Decomposing DAGs into Spanning Trees: A New Way to Compress Transitive Closures

Yangjun Chen#*1, Yibin Chen#2

#Dept. Applied Computer Science, University of Winnipeg, Canada
1y.chen@uwinnipeg.ca
2chenyibin@gmail.com

*School of Information Science and Engineering, Central South University, P.R. China

Abstract—Let $G(V, E)$ be a digraph (directed graph) with $n$ nodes and $e$ edges. Digraph $G^* = (V, E^*)$ is the reflexive, transitive closure of $G$ if there is a path from $v$ to $u$ in $G$. Efficient storage of $G^*$ is important for supporting reachability queries which are not only common on graph databases, but also serve as fundamental operations used in many graph algorithms. A lot of strategies have been suggested based on the graph labeling, by which each node is assigned with certain labels such that the reachability of any two nodes through a path can be determined by their labels. Among them are interval labelling, chain decomposition, and 2-hop labeling. However, due to the very large size of many real world graphs, the computational cost and size of labels using existing methods would prove too expensive to be practical. In this paper, we propose a new approach to decompose a graph into a series of spanning trees which may share common edges, to transform a reachability query over a graph into a set of queries over trees. We demonstrate both analytically and empirically the efficiency and effectiveness of our method.

Key words: Directed graphs; spanning trees; reachability queries; transitive closure compression.

I. INTRODUCTION

Given two nodes $u$ and $v$ in a directed graph $G(V, E)$, we want to know if there is a path from $u$ to $v$. The problem is known as graph reachability. In many applications, such as evaluation of recursive queries in deductive databases [7, 18, 33, 34], type checking in object-oriented databases [22], XML query processing, transportation network, internet traffic analyzing, semantic web, and metabolic network [35], graph reachability is one of the most basic operations, and therefore needs to be efficiently supported.

A naive method is to precompute the reachability between every pair of nodes – in other words, to compute and store the transitive closure ($TC$ for short) of a graph as a boolean matrix $M$ such that $M[i, j] = 1$ if there is path from $i$ to $j$; otherwise, $M[i, j] = 0$. Then, a reachability query can be answered in constant time. However, this requires $O(n^2)$ space, which makes it impractical for massive graphs, where $n = |V|$. Another method is to compute the shortest path from $u$ to $v$ over a graph on demand. Therefore, it needs only $O(e)$ space, but with high query processing cost - $O(e)$ time in the worst case, where $e = |E|$.

There is much research on this issue to reduce space overhead but still keep constant query time, such as those discussed in [1, 4, 6, 8, 9, 19, 35]. All of them reduce the space requirement to some extent. But the worst space cost is still in the order of $O(n^2)$. In the case of large graphs, they cannot be used.

In this paper, we investigate the problem from a different angle: to decompose $G$ into several components such that the existing labeling techniques can be utilized for each smaller graph without sacrificing too much query time.

Concretely, we decompose $G$ into a series of spanning trees: $T_0, \ldots, T_k$ (for some $k \geq 1$), and a remaining graph $G$. They may share common edges, but $G$ is in general much smaller than $G$. If we assign intervals [35] to the nodes in $T_i (i = 0, \ldots, k - 1)$ and use Chen’s method [8] to label $G$, the total size of labels is reduced to $O(kn + n^2\omega)$, where $n$ stands for the number of the nodes in $G$, and $\omega$ for the width of $G$, defined to be the size of a largest node subset $U$ of $G$ such that for any pair of nodes $u, v \in U$ there does not exist a path from $u$ to $v$ or from $v$ to $u$. The query time is bounded by $O(k)$.

More importantly, it is a very flexible method. For different applications, we can control the graph decomposition, i.e., to set $k$ to different constants, to get a trade-off of query time for space. We will show that it is a biased trade-off of time for space. While the query time increases linearly, the space overhead decreases quadratically, in the sense that both the number of the nodes and the width of $G$ are decreased.

The remainder of the paper is organized as follows. In Section II, we review the related work. In Section III, we discuss the main step to decompose a directed acyclic graph, based on which a transitive closure can be effectively compressed. In Section IV, we show a recursive graph decomposition to generate a series of spanning trees which may share common edges. Also, how to evaluate reachability queries using such trees is discussed. Section V is devoted to the experiments. Finally, a short conclusion is set forth in Section VI.

II. RELATED WORK

In the past two decades, many interesting labeling-based strategies have been proposed to reduce both the precomputation time and storage cost with reasonable answering time. In the following, we review some representative ones.

Chain decomposition methods. In [19], Jagadish suggested an interesting method to decompose a DAG (directed acyclic graph) into node-disjoint chains. On a chain, if node $v$ appears...
above node \( u \), there is a path from \( v \) to \( u \) in \( G \). Then, each node \( v \) is assigned an index \((i, j)\), where \( i \) is a chain number, on which \( v \) appears, and \( j \) indicates \( v \)'s position on the chain. These indexes can be used to check reachability efficiently with \( O(\kappa n) \) space overhead and \( O(1) \) query time, where \( \kappa \) is the number of chains. However, to find a minimized set of chains for a graph, Jagadish's algorithm needs \( O(n^2) \) time (see page 566 in [19]). For this reason, Jagadish suggested a heuristic method to find all the node-disjoint paths of \( G \) and then stitch some paths together to form a chain. In doing so, the number of the produced chains is normally much larger than the minimum number of chains, increasing significantly both space and query time.

The method discussed in [8] greatly improves Jagadish's method. It needs only \( O(n^2 + \omega^{3/2} n) \) time to decompose a DAG into a minimum set of node-disjoint chains, where \( \omega \) represents \( G \)'s width. Its space overhead is \( O(\omega n) \) and its query time is bounded by a constant. In [9], the concept of the so-called general spanning tree is introduced, in which each edge corresponds to a path in \( G \). Based on this data structure, the real space requirement becomes smaller than \( O(\omega n) \), but the query time increases to \( \log \omega \).

Interval based methods. In [1], Agrawal et al. proposed a method based on interval labeling. This method first figures out a spanning tree \( T \) and assigns to each node \( v \) in \( T \) an interval \((a, b)\), where \( b \) is \( v \)'s postorder number (which reflects \( v \)'s relative position in a postorder traversal of \( T \)); and \( a \) is the smallest postorder number among \( v \) and \( v \)'s descendants with respect to \( T \) (i.e., all the nodes in \( T[v] \), the subtree rooted at \( v \)). Another node \( u \) labeled \((a', b')\) is a descendant of \( v \) (with respect to \( T \)) iff \( a \leq b' < b \). This idea originates from Schubert et al. [28]. In a next step, each node \( v \) in \( G \) will be assigned a sequence \( L(v) \) of intervals such that another node \( u \) in \( G \) with interval \((x, y)\) is a descendant of \( v \) (with respect to \( G \)) iff there exists an interval \((a, b) \in L(v) \) such that \( a \leq y < b \). The length of such a sequence (associated with a node in \( G \)) is bounded by \( O(\lambda) \), where \( \lambda \) is the number of the leaf nodes in \( T \). So the time and space complexities are bounded by \( O(\lambda n) \) and \( O(\lambda n) \), respectively. The querying time is bounded by \( O(\log \lambda) \). In the worst case, \( \lambda = O(n) \). (See [10, 11, 13].)

The method discussed in [35] can be considered as a variant of the interval based method, and called Dual-I, specifically designed for sparse graphs \( G(V, E) \). As with Agrawal et al.'s, it first finds a spanning tree \( T \), and then assigns to each node \( v \) a dual label \([a_v, b_v]\) and \((x_v, y_v, z_v)\). In addition, a \( t \times t \) matrix \( N \) (called a TLC matrix) is maintained, where \( t \) is the number of non-tree edges (edges not appearing in \( T \)). Another node \( u \) with \([a_u, b_u]\) and \((x_u, y_u, z_u)\) is reachable from \( v \) iff \( a_u \in [a_v, b_v] \), or \( N(x_u, z_u) \cdot N(y_v, z_v) > 0 \). The size of all labels is bounded by \( O(n + t^2) \) and can be produced in \( O(n + e + t^2) \) time. The query time is \( O(1) \). As a variant of Dual-I, one can also store \( N \) as a tree (called a TLC search tree), which can reduce the space overhead from a practical viewpoint, but increases the query time to \( \log t \). This scheme is referred to as Dual-II.

