Efficient Graph Reachability Query Answering using Tree Decomposition

Fang Wei

Computer Science Department, University of Freiburg, Germany

Abstract. Efficient reachability query answering in large directed graphs has been intensively investigated because of its fundamental importance in many application fields such as XML data processing, ontology reasoning and bioinformatics.
In this paper, we present a novel indexing method based on the concept of tree decomposition. We show analytically that this intuitive approach is both time and space efficient. We demonstrate empirically the efficiency and the effectiveness of our method.

1 Introduction

Querying and manipulating large scale graph-like data has attracted much attention in the database community, due to the wide application areas of graph data, such as GIS, XML databases, bioinformatics, social network, and ontologies.

The problem of reachability test in a directed graph is among the fundamental operations on the graph data. Given a digraph $G = (V, E)$ and $u, v \in V$, a reachability query, denoted as $u \rightarrow v$, ask: is there a path from u to v? One of the fundamental queries on biological networks is for instance, to *find all genes whose expressions are directly or indirectly influenced by a given molecule* [15]. Given the graph representation of the genes and regulation events, the question can also be reduced to the reachability query in a directed graph.

Recently, tree decomposition methodologies have been successfully applied to solving shortest path query answering over undirected graphs [17]. Briefly stated, the vertices in a graph G are decomposed into a tree in which each node contains a set of vertices in G. Different from other partitioning based methods, there are overlapping between the tree nodes, i.e., for any vertex v in G, there could be more than one node in the tree which contains v. However, it is required that all these nodes constitute a connected subtree (see Definition 1 for the formal definition). Based on this decomposed structure, many otherwise intractable problems can be solved if the underlying tree decomposition has bounded treewidth.

In this paper we make an attempt to solve reachability problems over directed graphs by using tree decomposition based index structures. In comparison to shortest path queries, reachability query answering enjoys some nice properties. For instance, the existing BFS or DFS algorithms are highly efficient. However, these properties might cause challenging problems to occur, if

substantial improvement on time complexity is desired. Note that one extreme scheme is to store all the transitive closures in the pre-processing stage, thus the reachability queries can be answered in constant time. However this requires an index of size $O(n^2)$, which is unrealistic for large scale graphs. Therefore, finding a better trade-off between time and storage is the ultimate goal of many reachability query answering algorithms. Surprisingly, we have found that the tree decomposition-based methodology can be adapted on directed graphs and moreover, the efficiency of the query algorithm is substantially improved, based on the index which is much smaller than $O(n^2)$. Our main contributions are the following:

- **Linear time tree decomposition algorithm.** In spite of the theoretical importance of the tree decomposition concept, many results are practically useless due to the fact that finding a tree decomposition with optimal treewidth is an NP-hard problem, w.r.t. the size of the graph. To overcome this difficulty, we propose a simple heuristics to achieve a linear time tree decomposition algorithm.
- **Flexibility of balancing the time and space efficiency.** From the proposed tree decomposition algorithm, we discover an important correlation between the query time and the index size. This flexibility enables the users to choose the best time/space trade-off according to the system requirements.

1.1 Related Work

Most of the current research of reachability query answering concentrates on methods that first build an index structure to store part of the transitive closures, then speed up the query answering process, thus to find better trade-offs of index size and the query answering time. They can be categorized into the two main groups. The first group of algorithms are based on the 2-Hop approach first proposed by Cohen et al. [6]. The second are based on the *interval labeling* approach by Agrawal et al. [1].

2-Hop based algorithms. The basic idea of the 2-Hop approach is to assign for each vertex v a list of vertices which are reachable from v, denoted as $L_{in}(v)$, and a list of vertices to which v can reach, denoted as $L_{out}(v)$, so that for any two vertices u and v, $u \rightarrow v$ if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. The ultimate goal of the algorithm is to minimize the index size of the form $\sum_{v \in V} L_{out}(v) + L_{in}(v)$. Clearly if the index is available, the reachability query answering requires only two lookups. However, this optimization problem is NP-hard. Improvements on the 2-Hop algorithm can be found in [13, 14] Generally 2-Hop based algorithms do not scale for large size graphs.

Interval labeling based algorithms. Interval labeling based approaches utilize the efficient method of indexing and querying trees which was applied to XML query processing in recent years [18]. It is well known that given a tree, we can label each node v by an interval $[start(v), end(v)]$. Thus the reachability query can be answered by comparing the start and the end labels of u and v in constant time. The labeling process takes linear time and space. The Dual

Labeling algorithm proposed by Wang et al. [16] achieved to answer reachability queries in constant time. They first identify a spanning tree from the graph and label the vertices in the tree with pre- and post-order values. Then the transitive closure for the rest of the edges is stored. Clearly, the price for the constant query time is paid by the storage cost of t^2 where t is number of the non-tree edges in the graph. Therefore the Dual Labeling approach achieves good performance only if the graph is extremely sparse where $t \ll n$.

