A hierarchical approach to reachability query answering in very large graph databases

Conference Paper · January 2010

DOI: 10.1145/1871437.1871625 · Source: DBLP

Hasan M. Jamil
University of Idaho
167 PUBLICATIONS **847** CITATIONS
SEE PROFILE

Paper Number 1300

A Hierarchical Approach to Reachability Query Answering in Very Large Graph Databases [*]

Saikat Kumer Dey
Department of Computer Science
Wayne State University, USA
saikat@wayne.edu

Hasan Jamil
Department of Computer Science
Wayne State University, USA
jamil@cs.wayne.edu

ABSTRACT

The cost of reachability query computation using traditional algorithms such as depth first search or transitive closure has been found to be prohibitive and unacceptable in massive graphs such as biological interaction networks, or pathways. Contemporary solutions mainly take two distinct approaches - precompute reachability in the form of transitive closure (trade space for time) or use state space search (trade time for space). A middle ground among the two approaches has recently gained popularity. It precomputes part of the reachability information as a complex index so that most queries can be answered within a reasonable time. In this approach, the main cost now is creation of the index, and response generation using the index, and the space needed to materialize the index. Most contemporary solutions favor a combination of these costs to be efficient for a class of applications. In this paper, we propose a hierarchical index based on graph segmentation to reduce index size without sacrificing query efficiency. We demonstrate experimentally that our approach can save more than 25% space, and thus improve efficiency. The algorithm proposed to build the index is also significantly faster without compromising efficiency. We also show that our index need not be rebuilt for a large class of updates, a feature missing in all other contemporary systems.

1. INTRODUCTION

Reachability in massive graphs has always been a challenging question partly because neither online query processing nor index pre-computation offer a pragmatic solution for this well researched problem. Online query processing has unacceptable time complexity, while index precomputation suffers from very high space complexity. So, there has always been a need for innovative methods that will answer graph reachability within a short time while having a reasonable index size. In this article, we propose a novel approach to achieve this goal. By graph reachability we mean: *given two vertices u and v in a directed graph G, is there a path from u to v in G?*

Graph reachability has its applicability in many diverse areas ranging from biological domain to social networks. But in this research, our main focus is to facilitate the analysis of ever increasing data in the life sciences domain. It is self-evident that if we want to analyze and comprehend the complex phenomena that take place in living organisms, the corresponding data must have a computational representation. Fortunately, biological systems such as metabolic pathways [10], protein-protein-interaction networks [16] have an inherent graph oriented representation. Molecules and enzymes can be represented as nodes, while edges can be used to depict relationships or interactions, such as catalysis or gene regulation. Typically, these biological networks are very large in size, with tens of thousands of nodes. Moreover, if we move from molecular organisms (like bacteria) to higher level organisms (like humans), the size of the graph increases rapidly.

Helden and his colleagues pointed out (in [18]) many important queries applicable to these biological networks. One such query is "find all genes whose expression is directly or indirectly influenced by a given molecule". This query can be easily mapped as a reachability query on a directed graph that has genes as nodes and regulations as edges. Similar queries are:

- List the reactions catalyzed by a given protein.
- Find whether a reaction is influenced by a specific gene product.
- Find a metabolic pathway that converts compound A into compound B in less than X steps.
- Find a gene whose expression is affected by a given compound.

All these queries can be mapped to graph reachability query. So, efficient answering of graph reachability can be a great boost for biological network analysis.

Besides biological networks, social networks (e.g. a terrorist network) can be modeled as graphs, and reachability query can play a vital role in that domain also. Terrorists can be represented as nodes and each node can be annotated with

[*]Research supported in part by National Science Foundation grants XXXXXXX and YYYYYYY.

the genre of crimes committed by that individual. Edges, on the other hand, can depict any common relationship existing between two corresponding terrorists. So, queries like "Whether terrorist A is an associate of terrorist B for a specific kind of crime" can be mapped as a reachability query on this kind of graph.

Despite its applicability in many diverse areas of practical interest, graph reachability lacks a pragmatic solution that can answer reachability queries speedily while requiring a moderate size of indexing. One intuitive and traditional way to answer reachability queries is to traverse the graph recursively at query time (using Depth First Search (DFS) or Breadth First Search (BFS)) until the desired node is reached (reachable), or no more edges remain to traverse (not reachable) [6]. For a graph with n nodes and m edges this lookup has a complexity of $O(m)$. Though no index is required, for graphs with a large number of edges (like biological networks), it is not feasible to run depth first search or breadth first search each time a query appears.

Another extreme is to pre-compute the *Transitive Closure* (TC) of the whole graph. If transitive closure is available, then reachability queries can be answered in $O(1)$ time. The main downside of this approach is the size of the necessary index. For a graph with n nodes, the size of transitive closure is $O(n^2)$, which may not be a problem for smaller graphs, but for larger graphs, it can be overwhelming. Moreover, in case of node (or edge) addition (or deletion), the entire index has to be recalculated which takes $O(n^3)$ time.

Some recent approaches tried to overcome the difficulties faced by depth first searching and transitive closure precomputation. In [4] (Labeling+SSPI) Chen et al. index only the spanning tree of the graph, while *non-tree edges* (edges excluded from the spanning tree) are stored in a separate index structure (called SSPI). Index size is $O(n+m)$, but the problem is that a portion of the index must be traversed recursively at query time. The Dual Labeling approach [19] by Wang et al. can answer queries in $O(1)$ time. Their method is also based on building a spanning tree (for answering tree reachability), and a condensed transitive closure over the remaining t non-tree edges (edges outside the spanning tree). Dual Labeling requires $O(n + m + t^3)$ time to build the non-tree index and its space complexity is $O(n + t^2)$. For sparse graphs (with $t \ll n$) this approach works fine, but for denser graphs ($t > 2n$) it also requires a prohibitively large amount of indexing. In [17], Trissl et al. strived to eliminate the indexing problem featured by Dual Labeling. Their method (GRIPP) also used spanning tree labeling for answering tree reachability. Unlike Wang's method, Trissl et al. did not create any index on the non-tree edges. Rather, they used an iterative search approach for answering non-tree reachability, hence lower index size ($O(n + m)$). In a nutshell, Dual Labeling can answer reachability in $O(1)$ time but has indexing problem as it has to compute transitive closure on non-tree edges. On the other hand, GRIPP has no space problem, but does recursive searching for answering non-tree reachability. So, GRIPP has lower index size, but takes more time ($O(m - n)$) to answer reachability queries involving non-tree edges.