2-hop labeling. The method proposed by Cohen et al. [6] labels a graph based on the so-called 2-hop covers. It is also designed for sparse graphs. A hop is a pair \((h, v)\), where \( h \) is a path in \( G \) and \( v \) is one of the endpoints of \( h \). A 2-hop cover is a collection of hops \( H \) such that if there are some paths from \( v \) to \( u \), there must exist \((h_1, v) \in H \) and \((h_2, u) \in H \) and one of the paths between \( v \) and \( u \) is the concatenation \( h_1 h_2 \). Using this method to label a graph, the worst space overhead is in the order of \( O(n) \). The main theoretical barrier of this method is that finding a 2-hop cover of minimum size is an \( NP \)-hard problem. So a heuristic method is suggested in [6], by which each node \( v \) is assigned two labels, \( C_{in}(v) \) and \( C_{out}(v) \), where \( C_{in}(v) \) contains a set of nodes that can reach \( v \), and \( C_{out}(v) \) contains a set of nodes reachable from \( v \). Then, a node \( u \) is reachable from node \( v \) if \( C_{in}(v) \cap C_{out}(v) \neq \emptyset \). Using this method, the overall label size is increased to \( O(n \sqrt{\log n}) \). In addition, a reachability query takes \( O(\sqrt{\log n}) \) time because the average size of each label is above \( O(\sqrt{\log n}) \). The time for generating labels is \( O(n^2) \).

Path-tree decomposition. Recently, Jin et al. [20] discussed a new method, by which a DAG \( G \) is decomposed into a set of node-disjoint paths. Then, a weighted directed graph \( G_w \) is constructed, in which each node represents a path and there is an edge \((i, j)\) if on path \( i \) there is a node connected to a node on path \( j \). The weight associated with \((i, j)\) is the number of such connections. \( G_w \) is then labeled in a way similar to Agrawal et al.'s. Unfortunately, this method does not work in some cases because \( G_w \) is in general a cyclic graph, but Agrawal et al.'s method is applicable only to acyclic graphs. By Agrawal et al.'s and also all the above methods, a preprocessor is needed to recognize all the strongly connected components (SCCs) in a graph by using Tajan's algorithm [29] and collapse each of them into a single node. It is because any two nodes in an SCC are reachable from each other.

For illustration, see the following example.

![Fig. 1 Illustration for the path-tree](image-url)
of these two paths may not be reachable from some node on the other path in $G$.

A possible correction of this method is to consider all the possible spanning trees of each SCC in $G_c$. But in this way, the labeling time will be dramatically increased. In the worst case, it can be exponential in the size of $G_c$. For example, for the SCC shown in Fig. 1(b), we need to consider altogether six spanning trees. (In [20], only one spanning tree is considered. Obviously, it cannot be correct.)

There are some other graph labeling methods, such as the method using signatures [31], PE-Encoding [5] and PQ-Encoding [36]. The idea of the signature-based method [31] is to assign to each node a signature (which is in fact a bit string) generated using a set of hash functions. The space complexity is $O(ln)$, where $l$ is the length of a signature. But this encoding method suffers from the so-called signature conflicts (two nodes are assigned the same signature). Moreover, in the case of DAGs, a graph needs to be decomposed into a series of trees; and no formal decomposition was reported in that paper. The PE-Encoding [5] and the PQ-Encoding [36] are similar to the 2-hop labeling, but with higher computational complexities. The methods discussed in [25, 26] reduces 2-hop’s labeling complexity from $O(n^3)$ to $O(n'^3)$, but are still not applicable to massive graphs. The method proposed in [4] is a geometry-based algorithm to find high-quality 2-hop covers. It also improves the 2-hop labeling by avoiding the computation of transitive closures, which is required by Cohen’s method. However, it has the same theoretical computational complexities as Cohen’s method [6].

Finally, the method discussed in [32] is suitable only for planar graphs with $O(n\log n)$ labeling time and $O(n\log n)$ space. The query time is $O(1)$. Finally, deductive databases can be considered as a quite different extension to handle this problem [12, 14, 15].

In the following table, we compare our labeling method with the representative approaches.

<table>
<thead>
<tr>
<th>Table I</th>
<th>Comparison of strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph traversal</td>
<td>Query time</td>
</tr>
<tr>
<td>Jagdish [19]</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Interval-based [1]</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Dual-I [35]</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Dual-II [35]</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>2-hop [6]</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Matrix-based [37]</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Chen [8]</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>ours</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

III. GRAPH DECOMPOSITION

In this section, we discuss a new graph decomposition approach to compress transitive closures. First, we give some basic definitions related to spanning trees in Subsection A. Then, in Subsection B, we demonstrate our basic graph decomposition based on the concept of critical nodes, as well as a method for checking the reachability by using such graph decomposition. Finally, we show how to efficiently recognize the critical nodes in a graph in Subsection C.

A. Basic Definition

Without loss of generality, we assume that $G$ is acyclic (i.e., $G$ is a DAG). If not, we will find all SCCs of $G$ and collapse each of them into a representative node. Obviously, each node in an SCC is equivalent to its representative node as far as reachability is concerned. This process takes $O(e)$ time using Tarjan’s algorithm [29].

We also use $V(G)$ and $E(G)$ to represent its node set and edge set, respectively.

It is well known that the preorder traversal of $G$ introduces a spanning tree (forest) $T$. With respect to $T$, $E(G)$ can be classified into four groups:

- **tree edges** ($E_{tree}$): edges appearing in $T$.
- **cross edges** ($E_{cross}$): any edge $(u, v)$ such that $u$ and $v$ are not on the same path in $T$.
- **forward edges** ($E_{forward}$): any edge $(u, v)$ not appearing in $T$, but there exists a path from $u$ to $v$ in $T$.
- **back edges** ($E_{back}$): any edge $(u, v)$ not appearing in $T$, but there exists a path from $v$ to $u$ in $T$.

All cross, forward, and back edges are referred to as non-tree edges. (But in a DAG, we do not have back edges since a back edge implies a cycle.) For illustration, consider the DAG shown in Fig. 2. For it, we may find a spanning tree as shown by the solid arrows. (In the figure, each non-tree edge is represented by a dashed arrow.)

As in [35], we can assign each node $v$ in $T$ an interval $[\alpha_v, \beta_v]$, where $\alpha_v$ is $v$’s preorder number (denoted $pre(v)$) and $\beta_v - 1$ is equal to the largest preorder number among all the nodes in $T[v]$. So another node $u$ labeled $[\alpha_u, \beta_u]$ is a descendant of $v$ (with respect to $T$) iff $\alpha_u \in [\alpha_v, \beta_v]$ [35], as shown in Fig. 2. If $\alpha_u \in [\alpha_v, \beta_v]$, we say, $[\alpha_u, \beta_u]$ is subsumed by $[\alpha_v, \beta_v]$. This method is called the tree labeling.

B. Graph Decomposition and Reachability Checking

In this subsection, we discuss a decomposition of $G(V, E)$: a spanning tree $T$ and a subgraph $G_c$ such that $|V(G_c)| < |V|$. What we want is to transform the reachability checking of any two nodes in $G$ to a checking over $T$ and a checking over $G_c$. Obviously, $G_c$ has to contain $E_{cross}$. But some more edges need to be included and carefully recognized. For this purpose, we introduce some new concepts.