Jin et al. [9] proposed a different index structure called Path Tree. Like other interval labeling based methods, they extract a tree from the original graph. But every node in the tree contains a path, instead of a single vertex. This index structure is superior to the previous ones since it can encode some non-tree structures such as grid in an elegant way.

All of these algorithms in common is that the performance deteriorate for non-sparse graphs. In contrast, the index structure proposed in this paper scales for dense graphs as well.

2 Graph Indexing with Tree Decomposition

2.1 Tree Decomposition of Directed Graphs

A directed graph is defined as $G = (V, E)$, where $V = \{0, 1, \ldots, n-1\}$ is the vertex set and $E \subseteq V \times V$ is the edge set. Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges.

For each directed graph, its tree decomposition is defined as follows:

Definition 1. *A tree decomposition of $G = (V, E)$, denoted as T_G, is a pair $(\{X_i \mid i \in I\}, T)$, where $\{X_i \mid i \in I\}$ is a collection of subsets of V and $T = (I, F)$ is a tree such that:*

1. $\bigcup_{i \in I} X_i = V$.
2. *for every $(u, v) \in E$, there is $i \in I : u, w \in X_i$.*
3. *for all $v \in V$, the set $\{i \mid v \in X_i\}$ induces a subtree of T.*

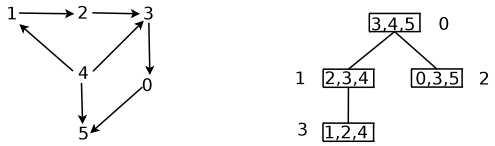
A tree decomposition contains a set of tree nodes, where each node contains a set of vertices in V. We call the sets X_i *bags*. It is required that every vertex in V should occur in at least one bag (condition 1), and for every edge in E, both vertices of the edge should occur together in at least one bag (condition 2). The third condition is usually referred to as the *connectedness condition*, which requires that given a vertex v in the graph, all the bags which contain v should be connected.

Note that from now on, the node in the directed graph G is referred to as *vertex*, and the node in the tree decomposition is referred to as *tree node* or simply *node*. For each tree node i, there is a bag X_i consisting of vertices. To simplify the presentation, we will sometimes use the term *node* and its corresponding *bag* interchangeably.

Given any graph G, there may exist many tree decompositions which fulfill all the conditions in Definition 1. However, we are interested in those tree

decompositions with smaller bag sizes. The *width* of a bag is the cardinality of the bag. The *width* of a tree decomposition $(\{X_i \mid i \in I\}, T)$ is defined as $\max\{|X_i| \mid i \in I\}$ [1]. The *treewidth* of G is the minimal width of all tree decompositions of G. It is denoted as $tw(G)$. Note that trees and forests are precisely the structures with treewidth 2.

Example 1. Consider the graph illustrated in Figure 1(a). One of the tree decompositions is shown in Figure 1(b) . Recall that only trees and forests have treewidth 2, therefore this tree decomposition is optimal and we have $tw(G) = 3$.



(a) (b)

Fig. 1. The graph G (a) and one tree decomposition T_G (b) with $tw(G) = 3$

Let $G = (V, E)$ be a graph and $T_G = (\{X_i \mid i \in I\}, T)$ its tree decomposition. Due to the third condition in Definition 1, for any vertex v in V there exists an induced subtree of T_G in which every bag contains v. We call it the *induced subtree* of v and denote it as T_v. Furthermore, we denote the root of T_v as r_v and its corresponding bag as X_{r_v}. For instance, the induced subtree of vertex 3 in Figure 1(b) contains the bags X_0, X_1 and X_2, where $r_3 = 0$.

2.2 Tree Path

Let $G = (V, E)$ be a directed graph, and $u, v \in V$. We say v is reachable from vertex u, denoted as $u \to v$, if there is a path starting from u and ending at v with the form $(u, v_1, \ldots, v_n, v)$, where $(u, v_1), (v_i, v_{i+1}, (v_n, v) \in E$. Note that in this paper, we consider the more general definition of path, that is, a path is not necessarily a simple path.

Let us consider the graph vertices in the tree nodes. Since each vertex occurs in more than one bag, a vertex can be identified with $\{v, i\}$, where v is a vertex and i the node in the tree, meaning that vertex v is located in the tree node i. We denote it as *tree vertex*. Now we define the so-called *inner edge* and *inter edge* in the tree decomposition.

