Evaluation of Reachability Queries Based on Recursive DAG Decomposition

Yangjun Chen, Yibin Chen, and Yifeng Zhang

Abstract—Let $G(V, E)$ be a digraph (directed graph) with n nodes and e arcs. Digraph $G^* = (V, E^*)$ is the reflexive, transitive closure if $v \rightarrow u \in E^*$ iff 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 in graph databases, but also serve as fundamental operations used in many graph algorithms. A lot of strategies have been proposed 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 labeling, chain decomposition, 2-hop labeling, and path-trees, as well as partial index based methods. 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 deduct and decompose a graph into a series of spanning trees and transform a query q to a series of subqueries each evaluated against a spanning tree. Using the so-called tree labeling, each subquery needs only O(1) time. More importantly, the number of such subqueries is $\ll n$. Thus, q can be evaluated very efficiently. We demonstrate both analytically and empirically the efficiency and effectiveness of our method. While the query time of our method is orders of magnitude better than almost all the existing strategies, its indexing time and index sizes are comparable to them.

Index Terms—Reachability, spanning trees, graph decomposition, recursive graph decomposition, random graph analysis

1 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, denoted as $u \Rightarrow v$. The problem is known as *graph reachability*. In many applications, such as evaluation of recursive queries in deductive databases, type checking in object-oriented databases, XML query processing, social network, transportation network, internet traffic analyzing, semantic web, trace of infectious diseases, and metabolic network [24], 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 a 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 at very high query processing cost - O(e) time in the worst case, where $e = |E|$.

- *Yangjun Chen is with the Deptartment of Applied Computer Science, University of Winnipeg, Winnipeg, MB R3B 2E9, Canada. E-mail: y.chen@uwinnipeg.ca.*
- *Yibin Chen is with Intel Corp., Mountain View, CA 95054-1549 USA. E-mail: chenyibin@gmail.com.*
- *Yifeng Zhang is with the Department Computing Science, University of Alberta, Edmonton, AB T6G 2R3, Canada. E-mail: zh.sledge@gmail.com.*

Manuscript received 15 March 2021; revised 27 July 2021; accepted 3 November 2022. Date of publication 0 2022; date of current version 0 2022.
This work was supported by NSERC, Canada, under Grant 239074-01 (242523).
(Corresponding author: Yangjun Chen.)
Recommended for acceptance by G. Chen.
Digital Object Identifier no. 10.1109/TKDE.2022.3220191

There is much research on this issue to reduce space overhead but still keep a short query time, such as those discussed in [1], [2], [4], [5], [7], [10], [11]. All of them reduce the space requirement to some extent. However, the worst space overhead is yet in the order of O(n^2), or in the order of O(e), but with the query time near to O(e). In the case of large graphs, they cannot be efficient.

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

Specifically, we will decompose G into a series of subgraphs $G = G_0, G_1, \ldots, G_{k\text{-}1}$ ($k \geq 1$) and find their respective spanning trees (forests) T_0, T_1, …., $T_{k\text{-}1}$. Accordingly, we will associate each node u with two node sequences: $A_u = a_0, \ldots, a_l$ and $B_u = b_0, \ldots, b_l$ ($l \leq k$ - 1) with $a_0 = b_0 = u$, $a_j \Rightarrow a_{j+1}$ and $b_j \Leftarrow b_{j+1}$ for $j = 0, \ldots, l$ - 1. A_u is used to check reachabilty from u to any other node while B_u is used to check reachabilty from any other node to u. Thus, to check whether a node v is reachable from u, we will decompose the query into a series of subqueries as described below:

- Assume that the two node sequences associated with v are $A_v = a_0', \ldots, a_r'$ and $B_v = b_0', \ldots, b_r'$ ($r \leq k$).
- To answer query q: $u \Rightarrow v$?, we will evaluate a series of subqueries q_j ($j = 0, \ldots, s$, $s \leq min\{l, r\}$), by which we will test whether $a_j \Rightarrow b_j'$ within T_j.
- We evaluate these subqueries in turn, starting from q_0, until some q_j returns true, or all the subqueries are exhausted with each evaluating to *false*. In the former case, the answer is *true*. In the latter case, *false*.

Besides, we will also associate each u with an extra pair of integers (κ_u, μ_u), used as a filter. They are in fact two topological numbers with a very nice property [31]: If another node v, associated with (κ_v, μ_v), is reachable from u, we must have $\kappa_v \leq \kappa_u$ and $\mu_v \leq \mu_u$. Thus, $\kappa_v \nleq \kappa_u$ or $\mu_v \nleq \mu_u$ indicates a negation, and then in this case the scanning of the *A*- and *B*-sequences is unnecessary to provide a negative answer to the reachability query from u to v.

We decompose a query in this way since the evaluation of each q_j (checking whether $a_j \Rightarrow b_j'$ within T_j) can be done in constant time. Hence, the time complexity of a query evaluation is bounded by O(k) with O(kn) space requirement. Theoretically, $k = \mathrm{O}(\sqrt{n})$. However, our experiments show that $k \ll \sqrt{n}$ for all the tested graphs.

The remainder of the paper is organized as follows. In Section 2, we summarize the symbols and notations used in this paper. In Section 3, we review the related work. In Section 4, we discuss the main working process of our method to reduce and decompose a directed acyclic graph (*DAG*), based on which a transitive closure can be effectively compressed. In Section 5, we present some important technical details used by the main algorithm, such as recognition of critical nodes, and an efficient approach to find spanning trees of a DAG with more forward arcs to reduce the depth of recursive graph decomposition. Section 6 is devoted to the experiments. Finally, a short conclusion is set forth in Section 7.

2 Notations

In this section, we summarize all the symbols and notations used throughout the paper, in Table 1.