We denote by $E'$ the set of all cross edges. Denote by $V'$ the set of all the end points of the cross edges. That is, $V' = V_{start} \cup V_{end}$ where $V_{start}$ contains all the start nodes while $V_{end}$ all the end nodes of the cross edges. For example, for the graph shown in Fig. 2, we have $V_{start} = \{h, g, f, d\}$ and $V_{end} =$
\{e, g, c, d, k\}. No attention is paid to the forward edge \((a, e)\) in the graph since it can simply be removed as far as the reachability is concerned.

The first concept is the so-called crossing range, which is a second pair of integers associated with each node \(v \in V\), defined below.

**Definition 1 (crossing range)** Let \(T\) be a spanning tree (forest) of \(G\). Let \(v\) be a node in \(V\), and \(v_1, \ldots, v_j\) the children of \(v\) in \(G\). Let \(\{\alpha_i, \beta_i\}\) \((i = 1, \ldots, j)\) be the interval of \(v_i\). Set \(a_i = \min\{\alpha_i\}\) and \(b_i = \max\{\alpha_i\}\). Then, \(\{a_v, b_v\}\) is called the crossing range of \(v\).

For technical convenience, for any node \(v\) without child nodes in \(G\), both its \(a_v\) and \(b_v\) are set to be \(\alpha_v\). For example, with respect to the spanning tree shown in Fig. 2, the crossing ranges of the nodes in \(G\) can easily be computed, as shown in Fig. 3.

We notice that the crossing range of node \(f\) in \(T\) shown in Fig. 3 is \(\{5, 5\}\). It is because \(f\) has only one child \(d\) in \(G\), whose interval is \(\{5, 6\}\). But node \(g\)'s crossing range is \(\{2, 5\}\) since it has two children \(c\) and \(d\) with intervals \(\{2, 5\}\) and \(\{5, 6\}\), respectively. The purpose of crossing edges is to define the so-called critical nodes, which are used to determine all those nodes \(\not\in V_{\text{start}} \cup V_{\text{end}}\) but should be included in \(G_c\).

**Definition 2 (critical nodes)** A node \(v\) in a spanning tree \(T\) of \(G\) is critical if the following conditions are satisfied:

1. There is a subset \(U\) of \(V_{\text{start}}\) with \(|U| > 1\) such that for any two nodes \(u_i, u_j \in U\) they are not related by the ancestor/descendant relationship and \(v\) is the lowest common ancestor of all the nodes in \(U\).
2. For each \(u \in U\), its crossing range \(\{a_u, b_u\}\) is not within \(T[v]\). That is, \(a_u\) or \(b_u\) is a preorder number not appearing in \(T[v]\).

All the critical nodes with respect to \(T\) are denoted by \(V_{\text{critical}}\). For example, in the spanning tree shown in Fig. 2, node \(e\) is the lowest common ancestor of \(\{f, g\}\) and both \(f\) and \(g\) are in \(V_{\text{start}}\). In addition, the crossing ranges of \(f\) and \(g\) are not within in \(T[e]\) (see Fig. 3). So \(e\) is a critical node. We also notice that node \(a\) is the lowest common ancestor of \(\{d, f, g, h\}\). But the crossing ranges of all the four nodes are in \(T[a]\). Thus, \(a\) is not a critical node. In the same way, we can check all the other nodes and find that \(V_{\text{critical}} = \{e\}\).

The reason for imposing condition (2) in the above definition is that if any cross edge going out of a node in \(T[v]\) reaches only a node in \(T[v]\), then the reachability between \(v\) and any other node in \(G\) can be checked by the tree labeling. So it is not necessary to include \(v\) in \(G_c\), if \(v \not\in V_{\text{start}} \cup V_{\text{end}}\).

Now we consider a tree (forest) structure \(T_c\), called a critical tree of \(G\) (with respect to \(T\)), which contains all the nodes in \(V_{\text{critical}} \cup V_{\text{start}} \cup V_{\text{end}}\). In \(T_c\), there is an edge from \(u\) to \(v\) iff there is a path \(P\) from \(u\) to \(v\) in \(T\) and \(P\) contains no other node in \(V_{\text{critical}} \cup V_{\text{start}} \cup V_{\text{end}}\), as illustrated in Fig. 4(a).

Denote \(T_c \cup E_{\text{cross}}\) by \(G_c\) (see Fig. 4(b)). Then, \(T\) and \(G_c\) make up a decomposition of \(G\). It can be seen that \(V(G_c)\) is much smaller than \(V\).

For any two node \(u, v\) appearing on a path in \(T\), their reachability can be checked using their associated intervals. However, our question is, if they are not on the same path in \(T\), can we check their reachability by using \(G_c\)?

To answer this question, we need another concept, the so-called anchor nodes.

First, for any critical node \(v\), we will change its crossing range as follows.

1. Assume that \(U\) is a subset of \(V_{\text{start}}\) such that \(v\) is the lowest common ancestor of all the nodes in it and satisfies condition (1) and (2) in Definition 2.
2. Set \(a_v \leftarrow \min\{\min_{u \in U}\{a_u\}, a_v\}\); \(b_v \leftarrow \max\{\max_{u \in U}\{b_u\}, b_v\}\).

For instance, node \(e\)'s original crossing range is \(\{8, 9\}\) (see Fig. 3(b)). The crossing ranges of node \(f\) and \(g\) are \(\{5, 5\}\) and \(\{2, 5\}\), respectively. So \(e\)'s original range will be changed to \(\{2, 9\}\).

Next, we denote by \(C(v)\) all the critical nodes in \(T[v]\) plus all those start nodes of the cross edges which appear in \(T[v]\).

We consider a maximal subset of \(C(v)\) such that each node in it does not have an ancestor in \(C(v)\). Denote such a subset as \(C_s(v)\). It can be seen that \(C_s(v)\) is at most one node \(u\) such that its crossing range is not within \(T[v]\). Otherwise, a new critical node in \(T[v]\) will be created (see Definition 2), which is an ancestor of \(u\) and in \(C(v)\), contradicting the fact that \(u \in C_s(v)\) and thus has no ancestor in \(C(v)\).

**Definition 3 (anchor nodes)** Let \(G\) be a DAG and \(T\) a spanning tree of \(G\). Let \(v\) be a node in \(T\). We associate two nodes with \(v\) as below.

1. A node \(y \in C_s(v)\) is called an anchor node (of the first kind) of \(v\) if its crossing range is not within \(T[v]\), denoted \(v^*\). If such a node does not exist, \(v^*\) is set to be the special symbol \(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.\)
2. A node \(w\) is called an anchor node (of the second kind) of \(v\) if it is the lowest common ancestor of \(v\) (in \(T\)), which has a cross incoming edge. \(w\) is denoted \(v^{**}\). If such a node does not exist, \(v^{**}\) is set to be \(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.\)

For example, in the graph shown in Fig. 2, \(r^* = e\). It is because node \(e\) is a critical node in \(C_s(e)\) and its crossing range \(\{2, 9\}\) (note that the crossing range of a critical node is changed) is not within \(T[r]\). But \(r^{**}\) does not exist since it does not have an ancestor which has a cross incoming edge. In the same way, we find that \(e^* = e^{**} = e\). That is, both the first and second kinds of anchor nodes of \(e\) are \(e\) itself. We can easily recognize the anchor nodes for all the other nodes in that graph.
The following two lemmas are critical to the reachability checking using $G_c$.

**Lemma 1** Let $u$ be a node, which is not a descendant of $v$ in $T$, but $u$ is reachable from $v$ via some non-tree edges. Then, any way for $v$ to reach $u$ must be through $v^*$.  

*Proof.* According to Definition 4, $v^*$ is the only node in $C_i(v)$ such that its crossing range is not within $T[v]$. It indicates that any start node in $T[v]$ such that its crossing range is outside of $T[v]$ must be a descendant of $v^*$ in $T$. So any node that is not a descendant of $v$ but reachable from $v$ via some cross edges must be through $v^*$.  

**Lemma 2** Let $u$ be a node, which is not an ancestor of $v$ in $T$, but $v$ is reachable from $u$ via some non-tree edges. Then, any way for $v$ to reach $u$ must be through $v^{**}$.  