Definition 2 (Inner edge, Inter edge, Tree path). *Let $G = (V, E)$ be a directed graph and $T_G = (\{X_i \mid i \in I\}, T)$ its tree decomposition.*

[1] The original definition of the width is $max\{|X_i| \mid i \in I\} - 1$, due to esthetic reasons.

- *The* inner edges *of T_G are precisely the pairs of tree vertices defined as follows: $\{(\{u,i\},\{v,i\}) \mid (u,v) \in E, u,v \in X_i (i \in I)\}$.*
- *The* inter edges *of T_G are the pairs of tree vertices with the form $(\{v,i\},\{v,j\})$ where $v \in X_i$ and $v \in X_j$, and either $(i,j) \in F$ or $(j,i) \in F$ holds.*
- *A* tree path *from $\{u,i\}$ to $\{v,j\}$ is a sequence of tree vertices connected with either inter or inner edges.*

Intuitively, the set of inner edges consists precisely of those edges in E, with the extra information of the bags in which the edges are located. For instance, the inner edges of the tree decomposition of the graph in Example 1 are: $(\{0,2\},\{5,2\})$, $(\{1,3\},\{2,3\})$, $(\{2,1\},\{3,1\})$, $(\{3,2\},\{0,2\})$, ... Note that it happens that the same pair of vertices occurs in more than one bag. For instance, the edge $(4,3)$ occurs in both bags X_0 and X_1. Thus there are two inner edges: $(\{4,1\},\{3,1\})$ and $(\{4,0\},\{3,0\})$ For instance, in Example 1, $(\{5,0\},\{5,2\})$ is an inter edge, as well as $(\{5,2\},\{5,0\})$.

Lemma 1. *Let $G = (V,E)$ be a directed graph and $T_G = (\{X_i \mid i \in I\}, T)$ its tree decomposition. Let $u,v \in V$. Let further $\{u,i\}$ and $\{v,j\}$ be tree vertices in T_G. There is a path from u to v in G if and only if there is a tree path from $\{u,i\}$ to $\{v,j\}$.*

Example 2. Consider the graph in Figure 1(a). Vertex 4 reaches vertex 0 with the path $\{4,1,2,3,0\}$. In the tree decomposition in Figure 1(b), there is a tree path from $\{4,1\}$ to $\{0,2\}$ as follows: { $\{4,1\}$, $\{4,3\}$, $\{1,3\}$, $\{2,3\}$, $\{2,1\}$, $\{3,1\}$, $\{3,0\}$, $\{3,2\}$, $\{0,2\}$ }.

2.3 Reachability Test on Tree Decomposition

With the definition of tree path, to find a path from u to v, we can simply search in the tree decomposition for a corresponding tree path. Moreover, over the tree decomposition, we only need to concentrate on the simple path between the corresponding tree vertices. There is a well known property of trees that says for any two nodes i and j in a tree, there exists a unique simple path, denoted as $SP_{i,j}$, such that every path from i to j contains all the nodes in $SP_{i,j}$.

Proposition 1. *Let $G = (V,E)$ be a directed graph and $T_G = (\{X_i \mid i \in I\}, T)$ its tree decomposition. Let $u,v \in V$. Let further r_u (resp. r_v) be the root node of the induced subtree of u (resp. v). Then $u \to v$ if and only if for every node n in SP_{r_u,r_v}, there is at least one vertex $t \in X_n$, such that $u \to t$ and $t \to v$.*

Proof. The "if" direction is trivial: given a tree path from $\{u,i\}$ to $\{v,j\}$, we only need to consider the inner edges. Since for each inner edge $\{u,i\},\{v,i\}$, there is an edge $(u,v) \in E$, the path from u to v can be easily constructed.

Now we prove the "only if" direction: assume that there is a path from u to v in G. We prove it by induction on the length of the path.

– Basis: if u reaches v with a path of length 1, that is, $(u, v) \in E$. Then there exists a node k in the tree decomposition, s.t. $u \in X_k$ and $v \in X_k$. We start from $\{u, i\}$, traverse along the induced subtree of u, till we reach $\{u, k\}$. Since the induced subtree is connected, the path from $\{u, i\}$ to $\{u, k\}$ can be constructed with inter edges. Then we reach from $\{u, k\}$ to $\{v, k\}$ with an inner edge. Now we traverse from $\{v, k\}$ to $\{v, j\}$ along the induced subtree of v, which can again be constructed with inter edges. The tree path from $\{u, i\}$ to $\{v, j\}$ is thus completed.
– Induction: assume that the lemma holds with paths whose length is less than or equal to $n-1$, we prove that it holds for paths with length of n. Assume that there is a path from u to v with length n, where u reaches w with length $n-1$ and $(w, v) \in E$. From induction hypothesis, we know that there is a tree path form $\{u, i\}$ to $\{w, l\}$ in the tree decomposition, where l is a node in the induced subtree of w. Since $(w, v) \in E$, there is a node n such that $w \in X_n$ and $v \in X_n$. Thus $\{w, n\}$ can be reached from $\{w, l\}$ with inter edges. Then $\{w, n\}$ can reach $\{v, n\}$ with an inner edge. Finally $\{v, n\}$ can reach $\{v, j\}$ with a sequence of inter edges. This completes the proof.