Our Contributions: Recent methods for answering graph reachability are hindered either by the size of the index (e.g. [19]) or by the necessity of performing some recursive searches when a query arrives (e.g. [17]). In this research, we propose a novel approach for answering graph reachability that will reduce the size of the index and at the same time, no recursive searching will be needed during query time. Our approach is similar to *Dual Labeling* and *GRIPP* in a sense that we also perform interval based labeling on the underlying spanning tree. Unlike both these approahes, we neither create a transitive closure on the non-tree edges (edges excluded from the spanning tree), nor do we perform recursive searching during query time. On the contrary, we employ a novel method based on graph segmentation to approach reachability answering. We use the non-tree edges to segment the tree cover into multiple autonomous segments, and then maintain a global segment graph to preserve reachability among the partitioned segments. Through this segmentation we reduce the number of nodes (in comparison to [19]) for whom non-tree reachability has to be stored. Moreover, when we make the segment graph from the original graph, for multiple non-tree edges in the original graph, there can be only one non-tree edge in the segment graph. As we create *transitive closure* on the non-tree edges of the segment graph, our index size and index creation time will be smaller and faster respectively than those of Dual Labeling [19]. The indexing time and space complexity for our approach is $O(m+t+m_s+{t_s}^3)$ and $O(n+s+{t_s}^2)$, where s is the number of segments, m_s is the number of edges in the segment graph and t_s is the number of non-tree edges in the segment graph. We will show through experiment that in case of sparse biological graphs, t_s will be lot smaller than t. As both time and space complexities are dominated by the cubic (${t_s}^3$) and square terms (${t_s}^2$) respectively, our approach will have better indexing performance than Dual Labeling (when $t_s \ll t$). We also show through experiment that our method has acceptable query performance, though it is not $O(1)$. Moreover, using the benefit of the segmentation, updates in the graph can be easily incorporated. In contrast, the entire index structure has to be recomputed (in presence of an update) if we use [17] or [19].

The rest of the paper is organized as follows. In section 2, we briefly describe relevant research in the area of graph reachability. We introduce our approach to answering graph reachability based on a graph segmentation approach in Section 3. We present a complete set of algorithms for the creation of index structure based on segmentation and establish theoretical correctness results. In Section 4, we discuss experimental results and perform comparative analysis with leading contemporary systems and approaches. We show that we achieve substantial space reduction and show that our algorithm is significantly faster than all leading algorithms. Finally, we conlcude in section 5.

2. RELATED RESEARCH

There have been many works in the field of graph reachability, ranging from simple depth first search based method to path based segmentation of the graph. Let $G = (V, E)$ be the graph for reachability query. Here, n is the number of vertices ($n = |V|$) and m is the number of edges ($m = |E|$). And t is the number of (non-tree) edges remaining after removing all the edges of a spanning tree of G.

DFS/BFS and Transitive Closure Computation: At first, we describe two extreme approaches to query time and index size. One way to answer reachability questions is to run depth first search or breadth first search each time a query arrives. Breadth first search and depth first search require online traversal of the graph and may take up to $O(n+m)$ time to answer. For very large graphs, like metabolic pathways and social networks, however, running depth first search and breadth first search for every query is not feasible (because of the time needed).

Another way of answering reachability query is to pre-compute transitive closure for the entire graph [3]. The transitive closure of a graph with n vertices is a $(n\times\ n)$ matrix where if (u,v)=1 then v is reachable from u; otherwise not. The main problem with transitive closure is that it requires $O(n^2)$ storage and its computation time is $O(n^3)$, making it impractical for large graphs. For example, using the method described in [13] the computation of transitive closure on a graph of 50,000 nodes and 100,000 edges did not finish in 24 hours [17]. Tackling the storage and computation cost of transitive closure is an popular ongoing research field. There are several compression technique available that reduces the storage cost of transitive closure. But, this compression comes at the cost of reduced query answering time. So, researchers are tying to find a balance between transitive closure compression and query answering time.

Chain Decomposition Approach: Chains have been used before to improve the efficiency of transitive closure computation [15] and for compressing the transitive closure matrix [7]. The idea behind chain decomposition is to partition the given *Directed Acyclic Graph* (DAG) G into several pairwise disjoint chains so that each vertex appears in one and only one chain. Every vertex is then labeled with the corresponding chain number and its sequence number in the chain. Then for each vertex v and each chain c, at most one vertex u is recorded such that u has the smallest sequence number (so the closest) on chain c that is reachable from v. For answering whether any vertex u reaches any vertex v, we only need to check if u reaches any vertex p in v's chain and p has a smaller sequence number than v. Simon used chain decomposition to compute transitive closure and achieved worst case complexity of $O(k.e_{red})$. Here k is the width of the chain decomposition (number of chains) and e_{red} stands for the number of edges in the transitive reduction of G. The transitive reduction of a graph G is the smallest subgraph of G with the same transitive closure. Jagadish et al. reduced the size of the transitive closure by using chain decomposition [7]. He transformed the chain decomposition problem into a network flow problem which he solved in $O(n^3)$ complexity (n = number of vertices).

By compressing the transitive closure, chain decomposition can help to answer reachability queries. Its disadvantage is the compression rate which is limited by the constraint that each node can have only one immediate successor. In biological networks, even in sparse ones, each node can have more than one successor. Chain decomposition considers only one of them. Hence, this approach cannot be applied to biological and social networks.

Optimal Tree Cover: Some recent methods focus on the tree cover approach and try to focus on improving either the query processing time or (and) provide a smaller index size. Wang et al. [19] developed a Dual-Labeling approach which improves the query time and index size. However, the graph has to be sparse, meaning that the number of non-tree edges t is much smaller than the number of vertices n ($t \ll n$). They reduced the index size to $O(n+t^2)$ and achieved constant query answering time. But for many real world graphs, this condition (being sparse) is not true, and when $t > n$, the index size remains uncompressed.

Trissl et al. developed a method called GRIPP [17] that uses depth first search labeling of the graph vertices to answer reachability query . GRIPP extracts the tree cover of the graph using depth first search. In case of reachability query comprising only tree edges, it can answer in $O(1)$ time. But for reachability queries having non-tree edges it starts an recursive search with heuristics. So, while index creation time and index size are low for this method, it sacrifices query speed which can be a problem for larger graphs.