*Proof.* This can be seen from the fact that any node which reaches $v$ via some cross edges is through $v^{**}$ to reach $v$.

In terms of the above discussion, we associate each $v \in G$ with a triplet $<x, y, z>$:  

- $x = [\alpha, \beta]$, an interval created by labeling the nodes in $T$;  
- $y = v^*$; and  
- $z = v^{**}$.  

**Proposition 1** Let $u$ and $v$ be two nodes in $G$, labeled $(\alpha_u, \beta_u, y_u, z_u)$ and $(\alpha_v, \beta_v, y_v, z_v)$, respectively. Node $u$ is reachable from $v$ iff one of the following holds:  

(i) $[\alpha_u, \beta_u]$ is subsumed by $[\alpha_v, \beta_v]$ (i.e., $\alpha_u \in [\alpha_v, \beta_v]$), or  
(ii) $z_u$ is reachable from $y_v$ through a path in $G_c$.  

*Proof.* The proposition can be derived from the following two facts:  

(1) $u$ is reachable from $v$ through tree edges iff $[\alpha_u, \beta_u]$ is subsumed by $[\alpha_v, \beta_v]$.  

(2) In terms of Lemma 1 and 2, $u$ is reachable from $v$ via non-tree edges iff $z_u = u^{**}$ and $y_v = v^*$ exist and $u^{**}$ is reachable from $v^*$ through a path in $G_c$.

In a triplet $(x, y, z)$ associated with a node, $y$ and $z$ are referred to as non-tree labels.  

**Example 1** Consider $G$ and $T$ shown in Fig. 2 once again. The non-tree labels of the nodes are shown in Fig. 5.

![Fig. 5 Non-tree labels](image)

In this figure, we can see that the non-tree label of node $r$ is $<e, e>$ because (1) $r^* = e$; and (2) $r^{**}$ does not exist. Similarly, the non-tree label of node $f$ is $<f, e>$. It is because $f^*$ is $f$ itself; but $f^{**}$ is $e$.

Especially, we notice that node $r$ and node $d$ are not on the same path in $T$. But $d$ is a descendant of $r$. Such reachability has to be checked by using their anchor nodes. In fact, we have a path: $e \rightarrow f \rightarrow d$ in $G_c$. But $d^{**} = d$ and $r^* = e$, which shows that $d$ is reachable from $r$ by Proposition 1.

In order to check the reachability in $G_c$, we can use any existing method. For example, we can employ Chen’s algorithm [8] to decompose $G_c$ into two chains as shown in Fig. 6(a).

Recall that on each chain if node $v$ appears above node $u$, there is a path from $v$ to $u$ in $G_c$.

Below is a brief description of Chen’s algorithm [8], which is given for the purpose of self-explanation.

1. Each node in $G_c$ will be assigned an index $(i, j)$ to show that it is the $j$th node on the $i$th chain.

2. In addition, each node $v$ on the $i$th chain will be associated with an index sequence of length $\alpha_i$: $(1, j_1) \ldots (1, j_l)$ (as illustrated in Fig. 6(a)) such that any node with index $(x, y)$ is a descendant of $v$ if $x = i$ and $y \geq j$ or $x \neq i$ but $y \geq j$, where $\alpha_i$ is the number of the node-disjoint chains, equal to the width of $G_c$.

We can also store all the index sequences as a matrix $M$ as illustrated in Fig. 6(b), in which each entry $M(v, j)$ is the $j$th element in the index sequence associated with node $v$. So, a node $u$ with index $(u) = (i, j)$ is a descendant of another node $v$ if $M(v, i) \leq j$. Thus, using $M$, a reachability checking can be done in $O(1)$ time.

However, if we don’t decompose $G$, but directly apply Chen’s algorithm to it, at least five chains will be produced since there exists a subset of $U = \{b, f, g, i, j\}$ in $G$ such that each pair of nodes in it are not connected through a path in $G$. So it is not possible to decompose $G$ into a set with fewer chains. Therefore, a $13 \times 5$ matrix has to be created, which is much larger than the $7 \times 2$ matrix shown in Fig. 6(b), generated for $G_c$. We notice that $G$ contains 13 nodes.

**C. Recognizing Critical Nodes**

From the discussion in the previous subsection, we know that all the critical nodes need to be recognized to construct $G_c$. Now we discuss an efficient algorithm for this task.

We will search $T$ bottom up and produce a subtree $T'$ of $T$ such that only the critical nodes and the nodes from $V_{start}$ are included. Initially, $T'$ is set to $\emptyset$, and all the nodes in $V_{start}$ are marked. Then, during the traversal of $T$, any node belonging to $V_{start}$ or any critical node, once it is recognized, will be inserted into $T'$. To this end, each node $v$ inserted into $T'$ will be associate with two links, denoted $parent(v)$ and $left-sibling(v)$, respectively. $parent(v)$ is used to point to the parent of $v$ in $T'$ while $left-sibling(v)$ points to a node in $T'$ created just before $v$, which is not a descendant of $v$ in $T$.

Concretely, $parent(v)$ and $left-sibling(v)$ will be created as follows.

(i) Let $v$ be the node currently inserted into $T'$.

(ii) If $v$ is not the first node inserted into $T'$, we do the following:  

![Fig. 6 A $G_c$, its decomposition and its reachability matrix](image)
Let $v'$ be the node inserted just before $v$. If $v'$ is not a child (descendant) of $v$, create a link from $v$ to $v'$, called a left-sibling link and denoted as left-sibling($v) = v'$. If $v'$ is a child (descendant) of $v$, we will first create a link from $v'$ to $v$, called a parent link and denoted as parent($v) = v$. Then, we will go along the left-sibling chain starting from $v'$ until we meet a node $v''$ which is not a child (descendant) of $v$. For each encountered node $u$ except $v''$, set parent($u) = v$. Finally, set left-sibling($v) = v''$.

Fig. 7 is a pictorial illustration of this process.

In Fig. 7(a), we show the navigation along a left-sibling chain starting from $v'$ when we find that $v'$ is a child ( descendant) of $v$. This process stops whenever we meet $v''$, a node that is not a child (descendant) of $v$. Fig. 7(b) shows that the left-sibling link of $v$ is set to point to $v''$, which is previously pointed to by the left-sibling link of $v$’s left-most child.

Extending the above process with the recognition of critical nodes and the computation of crossing ranges, we get an efficient algorithm for finding all the critical nodes.

Algorithm find-critical($T$)

begin
1. $T' \leftarrow \emptyset$. Mark any node in $T$, which belongs to $V_{start}$.
2. Let $v$ be the first marked node encountered during the bottom-up searching of $T$. Insert $v$ in $T'$.
3. Let $u$ be the currently encountered node in $T$. Let $u'$ be the node inserted into $T'$ just before $u$. Do (4) or (5), depending on whether $u$ is a marked node or not.
4. If $u$ is marked, then insert $u$ into $T'$ and do the following.
   (a) If $u'$ is not a child (descendant) of $u$, set left-sibling($u) = u'$ (i.e., a link from $u$ to $u'$).
   (b) If $u'$ is a child (descendant) of $u$, we will first set parent($u') = u$. Then, we will go along a left-sibling chain starting from $u'$ until we meet a node $u''$ which is not a child (descendant) of $u$. For each encountered node $w$ except $u''$, set parent($w) = u$. Also, set left-sibling($u) = u''$. (See Fig. 7(b) for illustration.)
   Calculate initial $a_u$ and $b_u$ according to Definition 1. Let $W$ be the set of all the encountered nodes during the navigation along the left-sibling chain (not including $u'$). Set $a_u \leftarrow \min\{\min_{w \in W}\{a_w\}, a_u\}$ and $b_u \leftarrow \max\{\max_{w \in W}\{b_w\}, b_u\}$.
5. If $u$ is a non-marked node, then do the following.
   (c) If $u'$ is not a child (descendant) of $u$, $u$ is ignored.
   (d) If $u'$ is a child (descendant) of $u$, we will go along a left-sibling chain starting from $u'$ until we meet a node $u''$ which is not a child (descendant) of $u$. If there are more than one node in $W$ such that crossing ranges not within $T[u]$, insert $u$ into $T'$, and compute $a_u$ and $b_u$ as (4.b). Otherwise, $u$ is ignored.

end

In the algorithm, each node $v$ belonging to $V_{start}$ is simply inserted into $T'$, by which its $\{a_v, b_v\}$ is computed. (See 4.a and 4.b. in the algorithm.) For a node not belonging to $V_{start}$ we will check whether it satisfy the conditions given in Definition 2. If it is the case, it will be inserted into $T'$. At the same time, its crossing range will be calculated. Otherwise, it will be ignored. (See 5.c and 5.d in the algorithm.)