□

Proposition 1 shows that for the reachability test from u to v, although the tree path from $\{u, r_u\}$ to $\{v, r_v\}$ may possibly visit any node in the tree, we only need to concentrate on the reachability test for those vertices which occur in the simple path SP_{r_u, r_v}. More precisely, we can simply take *any* node n from SP_{r_u, r_v}, and check whether there is a vertex $t \in X_n$, such that $u \to t$ and $t \to v$ hold. In order to further accelerate the query process, we can execute the reachability test along the path tree in a bottom-up manner, as shown in Figure 2. In order to enable the bottom up operation, we need to store the transitive closure for each bag in the tree decomposition. That is, in every bag X, for every pair of vertices $x, y \in X$, the boolean values of $x \to y$ and $y \to x$ are pre-computed. We show in the following proposition how the reachability queries from u to all the vertices in $SP_{r_u, k}$ can be answered.

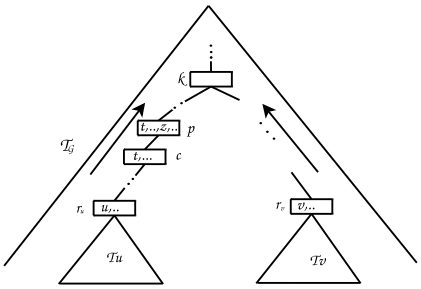


Fig. 2. Bottom-up processing on the simple tree path

Proposition 2. *Let $G = (V, E)$ be a directed graph and $T_G = (\{X_i \mid i \in I\}, T)$ its tree decomposition. Let $u, v \in V$. Let k be the lowest common ancestor of r_u and r_v. The reachability queries from u to all the vertices in $SP_{r_u,k}$ can be answered in $O(w^2h)$, where $h = |SP_{r_u,k}|$ and w is the maximal width of the bags in $SP_{r_u,k}$.*

Proof. Assume that the transitive closure in every bag from $SP_{r_u,k}$ is available. The reachability test starts with node r_u. From the information of transitive closure, we can simply obtain the set $Y_{r_u} \subseteq X_{r_u}$ such that every vertex in Y_{r_u} can be reached from u. Next, we consider r_u as the child node and process its parent node, with the available reachability information. This process is recursively executed h times, until k is reached.

Next we show that at each step of the processing, all the vertices in the current bag reachable from u can be found in w^2 time, where w is the width of the current bag. Assume p is the current node, c its child node, and we have obtained $Y_c \subseteq X_c$, where Y_c contains all the vertices reachable from u. Now we have to decide the set $Y_p \subseteq X_p$, i.e. identify all the vertices reachable from u in X_p.

Let z be a vertex in X_p. We want to decide whether $u \to z$. We have the following two cases:

1. $z \in X_p$ and $z \in X_c$. Since at the child node we know whether $z \in Y_c$, we set $z \in Y_p$ if $z \in Y_c$.
2. $z \in X_p$ and $z \notin X_c$. This is a more complex case. We show that $z \in Y_p$ (i.e. z is reachable from u) if and only if there exists a vertex t, such that $t \in X_p$, $t \in Y_c$ and $t \to z$ holds.
   (a) "if" direction is trivial.
   (b) "only if: Assume that $u \to z$ holds. Since z does not occur in X_c, according to the connectedness condition, z does not occur in any bag in the subtree rooted with c. Thus the induced subtrees of u and z do not share any common node in T_G. Since $u \to z$, there is a tree path from $\{u, r_u\}$ to $\{z, r_z\}$, and $c, p \in SP_{r_u,r_z}$. The tree path from $\{u, r_u\}$ to $\{z, r_z\}$ must contain an inter edge of the form $(\{t, c\}, \{t, p\})$, where $t \in X_p, X_c$, because this is the only possible edge to traverse from c to p. Clearly $u \to t$ holds. From the assumption $u \to z$, we obtain that $t \to z$ must hold.