3 Related Work

In the past three decades, many interesting labeling-based strategies have been proposed to reduce both the precomputation time and storage cost with reasonable answering time. In general, all those methods can be categorized into two groups: *full indexing* and *partial indexing*. By full indexing, for both positive queries (*reachability* checking) and negative queries (*non-reachability* checking), the created index can be fully used. By partial indexing, however, only for some negative queries the index can be employed while for any positive query, as well as a large part of negative queries the index can be used only for doing some kinds of pruning of space when searching G, or totally useless. So the worst-case querying time complexity of all the partial index based methods is bounded by O($n + e$).

TABLE 1
Symbols and Notations

G	a directed graph
$u \Rightarrow v$	representing that v is reachable form u
T	a spanning tree of G
$T[v]$	subtree of T rooted at v
$[\alpha_v, \beta_v)$	interval associated with v, where α_v is v's preorder number (denoted as *pre*(v)) and β_v - 1 is equal to the largest preorder number among all the nodes in T [v]
$\{a_v, b_v\}$	cross range of node v
V_{start}	set of all the start nodes of cross arcs
V_{end}	set of all the end nodes of cross arcs
$V_{critical}$	set of all the critical nodes
T_c	critical tree of G (with respect to T), which contains all the nodes in $V_{critical} \cup V_{start} \cup V_{end}$
G_c	summary graph of G. G is decomposed into T and G_c. But in general, $G \neq T \cup G_c$.
T_c^i	critical tree of G_i
G_c^i	summary graph of G_i
v^*	v's anchor node of the first kind
v^{**}	v's anchor node of the second kind
ω_v	interval sequence associated with node v
ϖ_v	sequence of anchor node pairs associate with node v
A_v	node sequence associated with v, used to check reachability from node v
B_v	node sequence associated with v, used to check reachability to node v
parent(v)	link pointing to parent of v in T
left-sibling (v)	link pointing to the left sibling of v in T

In the following, we will review some of these two kinds of methods.

Full indexing

Chain decomposition methods. In [10], Jagadish suggested a method to decompose a DAG 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(1) query time and O ($\boldsymbol{\mu} n$) space overhead, where $\boldsymbol{\mu}$ is the number of chains. However, to find a minimum set of chains for a graph, Jagadish's algorithm needs O(n^3) time (see page 566 in [10]), and in the worst case, $\boldsymbol{\mu}$ is O(n).

The method discussed in [5] greatly improves Jagadish's method. It needs only O($n^2 + \boldsymbol{\omega}^{1.5}n$) time to decompose a DAG into a minimum set of node-disjoint chains, where $\boldsymbol{\omega}$ represents G's width. Its space overhead is O($\boldsymbol{\omega}n$) and its query time is bounded by a constant. In [7], the concept of the so-called general spanning tree is introduced, in which each arc corresponds to a path in G. Based on this data structure, the real space requirement becomes smaller than O($\boldsymbol{\omega}n$), but the query time increases to log $\boldsymbol{\omega}$.

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. [19]. 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 ($\boldsymbol{\lambda}$), where $\boldsymbol{\lambda}$ is the number of the leaf nodes in T. So the time and space complexities are bounded by O($\boldsymbol{\lambda}e$) and O ($\boldsymbol{\lambda}n$), respectively. The querying time is bounded by O(log $\boldsymbol{\lambda}$). In the worst case, $\boldsymbol{\lambda} = \mathrm{O}(n)$.

The method discussed in [24] 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 $\boldsymbol{N}$ (called a *TLC* matrix) is maintained, where t is the number of non-tree arcs (arcs 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 $\mathbf{N}(x_v, z_u) - \mathbf{N}(y_v, z_u) > 0$. The size of all labels is bounded by $O(n + t^2)$ and can be produced in $O(n + e + t^3)$ time. The query time is $O(1)$. As an improvement of *Dual-I*, *Dual-II* can reduce the space overhead from a practical viewpoint, but increases the query time to $\log t$.

2-hop labeling. The method proposed by Cohen et al. [4] 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_1h_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 [4], by which the overall label size is bounded by $O(n\sqrt{e}\log n)$, and the query time by $O(\sqrt{e})$ since the average size of each label is above $O(\sqrt{e})$. The time for generating labels is $O(n^4)$. The 2-hop labeling is improved by the so-called *3-hop labeling* [37] and *path-hop labeling* [38]. The path-hop labeling is slightly better than the 3-hop labeling with its indexing time and index size bounded by $O(ne)$ and $O(\boldsymbol{\lambda} n)$, respectively. Its query time is in the order of $O(\log^2 \boldsymbol{\lambda})$.

Path-tree decomposition. In 2011, Jin et al. [11], [12] discussed a method, by which a DAG G is decomposed into a set of node-disjoint paths. Then, a weighted directed graph G_w (called *path-graph* in [11]) is constructed, in which each node represents a path and there is an arc $i \rightarrow j$ if on path i there is a node connected to a node on path j. The weight associated with $i \rightarrow j$ is the number of such connections. Then, find a maximum spanning tree T_w (called a path-tree) of G_w and label the nodes in T_w with an interval in a way similar to Agrawal et al.'s. Together with the labels assigned to the nodes on all the paths, the intervals can be utilized to check part of reachability. To be a complete strategy, each node v has to be associated with a set, denoted $R^c(v)$, such that all the descendants of v, which appear on a path are dominated by a node in $R^c(v)$. In the worst case, the size of $R^c(v)$ is bounded by $\boldsymbol{\lambda}$. Therefore, the space complexity of this method is $O(\boldsymbol{\lambda} n)$. The query time and the labeling time are bounded by $O(\log^2 \boldsymbol{\lambda})$ and $O(\boldsymbol{\lambda} e)$, respectively (see the analysis of [12]). As mentioned above, $\boldsymbol{\lambda}$ is bounded by $O(n)$ in the worst case. Thus, theoretically, both the space requirement and the query time of this method are worse than Agrawal's [1].