2-Hop Approach: An approach quite different from the previously mentioned tree covering approaches is the 2-hop labeling method proposed by Cohen in [5]. In 2-hop approach each node u is assigned two labels, $C_{in}(u)$ and $C_{out}(u)$, where $C_{in}(u)$ is the set of nodes that can reach u, and $C_{out}(u)$ is the set of nodes reachable from u. Then v is reachable from u if $C_{out}(u) \cap C_{in}(v) \neq \phi$. The overall index size in this approach is $O(nm^{1/2})$ which in worst cases can rise upto $O(n^2)$ (same as transitive closure). Answering reachability query may take upto $O(m^{1/2})$ as the average size of each index (label) is $O(m^{1/2})$.

The main problem with the 2-hop labeling approach is its labeling time. Finding optimum 2-hop labeling can be reduced to solving the weighted set covering problem, which is NP-hard. Cohen et al. (in [5]) used an approximation algorithm based on greedy approach to device a labeling scheme. Still, the scheme is extremely time consuming for large datasets as the complexity is in the range of $O(n^4)$. Several other approaches have been devised to improve 2-hop labeling scheme. Schenkel et al. proposed the HOPI algorithm [14] which reduces the labeling complexity from $O(n^4)$ to $O(n^3)$. But this complexity is still inapplicable for larger graphs.

Path Tree Approach: The path tree approach, proposed by Rouming Jin et al. in [8], was motivated by the fact that the computational cost of finding a good tree can be expensive. In Agrawal's algorithm [2] the cost of finding an optimal tree cover is $O(mn)$. According to them, the tree structures are too strict and limited to express many different types of DAGs even the sparse ones. Most existing approaches are hindered by the number of edges left uncovered (non-tree edges). To overcome these challenges, they proposed a novel graph representation (called Path Tree) to cover the DAG. They created a tree structure where each node represents a path of the original graph. Then a labeling scheme was used to given each node a 3-tuple label. They showed that a good path-tree cover can be constructed in $O(m+nlogn)$ time. They also proved that the path-tree cover can always perform the compression of transitive closure.

3. A NOVEL INDEX STRUCTURE BASED ON GRAPH SEGMENTATION

The method we propose can be divided into three parts. First, we create the tree cover of the given network G using depth first search and simultaneously creating the list of non-tree edges. Second, we segment the tree cover based on the non-tree edges, and thus creating the segment graph G_s. Finally, we apply the first step on the segment graph and create the transitive closure of the non-tree edges of G_s. When the index structure is ready, we can answer reachability queries using G_s first, and then the corresponding segments. In our work, we focus on *Directed Acyclic Graph*. However, this does not sacrifice the generality of our method in any way since a cyclic graph can be converted into a DAG by coalescing each strongly connected component into a node.

3.1 Building and Labeling the Tree Cover

We used depth first traversal for building the tree cover (T) of the graph and labeling its nodes. We assume that the graph has exactly one root node, i.e., one node without incoming edges. For graphs with multiple root nodes, we can make one virtual root node and apply our method. The graph is represented in adjacency list form . Thus, there is a fixed order among the child nodes (achieved using node ID).

Depth first search starts from the root node. During depth first search of G, each node is assigned a preorder and a postorder value. The children are traversed according to their order. A node v with n incoming edges, where $n > 1$, is reached n times on edges e_i, $1 \leq i \leq n$. The edge e_i on which we reach v for the first time is called a tree edge. We assign a preorder value to v and proceed up the depth-first traversal. After all the child nodes have been labeled, v receives its postorder value. Then we reach v for $n-1$ times more. Suppose we arrive at v over an edge e_j , $e_j \neq e_i$. Then, e_j will be labeled as a non-tree edge. For all non-tree edges we prune the depth first search traversal. So, when we reach v using all $e_j, j \neq i$; we prune the depth first search traversal and move to the subsequent path. In this way, each node of the graph will have a unique label.

The reachability information contained in graph G and tree cover are not the same. Since, tree cover answers reachability queries using tree edges only, it knows nothing about the non-tree edges which also contain crucial reachability information. Thus, in addition to the tree, we must also keep track of the non-tree edges (edges outside the tree). So during depth first search, if there is a non-tree edge from a node labeled $[a, b]$ to a node labeled $[c, d]$, then we record the edge in a link table (called Non-Tree Link Table). We denote this link by: $a \to [c, d]$. In this fashion, all the nontree edges are collected in a list during the depth first search traversal. The entire labeling mechanism is outlined in Algorithm 1.

Definition 1 (Tree and Non-tree Edge): *Let G be the graph representation of the network in question and $(u, v) \in E[G]$. Then (u, v) is a tree edge if and only if v was first visited by the edge (u, v) using Algorithm 1. Otherwise it is a non-tree edge.*

Algorithm 1: DFS(u)

```
Data: Node $u$
Result: Non-tree edge list $NTList$
begin
    $u \leftarrow visited$;
    $u.preOrder \leftarrow label++$;
    foreach $v$ such that $v$ is a neighbor of $u$ do
        if $v$ not visited then
            DFS(v);
        else
            NTList = NTList ∪ (u,v);
        end
    end
    $u.postOrder \leftarrow label++$;
end
```

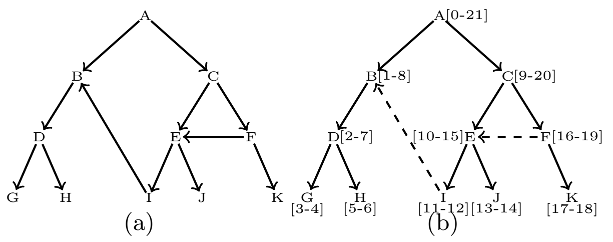


Figure 1: (a) Example Graph (b) Graph after labeling

Bellow is an example of the above mentioned Tree Cover Creation and Node Labeling method. The primary motivation behind using this labeling approach is that we answer (in $O(1)$ time) some straight forward reachability queries using only the tree labels:

Theorem 1: *Let u and v be two nodes of G. Let [a,b] and [c,d] be the pre-order and post-order values of u and v respectively. Now if $a \leq c < d \leq b$ then there must be a path between u and v using only the tree edges.*

The proof of Theorem 1 is intuitive as the labels are created using depth first search. If we look at the labels of node A and H (in figure 1(b)) the whole picture becomes clear.

As we mentioned earlier, we can infer additional reachability information using non-tree edges also. The following lemma demonstrates the condition for a path using non-tree edges.