Obviously, the algorithm requires only $O(e)$ time since each node in $T$ is accessed at most two times and for each node $v$ out-degree($v)$ edges will be visited. We have

$$\sum_{v \in V_{tot}} \text{out} - \text{deg}(v) = e.$$ 

Example 2 Consider the spanning tree $T$ shown in Fig. 2. Applying the above algorithm to $T$, we will generate a series of data structures shown in Fig. 8.

First, the nodes $d$, $f$, $g$, and $h$ in $T$ are marked. During the bottom-up search of $T$ the first node created for $T'$ is node $d$ (see Fig. 8(a)). In a next step, node $b$ is met. But no node for $b$ in $T'$ is created since $b$ is not marked and has only one child in the current $T'$ (see 5.d in Algorithm find-critical( )). In the third step, node $f$ is encountered. It is a marked node and to the right of node $d$. So a link left-sibling($f) = d$ is created (see Fig. 8(b)). In the fourth step, node $g$ is encountered and a second left-sibling link is generated (see Fig. 8(c)). In the fifth step, node $e$ is met. It is not marked. But it is the parent of node $g$. So the left-sibling chain starting from node $g$ will be searched to find all the children (descendants) of $e$ along the chain, which appear in $T'$.

Furthermore, the number of such nodes is 2 and the crossing ranges of both nodes $f$ and $g$ are outside of $T[e]$. Thus, node $e$ is inserted into $T'$ (see Fig. 8(d)).

Here, special attention should be paid to the replacement of left-sibling($f) = d$ with left-sibling($e) = d$, which enable us to easily find the lowest common ancestor of $d$ and some other nodes from $V_{start}$ if any. In the next two steps, we will meet node $i$ and $j$. But no nodes will be created for them. Fig. 8(e) demonstrates the last step of the whole process. Especially, the tree shown in Fig. 8(e) is $T'$, which contains all the critical nodes and the nodes from $V_{start}$.

Form $T'$, $T_c$ and $G_c$ can be easily constructed as shown in Fig. 8.

The following proposition shows the correctness of the algorithm.

Proposition 2 Let $G = (V, E)$ be a DAG. Let $T$ be a spanning tree (or a spanning forest) of $G$. Algorithm find-critical( ) generates $T'$ of $G$ with respect to $T$ correctly.

Proof. To show the correctness of the algorithm, we should prove the following: (1) each node in $T'$ is a critical node or a node from $V_{start}$, (2) any node not in $T'$ is neither a critical
node nor a node from $V_{start}$; (3) for each edge $(u, v)$ in $T'$ there is a path from $u$ to $v$ in $T$, which does not contain a critical node or a node from $V_{start}$ (except the two end points).

First, we prove (1) by induction on the height $h$ of $T'$. The height of a node $v$ in $T'$ is defined to be the longest path from $v$ to a leaf node in $T'$.

**Basis step.** When $h = 0$, each leaf node in $T'$ is a node in $V_{start}$. So it is correct.

**Induction hypothesis.** Assume that every node appearing at height $h = k$ in $T'$ is a critical node or a node from $V_{start}$. We prove that every node $v$ at height $k + 1$ in $T'$ is also a critical node or a node from $V_{start}$.

If $v \in V_{start}$, the proof is trivial. Assume that $v \notin V_{start}$. According to the algorithm, $v$ has at least two children with their crossing ranges not within $T[v]$ (see 5.d in Algorithm find-critical()). Assume that $v_1$ and $v_2$ are two such nodes. If these two children belong to $V_{start}$, the claim holds. Now we assume that $v_1$ does not belong to $V_{start}$. Then, its height must be the same as or lower than $k$. According to the induction hypothesis, it is a critical node. Therefore, there must exist a subset $S \subseteq V_{start}$ such that $v_1$ is the lowest common ancestor of all the nodes in $S$. Therefore, $v$ is an ancestor of all the nodes in $S$, which contains at least one node whose crossing range is outside of $T[v]$. Let $v_3$ be such a node. Thus, $v$ is the lowest common ancestor of $v_2$ and $v_3$. (Here, we assume that $v_2$ is from $V_{start}$; If $v_2$ does not belong to $V_{start}$, repeating the above argument for $v_2$ will prove the claim.)

In order to prove (2), we notice that only in two cases no node is generated in $T'$ for a node $v \notin V_{start}$: (i) $v$ is to the right of a node, for which a node in $T'$ is created just before $v$ is encountered (see 5.c in Algorithm find-critical()); (ii) $v$ is the parent (ancestor) of a node $u$, for which a node in $T'$ is generated; but $u$ is the only node encountered when navigating the corresponding left-sibling chain (see 5.d in Algorithm find-critical()) or there are not more than one children such that their crossing ranges are outside of $v$'s interval. Obviously, in both cases, $v$ cannot be a critical node.

(3) can be seen from the fact that each parent link corresponds to a path in $T$ and such a path cannot contain any critical node (except the two end points) since the nodes in $T$ are checked level by level bottom-up.

In the following, we show that for any DAG $G(V, E)$ we always have:

$$|V_{critical}| < |V| - |V_{start} \cup V_{end}|.$$

Since $G$ is a DAG, it has at least one node whose in-degree is 0. Using this node as the starting point to search $G$ in preorder, we get a spanning tree (forest) $T$. Then, with respect to $T$, this node cannot be a critical node. Also, it does not belong to $V_{start} \cup V_{end}$. Thus, the above inequality holds, which implies the following proposition.

**Proposition 3** The number of the nodes in $G$ is strictly larger than the number of the nodes in $G_c$.

**Proof.** Remember that $G_c = T_c \cup E_{cross}$. So the node set of $G_c$ is $V_{critical} \cap V_{start} \cap V_{end}$. We notice that $V_{critical} \cap (V_{start} \cup V_{end}) = \emptyset$, which indicates that $|V_{critical} \cup V_{start} \cup V_{end}| = |V_{critical}| + |V_{start} \cup V_{end}| < |V|$.

IV. RECURSIVE GRAPH DECOMPOSITION

We note that $G_c$ itself can be decomposed, leading to a further space decrement. Repeating this operation, we will get a recursive decomposition of $G$. In this subsection, we elaborate this process.

A. Recursive Decomposition

Let $G_0$ be a DAG. Denote by $T_0$ a spanning tree of $G_0$. Denote by $E_{cross}$ the set of all the cross edges with respect to $T_0$. Then, as discussed in the previous section, $T_0$ and $G_1 = T_0 \cup E_{cross}$ make up a decomposition of $G_0$, where $T_0$ is the critical tree of $G_0$. Our purpose is to find a series of tree structures:

$$T_0, T_1, ..., T_k, (k \geq 1)$$

such that $T_0$ is a spanning tree of $G_0$ and each $T_i (i = 1, ..., k-1)$ is a spanning tree of $G_i = T_i \cup E_{cross}^i$, where $T_0$ is the critical tree of $G_1$, and $E_{cross}^i$ is a set of all the cross edges with respect to $T_i$.

The following example helps for illustration.