   Given the set $Y_c \subseteq X_c$ and the transitive closure in X_p, we can obtain Y_p as follows: First set Y_p as $Y_c \cap X_p$. Then for each vertex $t \in Y_p$, we add the vertex s into Y_p, if $t \to s$ holds. Clearly the time consumption is in the worst case $O(w^2)$ where w is the width of X_p.

□

3 Algorithms and Complexity Results

In this section, we present the detailed algorithms for both the index construction and the reachability query answering. In Section 3.1 we begin with the introduction of algorithmic issues on the tree decomposition from a complexity

theory perspective, and then justify our choice of an efficient but suboptimal decomposing algorithm. In Section 3.2 we first analyze the reachability query answering algorithm proposed in Theorem 2 from the previous section. Then, we point out that the time and space improvement can be made to achieve higher efficiency of our algorithm.

3.1 Index Construction via Tree Decomposition

Since its introduction by Robertson and Seymour [12], the concepts of tree decomposition has been proved to be of great importance in computational complexity theory [4]. The theoretical significance of the tree decomposition based approach lies in the fact that many intractable problems can be solved in polynomial time (or even in linear time) for graphs with treewidth bounded by a constant. Problems which can be dealt with in this way include many well known NP-complete problems, such as the Independent Set, the Hamiltonian Circuits, etc. Recent applications of tree decomposition based approaches can be found in Constraint Satisfaction [10] and database design [7].

However, the practical usefulness of tree decomposition based approaches has been limited due to the following two problems: (1) Calculating the treewidth of a graph is hard. In fact, determining whether the treewidth of a given graph is at most a given integer w is NP-complete [2]. Although for fixed w, linear time algorithms exist to solve the decision problem "treewidth $\leq w$" [3], there is a huge hidden constant factor, which prevents it to be useful in practice. There exist many heuristics and approximation algorithms for determining the treewidth, unfortunately few of them can deal with graphs containing more than 1000 nodes [11]. (2) The second problem lies in the fact that even if the treewidth can be determined, it still can not be guaranteed that good performance will be obtained since the time complexity of most of the algorithms is exponential to the treewidth. Therefore, to solve really hard problems efficiently by using the tree decomposition based approaches, we have to require that the underlying graphs have *bounded* treewidth (i.e. less than 10).

As far as the efficiency is concerned, we can only search for an approximate solution, which yields a tree decomposition whose width is greater than the treewidth. On the other hand, we can tolerate a tree decomposition whose treewidth is not bounded. As we have seen from Proposition 2, the time complexity is in the worst case quadratic of the maximal bag size. We will show later in this section that our query answering algorithm does not depend on the treewidth, but with some parameter which can be enforced to be bounded, due to the nice property of our dedicated decomposing algorithm, and the height of the tree.

Inspired from the so-called pre-processing methods by Bodlaender et al. [5], we apply the reduction rules on the graph by reducing stepwise a graph to another one with fewer vertices, due to the following simple fact.

Definition 3 (Simplicial). *A vertex v is simplicial in an undirected graph G if the set of neighbors of v form a clique in G.*

Figure 3 shows some special cases. If a vertex v has degree of one (Figure 3(a)), then we can remove v without increasing the treewidth. Figure 3(b), 3(c) illustrate the cases of degree 2 and 3 respectively.

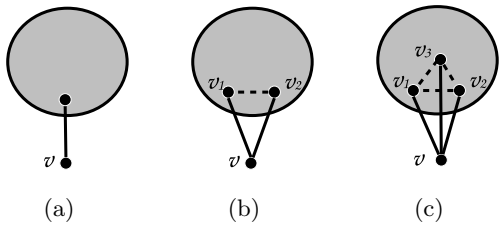


Fig. 3. A undirected graph containing a vertex v with degree 1 (a), 2 (b) and 3 (c)

The main idea of our decomposition algorithm is to reduce the graph by removing the vertices one by one from the graph, and at the same time push the removed vertices into a stack, so that later on the tree can be constructed with the information from the stack. First a vertex v with a specific degree is identified. We first check whether all its neighbors form a clique, if not, we add the missing edges to construct a clique. Then v together with its neighbors are pushed into the stack, which is followed by the deletion of v and its edges in the graph. See Algorithm 2.

Algorithm 1 *tree_decomp*(G)

Input: $G = (V, E)$ is a directed graph.
Output: return the tree decomposition T_G.
1: Transform G into an undirected graph UG;
2: *graph_reduction*(UG); {output the vertex stack S}
3: *tree_construction*(S, G); {output the tree decomposition}

The program begins with removing isolated vertices and vertices with degree 1. Then, the reduction process proceeds with the vertices with degree of 2, 3, ... We denote such procedure of removing all the vertices with degree x as *degree-x reduction*.