Lemma 1: *Assume two nodes u and v are labeled $[a, b]$ and $[c, d]$ respectively. There is a non tree path from u to v iff the non-tree link table contains a series of m non-tree edges*

$$i_1 \to [j_1, k_1], ..., i_m \to [j_m, k_m] \qquad (1)$$

such that $i_1 \in [a, b]$, $c \in [j_m, k_m]$, and $i_{m'} \in [j_{m'-1}, k_{m'-1}]$, for all $1 < m' \leq m$.

Proof: Proof for this theorem can be developed analogously to the proof in [19], and hence omitted.

As in the example above, the path from F to E involves the non-tree edge $16 \to [10, 15]$, and the path from F to G involves two non-tree edges $16 \to [10, 15]$ and $11 \to [1, 8]$.

But unlike Wang et al. [19], we do not create index on original graph non-tree edges to answer reachability. Rather, we use the non-tree edges to partition the graph into multiple segments. Hence a higher level segment graph is derived (implying the use of a hierarchical approach). This means that reachability (in original graph) through non-tree edges is propagated to the segment layer. As a result of this propagation, the number of non-tree edges in the segment graph can be considerably smaller than the number of non-tree edges in the original data graph, leading to a smaller index size. But no reachability information is lost through this segmentation. The segmentation process is described in detail in the next section.

3.2 Segmentation of the Graph

The entire data graph is made up of an underlying spanning tree and a set of non tree edges. Each non-tree edge adds some extra reachability among the subtrees in the underlying spanning tree. In some previous works (Wang et al. in [19]), this extra reachability information is stored in all nodes of the two subtrees the corresponding non-tree edge is linking. We deduce however, that there is no need to propagate this reachability information to that many nodes. As each non tree link joins two subtrees of the spanning tree, we can think of each subtree as a single entity (a single node in our case), not as a collection of nodes. So, we don't have to propagate reachability information to all the nodes. We just shred the spanning tree into multiple segments (based on the non-tree links) and maintain an upper level graph, called the *Segment Graph*, G_s. The segment graph tells us whether segment S_1 is reachable from segment S_2 using tree or non-tree edges. During this segmentation process we can also reduce the number of non-tree edges propagating from the original graph to the segment graph. We will present this phenomenon in the discussion section. Now we present the segmentation procedure in a more formal way.

First we want to establish the claim that the list of non-tree edges received after the tree covering algorithm are ordered in a specific direction. That is, either they are ordered from right to left (if the leftmost subtree is traversed first in depth first search), or they are ordered from left to right (if the rightmost subtree is traversed first in depth first search). If the leftmost subtree is traversed first in depth first search, there can be no non-tree edge from left to right direction, as it is going to be a tree edge in that case. The same is true for rightmost depth first search traversal. As non-tree edges are collected during depth first search traversal, we can safely say that they will be ordered in a specific direction. Without loss of generality, we assume that they are ordered from right to left direction. Figure 2(a) and figure 2(b) demonstrate this issue.

Segmentation Method: Take the non-tree edges from right to left, which is possible as they are ordered in that fashion. If (u, v) is a non-tree edge then we make the tree rooted at v a new segment (a new node in G_s) unless v has already been made the root of an existing segment (node already in G_s). For each (u, v), we add an edge (segment(u), segment(v)) in the segment graph unless there is already such an edge in G_s. In that case, we just keep the record that (segment(u), segment(v)) $\in G_s$ actually represent more than one edge in G. Lastly, we mark (u, v) as a boundary edge for segment(u).

An example of the segmentation process is given in figure 2(c). This example uses the graph in figure 1(b). The overall algorithm is given in Algorithm 2. As the picture shows, the non-tree edges 16→[10,15],11→[1,8] are ordered right to left. When we reach 16→[10,15], we make the tree rooted at E a new segment. In the same way, for 11→[1,8], we make the tree rooted at B a new segment. So, at the end we have three segments S_0=$\{A, C, F, K\}$, S_1=$\{E, I, J\}$ and $S_2 = \{B, D, G, H\}$. The segment graph will be like figure 2(d). In the segment graph G_s, the edge (S_1,S_2) represent two edges in G; (C, E) and (F, E). In similar manner, (S_0,S_2) represents (A, B) and (S_1,S_2) represents (I, B). The segmentation algorithm is presented in Algorithm 2.

3.3 Labeling G_s and Creating Transitive Link Table

After creating the hierarchical segment graph, we now have to devise a mechanism so that we can answer reachability in the segment graph G_s also. We use the same method that we employed in the data graph G. The interval based node labeling method creates the spanning tree of G_s (using depth first search). As in the case of G, a list of non-tree edges of G_s is also created during this time.

We have described earlier (in section 3.1) how interval based node labeling can be used to answer reachability queries that only use tree edges. That same method is also used here. For reachability using non-tree edges, Lemma 1 specifies the necessary condition. As we stated earlier, we do not use the non-tree reachability information directly on the data graph. Rather we propagate this information to the segment graph G_s (which has lesser nodes) and use it to answer reachability there.

It is evident from Lemma 1 that for reachability using non-tree edge, we need to find a sequence of non-tree edges that satisfy the following condition, assuming the number of non-tree edges in the path are m.

$$i_{m'} \in [j_{m'-1}, k_{m'-1}], \text{ for all } 1 < m' \leq m \qquad (2)$$

Using Lemma 1 to answer reachability query requires traversing and exploring the non-tree edges in an iterative fashion, which can be extremely costly. To do the search efficiently, we compute the transitive closure of the link table. Thus, given two non-tree links $p_1 \to [q_1, r_1]$ and $p_2 \to [q_2, r_2]$ in the link table, if $p_2 \in [q_1, r_1]$, we add a new link $p_1 \to [q2, r2]$ to the table. For example, consider the graph in figure 2(c) (though its G, but the same is applicable to G_s). It contains two non-tree edges 16→[10,15] and 11→[1,8]. Now, as 11∈[10,15], 16→[1,8] is also in the transitive closure. This process is repeated until no new links can be found. The resulting table is called the transitive link table. The theorem below directly follows from the definition of *Transitive Link Table* and Lemma 1.

Theorem 2: If nodes u and v are labeled $[a, b]$ and $[c, d]$ respectively, then there is a path between u and v iff $a < c$ and $d < b$, or there is a non-tree link $p \to [q, r]$ in the transitive link table such that $p \in [a, b]$ and $c \in [q, r]$.

