimal. We can solve it using an algorithm for the *minimum-cost flow* problem for the case where the amount of flow being sent from the source to the sink is given. Let i-flow be the solution for the minimum-cost flow problem when the total amount of flow from the source to the sink is fixed at i units. We can then vary the amount of flow from 1 to n units and choose the largest one i-flow which achieves the minimum cost. It is apparent that i-flow goes through all the vertex-edges (s_v, e_v).

THEOREM 4. *Let G_N be the flow-network for DAG G. Let F_k be the minimal cost of the amount of k-flow from the source to the sink, $1 \leq k \leq n$. Let i-flow from the source to the sink minimize all the n-flow, $F_i \leq F_k, 1 \leq k \leq n$. The i-flow corresponds to the best index size if we utilize only the path-decomposition to compress the transitive closure.*

Proof Sketch: First, we can prove that for any given i-flow, where i is an integer, the flow with minimal-cost will pass each edge either with 1 or 0 (similar to the Integer Flow property [6]). Basically, the flow can be treated as a binary flow. In other words, any flow going from the source to the sink will not split into branches (note that each edge has only capacity one). Thus, applying Theorem 2, we can see that the total cost of the flow (multiplied by negative one) corresponds to the savings for the *Index_cost*

$$\sum_{1 \leq i \leq k} \Big(\sum_{v \in P_i \setminus \{v_l\}} |S(v) \cup \{v\}| \Big) = \sum_{(u,v) \in G_N} c(u, v) \times F(u, v)$$

where v_l is the vertex with largest *sid* in path P_i. Then, let i-flow be the one which achieves minimal cost (the most negative cost) from the source to the sink. Thus, when we invoke the algorithm which solves the *minimal-cost maximal flow*, we will achieve the minimal cost with i-flow. It is apparent that the i-flow goes through all the vertex-edges (s_v, e_v) because i is largest. Thus, we identify the flow and find our path-decomposition. □

Note that there are several algorithms which can solve the minimal-cost maximal-flow problem with different time complexities. Interested readers can refer to [10] for more details. Our methods can utilize any of these algorithms.

3.3 Superiority of Path-Tree Cover Approach

In the following, we consider how to build a path-tree which utilizes the existing chain decomposition or tree cover (if they are already computed) to achieve better compression results. Note that both chain decomposition and tree cover approaches are generally as expensive as the computation of transitive closure.

A major difference between chain decomposition and path-decomposition is that each path P_i in the path-decomposition is a subgraph of G. However, a chain may not be a subgraph of G. It is a subgraph of the transitive closure. Therefore, several methods developed by Jagadish actually require the knowledge of transitive closure [11]. We can easily apply any chain decomposition into our path-tree framework as follows. For any chain $C_i = (v_1, \cdots, v_k)$, if (v_i, v_{i+1}) is not an edge in G, then we add (v_i, v_{i+1}) into the edge set $E(G)$. The resulting graph G' has the same transitive closure as G. Now, the chain decomposition of G becomes a path decomposition of G' and we can then apply the path-tree construction based on this decomposition.

The path-tree built upon a chain decomposition C will achieve better compression than C itself (since at least one cross-path edge can be utilized to compress the transitive closure; see the formula in Theorem 2) if there is any edge connecting two chains in C in DAG G. Note that if there are no edges connecting two chains in G, then both path-tree and chain decomposition completely represent G and thus maximally compress G.

For any tree cover, we can also transform it into a path-decomposition. We extract the first path by taking the shortest path from the tree cover root to any of its leaves. After we remove this path, the tree will then break into several subtrees. We perform the same extraction for each subtree until each subtree is empty. Thus, we can have the path-decomposition based on the tree cover. In addition, we note that there is at most one edge linking two paths.

Given this, we can prove the following theorem.