**Example 3** Denote by $G_0$, the graph shown in Fig. 2. Denote by $T_0$, the spanning tree represented by the solid arrows in the graph. With respect to $T_0$, $E_{cross}$ is a graph as shown by the dashed arrows in the same figure, and $T_1$ is a forest as shown in Fig. 4(a). Then, $G_1 = T_1 \cup E_{cross}$ is a graph as shown in Fig. 4(b).

A spanning tree $T_1$ of it is shown by the solid arrows in Fig. 9(a). With respect to this spanning tree, $(h, g)$ and $(h, k)$ are two forward edges and can be removed. So $E_{cross}$ is a graph as shown in Fig. 9(b), containing only two disconnected edges. Their respective start nodes are $g$ and $c$. Accordingly, $T_2$ is also a graph containing two disconnected edges, as shown in Fig. 9(c).

![Fig. 9 Illustration for recursive graph decomposition](image)

$G_2$ will be constructed in the same way as $G_1$. That is, $G_2$ is equal to $T_2 \cup E_{cross}^2$, as shown in Fig. 10(a).

A spanning tree $T_2$ of $G_2$ is shown in Fig. 10(b). With respect to $T_3$, $E_{cross}^3$ is a graph containing only one edge, and $T_3$ contains only two single nodes, as shown in Fig. 10(c) and (d).
respectively. So, we have $G_3 = T_3^1 \cup E^2_{rem} = E^2_{rem}$, and $T_3$ is the same as $G_3$ (see Fig. 10(e)).

B. Reachability Checking Based on Recursive Decomposition

From the above discussion, we can see that for a given DAG $G$, we can always find a series of trees: $T_0, T_1, ..., T_{k-1}, (k \geq 1)$, and a series of subgraphs: $G_0 = G, G_1, ..., G_k$ such that $T_i$ is a spanning tree of $G_i (i = 0, ..., k - 1)$. We refer to $G_k$ as the remaining graph of $G$, denoted as $G$. It can be a graph or a tree.

In terms of the recursive graph decomposition, we are able to associate each node $v$ in $G_0$ with two sequences: an interval sequence and an anchor node sequence to check reachability:
1) $[a^v_0, b^v_0], ..., [a^v_{j-1}, b^v_{j-1}], (j \leq k - 1)$ where each $[a^v_j, b^v_j]$ is an interval generated by labeling $T_i$;
2) $(x^v_0, y^v_0), ..., (x^v_{j-1}, y^v_{j-1}), (I \leq j)$ where each $x^v_i$ is a pointer to an anchor node of the first kind (a node appearing in $G_{i+1}$) while each $y^v_i$ a pointer to an anchor node of the second kind. Since a node may appear in more than one spanning trees, its anchor node sequence may contain more than one entries.

Example 4 Continued with Example 3. By creating intervals for the nodes in $T_0, T_1, T_2$ and $T_3$ (see Fig. 2, Fig. 9(a), Fig. 10(b) and (e), respectively), we will generate an interval sequence for each node as shown in Fig. 12(a).

\[
\begin{array}{c|c|c|c}
T_0 & T_1 & T_2 & T_3 \\
\hline
a & 0, 13 & a & \\
{b} & 1, 5 & b & <d, > \\
c & 2, 5 & [3, 4] & [1, 3] & c & \langle <e, > \rangle \langle <g, > \rangle \\
d & 5, 6 & [5, 7] & [3, 4] & 0, 2 & d & \langle <d, > \rangle \langle <d, > \rangle \langle <d, > \rangle \\
e & [7, 10] & [1, 7] & e & \langle <e, > \rangle \langle <g, > \rangle \\
f & 8, 9 & [4, 7] & f & \langle <e, > \rangle \\
g & [9, 10] & [2, 4] & 0, 4 & g & \langle <g, > \rangle \langle <g, > \rangle \langle <d, > \rangle \\
h & [10, 13] & [0, 7] & h & \langle <h, > \rangle \langle <g, > \rangle \\
i & [11, 12] & i & \\
j & [12, 13] & j & \\
k & [4, 5] & [6, 7] & [2, 4] & [1, 2] & k & \langle <k, > \rangle \langle <k, > \rangle \langle <d, > \rangle \\
p & [3, 5] & p & \langle <p, > \rangle \\
r & [6, 10] & r & \langle <r, > \rangle \\
\end{array}
\]

For this example, the remaining graph is $\emptyset$.

Now we discuss how to use the interval sequences and anchor node sequences to check reachability. First, we notice that the anchor node sequences imply a graph, in which there exists an edge $(u, v)$ iff there is an entry $<x, y>$ in the anchor node sequence associated with $u$ such that $x = v$, or $y = v$. The edge is labeled with $\{i, *\}$ or $\{i, **\}$ with $i$ used to indicate that $<x, y>$ is the $i$th entry in the corresponding anchor node sequence. If $x = v$, the edge is labeled with $\{i, *\}$. If $y = v$, the edge is labeled with $\{i, **\}$. We refer to such a graph as a transitive closure core graph of $G$ (or simply core graph of $G$) and denote it by $G_{core}$. In Fig. 14, we show a $G_{core}$ corresponding to the anchor node sequences shown in Fig. 12(b).

In the graph, each loop represents an edge from a node to itself, and each edge is labeled with one or more pairs. For example, edge $(b, d)$ labeled with $\{1, *\}$ represents that $d$ is an anchor node (of the first kind) of $b$ and $d$ appears in $G_i$ while edge $(f, e)$ labeled with $\{1, **\}$ represents that $e$ is an anchor node (of the second kind) of $f$ and $e$ appears in $G_i$. An edge with multiple labels represents several edges with different labels. For example, the edge $(g, g)$ (represented as a loop) labeled with $\{1, *, \}, \{1, **\}$, and $\{2, *\}$ stands for three edges with each going from $g$ to $g$, but labeled differently.

In order to check whether $v$ is an ancestor of $u$, we will search two paths in $G_{core}$ starting from $v$ and $u$, respectively. The path starting from $v$, referred to as $P_v$, contains only the edges labeled with $\{i, *\}$ while the path starting from $u$, referred to as $P_u$, contains only the edges labeled with $\{i, **\}$. Each time we reach two nodes $v'$ and $u'$ through two edges labeled respectively with $\{i, *\}$ and $\{i, **\}$, we will check whether $\{a^v_{i'}, b^v_{i'}\}$ subsumes $\{a^u_{i''}, b^u_{i''}\}$ (Remember that each node in $G_0 = G$ is associated with an interval sequence $\{a^v_{i'}, b^v_{i'}\}, ..., \{a^v_m, b^v_m\}$ for some $m \geq 0$). If it is the case, $v$ is an ancestor of $u$. Otherwise, we traverse along $P_v$ and $P_u$, reaching $v''$ and $u''$ through two edges labeled respectively with $\{i + 1, *\}$ and $\{i + 1, **\}$ and checking $\{a^v_{i''}, b^v_{i''}\}$ against $\{a^u_{i''}, b^u_{i''}\}$. We continue this process. After $I$ steps for some $I$,
we will meet two nodes $v'''$ and $u'''$ such that $v'''$ does not have an out-going edge labeled with $(l + 1, *)$ or $u'''$ does not have an out-going edge labeled with $(l + 1, **)$. If $[a_i^\ell, b_i^\ell]$ subsumes $[a_i^{\ell'}, b_i^{\ell'}]$, $v$ is an ancestor of $u$. Otherwise, we further check whether $l = k$. If it is the case, we will check whether $u'''$ is reachable from $v'''$ in $G$.

**Example 5** Continued with Example 4. To check whether $r$ is an ancestor of $p$, we will first explore two paths in the graph shown in Fig. 14, starting from $r$ and $p$, respectively. First, we check $[a_i^\ell, b_i^\ell] = [6, 10]$ against $[a_i^\ell, b_i^\ell] = [3, 5]$ (see Fig. 2 to know the interval values) and find that $[6, 10]$ does not subsume $[3, 5]$. Then, we go from $r$ along an edge labeled with $(1, *)$ to $e$; and from $p$ along an edge labeled with $(1, **)$. If it is the case, we will check $[a_i^{\ell'}, b_i^{\ell'}] = [1, 7]$ against $[a_i^{\ell'}, b_i^{\ell'}] = [3, 4]$ (see Fig. 9(a) to know the interval values). Since $[1, 7]$ subsumes $[3, 4]$, we know that $e$ is an ancestor of $c$, which implies that $r$ is an ancestor of $p$.