So, for answering non-tree reachability, we create a transitive

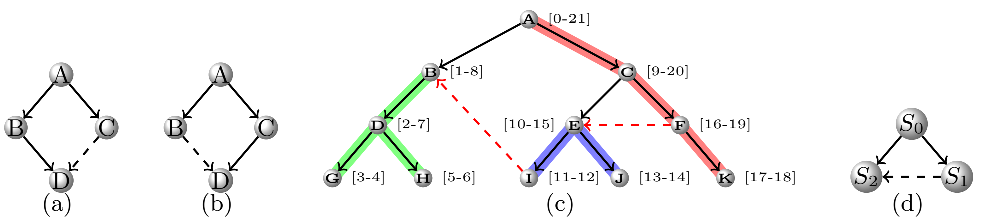


Figure 2: (a) Leftmost DFS (b) Rightmost DFS (c) Segmentation of figure 1(b) (d) Segment Graph G_s

```
Algorithm 2: CreateSegmentGraph(G , NTList)
Data: Graph G, Non-tree edge list NTList
begin
    forall the $(u, v)$ : $(u, v) \in NTList$ do
        if $v \neq$ root of subtree $T$ : $T \in$ any segment of $G_s$ then
            Create a new segment $S$;
            Make $v$ the root of the subtree represented by $S$;
            forall the $w$ : $w \in$ subtree rooted by $v$ do
                Make $S$ the segment of $w$;
            end
        end
        Add a new edge ($u$.segment,$v$.segment) in the segment graph $G_s$;
        Mark ($u$,$v$) as a boundary edge for $u$.segment;
    end
end
```

link table on the non-tree edges of the segment graph. The space requirement for the transitive link table is $O({t_s}^2)$ [19], where t_s is the number of non-tree edges in the segment graph. In the presence of a query, this link table will be searched for finding a non-tree link satisfying the later part of Theorem 2. This search will cost $O({t_s}^2)$ time, if we use linear searching. But, if we employ *Transitive Link Counting* proposed by Wang et al. [19], this search can be done in $O(1)$ time.

3.4 Answering Reachability Queries: The Overall Procedure

Now that all the index structures are ready, we can answer reachability query. At first, we check whether the source and the destination are in the same segment or not. If they are in the same segment, then we go to that segment and answer reachability using the interval labels of source and destination. As each segment is a tree, it is possible to answer reachability using interval based labels only. If they are in different segments, then let S_1 be the source segment and S_2 be the destination segment. We then check (using tree and non-tree link) whether there is a path from S_1 to S_2 in G_s (see Theorem 2). If not, they are unreachable. If yes, we check whether there is a boundary edge (u, v) of S_1 such that u is reachable from source and (u, v) leads a path from S_1 to S_2. If such an edge exists, then the two nodes are reachable, otherwise not. For each segment, we mark all boundary edges and which segment they lead to at segment creation time.

Let us now focus on figures 2(c) and 2(d). If $source = A$ and $destination = K$, then we go to segment S_0 as $segment(A) = segment(K) = S_0$. Then based on the labeling of A and K we can say that $reachability(A, K) =$

```
Algorithm 3: Reachability(source,destination)
Data: Node source, Node destination
begin
    if source.segment=destination.segment then
        ReachabilityIntervalBased(source,destination);
    else
        if source.segment is reachable from destination.segment then
            forall the $(u, v)$: (u,v) is a boundary edge leading to destination.segment do
                if ReachabilityIntervalBased(source,u) then
                    return True;
                end
            end
        end
        return False;
    end
end
ReachabilityIntervalBased(source, destination)
begin
    if source.preOrder<destination.preOrder && destination.postOrder<source.postOrder then
        return True;
    else
        return False;
    end
end
```

$true$. Similarly, we can say that $reachability(G, H) = false$. For $source = A$ and $destination = G$, we first see that $segment(A) = S_0$ and $segment(G) = S_2$ and there is a path between S_1 and S_2 in G_s. Now, (A, B) is a boundary edge of S_0 that leads a path from S_0 to S_2 and A is reachable from A. So, $reachability(A, G) = true$. The reachability answering algorithm is given in Algorithm 3

We assume that if two nodes are in the same segment, then their reachability can be answered using interval based labeling of those two nodes only. But it is not true for all segmentation. Please see at the segmentation in figure 3(a) and the segment graph in figure 3(b).

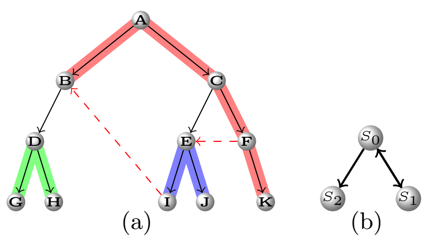


Figure 3: (a) Segmentation with cycle (b) Segment Graph G_s

Now, both B and C are in the same segment (S_0) and there is no path between them in S_0. But there is a path from C to B using segment S_1. Hence, only the labeling of B and C does not give right answer for this segmentation. Consequently, this kind of segmentation will not work for our method. A closer look at this example tells us that this kind of problem can only occur when the segment graph has a cycle (S_0, S_1, S_0). Thus, we have to find a segmentation method that will prevent the creating of cycle in the segment graph G_s. Next we prove that our segmentation method possesses that capability. But before we do so, we need to establish a few auxiliary results in the form of the theorem and the lemma below.

Theorem 3: *There can be no cycle in G_s if G is a tree.*

Proof: We prove this by contradiction. Let G_s be the segment graph of tree G and contains a cycle. Without losing any generality, lets assume that segment S is a part of that cycle. As S is a part of the cycle, we start from a vertex u in segment S and can come back to another vertex v also belonging to S. By definition, S is a subtree of G. Consequently, u and v have a common ancestor in S. So, the path from u to v completes a circuit in G. Which is a contradiction as G is a Tree and it cannot have a circuit.

Lemma 2: *If G_s has a cycle then there must be at least one non-tree edge present in the cycle.*

Proof: Directly follows from the proof of Theorem 3.

We are now ready to establish the soundness of our segmentation method and state our main result as the following theorem based on Theorem 3 and Lemma 2.

Theorem 4: *If G_s is created using the proposed segmentation method then it cannot have a cycle.*

Proof: We also prove this by contradiction. Let G_s be the segment graph created using Algorithm 2 and contains a cycle (see figure 4(a)). From *Lemma 2*, it can be said that the cycle has at least one non-tree edge. Let the edge be (S_2, S_1). That means S_2 has a vertex u and S_1 has a vertex v and (u, v) is a non tree edge of G. But whenever the proposed method finds (u, v) (and it will find (u, v)), it will make the subtree rooted at v a new segment (Let it be S_4). So, (S_2, S_1) will not be in G_s, rather (S_2, S_4) and (S_1, S_4) will be (see figure 4(b)). So, G_s cannot contain a cycle.