THEOREM 5. *For any tree cover, the path-tree cover which uses the path-decomposition from the tree cover and is built by OptIndex has an index size lower than or equal to the index size of the corresponding tree cover.*

Proof Sketch: This follows Theorem 2. Further, there is a path-tree cover which can represent the same shape as the original tree cover, i.e., if there is an edge in the original tree cover linking path P_i to P_j, there is a path-tree that can preserve its path-path relationship by adding this edge and possibly more edges from path P_i to P_j in DAG G to the path-tree. □

3.4 Very Fast Query Processing

Here we describe an efficient query processing procedure algorithm with $O(\log^2 k)$ query time, where k is the number of paths in the path-decomposition. We first build an interval tree based on the intervals of each vertex in $R(u)$ as follows (this is slightly different from [7]): Let x_{mid} be the median of the end points of the intervals. Let $u.I.begin$ and $u.I.end$ be the starting point and ending point of the interval $u.I$, respectively. We define the interval tree recursively. The root node v of the tree stores x_{mid}, such that

$$\begin{aligned} \mathcal{I}_{left} &= \{u | u.I.end < x_{mid}\} \\ \mathcal{I}_{mid} &= \{u | x_{mid} \in u.I\} \\ \mathcal{I}_{right} &= \{u | u.I.begin > x_{mid}\} \end{aligned}$$

Then, the left subtree of v is an interval tree for the set of $\mathcal{I}_{left}$ and the right subtree of v is an interval tree for the set of $\mathcal{I}_{right}$. We store $\mathcal{I}_{mid}$ only once and order it by $u.X$ (the X label of the vertices in $\mathcal{I}_{mid}$). This is the key difference between our interval tree and the standard one from [7]. In addition, when $\mathcal{I}_{mid} = \emptyset$, the interval tree is a leaf. Following [7], we can easily construct in $O(k \log k)$ time an interval tree using $O(k)$ storage and depth $O(\log k)$.

Algorithm 4 Query(r,v)

```
1: if $r$ is not a leaf then
2:    Perform BinarySearch to find a vertex $w$ on $r.\mathcal{I}_{mid}$
      whose $w.X$ is the closest one such that $w.X \leq v.X$
3: else
4:    return $FALSE$;
5: end if
6: if $v.I \subseteq w.I$ then
7:    return $TRUE$;
8: end if
9: if $v.I.end < r.x_{mid}$ then
10:   Query($r.left$,$v$);
11: else
12:   if $v.I.begin > r.x_{mid}$ then
13:      Query($r.right$,$v$);
14:   end if
15: end if
```

The query procedure using this interval tree is shown in Algorithm 4 when u does not reach v through the path-tree. The complexity of the algorithm is $O(\log^2 k)$. The correctness of the Algorithm 4 can be proved as follows.

Lemma 6. *For the case where u cannot reach v using only the path-tree, then Algorithm 4 correctly answer the reachability query.*

Proof Sketch: The key property is that for any two vertices w and w' in $\mathcal{I}_{mid}$, either $w.I \subseteq w'.I$ or $w'.I \subseteq w.I$. This is because the intervals are extracted from the tree structure, and it is easy to see that:

$$w.I \cap w'.I \neq \emptyset \Rightarrow (w.I \subseteq w'.I) \vee (w'.I \subseteq w.I)$$

Thus, we can order the intervals based on the inclusion relationship. Further, if $w.I \subseteq w'.I$, then we have $w.X < w'.X$ (Otherwise, we can drop $w.I$ if $w.X > w'.X$). Following this, we can see that for any w from Line 2 if $w.I$ does not include $v.I$, then no other vertex in $\mathcal{I}_{mid}$ can include $v.I$ with $w.X \leq v.X$. □

4. EXPERIMENTS

In this section, we empirically evaluate our path-tree cover approach on both real and synthetic datasets. We are particularly interested in the following two questions:

1. What is the compression rate of the path-tree cover approach, compared to that of the optimal tree cover approach?
2. What are the construction time and query time for the path-tree approach, compared to those of existing approaches?