SCARAB. In [29], a different method is discussed, in which a deducted *TC* over a subset V^* of nodes, called a *backbone* and denoted as $TC(V^*)$, is created. Then, for any pair (u, v), if u can reach v but through at least $\delta + 1$ intermediate nodes (where δ is a pre-determined constant), i.e., their distance is greater than δ, there must exist two nodes u^* and v^* in V^* such that u can reach u^*, v^* can reach v within δ steps, and u^* can reach v^* in $TC(V^*)$. To find $TC(V^*)$, an approximative algorithm is proposed in [29], which is based on the set-cover algorithm [32] and needs $O(\sum_{v \in V}(N_\delta(v) + E_\delta(v))$ time, where $N_\delta(v)$ and $E_\delta(v)$ denote the nodes and the arcs in v's forward δ-neighborhood, respectively. In the worst case, it is $O(nd^\delta)$, where d is the maximum out-degree

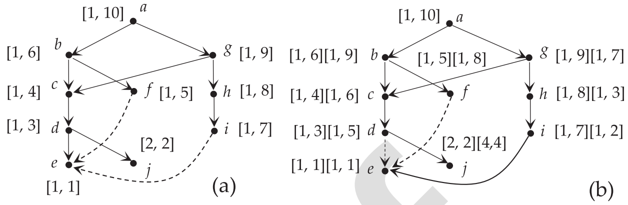

Fig. 1. Illustration for Grail labeling.

of a node in G. This running time is slightly improved by using the so-called *one-side* condition, by which V^* is defined to be a subset covering any pair (u, v) with *distance* $(u, v) = \delta$, where *distance*(u, v) is the length of a shortest path from u to v. The index size is obviously bounded by $O(n + e + |V^*|^2)$, but with a very high query time $O(d^{\lceil \delta/2 \rceil} + d^{2\delta}\log|V^*|)$. This method is further improved by Jin et al. [34]. Two new strategies are proposed. One is called *hierarchical-labeling* (*HL*) and the other is called *distribution-labeling* (*DL*). They are in fact two variants of backbones. By the *HL*, a node hierarchy is defined as $V_0 = V \not\supset V_1 \not\supset V_2 \not\supset \cdots \not\supset V_h$, with corresponding arc sets $E_0, E_1, E_2, \cdots, E_h$, such that $G_i = (V_i, E_i)$ is the (*one-side*) reachability backbone of $G_{i-1} = (V_{i-1}, E_{i-1})$, where $0 < i \le h$. Its theoretical labeling time is slightly better than *SCARAB* since G_i is constructed from $G_{i\text{-}1}$ and for the whole working process some time can be saved. However, the backbone is used in the same way as *SCARAB*. So it has almost the same index size and query time as *SCARAB*. By the *DL*, each single node makes up a layer, but with very high labeling time $O(n(n + e)L)$, where L is the maximal labeling size. Also, its index size and query time are comparable to *SCARAB*.

PWAH. The method discussed in [28] works in two phases. In the first phase, a deducted transitive closure of G will be created, by which for each node a bit vector is used to represent all those nodes reachable from it. In the second phase, each of such vectors will be compressed using the *PWAH*-8 encoding. In this way, the size of *TC* can be effectively reduced at cost of more query time since to check reachability the relevant compressed bit vectors have to be partially decompressed.

Partial labeling

GRAIL. The first partial labeling method proposed by Yildirim et al. [25] is a light-weight indexing structure. It traverses G for several times to create an interval sequence for each node, used as a filter. The interval for a node u, generated by a traversal, is of the form $L_u = [r_x, r_u]$, where r_u denotes the rank of the node u in a post-order traversal of the *DFS* tree of G (the tree created by exploring G in the depth-first fashion.) Here, the ranks are assumed to begin at 1, and all the children of a node are assumed to be ordered and fixed for that traversal. Further, r_x denotes the lowest rank for any node x in the subgraph rooted at u (i.e., including u.) For illustration, Fig. 1a shows an interval labeling on a DAG, assuming a left to right ordering of the children. In the figure, the solid arrows stand for the *DFS* tree while the dashed arrows for non-tree edges. As one can see, interval containment of nodes in a DAG is not exactly equivalent to reachability. For example, $L_h = [1], [8] \not\supset [1], [4] = L_c$, but $h \not\Rightarrow c$.

However, $L_u \not\subset L_v$ implies $v \not\Rightarrow u$. This shows that the intervals generated in this way can be used only as a filter.

For this reason, GRAIL employs multiple intervals that are obtained via random graph traversals to get stronger filtering power, as illustrated in Fig. 1b.

Let $L_u = L_u^1, \ldots, L_u^k$ and $L_v = L_v^1, \ldots, L_v^k$ be the interval sequences of u and v, respectively. If there exists i ($i \in \{1, \ldots, k\}$) such that $L_u^i \not\subset L_v^i$, u is definitely not a descendant of v. But if for all $i \in \{1, \ldots, k\}$ $L_u^i \subseteq L_v^i$, it cannot be determined whether u is a descendant of v, or *vice versa*. In this case, the whole G will be searched in the depth-first manner, but with the label sequences used to prune the search space. The labeling time of this method is bounded by $O(k(n + e))$. If k is chosen as a constant, the index size is proportional to $O(n)$ and can be established very fast. But in the worst case, the query time is $O(e)$ as if no index is established at all.

Feline. The method discussed in [31] is inspired by Dominance Graph Drawing, and uses two topological orders to label every node. Similar to *GRAIL*, each node v is labeled, but associated with a single pair of integers (x, y). If v is reachable from another node u, associated with (x', y'), we must have $x \le x'$ and $y \le y'$. Thus, $x \not\le x'$ or $y \not\le y'$ indicates a negation, and then no traversal of G is needed to negatively answer the reachability query from u to v. Otherwise, G will be searched in the *DFS* fashion.