**Proposition 4** Let $G$ be a DAG, and $G_0 = G, G_1, ..., G_k (k \geq 1)$, be a series of subgraphs as defined in the previous subsection. $T_0, T_1, ..., T_k$ be a series of trees such that each $T_i$ is a spanning tree of $G_i$. Let $u$ and $v$ be two nodes in $G$. $u$ is reachable from $v$ through a path in $G$ if and only if there exist two paths in $G_{core}$:

- $v_0 = v \rightarrow v_1 \rightarrow \cdots \rightarrow v_j \quad (0 \leq j \leq k)$
- $u_0 = u \rightarrow u_1 \rightarrow \cdots \rightarrow u_j$

such that each $(v_i, v_j)$ is labeled with $(i, *)$, each $(u_i, u_j)$ is labeled with $(i, **)$, and one of the following two conditions is satisfied:

1. $j < k$, and $u_i$ is reachable from $v_i$ through a path in $T_i$; or
2. $j = k$, and $u_i$ is reachable from $v_i$ through a path in $G_i$.

**Proof.** If-part. We prove the if-part by induction on $k$.

Basis step. When $k = 0$, the proof is trivial.

Induction hypothesis. Assume that when $k = l$ if the if-part holds. We consider the case when $k = l + 1$. If $j \leq l$, in terms of the induction hypothesis, the if-part holds. Assume that $j = l + 1$.

Since $u_{l+1}$ is reachable from $v_{l+1}$ through a path in $G_{l+1}$, $u_i$ must be reachable from $v_i$ in $G_i$ by Lemma 1 and 2. (Note that $v_{l+1}$ is an anchor node of the first kind of $v_i$ and $u_{l+1}$ is an anchor node of the second kind of $u_i$.) In terms of the induction hypothesis, $u$ is reachable from $v$.

Only-if-part. If $u_0 = u$ is reachable from $v_0 = v$, there will be a path in $T_0$ from $v_0$ to $u_0$ or $u_1$ is reachable from $v_1$ in $G_1$. Similarly, $u_i$ is reachable from $v_i$ in $G_i$, there will be a path in $T_i$ from $v_i$ to $u_i$, or $u_2$ is reachable from $v_2$ in $G_2$. Repeating this argument, we will get the proof.

The above proposition shows that to check whether $u$ is reachable from $v$, we need to search two paths in $G_{core}$ and at each step to examine whether $a^{\ell'} \in [a_i^\ell, b_i^\ell]$. Clearly, this process needs only $O(k)$ time and the space requirement for all the interval sequences and anchor node sequences is bounded by $O(kn)$. In addition, we need $O(n \cdot \omega)$ to store the matrix created for the remaining graph $G = G_k$, where $n$ stands for the number of the nodes in $G$, and $\omega$ for the width of $G$. Since $O(n^{1.5} \omega)$ time is needed to decompose $G$ into node-disjoint chains by using Chen’s method [8], the total cost for generating a compressed transitive closure is bounded by $O(ke + \omega^{1.5} \omega)$.

For different applications, we can set $k$ to be different constants to get effective space deduction, but still have a constant query time. We also notice that this is a biased trade-off of time for space since each step of decomposition will reduce both the number of the nodes and the width of $G$.

**V. EXPERIMENTS**

In this section, we report the test results. We conducted our experiments on a DELL desktop PC equipped with Pentium III 1.0 Ghz processor, 512 MB RAM and 20GB hard disk. The programs are compiled using Microsoft virtual C++ compiler version 6.0, running standalone.

**A. On the Tested Methods**

In the experiments, we have tested six methods:

- Chain decomposition by Chen et al. (CD for short) [8],
- Tree encoding by Agrawal et al. (TE for short) [6],
- 2-hop labeling by Cohn et al. (2-hop for short) [9],
- Dual labeling by Wang et al. (Dual-II for short) [35],
- Matrix multiplication by Warren (MM for short) [37],
- Recursive DAG decomposition (discussed in this paper, RDD for short).

The theoretical computational complexities of these methods are listed in Table 1 (in Section II).

In the experiments, the tree-path cover [20] is not included since it does not work in some cases. In fact, for all the graphs tested in our experiments, their weighted directed graphs contain SCCs; but how to handle them is not discussed in [20]. Jagadish’s chain decomposition is not included, either. It is because Chen’s method works in a similar way, but has a much better labeling time. For the dual labeling, we implemented Dual-II, instead of Dual-I for tests. For non-sparse graphs, Dual-I needs even more space than any traditional matrix-based method; no compression in any sense.

**B. Test Results**

The experiments altogether tested three groups of data: large but sparse DAGs, large and non-sparse DAGs, and dense DAGs (but with relatively small number of nodes) to make a proper comparison. In these tests, we measured the space overhead, and the time spent on the generation of compressed transitive closures (i.e., labeling time), as well as the time for checking reachability.

**1) Tests on Sparse Graphs:** In this group of tests, we first generate a binary tree of 15000 nodes. Then, add randomly edges to the tree. The number of the added edges ranges from 1000 to 5000 to create different graphs. For each generated graph, Tarjan’s algorithm is used to find SCCs as a preprocessor. All SCCs are then removed.

In Table II, we show the average size of the data structures generated by the different methods, and the average times spent on generating such data structures.
In this table, RDD(k) (k = 1, 2, 3) represents a k-level recursive DAG decomposition, by which a series of spanning trees: T0, ..., Tk, and a remaining graph are created.

<table>
<thead>
<tr>
<th>DATA SIZE AND LABELING TIME – SPARSE GRAPH</th>
<th>Data size (16 bits)</th>
<th>Labeling time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>30254</td>
<td>15.764</td>
</tr>
<tr>
<td>TE</td>
<td>39247</td>
<td>12.023</td>
</tr>
<tr>
<td>2-hop</td>
<td>801217</td>
<td>24145</td>
</tr>
<tr>
<td>Dual-II</td>
<td>36380</td>
<td>42.227</td>
</tr>
<tr>
<td>MM</td>
<td>14063750</td>
<td>675.812</td>
</tr>
<tr>
<td>RDD(1)</td>
<td>28764</td>
<td>11.564</td>
</tr>
<tr>
<td>RDD(2)</td>
<td>21673</td>
<td>10.786</td>
</tr>
<tr>
<td>RDD(3)</td>
<td>18765</td>
<td>10.988</td>
</tr>
</tbody>
</table>

From the table, we can see that RDD(3) has the lowest space overhead, but needs a little bit more labeling time than RDD(2). But RDD(2) is better than RDD(1) in both space overhead and labeling time. Although RDD(2) needs more time to generate one more spanning tree than RDD(1), it spends less time to decompose a smaller remaining graph than RDD(1). Chen’s method is better than all the other four approaches in space overhead. It is because for this kind of graphs, the pair sequences associated with the nodes are quite short. But Agrawal’s uses less time than it to label a graph since generating the interval sequences for the nodes in a graph by Agrawal’s needs much less time than decomposing that graph into node-disjoint chains by Chen’s. Dual-II also has very good performance since the TLC search trees created by it are very small, which are proportional to the number of non-tree edges. 2-hop can somehow reduce the size of transitive closures stored as matrices. But it took too much time (more than 6 hours) for the task.

Fig. 15 shows the average query time over the tested graphs. Each query is a pair (x, y) to check whether node x is an ancestor of node y. For each graph, we have checked up to 100,000 queries randomly generated and recorded the accumulated time.