All tests were run on an AMD Opteron 2.0GHz machine with 2GB of main memory, running Linux (Fedora Core 4), with a 2.6.17 x86_64 kernel. All algorithms are implemented in C++. Table 2 lists the real graphs we use. Among them, AgroCyc, Anthra, Ecoo157, HpyCyc, Human,Mtbra, and VchoCyc are from EcoCyc [2]; Xmark and Nasa are XML documents; and Reactome, aMaze, and KEGG are metabolic networks provided by Trißl [15]. The first two columns are the number of vertices and edges in the original graphs, and the last two columns are the number of vertices and edges in the DAG after compressing the strongly connected components.

Table 2: Real Datasets

Graph Name	#V	#E	DAG #V	DAG #E
AgroCyc	13969	17694	12684	13408
aMaze	11877	28700	3710	3600
Anthra	13736	17307	12499	13104
Ecoo157	13800	17308	12620	13350
HpyCyc	5565	8474	4771	5859
Human	40051	43879	38811	39576
Kegg	14271	35170	3617	3908
Mtbrv	10697	13922	9602	10245
Nasa	5704	7942	5605	7735
Reactome	3678	14447	901	846
Vchocyc	10694	14207	9491	10143
Xmark	6483	7654	6080	7028

We apply three different methods: 1) *Tree*, which corresponds to the optimal tree cover approach by Agrawal [1], 2) *PTree-1*, which corresponds to the path-tree approach utilizing optimal tree cover together with *OptIndex* (Subsection 3.1), and 3) *PTree-2*, which corresponds to the path tree approach described in Section 2 and utilizing the *MinPathIndex* criteria. For each method, we measure three parameters: compressed transitive closure size, construction time, and query time. (*Here, construction time is the total processing time of a DAG. As an example, for the path-tree cover approach, it includes the construction of both the path-tree cover and the compressed transitive closure.*)

Both PTree-1 and PTree-2 use Algorithm 3 to calculate compressed transitive closures. A query is generated by randomly picking a pair of nodes for a reachability test. We measure the query time by answering a total of $100,000$ randomly generated reachability queries.

We compare our approaches with only the optimal tree cover approach because we find it has the best performance on most of the real datasets. Under Dual-Labeling I and II [16] (using code provided by its authors), we found the performance on most of our experimental datasets to be worse than under Agrawal *et al.*'s tree-cover approach. This is because Dual-Labeling methods focus on very sparse or very small graphs with very few non-tree edges. Furthermore, we believe a comparison with Dual-Labeling may not be fair, since our implementation of PTree-1, PTree-2 and optimal tree cover is based on the C++ Standard Template Library, while the Dual-Labeling code provided by Dual-Labeling authors is based on the C++ Boost libraries. For GRIPP [15], we found that its query time is too long with our C++ implementation. On average, its query time can be several orders of magnitude slower than the time of the tree-cover approach and also our path-tree cover approach. This is primarily because their method focuses on how to implement index and answer reachability queries in a real database sys-

[2]http://ecocyc.org/

Table 3: Comparison between Optimal Tree Approach and Path-Tree Approach

	Transitive Closure Size			Construction Time (in ms)			Query Time (in ms)		
	Tree	PTree-1	PTree-2	Tree	PTree-1	PTree-2	Tree	PTree-1	PTree-2
AgroCyc	13550	962	2133	149.798	224.853	142.311	46.629	10	14.393
aMaze	5178	1571	17274	1062.15	834.697	63.748	19.478	21.529	61.925
Anthra	13155	733	2620	141.108	212.258	143.568	44.958	9.317	16.498
Ecoo	13493	973	3592	151.455	229.29	141.951	46.674	11.224	16.739
HpyCyc	5946	4224	4661	57.378	106.552	71.675	31.539	12.089	15.503
Human	39636	965	2910	446.321	648.005	465.148	70.107	20.008	23.008
Kegg	5121	1703	30344	746.025	1057.11	86.396	17.509	27.282	75.448
Mtbrv	10288	812	3664	111.479	173.382	106.583	40.391	9.81	19.815
Nasa	9162	5063	6670	85.291	111.397	53.139	37.037	16.214	20.771
Reactome	1293	383	1069	17.244	18.189	6.3	17.565	6.467	13.037
VchoCyc	10183	830	2262	109.465	170.714	103.036	40.026	8.999	14.274
Xmark	8237	2356	10614	204.762	247.628	68.358	37.834	17.122	41.549