Ferrari. The approach discussed in [36] uses up to k intervals for every node of G. Unlike *GRAIL*, some intervals are exact. But some are approximate, generated by merging several adjacent intervals to save space. It can be considered as a variant of *GRAIL*, but with no theoretical evidence that it is more pruning effective than *GRAIL*. Like *GRAIL*, part of G has to be searched when some approximate intervals are involved in a positive checking of reachability.

IP. The method discussed in [30] improves *GRAIL* by using k-min-wise independent permutation, by which each node u is associated with two labels: $L_{out}(u)$ and $L_{in}(u)$. $L_{out}(u)$ keeps up to k smallest numbers by the permutation π for $Out(u)$, denoted as $L_{out}(u) = min_k\{\pi(Out(u))\}$, whereas $L_{in}(u)$ keeps up to k smallest number by the same permutation π for $In(u)$, denoted as $L_{in}(u) = min_k\{\pi(In(u))\}$, where $Out(u)$ stands for a set containing all those nodes reachable from u, and $In(u)$ for a set containing all those nodes reachable to u. Together with this kind of permutation, it also uses two additional labels: the level label and the huge-node label, where the level label is used to stop *DFS* early as used in *GRAIL* while the huge-node label is used, together with the *topological folding label* discussed in [37], to handle high out-degree nodes. The size of a huge-node label is limited by the largest out-degree of nodes in G. Again, G may be searched to answer positive queries and some negative queries.

BFL. This method [35] works in a similar way to *IP* [30]. The only difference is that $L_{out}(u)$ and $L_{in}(u)$ are stored as two subsets of $\{1, \ldots, s\}$, generated by using a hash function applied over $Out(u)$ and $In(u)$, respectively, where s is a user-given number. It improves *IP* by the so-called 'bit-pruning' using the 'signatures' created by applying the hash function. The disadvantage of this method is the false positives caused by the signatures generated by the used hash function and a lot of time is needed to remove them. As with *IP*, the whole G may be searched whenever the index is useless for a query.

In Table 2, we compare our labeling method with all the other representative approaches.

TABLE 2
Comparison of Strategies

	Query time	Labeling time	Space overhead
Graph traversal	$O(e)$	0	$O(e)$
Matrix-based [27]	$O(1)$	$O(n^3)$	$O(n^2)$
Jagadish [10]	$O(1)$	$O(n^3)$	$O(\boldsymbol{\mu}\, n)$
Chen [5]	$O(1)$	$O(n^2 + \boldsymbol{\omega}^{1.5} n)$	$O(\boldsymbol{\omega} n)$
Interval-based [1]	$O(\log n)$	$O(ne)$	$O(\lambda n)$
Dual-I [24]	$O(1)$	$O(n + e + t^3)$	$O(n + t^2)$
Dual-II [24]	$O(\log t)$	$O(n + e + t^3)$	$O(n + t^2)$
2-hop [4]	$O(e^{1/2})$	$O(n^4)$	$O(ne \log n)$
Path-tree [11]	$O(\log^2 \lambda)$	$O(\lambda e)$	$O(\lambda n)$
SCARAB [29]	$O(d^{\lceil \delta/2 \rceil} + d^{2\delta} \log n')$	$O(nd^{\delta})$	$O(n + e + n'^2)$
PWAH [28]	$O(\tau)$	$O(n^3)$	$O(n\tau)$
GRAIL [25]	$O(e)$	$O(\boldsymbol{\kappa} e)$	$O(\boldsymbol{\kappa} n)$
HL [34]	$O(d^{\lceil \delta/2 \rceil} + d^{2\delta} \log n')$	$O(nd^{\delta})$	$O(n + e + n'^2)$
Feline [31]	$O(n + e)$	$O(n \log n + e)$	$O(n)$
Ferrari [36]	$O(n + e)$	$O(\boldsymbol{\kappa}^2 e + S)$	$O((\boldsymbol{\kappa} + s)n)$
IP [30]		$O((\boldsymbol{\kappa} + d)(e + n))$	$O((\boldsymbol{\kappa} + d)n)$
BFL [35]	$O(sn + e)$	$O(s(e + n))$	$O(sn)$
ours	$O(k)$	$O(kn)$	$O(kn)$

In Table 2, the first 10 methods and ours are full index based methods while all the others are partial index based. In the table, $\boldsymbol{\mu}$ is the number of chains by the Jagadish's method [10]. $\boldsymbol{\omega}$ is the width of a digraph while $\boldsymbol{\lambda}$ is the number of leaf nodes of the spanning tree of a digraph. For *Dual-I* and *Dual-II* [24], t is in the order of $O(e)$ in the worst case. $\boldsymbol{\tau}$ is the length of a compressed bit string using the *PWAH*-8 encoding [28]. $\boldsymbol{\kappa}$ is the number of intervals associated with a node in a partial labeling method. In the worst case, $\boldsymbol{\kappa} = O(n)$. For *SCRAB* [29] and *HL* [34], n' is $\boldsymbol{\alpha} n$ with $\boldsymbol{\alpha} \le 1$ being a constant and d is the largest out-degree of nodes in G. S is the time complexity to find the top s largest degree nodes in *Ferrari* [36]. r is the false positive rate in *BFL* [35]. Finally, k is the depth of recursive graph decomposition by our method (i.e., when we will stop the recursive decomposition.)

4 Main Algorithm

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 Section 4.1. Then, in Section 4.2, we demonstrate our basic graph decomposition based on the concept of *critical nodes*, as well as a method for checking the reachability based on such a graph decomposition. Finally, we show how a graph can be recursively decomposed in Section 4.3.