4. EXPERIMENTAL EVALUATION AND DISCUSSION

The primary aim of our method was to use a hierarchical segment graph to reduce the amount of precomputed information needed for answering reachability query. We only considered DAGs for our experiments as a cyclic graph can be converted into a DAG by coalescing the strongly connected components.

The motivating force behind our work is the fact that not all nodes in the graph may have a non-tree edge incident upon them. In that case, each non-tree edge basically connects unreachable subtrees in the underlying spanning tree.

Table 1: Compression of Graph Nodes

Node No.	Edge No.	Segment No.	Compression
5000	6000	912	81.76%
3000	3500	457	84.77%

Note that for answering reachability within a single subtree, we only need the pre-order and post-order labels of the corresponding nodes. Reachability achieved through non-tree edges are only needed when the corresponding nodes are in two different and unreachable subtrees. We argue that unlike Wang et al. [19], there is no need to propagate reachability achieved through non-tree edges to all the nodes. Rather we represent each subtree (joined by a non-tree edge) as a single node in a higher level segment graph where non-tree reachability is preserved. This hierarchical approach has twofold usefulness: one being the preservation of non-tree reachability in segment graph G_s, not in the original graph G. As the number of nodes in G_s are going to be a lot less than the total number of nodes in G (see Table 1), this approach saves space.

Looking at the the examples in figure 2(c) and figure 2(d), we see that segment S_2 is reachable from segment S_1 through a non-tree edge. But this information resides in G_s. There is no need for E,I, J to know that they can reach B, D, G, and H through a non-tree edge. That burden is taken away from them to S_1 and S_2 respectively requiring less information to be stored.

We used synthetic graphs of many size to test the level of compression that can be achieved using this hierarchical approach. The graphs were created using a tool built by Silke Trissl and used in [17]. If we examine Table 1, it can be seen that for a graph with 3000 nodes and 3500 edges, we achieved a compression level of 84.77% which means that the number of nodes in G_s are 15.23% of G. So, it is enough just to preserve intra-segment reachability for the nodes of G. Global reachability information (e.g. non-tree reachability) are stored in G_s which has notably lesser (15.23%) nodes in comparison to G. Wang et al. [19] stored such global reachability information for each graph node; hence their space requirement is significantly larger than our method. For a graph with 5000 nodes and 6000 edges, we achieve 81.76% compression. This decrease in compression level is due to the increase in edge-node ratio (number of edges per node). As the edge-node ratio increases, the possibility that a node has a non-tree incoming edge also increases. Hence, our system will perform worse when each node will have a non-tree incoming edge. In that case, G_s will have the same number of nodes as G; hence no reduction in space is achieved. But, not all real world graphs have every node with a non-tree incoming edge. There are many real world graphs (like Agrocyc, Xmark, etc.) which are sparse in nature. That means, graphs like *Agrocyc* comprise many stand-alone nodes (without any edge) as well as nodes with many incoming edges. Later in the section, we will show (through experimental results) the applicability of our approach in these graphs.

Another benefit of our approach is the update flexibility. As each segment is independent, some update operations can be performed without violating the entire index structure. If the update is entirely within a segment and it does not

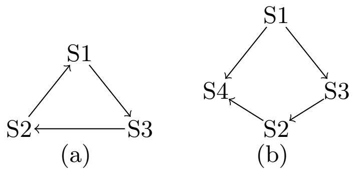


Figure 4: (a) Assumed cycle (b) Contradiction

hamper the tree property, then we just need to relabel the nodes of that segment only, not the entire graph. This is not the case for other tree cover based approaches like GRIPP [17] and Dual-Labeling [19]. They have to recompute the entire index structure to incorporate such updates.

For example in figure 2(c), if a new child is added to D (and it does not violate tree property of S_2), we just need to relabel the nodes of S_2, not the entire graph. If we use the update method described by Agarwal et al. in [2], we can even avoid relabeling the nodes of S_2.

For updates violating tree property within a segment (e.g. adding a non-tree edge), we just need to recompute reachability in the segment graph. Suppose, an edge is added to S_2 violating the tree property. Let the edge be (G,H); see figure 5. In that case, S_2 will get divided into two segments (as a non-tree edge has been added). Division of S_2 will create a new segment S_3 which will be connected to S_2 by edge (D, H) and (E, H). Now, the segment graph will be like figure 5. Then we need to relabel the segment graph, and in some cases recompute the transitive closure on the non-tree edges of G_s. As G_s is expected to have less number of nodes and non-tree edges than the original graph G, this update scheme is also computationally less costly. Also note that though we added a non-tree edge in segment S_2, the number of non-tree edges in G_s did not increase (see figure 5). As a result, transitive closure re-computation is not needed for this update.

Another important achievement of our hierarchical approach is that it considerably reduces the number of non-tree links without losing any reachability information. This reduction happens at the time of segment graph creation from the original data graph. Let S_1 and S_2 be two segments of the segment graph G_s. Now if there is more than one edge (in G) from vertices of S_1 to S_2, those multiple edges are represented as a single edge (in G_s) from S_1 to S_2. As multiple edges are represented by one edge, we achieve the aforementioned compression. From figure 5(c) and figure 5(d) this claim is quite evident. There are three edges from S_0 to S_1 in G; (C, G), (F, G) and (D, E). These three edges are represented by a single edge (S_0,S_1) in G_s. So, two non-tree edges get reduced when G_s is created from G. Dual Labeling [19] cannot get rid of these two non-tree edges even after applying Agrawal et al. [2] to find the optimum tree cover.

Reduction in the number of non-tree edges is very significant because we need to build the transitive closure on this list of edges. Building transitive closure is costly ($O(n^3)$) and consumes a lot of space ($O(n^2)$).So, reduction in the number of non-tree edges will reduce the index creation time. More importantly it will save space. Figure 6(a) shows the level of reduction achieved in non-tree edges, through our experiments. As transitive closure operation is very costly (both in time and space), the reduction that we have achieved is extremely significant.