Example 3. Consider the undirected version of the graph in Example 1. Figure 4 illustrates the reduction process. The process starts with a degree-2 reduction by removing vertex 0 and its edges, after adding the edge between 3 and 5. Vertex 0 and its neighbors are then pushed in the stack. Next vertex 1 is removed, following the same principle as of 0. After vertex 2 is removed, a single triangle is then left.

The procedure *graph_reduction* will terminate when one of the following conditions is fulfilled. (1) The graph is reduced to an empty set. For instance, if

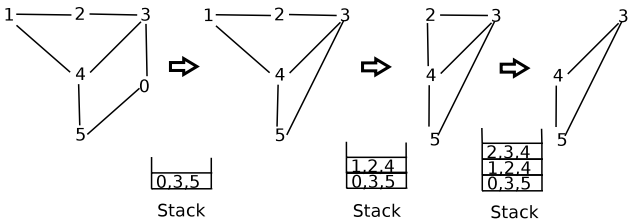


Fig. 4. The reduction process on the undirected graph of Example 1

the graph contains only simple cycles, it will be reduced to an empty set after degree-2 reductions. This is usually the case for extremely sparse graphs. (2) For graphs which are not sparse, one has to define a upper bound l for the reduction, so that the program stops after the degree-l reduction. Note that as the degree increases, the effectiveness of the reduction will decrease, because in the worst case, we need to add $x(x-1)/2$ edges in order to remove x edges.

Algorithm 2 *graph_reduction*(UG)

Input: UG is the undirected graph of G, l is the upper bound for the reduction.
Output: stack S and the reduced graph UG'

```
1: initialize stack S;
2: for i = 1 to l do
3:    remove_upto(i);
4: end for
5: return S, UG;
6: procedure remove_upto(x)
7: while TRUE do
8:    if there exists a vertex v with degree less than x then
9:       {v_1, ..., v_x} = neighbors of v;
10:      build a clique for {v_1, ..., v_x};
11:      push v, v_1, ..., v_x into S;
12:      delete v and all its edges from UG;
13:   else
14:      break;
15:   end if
16: end while
```

After the reduction process, the tree decomposition can be constructed as follows: (1) At first we collect all the vertices which were not removed by the reduction process and assign this set as the bag of the tree root. The size of the root depends on the structure of the graph (i.e. how many vertices are left after the reduction). (2) The rest of the tree is generated from the information stored in stack S. Let X_c be the set of vertices $\{v, v_1, \ldots, v_x\}$ which is popped up from the top of S. Here v is the removed vertex and $\{v_1, \ldots, v_x\}$ are the neighbors

of v which form a clique. After the parent bag X_p which contains $\{v_1, \ldots, v_x\}$ is located in the tree, X_c is added as a child bag of X_p. This process proceeds until S is empty. Algorithm 3 illustrates the process.

Algorithm 3 *tree_construction*(S, G, UG')

Input: S is the stack storing the removed vertices and their neighbors, G is the directed graph, UG' is the reduced graph of UG.
Output: return tree decomposition T_G

```
1: construct the root of T_G containing all the vertices of UG';
2: while S is not empty do
3:     pop up a bag X_c = {v, v_1, ..., v_x} from S;
4:     find the bag X_p containing {v_1, ..., v_x};
5:     add X_c into T as the child node of X_p;
6: end while
7: generate transitive closure in all bags;
```

The last step of the tree construction process is to generate the transitive closure for every bag.

The correctness of our tree decomposition algorithm can be shown by the induction on the reduction steps. Note that during the reduction process, edges are inserted into the original graph. Therefore, the tree decomposition we obtain according to the algorithm is based on a graph consisting of extra edges. However, this does not affect the correctness proof due to the following proposition.

Proposition 3. *Let $G = (V, E)$ and $G' = (V, E')$ be graphs where $E \subseteq E'$. Then any tree decomposition of G' is a tree decomposition of G.*

Proof. Let $T_{G'}$ be the tree decomposition of G'. By checking the three properties of Definition 1, it is obvious that $T_{G'}$ is also a tree decomposition of G.

3.2 Reachability Query Answering

Recall from Proposition 2 that the time complexity of the bottom-up query answering is $O(w^2 h)$. This upper bound is optimal, only if the following two conditions are fulfilled: (1) the treewidth of the underlying graph is bounded (that is, $w^2 \ll n$), and (2) there is an efficient tree decomposition algorithm for it. The first condition has to be fulfilled, since otherwise the linear time BFS algorithm would be more efficient. Unfortunately, as we have seen in the previous section, given an arbitrary graph, it is clear that neither (1) nor (2) can be fulfilled. Therefore, we have to inspect the tree decomposition heuristics applied in Section 3.1 for improvements.