4.1 Basic Definition

Without loss of generality, we assume that G is *acyclic* (i.e., G is a DAG), as assumed in the existing work [1], [2], [4], [5], [6], [10]. However, if G contains cycles, we can find all the *strongly connected components* (*SCCs*) of G by using Tarjan's algorithm in $O(e)$ time [20] and collapse each of them into a representative node, transforming G to a DAG [16]. Clearly, each node in an *SCC* is equivalent to its representative node as far as reachability is concerned.

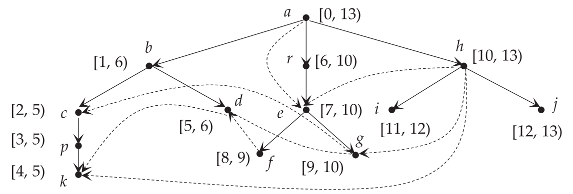

Fig. 2. A spanning tree and intervals.

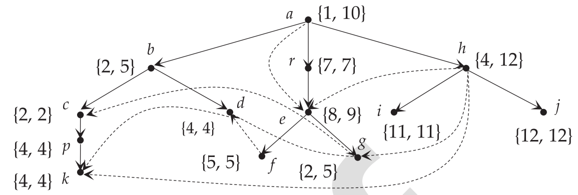

Fig. 3. Start nodes, end nodes, and crossing ranges.

We also use $u \rightarrow v$ to stand for an arc from u to v in a directed graph.

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

All cross, forward, and back arcs are referred to as non-tree arcs. (But in a DAG, we do not have back arcs since a back arc 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 (in which each non-tree arc is represented by a dashed arrow.)

As in [24], we can assign each node v in T an interval $[\alpha_v, \beta_v)$, where α_v is v's preorder number (denoted $pre(v)$) and β_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 $\alpha_u \in [\alpha_v, \beta_v)$ [24], as illustrated 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*.

Note that we may not be able to find a spanning tree, instead, a spanning forest T. In this case, we can always construct a spanning tree by creating a *virtual root* and connect it to the root of every tree in T with an arc. Therefore, we will not distinguish between spanning trees and spanning forests and always assume that there is a virtual root if what is found is a spanning forest.

4.2 Graph Decomposition and Reachability Checking

In this subsection, we discuss a kind of decomposition of $G(V, E)$: a spanning tree T and a summary graph 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. In general, G_c will contain E_{cross}. But some arcs from T are also included in G_c to transfer reachability information. For this purpose, we introduce some new concepts.

Denote by V' the set of all the *endpoints* of the cross arcs. Then, we have $V' = V_{start} \cup V_{end}$, where V_{start} contains all the *start nodes* while V_{end} contains all the *end nodes* of cross arcs. 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 arc (a, e) in the graph since it can be simply removed without impacting the checking of reachability.

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 with the children $v_1, \ldots, v_j$ in G. Let $[\alpha_i, \beta_i)$ $(i = 1, \ldots, j)$ be the interval of v_i. Set $a_v = min_i\{\alpha_i\}$ and $b_v = max_i\{\alpha_i\}$. 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 α_v itself.

For example, with respect to the spanning tree shown in Fig. 2, the crossing ranges of the nodes in G can be easily 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 ranges is to define the so-called *critical nodes*, which are used to determine all those nodes $\notin V_{start} \cup V_{end}$, 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_{start} with $|U| > 1$ such that for any two nodes $u_1, u_2 \in U$ they are not related by the ancestor/descendant relationship and v is the lowest common ancestor (*LCA*) 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 $V_{critical}$. 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_{start}. In addition, the crossing ranges of f and g are not within $T[e]$ (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 these 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 $V_{critical} = \{e\}$.

All the critical nodes can be recognized in linear time by using an algorithm to find *LCA*s. But we shift the discussion on this algorithm to Section 5.1.

The reason for imposing condition (2) in the above definition is that if any cross arc 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 $v \notin V_{start} \cup V_{end}$.

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 $V_{critical} \cup V_{start} \cup V_{end}$. In T_c, there is an arc from u to v iff there is a path P from u to v in T and P contains no other node in $V_{critical} \cup V_{start} \cup V_{end}$, as illustrated in Fig. 4a.

Denote $T_c \cup E_{cross}$ by G_c (see Fig. 4b.) Then, T and G_c make up a decomposition of G. Here, we notice that G_c is in

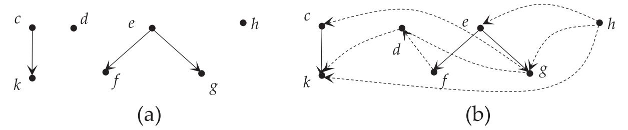

Fig. 4. Illustration for T_c and G_c..

general not a proper subgraph of G since in T_c some arcs each correspond to a path in T. We will, however, use the word 'decomposition' to refer to the transformation of G into T and G_c without causing confusion.

It can be seen that $V(G_c)$ (all the nodes in G_c) is much smaller than V.

For any two nodes 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 a 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 slightly change its crossing range as follows.

- Assume that U is a subset of V_{start} such that v is the *LCA* of all the nodes in it and satisfies condition (1) and (2) in Definition 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). 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}. In this way, we can quickly check whether there is any cross arc starting from a node in $T[v]$, which reaches out of $T[v]$.