To compare our approach to Dual Labeling [19], we used real life networks such as Agrocyc [11], Vchocyc [11], Hpycyc [11], Xmark [1], Ecoo157 [11]. As real life networks are not DAGs and are likely to contain cycles, we employed the preprocessing step used by Wang et al. [19] to merge the strongly connected components into representative nodes. Table 2 shows the compression achieved through merging the strongly connected components. The first two columns correspond to the number of vertices and edges in the original graph, while the last two columns represent the number of vertices and edges in the resulting DAG.

Table 2: Real Datasets

Graph	#V	#E	Dag #V	Dag #E
Agrocyc	13969	17694	12684	13408
Hpycyc	5565	8474	4771	5859
Vchocyc	10694	14207	9491	10143
Echo157	13800	17308	12620	13350
Xmark	6483	7654	6080	7028

We compare our approach against Dual Labeling [19] based on four criteria; (1) number of non-tree edges taking part in transitive closure computation, (2) Number of nodes having non-tree reachability information, (3) Index creation time and (4) Query time.

We have stated before that the proposed scheme reduces the number of non-tree edges without sacrificing reachability, resulting in smaller index size. This claim is established in figure 6(a). For all the datasets, segment graph G_s has significantly lower number of non-tree edges than the same in the original graph G. As Dual Labeling create index (transitive closure) on the non-tree edges of G, unlike our approach where index (transitive closure) is created on the non-tree edges of G_s, the proposed method has smaller index size and faster index creation time. For example, in case of Vchocyc, we achieve a reduction level of 94.75% which is a significant achievement when the computation time ($O(n^3)$) and space requirement ($O(n^2)$) is considered.

Moreover, Dual Labeling [19] propagates non-tree reachability to every node in the original graph G. But unlike Dual Labeling, the segmentation based approach that we propose needs to store non-tree reachability only in segment level nodes (i.e. nodes of G_s). If we examine figure 6(b), we see that G_s has significantly lesser number of nodes than G. So, our approach has notable advantage over Dual Labeling when the amount of information stored is considered. Hence index size will get reduced and index creation time will be faster.

Index creation time is an important issue with respect to massive biological networks. From figure 6(c) it is quite evident that our method outperforms Dual Labeling [19] by a substantial margin. In all datasets, the proposed approach achieved significantly faster index creation time. Through

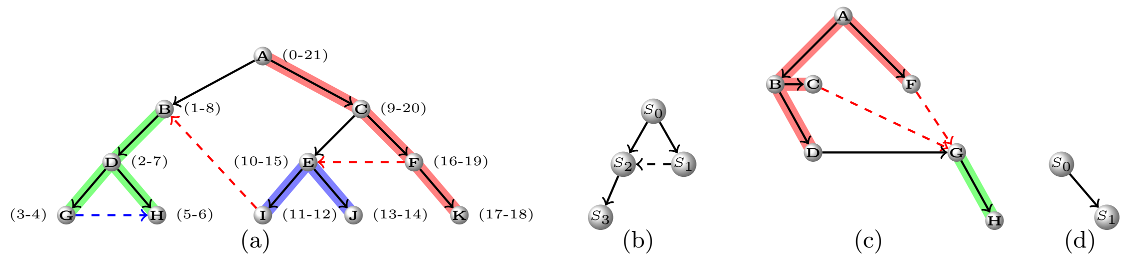
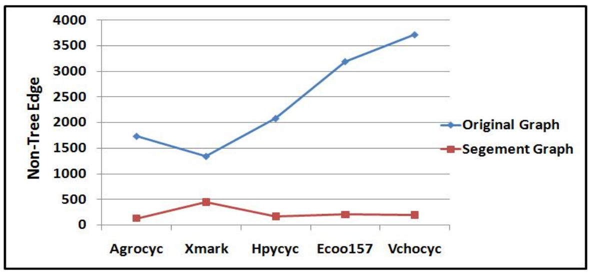
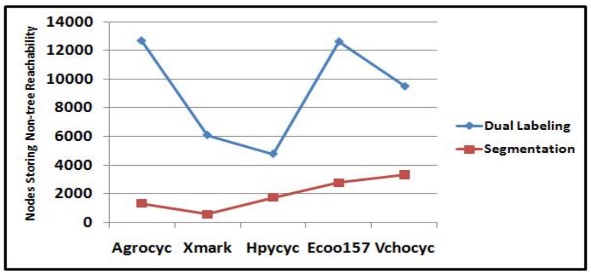


Figure 5: Update Flexibility and Non-tree Edge Compression

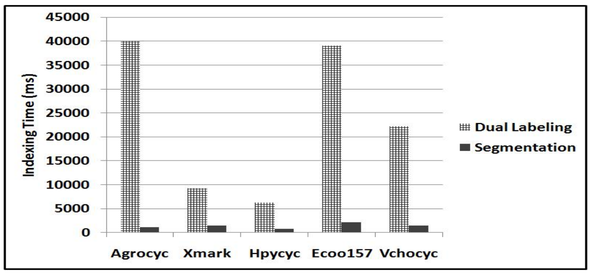
experiments performed on the datasets in figure 6(c), Wang et al. proved (in [19]) that Dual Labeling has better index creation time than Interval Based Labeling and 2-Hop cover. So, by applying the law of transitivity we can say that our method has better index creation time than those approaches also.



(a)



(b)



(c)

Figure 6: (a) Index Size (b) Node Compression (c) Index Computation Time

When query time is taken into account, Dual Labeling appears to perform better than our method. It was reported that Dual Labeling requires on the average 0.8 ms regardless the size of the graph [17][1]. In our case, query time range from 0.5ms to 3.8ms. The possible reason for Dual Labeling performing better is that it creates index for each individual node. As a result, it can answer queries in $O(1)$ time, but at the expense of higher index size and slower index creation time. In this research, we have attempted to reduce the index size and index creation time while having acceptable query time. Though we have slightly higher query processing time than Dual Labeling, we still achieve comparable performance when compared against GRIPP [17], a wellknown method for answering reachability. GRIPP was compared against our approach using two real life graphs; Reactome [9] and Amaze [12]. Query time was measured by averaging the time required to answer 100,000 randomly generated queries. Table 3 shows that our approach has the similar or better performance as GRIPP[2].