From Treewidth to $|R|$ and l According to Algorithm 2, a graph G can be decomposed by the *degree-l reductions* by increasing x from 1 to l. As soon

as the *degree-l reduction* is done, all the vertices which are not yet removed are the elements in R of the tree decomposition. Usually if the graph is not extremely sparse, the relationship $l \ll |R|$ holds. In fact, we could even enforce such a relationship by setting l to be small enough in the tree decomposition algorithm. Hence, the resulting tree decomposition has the following properties: (1) the root is of big size ($|R|$), and (2) the rest of the bags have smaller size (the upper bound is l).

If we inspect the bottom-up query processing more carefully, we could observe that the quadratic time computation over the root can be *always* be avoided. To see this, let us consider the vertices u and v and the lowest common ancestor of r_u and r_v is the root R. Assume that X_1 (resp. X_2) is the child node of R which locates in the simple path from r_u (resp. r_v) to R. Consider now that for all $x \in X_1$, $reach(u, x)$ (resp. all $y \in X_2$, $reach(y, v)$) have been computed. Clearly, any path from u to v has to pass through a vertex in X_1 and X_2 respectively. Therefore, at the root node R, we can first calculate $X_1 \cap R$ and $X_2 \cap R$. Since all the paths from u to v has to pass one vertex in $X_1 \cap R$ and another vertex in $X_2 \cap R$, we only need to execute a nested loop on $X_1 \cap R$ and $X_2 \cap R$ to decide the reachability. Since both $|X_1|$ and $|X_2|$ have the upper bound of l, the overall time consumption is of $O(l^2 h)$, thus independent of $|R|$. Note that if both u and v are located in R, then the shortest path can be immediately obtained from the local shortest path from u to v, which are pre-computed.

Algorithm 4 $reach(T_G, u, v)$

Input: T_G is the tree decomposition of G and u, v vertices in G.
Output: return TRUE if $u \rightarrow v$, otherwise FALSE

1: $c = r_u$ = root of induced subtree of u; $c = r_v$ = root of induced subtree of v;
2: k = lowest common ancestor of r_u and r_v;
3: R_u = reachable vertices from u in X_c;
4: **while** $c.parent \neq k$ **do**
5: $p = c.parent$; $R_u = R_u \cap X_c \cap X_p$;
6: **for all** t in R_u **do**
7: R_t= set of vertices reachable from t in X_p; $R_u = R_u \cup R_t$;
8: **end for**
9: $c = p$;
10: **end while**
11: R_v = all vertices that reach v in X_c;
12: **while** $c.parent \neq k$ **do**
13: $p = c.parent$; $R_v = R_v \cap X_c \cap X_p$;
14: **for all** t in R_v **do**
15: R_t= set of vertices reach t in X_p; $R_v = R_v \cup R_t$;
16: **end for**
17: $c = p$;
18: **end while**
19: $R_u = R_u \cup X_k$; $R_v = R_v \cup X_k$;
20: **return** $(reach(x, y) \mid \exists x \in R_u \wedge \exists y \in R_v)$;

The algorithm for the reachability query answering is presented in Algorithm 4. Comparing with the bottom-up query processing shown in Proposition 2, Algorithm 4 is customized with respect to our dedicated tree decomposition algorithm, in the sense that the query time complexity is adapted to be related to l, instead of the treewidth.

3.3 Complexity

Index construction time. For the index construction, we have to (1) generate the tree decomposition, and (2) at each tree node, generate the local transitive closures. For (1), both of the reduction step and the tree construction procedure take time $O(n)$. For (2), we deploy the classic BFS algorithm, which costs in worst case $O(m)$. In fact, we need to run for each vertex in G exactly one BFS procedure. Therefore, the overall index construction time is $O(nm)$.
Index size. In each bag X, for each pair of vertices u, v in X, if u reaches v, we need to store a boolean value. Thus the index size is $|X|^2$, Since the relationship $l \ll |R|$ holds, the root size ($|R|$) is dominant among all the bags. Therefore, the index size is $|R|^2$. The index size consists of the tree structure, constructed by using the tree decomposition algorithm. However, this space overhead is linear to n, thus can be ignored.
Query. The bottom-up query processing for reachability query answering takes time $O(l^2h)$, where l is the number of the reductions and h is the height of the tree decomposition. Note that the proposed tree decomposition algorithm is independent of the treewidth of the underlying graph, since the reduction parameter l can be adjusted according to the property of the graph. On the other hand, there is no guarantee that the optimal tree decomposition can be obtained. In the worst case, if tree-width is approximately n, there are $\Theta(n^2)$ edges to be stored. So the running time of the query algorithm in the worst case is worse than the one of the BFS (or DFS): if $tw(G) = \Theta(|G|)$. Clearly our algorithm is not suitable for such graphs.