Next, we denote by S_1 all the critical nodes in $T[v]$, and by S_2 all those start nodes of the cross arcs which appear in $T[v]$. Let $C(v) = S_1 \cup S_2$. We consider a maximal subset $C_s(v)$ of $C(v)$ such that each node in it does not have an ancestor in $C(v)$. It can be immediately seen that in $C_s(v)$ there is at most one node u such that its crossing range is not within $T[v]$. Otherwise, a new critical node in $T[v]$ can be recognized (see Definition 2), which is an ancestor of u 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.

i) A node $x \in C_s(v)$ is called an anchor node (of the first kind) of v if its crossing range is not within $T[v]$, denoted by v^*. If such a node does not exist, v^* is set to be the special symbol "-".

ii) A node y is called an anchor node (of the second kind) of v if it is the lowest ancestor of v (in T), which has a cross incoming arc. y is denoted by 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 $C_s(r)$ 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 arc. 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.

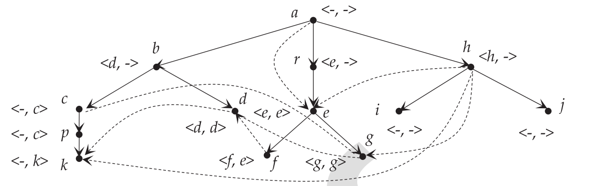

Fig. 5. Non-tree labels.

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 cross arcs. Then, any way for v to reach u must be through v^*.

Proof. According to Definition 3, v^* is the only node in $C_s(v)$ 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^* or v^* itself in T. So any node that is not a descendant of v but reachable from v via some cross arcs 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 cross arcs. Then, any way for u to reach v must be through v^{**}.

Proof. This can be seen from the fact that any node which reaches v via some cross arcs 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
- $z = v^{**}$.

In $<x, y, z>$, y and z together are referred to as non-tree labels.

Proposition 1. Let u and v be two nodes in G, labeled $([\boldsymbol{\alpha}_u, \boldsymbol{\beta}_u), y_u, z_u)$ and $([\boldsymbol{\alpha}_v, \boldsymbol{\beta}_v), y_v, z_v)$, respectively. Node u is reachable from v iff one of the following conditions holds:

i) $[\boldsymbol{\alpha}_u, \boldsymbol{\beta}_u)$ is subsumed by $[\boldsymbol{\alpha}_v, \boldsymbol{\beta}_v)$ (i.e., $\boldsymbol{\alpha}_u \in [\boldsymbol{\alpha}_v, \boldsymbol{\beta}_u)$), 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 arcs iff $[\boldsymbol{\alpha}_u, \beta_u)$ is subsumed by $[\alpha_v, \beta_v)$.
2) In terms of Lemmas 1 and 2, u is reachable from v via cross arcs iff $z_u = u^{**}$ and $y_v = v^*$ exist and u^{**} is reachable from v^* through a path in G_c.

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

In this figure, we can see that the non-tree label of node r is <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 [5] to do this task.

Obviously, the smaller G_c is, the better. But we know that the larger the number of forward edges is, the smaller G_c. Thus, we want to be able to find a spanning tree such that the number of forward edges is increased (and then the number of cross edges is decreased), which will eventually lead to a smaller G_c. In Section 4.2, we will discuss this issue in great detail.

Lastly, we notice that G_c itself can be very large. In this case, we need to decompose G_c again, leading to an elegant recursive graph decomposition, as discussed in the next subsection.

4.3 Recursive Graph Deduction

Let G_0 be a DAG. Denote by T_0 a spanning tree of G_0. Denote by E^0_{cross} the set of all the cross arcs with respect to T_0. Then, as discussed before, T_0 and $G^0_c = T^0_c \cup E^0_{cross}$ make up a decomposition of G_0, where T^0_c is the critical tree of G_0.

Denote G^0_c as G_1. Recursively decomposing G_1, we will figure out a sequence of tree structures:

$$T_0, T_1, \ldots, T_{k-1}, \quad (k \geq 1)$$

with each T_i being a spanning tree of the subgraph

$$G_i = G^{i-1}_c = T^{i-1}_c \cup E^{i-1}_{cross}, \tag{1}$$

where G^{i-1}_c is the summary graph of G_{i-1}, T^{i-1}_c is the critical tree of G_{i-1}, and E^{i-1}_{cross} is the set of all the cross arcs with respect to T_{i-1}.

In this way, we are able to associate each node v in G_0 with two sequences: an interval sequence ω_v and an anchor node sequence ϖ_v to check reachability:

1) ω_v: $[\alpha^v_0, \beta^v_0), \ldots, [\alpha^v_j, \beta^v_j), (j \leq k - 1)$

where each $[\alpha^v_i, \beta^v_i)$ is an interval generated by labeling T_i;

2) ϖ_v: $(v^*_0, v^{**}_0), \ldots, (v^*_j, v^{**}_j)$,

where each v^*_i is the anchor node (of the first kind) of v in T_i ($0 \leq i \leq j$) while v^{**}_i is the anchor node (of the second kind) of v in T_i, as discussed in Section 3.2.

The following example helps for illustration.

Example 2. 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^0_{cross} is all the cross arcs as shown by the dashed arrows (except the unique forward arc $a \rightarrow e$) in the same figure, and T^0_c is a forest as shown in Fig. 4a. Then, $G_1 = T^0_c \cup E^0_{cross}$ is a graph as shown in Fig. 4b.

One of its spanning tree T_1 is shown by the solid arrows in Fig. 6a. With respect to this spanning tree, $h \rightarrow g$ and $h \rightarrow k$ are two forward arcs and can be removed. So E^1_{cross} is a subgraph as shown in Fig. 6b, containing only two disconnected arcs. Their respective start nodes are g and c.

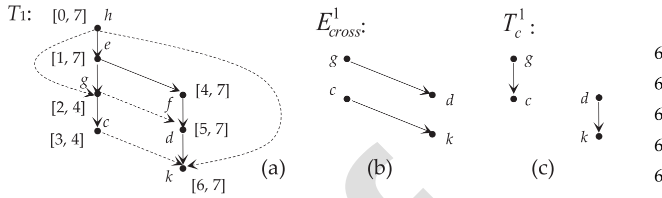

Fig. 6. Illustration for recursive graph decomposition.