Table 3: Comparing Query Time

Graph	Segmentation	GRIPP
Reactome	0.7 ms	4.63 ±4.016 ms
Amaze	5.07 ms	3.43 ±1.597 ms

The proposed method has a space complexity of $O(n+n_s+{t_s}^2)$, where n is the number of nodes in G, n_s is the number of nodes in G_s and t_s refers to the number of non-tree edges in G_s. Interval based labeling of the original graph nodes contribute $O(n)$, while $O(n_s)$ account for the segment graph as well as labeling of the nodes of G_s. Lastly, $O({t_s}^2)$ corresponds to the asymptotic space complexity of the transitive link table. For sparse graphs, n_s will be a lot smaller than n (see figure 6(b)). So, $O(n+n_s+{t_s}^2) \approx O(n+{t_s}^2)$, for sparse graphs. Now, we also know that for sparse graphs, $t_s \ll t$ where t is the number of non-tree edges in the original graph (see figure 6(a)). So, our method has smaller space complexity compared to Dual Labeling ($O(n+t^2)$) with respect to massive sparse graphs. The comparison shown in figure 7 substantiates the previous statement. We would like to add here that many biological networks, such as Agrocyc, and Vchocyc are sparse graphs.

It is clear that our approach offers an interesting and novel way to answer reachability queries. No previous work has approached graph reachability by segmenting the original graph. Through segmentation, we reduce the index size and creation time while having acceptable query time. In this section, we have demonstrated the applicability of our method in many real world graphs, especially in massive

[1]We should mention here that these times are one different machines. Although we have used the same dataset, we did not have access to the implementation to compare both on our lab machines.

[2]Again, we are using reported time in [17] and not actual computation times in our own machine.

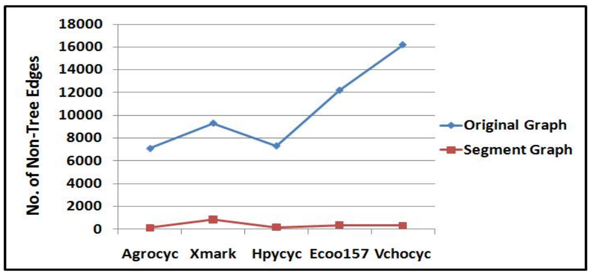


Figure 7: Size of transitive closure

sparse graphs. In future, we plan to modify the segmentation process so that we do not have to propagate every non-tree edge to the segment graph. We will try to propagate only those non-tree edges that can create a cycle in G_s. In that way, our method will work fine for denser massive graphs.

5. CONCLUSION

Biological and social networks are being discovered rapidly now a days. To analyze these complex networks efficiently and quickly we need innovative computational methods. Many such analysis need correct answering of graph reachability queries within a reasonable time. Due to the size of these massive graphs, conventional reachability answering algorithms like depth first search or transitive closure precomputation do not give any practical solution. One highly acclaimed approach called 2-hop labeling has relatively efficient query performance, but very high indexing complexity, which prevents it from being used on massive graphs. Wang et al. in [19] and Trissl et al. in [17] proposed a tree cover based approach, but the applicability of their method is hindered by index creation time (and size) and the need for recursive searching during query time respectively. In this work, we propose an approach based on graph segmentation that will reduce index creation time as well as index size for massive sparse graphs, and will answer reachability query within very short time (no recursive searching is necessary). No previous approach has tried to answer reachability through segmentation; hence, we bring a new dimension to this highly researched topic. Furthermore, our method allows flexible updates. For certain kinds of updates, we can avoid recomputing the entire index structure; a noteworthy advantage over existing methods like Dual Labeling [19] and GRIPP [17].

6. REFERENCES

[1] *XMARK: The XML-benchmark project.* http://monetdb.cwi.nl/ xml, 2002.

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

[3] Agrawal, R., and Jagadish, H. V. Direct algorithms for computing the transitive closure of database relations. *VLDB* (1987), 255–266.

[4] Chen, L., Gupta, A., and Kuru, M. E. Stack-based algorithms for pattern matching on dags. *VLDB* (2005), 493–504.

[5] Cohen, E., Halperin, E., Kaplan, H., and Zwick, U. Reachability and distance queries via 2-hop labels. *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete algorithms* (2002), 937–946.

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

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

[8] Jin, R., Xiang, Y., Ruan, N., and Wang, H. Efficiently answering reachability queries on very large directed graphs. *SIGMOD* (2008), 595–608.

[9] Joshi-Tope, G., Gillespie, M., Vastrik, I., D'Eustachio, P., Schmidt, E., de Bono, B., Jassal, B., Gopinath, G., Wu, G., Matthews, L., Lewis, S., Birney, E., and Stein, L. Reactome: a knowledgebase of biological pathways. *Nucl. Acids Res. 33*, suppl-1 (2005), D428–432.

[10] Kanehisa, M., Goto, S., Kawashima, S., Okuno, Y., and Hattori, M. The KEGG resource for deciphering the genome. *Nucl. Acids Res. 32*, suppl-1 (2004), D277–280.

[11] Keseler, I. M., Collado-Vides, J., Gama-Castro, S., Ingraham, J., Paley, S., Paulsen, I. T., Peralta-Gil, M., and Karp, P. D. EcoCyc: a comprehensive database resource for Escherichia coli. *Nucl. Acids Res. 33*, suppl-1 (2005), D334–337.

[12] Lemer, C., Antezana, E., Couche, F., Fays, F., Santolaria, X., Janky, R., Deville, Y., Richelle, J., and Wodak, S. J. The aMAZE LightBench: a web interface to a relational database of cellular processes. *Nucl. Acids Res. 32*, suppl-1 (2004), D443–448.

[13] Lu, H. New strategies for computing the transitive closure of a database relation. *VLDB* (1987), 267–274.

[14] Schenkel, R., Theobald, A., and Weikum, G. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. *In Proceedings of the 21st International Conference on Data Engineering (ICDE)* (2005), 360–371.

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

[16] Stelzl, U., et al. A Human Protein-Protein Interaction Network: A Resource for Annotating the Proteome. *Cell 122*, 6 (2005), 957Ű–968.

[17] Trissl, S., and Leser, U. Fast and practical indexing and querying of very large graphs. *SIGMOD 15*, 5 (june 2007), 795–825.

[18] van Helden, J., Naim, A., Mancuso, R., and Eldridge, M. Representing and analysing molecular and cellular function using the computer. *Journal of Biological Chemistry 381*, 9–10 (Sep–Oct 2000), 921–935.

[19] Wang, H., He, H., Yang, J., Yu, P. S., and Yu, J. X. Dual labeling: Answering graph reacability query in constant time. *Proceedings of the 22nd International Conference on Data Engineering* (2006), 75.