In this figure, we use RDD to represent all the three levels of the recursive DAG decomposition since they have almost the same query time. From this figure, we can see that Warren’s method is best. (In our implementation, a boolean matrix is simply stored as bit strings.) Chen’s method and the RDD are comparable; and Agrawal’s tree encoding is slightly better than Dual-II since each time to check reachability the TLC search tree may be explored by Dual-II. But by the tree encoding method, a quite short pair sequence is visited in a binary searching way. Although by Chen’s method the matrix maintained is much larger than the RDD, they both require a constant query time and no significant difference can be observed.

2) Test on Non-sparse Graphs: In the second group of experiments, two kinds of DAGs are tested.

(i) tree-based
Any graph of this type is generated by constructing a tree of 20000 nodes. In the tree, each node is of a random number of children from zero to six. Then, add randomly up to 10000 cross edges to the tree. On average, the outdegree of each node is 2.5, and the length of each path is 8.

(ii) layered graph
Any graph of this type contains 8 levels with each containing 680 nodes. Each node at a level (except for the lowest level) has a number of children from two to five. Altogether, it has 68786 edges.

Table III shows the average size of generated data structures and the average labeling time.

<table>
<thead>
<tr>
<th>DATA SIZE AND LABELING TIME – TREE-BASED GRAPH</th>
<th>Data size (16 bits)</th>
<th>Labeling time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>196506</td>
<td>13.764</td>
</tr>
<tr>
<td>TE</td>
<td>210356</td>
<td>17.125</td>
</tr>
<tr>
<td>Dual-II</td>
<td>3163420</td>
<td>591.227</td>
</tr>
<tr>
<td>MM</td>
<td>25010001</td>
<td>286.812</td>
</tr>
<tr>
<td>RDD(1)</td>
<td>109646</td>
<td>10.064</td>
</tr>
<tr>
<td>RDD(2)</td>
<td>62876</td>
<td>11.786</td>
</tr>
<tr>
<td>RDD(3)</td>
<td>65300</td>
<td>12.568</td>
</tr>
</tbody>
</table>

In the table, 2-hop is not included since it too long to generate labels. We only report the results of the other five methods. First, we remark that all the different levels of our DAG decomposition are much better than the other four strategies both in the space overhead and labeling time. Especially, a higher level of the DAG decomposition needs less space to store labels than a lower level of the decomposition although some more labeling time is required.

Our method is better than Chen’s method since the matrix constructed for a decomposed graph is much smaller than the matrix for the original graph. However, Chen’s method is better than Agrawal’s. It is because the width ω of a graph is in general much smaller than the number λ of the leaf nodes of a spanning tree. We notice that the number of the columns of a matrix generated by Chen’s is bounded by ω while the length of an interval sequence created by Agrawal’s is by λ. Dual-II even needs more space and more time than Warren’s. This shows that this method is totally not suitable for non-sparse graphs since the space complexity O((e - n)2) and the time complexity O((e - n)) of this method become respectively O(n2) and O(n) or more when a graph is not sparse. Although both Dual-II and Warren’s are of the same theoretical space and time complexities, the boolean operations by Warren’s make it more efficient than Dual-II.

The third level of the graph decomposition is just a little larger than the original graph while Agrawal’s needs more than 7 times of space, Chen’s about 6 times, and Warren’s about 800 times. Dual-II even needs more space and time than Warren’s.
In Fig. 16, we show the time spent on the query evaluation. From this figure, we can see that both our method and Chen’s are a little bit worse than Warren’s, but much better than Agrawal’s and Dual-II. The figure also shows that Agrawal’s is better than Dual-II. The reason for this is that the TLC search tree produced by Dual-II may not be balanced. Then, the query time of Dual-II may be larger than \( \log t \). This time complexity is derived based on the assumption that TLC is well balanced [35].

Table IV shows the sizes of the data structures generated by the different methods for storing the compressed transitive closure of the layered graphs, and the times spent on generating such data structures.

<table>
<thead>
<tr>
<th>Data size (16 bits)</th>
<th>Labeling time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>176000</td>
</tr>
<tr>
<td>TE</td>
<td>289784</td>
</tr>
<tr>
<td>Dual-II</td>
<td>74026442</td>
</tr>
<tr>
<td>MM</td>
<td>56250000</td>
</tr>
<tr>
<td>RDD(1)</td>
<td>92664</td>
</tr>
<tr>
<td>RDD(2)</td>
<td>82764</td>
</tr>
<tr>
<td>RDD(3)</td>
<td>80561</td>
</tr>
</tbody>
</table>

From this table, it can be observed that the time used by our method to generate a data structure for the layered graph’s transitive closure is again much less than all the other graph labeling strategies. More importantly, the discrepancy of the space overhead between ours and all the other strategies is huge.

We show the time for the query evaluation in Fig. 17. This figure demonstrates that our method needs slightly more time than Warren’s for checking reachability, but better than all the other graph labeling approaches. Together with Table 4, this shows that trading time for space by our method pays off.

3) Tests on Dense Graphs: In the third group of experiments, we have tested some DAGs with density near 0.25 (referred to as the dense-DAGs).

Any graph of this type contains 3000 nodes connected by 2230196 edges generated randomly. The density of the graph is 2230196/9000000 = 0.247.

In Table V, we show the sizes of the data structures generated by the different methods for storing the transitive closure of a dense-DAG, and the times spent on generating such data structures.

<table>
<thead>
<tr>
<th>Data size (16 bits)</th>
<th>Labeling time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>178654</td>
</tr>
<tr>
<td>TE</td>
<td>267838</td>
</tr>
<tr>
<td>Dual-II</td>
<td>771831</td>
</tr>
<tr>
<td>MM</td>
<td>25010001</td>
</tr>
<tr>
<td>RDD(1)</td>
<td>102654</td>
</tr>
<tr>
<td>RDD(2)</td>
<td>60764</td>
</tr>
<tr>
<td>RDD(3)</td>
<td>58561</td>
</tr>
</tbody>
</table>

As we can see, even for very dense graphs our method works well and compacts effectively the transitive closures. The time for generating data structures is also very low. In fact, a dense graph tends to have many forward edges, which can simply be moved without loss of any information on reachability. This may explain why our method has an advantage over the others. We also notice that the space overhead of Chen’s method is not much worse than ours. The reason for this is that the denser a graph is, the fewer chains will be generated.

Fig. 18 shows the query time. Again, our method works well. Although it is a little bit inferior to Warren’s, it is much more efficient than all the other graph labeling approaches. For a dense graph, the average size of the data structure by ours is small due to the large number of removed forward edges, leading to a reduction of average query time. Agrawal’s is in general worse than Chen’s since the number of the leaf nodes of any spanning tree is always larger than the number of chains found by Chen’s method. For this kind of graphs, Dual-II shows the worst performance.

VI. CONCLUSION

In this paper, a new method is proposed to compress transitive closures to support reachability queries. The main idea behind it is to decompose \( G \) into a series of spanning trees: \( T_0, \ldots, T_{k-1} \) (for some \( k \geq 1 \)), and a remaining graph \( \tilde{G} \), which
enables us to associate two sequences with each node in \( G \): an interval sequence and an anchor node sequence. Especially, in terms of the anchor sequences, a directed graph, called a core graph of \( G \), can be constructed, which can be used to control the process of reachability checking. The method needs \( O(k \cdot \omega^3 + n \cdot \omega) \) time to create a compressed transitive closure with \( O(k \cdot \omega + n \cdot \omega) \) space requirement, and \( O(k) \) query time, where \( n \) is the number of the nodes in \( G \), and \( \omega \) is the width of \( G \), defined to be the size of a largest node subset \( U \) of \( G \) such that for any pair of nodes \( u, v \in U \) there does not exist a path from \( u \) to \( v \) or from \( v \) to \( u \).

An extensive experiment is conducted to test different strategies over different kinds of graphs, which shows that our method is promising. Our method is also a flexible strategy. For different applications, \( k \) can be set to different constants to reduce space overhead. But the query time is still bounded by a constant.

REFERENCES