4 Experiments

In this section we evaluate the tree decomposition method on real datasets. We are interested in the following parameters: *Index size*, *Index construction time*, and *Query time*. Note that the index size is measured as the size of transitive closures, which takes up the major part of the overall index size. Besides the standard measurements, we are also interested in the structure of the tree decomposition, which may influence the performance of the algorithm. These are: the number of tree nodes (#TreeN), the number of all the vertices stored in the bags (#SumV), the height of the tree (h), the number of vertex reductions (l), and the root size of the tree ($|R|$). Note that we have chosen the optimal l, in order to achieve the best query time performance.

We tested our algorithm over real large datasets with density being larger than or close to 2 used in [8]. All graphs are extracted from real-world large

datasets with density being larger than or close to 2. Among them, arXiv is extracted from a dataset of citations among scientific papers from the arxiv.org website. Citeseer contains citations among scientific literature publications from the CiteSeer project, and pubmed was extracted from an XML registry of open access medical publications from the PubMed Central website. GO contains genetic terms and their relationships from the Gene Ontology project. Yago describes the structure of relationships among terms in the semantic knowledge database from the YAGO project. The details of the datasets can be found in [8]. All tests are run on an Intel(R) Core 2 Duo 2.4 GHz CPU, and 2 GB of main memory. All algorithms were implemented in C++ with the Standard Template Library (STL). 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 10000 randomly generated reachability queries. We make a comparison of the query time with the linear time Breadth First Search method (BFS).

Graph	$\#V$	$\#E$	#TreeN	#SumV	h	l	$\|R\|$	Index		Query Time	
								Time(s)	Size	TD (ms)	BFS (ms)
Arxiv	6000	66707	4713	28300	12	30	1288	12.5	362228	49.6	449.5
Citeseer	10720	44258	8291	33411	9	8	2430	3.6	91067	8.8	135.5
Go	6793	13361	5186	19262	9	5	1608	1.2	29674	5.8	77.1
Pubmed	9000	40028	6482	26746	6	9	2519	2.9	185065	5.8	127.4
Yago	6642	42392	6161	19677	8	8	482	1.2	11673	3.2	78.9

Table 1. Statistics of real graphs, the properties of the index and query performance

As shown in Table 1, the time costs for query answering are substantially improved with respect to the naive BFS algorithm. As expected, there is a correlation between the index size and the size of the root size of the tree decomposition $|R|$. Note that the size of the index structure should be approximately $|R|^2$. However, we can reduce the size by only store those pairs which are reachable from one to the other. We obtain a query time speedup with respect to the naive BFS approach between 11% (Arxiv) and 4% (Yago).

5 Conclusions and Future Work

In this paper, we introduced the tree decomposition as the index structure for large directed graphs to answer reachability queries efficiently. With both theoretical and empirical analysis, we demonstrated that our approach is intuitive and efficient. The algorithms achieve good transitive closure compression rates and scale well on large size graphs.

In the future we plan to investigate the following problems: (1) Development of scalable tree decomposition algorithms. We expect to investigate more heuristics and integrate them into our implementation. (2) How to update the

of the index structure is the underlying graph is changed. Furthermore, we will consider on-disk algorithms for both index construction and query answering.

References

1. R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, 1989.
2. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.
3. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *STOC*, 1993.
4. H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.
5. H. L. Bodlaender, A. M. C. A. Koster, and F. van den Eijkhof. Pre-processing rules for triangulation of probabilistic networks. *Computational Intelligence*, 21(3):286–305, 2005.
6. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.
7. G. Gottlob, R. Pichler, and F. Wei. Tractable database design through bounded treewidth. In *PODS*, pages 124–133, 2006.
8. R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.
9. R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, 2008.
10. K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying tree decompositions for reasoning in graphical models. *Artif. Intell.*, 166(1-2):165–193, 2005.
11. A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. V. Hoesel. Treewidth: Computational experiments. In *Electronic Notes in Discrete Mathematics*, 2001.
12. P. D. Robertson, Neil; Seymour. Graph minors iii: Planar tree-width. *Journal of Combinatorial Theory*, Series B 36:49–64, 1984.
13. R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *EDBT*, 2004.
14. R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371, 2005.
15. S. Trissl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, 2007.
16. H. Wang, H. He2, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, 2006.
17. F. Wei. Tedi: Efficient shortest path query answering on graphs. In *SIGMOD*, 2010.
18. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.