tem. Thus, we do not compare with GRIPP here because this comparison may not be fair in our experimental setting.

Real-Life Graphs. Table 3 shows the compressed transitive closure size, the construction time, and the query time. We can see that PTree-1 consistently has a better compression rate than the tree approaches do, which confirms our theoretical analysis in Subsection 3.1. PTree-2 in 9 out of 12 datasets has much better compression rate than the optimal tree cover approach does. Overall, PTree-1 and PTree-2 achieve, respectively, an average of 10 times and 3 times better compression rate than the optimal tree cover approach does. The compressed transitive closure size directly affects the query time for the reachability queries. This can be observed in the query time results (PTree-1 and PTree-2, respectively, are approximately 3 and 2 times as fast as the optimal tree cover approach at answering the reachability queries).

For the construction time, we do expect PTree-1 to be slower than the optimal tree cover approach since it uses the optimal tree cover as the first step for path-decomposition (Recall that we extract the paths from the optimal tree). However, PTree-2 uses less construction time than optimal tree cover in 9 out of 12 datasets, and on average is 3 times as fast as the optimal tree cover. This result is generally consistent with our analysis of the theoretical time complexity, which is $O(m + n \log n) + O(mk)$.

Random DAG. We also compare path-tree cover approaches with the tree cover approach on synthetic DAGs. Here, we generate a random DAG with average edge density of 2 (i.e. on average each vertex points to two other vertices), varying the number of vertices from $10,000$ to $100,000$. Figure 6 shows the compressed transitive closure size of the path-tree cover approaches (PTree-1 and PTree-2) and optimal tree cover approach (Tree). Figure 7 and Figure 8 show the construction time and query time of these three approaches respectively. Overall, the compressed transitive closure size of PTree-1 and PTree-2 is approximately 29% and 26%, respectively, of the one from the optimal tree cover.

In Figure 6 and Figure 8, PTree-1 and PTree-2 both perform significantly better than the tree cover approach. In Figure 7, PTree-1 takes longer construction time than the optimal tree cover because it uses the optimal tree cover as