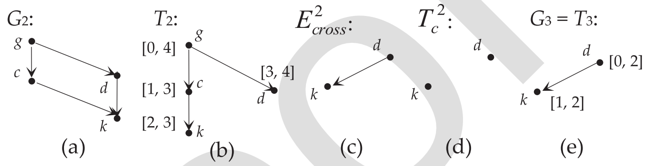

Fig. 7. Illustration for recursive graph decomposition.

(a)

	T_0:	T_1:	T_2:	T_3:
a	[0, 13)			
b	[1, 5)			
c	[2, 5)	[3, 4)	[1, 3)	
d	[5, 6)	[5, 7)	[3, 4)	[0, 2)
e	[7, 10)	[1, 7)		
f	[8, 9)	[4, 7)		
g	[9, 10)	[2, 4)	[0, 4)	
h	[10, 13)	[0, 7)		
i	[11, 12)			
j	[12, 13)			
k	[4, 5)	[6, 7)	[2, 4)	[1, 2)
p	[3, 5)			
r	[6, 10)			

(b)

	G_0:	G_1:	G_2:
a	<-, ->		
b	<d, ->		
c	<-, c>	<c, ->	
d	<d, d>	<-, d>	<d, ->
e	<e, e>	<g, ->	
f	<f, e>		
g	<g, g>	<g, ->	<d, ->
h	<h, ->	<g, ->	
i	<-, ->		
j	<-, ->		
k	<-, k>	<-, k>	<-, k>
p	<-, c>		
r	<e, ->		

Fig. 8. Non-tree labels and sequences associated with nodes.

Accordingly, T^1_c is also a subgraph containing two disconnected arcs, as shown in Fig. 6c.

G_2 will be constructed in the same way as G_1. That is, G_2 is equal to $T^1_c \cup E^1_{cross}$, as shown in Fig. 7a.

A spanning tree T_2 of G_2 is shown in Fig. 7b. With respect to T_2, E^2_{cross} is a subgraph containing only one arc, and T^2_c contains only two single nodes, as shown in Figs. 7c and 7d, respectively. So, we have $G_3 = T^2_c \cup E^2_{cross} = E^2_{cross}$, as shown in Fig. 7e. We notice that G_3 is a tree. So, T_3 is the same as G_3.

By creating intervals for the nodes in T_0, T_1, T_2 and T_3 (see Fig. 2, Fig. 6a, Figs. 7b and 7e, respectively), we will generate an interval sequence for each node as shown in Fig. 8a.

Fig. 8b shows all anchor node sequences, which are created by the non-tree labeling of the nodes in G_0 (see Fig. 2), G_1 (see Fig. 9a), and G_2 (see Fig. 9b). G_3 is a tree itself and no non-tree labels are established.

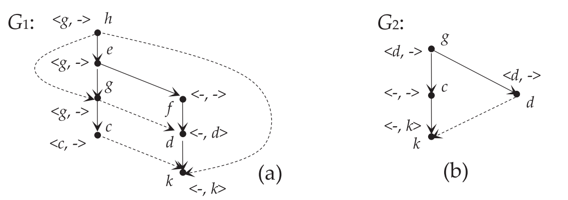

Fig. 9. Non-tree labels and sequences associated with nodes.

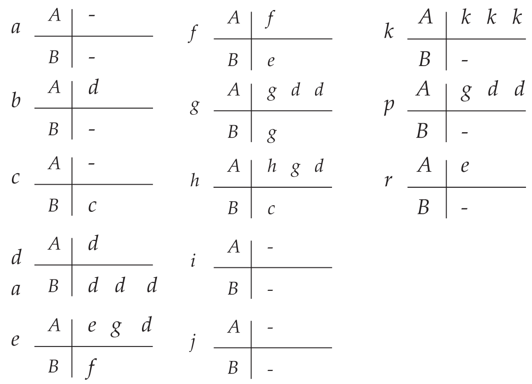

Fig. 10. *A*- and *B*-sequences.

Based on the interval sequences ω_v and anchor node sequences ϖ_v, we can generate A_v and B_v for node v:

$A_v = x_0, x_1, \ldots, x_l$,

$B_v = y_0, y_1, \ldots, y_l$, where $x_0 = y_0 = v$, $x_1 = (x_0)_1^*$, …, $x_l = (\ldots (x_0\ )_1^* \ldots)_l^*$, and $y_0 = y_1 = (y_0\ )_1^{**}$, …, $y_l = (\ldots (y_0\ )_1^{**} \ldots)_l^{**}$.

In Fig. 10, we show the *A* and *B* sequences for all the nodes of the graph shown in Fig. 2.

Now, we give the following algorithm to evaluate reachability queries.

Algorithm 1. *queryEval*(u, v) (*to check $u \Rightarrow v$?*)

---

begin
1. **if** $\kappa_v \not\leq \kappa_u$ or $\mu_v \not\leq \mu_v$ **then** return *false*;
2. i: = 0; x: = u; y: = v;
3. **while** $x \neq \phi$ and $y \neq \phi$ **do** {
4. use $[\alpha_x^i, \beta_x^i)$ and $[\alpha_y^i, \beta_y^i)$ to check whether y is reachable from x in T_i. If it is the case, return *true*.
5. $x: = x^*$. $y: = y^{**}$. $i: = i + 1$;
6. }
7. Return *false*;

end

In the above algorithm, in line 1 κ_u and μ_u are two topological numbers associated with u while κ_v and μ_v are two topological numbers for v, generated by using the algorithm discussed in [31]. If $\kappa_v \not\leq \kappa_u$ or $\mu_v \not\leq \mu_u$, v is definitely not reachable from u and the algorithm returns *false*. Otherwise, we will go into a *while*-loop (see lines 3 - 6), in which two node sequences A_u and B_v are searched.