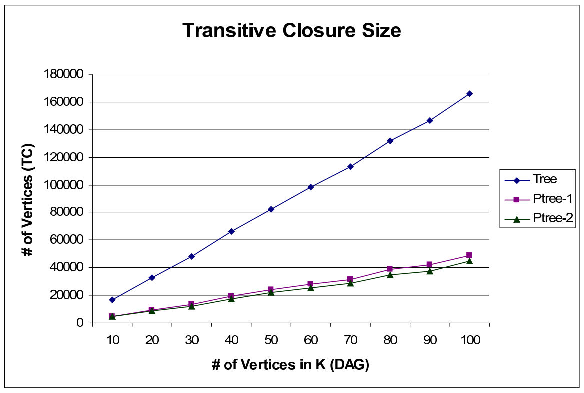

Figure 6: Transitive Closure Size for Random DAG

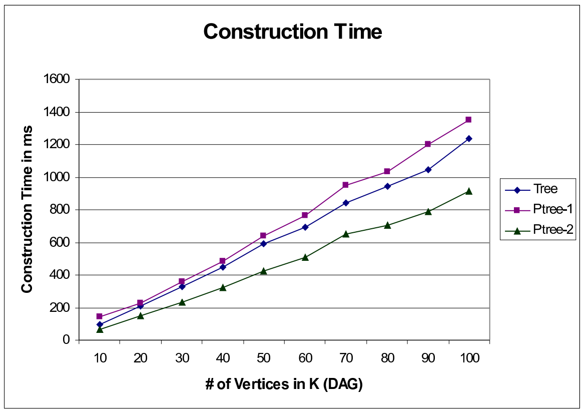

Figure 7: Construction Time for Random DAG

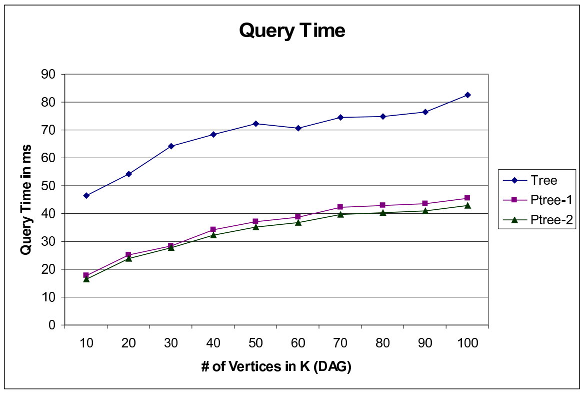

Figure 8: Query Time for Random DAG

the first step for path-decomposition. PTree-2 takes shorter construction time than the optimal tree cover as indicated by our theoretical analysis.

In the random DAG tests, PTree-2 has slightly smaller transitive closure size and shorter query time than PTree-1 does, whereas in the real dataset tests, PTree-1 has the smaller size and query time. We conjecture this is because many real datasets have tree-like structures while random DAGs do not. Therefore, tree cover and PTree-1 fit very well for these real datasets but less well for random DAGs. It also suggests that the *MinPathIndex* criteria may be well suited for DAGs that lack a tree-like structure.

5. CONCLUSION

In this paper, we introduce a novel *path-tree* structure to assist with the compression of transitive closure and answering reachability queries. Our path-tree generalizes the traditional tree cover approach and can produce a better compression rate for the transitive closure. We believe our approach opens up new possibilities for handling reachability queries on large graphs. Path-tree also has the potential to integrate with other existing methods, such as Dual-labeling and GRIPP, to further improve the efficiency of reachability query processing. In the future, we will develop disk-based path-tree approaches for reachability queries.

6. ACKNOWLEDGMENTS

The authors would like to thank Silke Trißl for providing the GRIPP code and datasets, Jeffrey Xu Yu for providing implementations of various reachability algorithms, Victor Lee, David Fuhry and Chibuike Muoh for suggesting manuscript revisions, and the anonymous reviewers for their very helpful comments.

7. REPEATABILITY ASSESSMENT RESULT

A previous version of the code and results were validated for repeatability by the repeatability committee; the results in these conference proceedings reflect a later (improved) version, and this version has been archived.

Code and data used in the paper are available at http://www.sigmod.org/codearchive/sigmod2008/

8. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.

[2] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 493–504, 2005.

[3] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[4] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[5] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete algorithms*, pages 937–946, 2002.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[7] Mark de Berg, M. van Krefeld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[8] J. Edmonds. Optimum branchings. *J. Research of the National Bureau of Standards*, 71B:233–240, 1967.

[9] H N Gabow, Z Galil, T Spencer, and R E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[10] A. V. Goldberg, E. Tardos, and R. E. Tarjan. *Network Flow Algorithms*, pages 101–164. Springer Verlag, 1990.

[11] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[12] T. Kameda. On the vector representation of the reachability in planar directed graphs. *Information Processing Letters*, 3(3), January 1975.

[13] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An efficient connection index for complex XML document collections. In *EDBT*, 2004.

[14] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1-3):325–346, 1988.

[15] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 845–856, 2007.

[16] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 75, 2006.