For each pair of x_i (in A_u) and y_i (in B_v), we will check whether y_i is reachable from x_i within T_i by using their intervals, which obviously requires only O(1) time.

Example 3 Continued with Example 2. To test whether $h \Rightarrow p$ in $G = G_0$ shown in Fig. 2, we will first check their topological numbers (see line 1). Since p is reachable from h, we must have $\kappa_p \leq \kappa_h$ and $\mu_p \leq \mu_h$. Thus, the *while*-loop will be executed, by which we will first check whether $h \Rightarrow p$ in T_0 (the spanning tree shown by the solid arrows in G_0.) Since $h \not\Rightarrow p$ in T_0, we will check whether $h^* = h \Rightarrow p^{**} = c$ in T_1 (see Fig. 10). It is the case and the query returns *true*.

By this query evaluation, the *A*-sequence associated with h and the *B*-sequence with p are demonstrated in Fig. 11a.

To test whether $h \Rightarrow k$, we will also scan two sequences as shown in Fig. 11b. Along the two sequences, the following tests will be carried out: $h \not\Rightarrow k$ in T_0, $(h\ )_0^* = h \not\Rightarrow (k)_0^{**} = k$ in T_1, $(h)_1^* = g \not\Rightarrow (k)_1^{**} = k$ in T_2, but $(g)_2^{**} = d \Rightarrow (k)_2^{**} = k$ in T_3.

The query returns *true*.

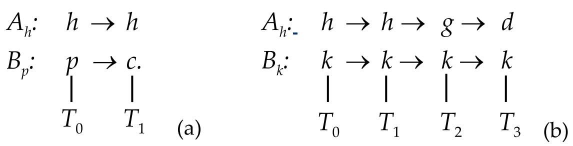

Fig. 11. *A*-sequences and *B*-sequences.

5. Technical Details

In the previous section, the main working process is described, but with some technique descriptions ignored. In this section, we get back to them. First, we discuss how to recognize critical nodes efficiently in Section 4.1. Then, how to find a better spanning tree with more forward arcs in Section 4.2. In Section 4.3, we give a probabilistic analysis of the algorithm discussed in Section 4.2.

5.1 Recognizing Critical Nodes

In order to recognize critical nodes efficiently, we will search T bottom-up to 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 associated 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 below.

i) Let v be the node currently inserted into T'.
ii) If v is the first node inserted into T', nothing will be done.
iii) If v is not the first node inserted into T', we do the following:

Let v' be the node inserted just before v. If v' is not a child (descendant) of v, create a *left-sibling* link from v to v', denoted as *left-sibling*$(v) = v'$. If v' is a child (descendant) of v, we will first create a *parent* link from v' to v, 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*$(u) \leftarrow v$. Finally, set *left-sibling*$(v) \leftarrow v''$.

Fig. 12 is a pictorial illustration of this process.

In Fig. 12a, 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. 12b 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.

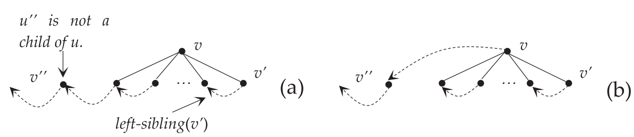

Fig. 12. Illustration for the construction of *T′*.

This is in essence a process to recognize *LCA*s, but more general than the algorithm discussed in [39] since we need to recognize the *LCA*s of any subsets of V_{start}, which contains two or more than two nodes not related by the ancestor/descendant relationship.

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 2. *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*$(w) \leftarrow u$. Also, set *left-sibling*$(u) \leftarrow u''$. (See Fig. 11b 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 their 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 cross range $\{a_v, b_v\}$ is computed. (See 4.a and 4.b in the algorithm.) For a node not belonging to V_{start}, we will check whether it satisfies 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 only *out-degree*(v) arcs are visited. Thus, we have

$$\sum_{v \in V} out\text{-}degree\,(v) = e.$$

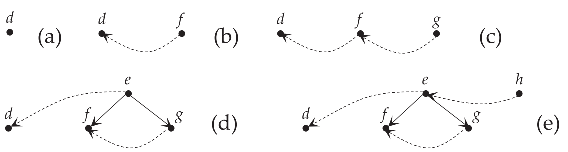

Fig. 13. A sample trace.

Example 4. Consider the spanning tree T shown in Fig. 2 again. Applying the above algorithm to T, we will generate a series of data structures as shown in Fig. 13.

First of all, 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. 13a.) 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. 13b.) In the fourth step, node g is encountered and a second left-sibling link is generated (see Fig. 13c.) 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. 13d.) Here, special attention should be paid to the replacement of *left-sibling*$(f) = d$ with *left-sibling*$(e) = d$, which enables 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. 13e demonstrates the last step of the whole process. Especially, the tree shown in Fig. 13e is T', which contains all the critical nodes and the nodes from V_{start}.

From T', T_c and G_c can be easily constructed as shown in Fig. 4.

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 arc $u \rightarrow v$ 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 endpoints).

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 child 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:

$$|V_{critical}| < |V| - |V_{start} \cup V_{end}|. \tag{2}$$

Since G is a DAG, it has at least one node whose indegree 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 nodes in G is strictly larger than the number of nodes in G_c.

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

The proposition implies that the length of the A- and B-sequences of any node must be $\leq n$.

5.2 Find Better Spanning Trees

It is obvious that the graph decomposition is definitely useful for sparse graphs. However, in practice, it can also be very useful for some dense graphs, but somehow related to what a spanning tree is found. To see this, let us have a look at a 'complete graph' shown in Fig. 14a. For this graph, we can find a spanning tree as shown by the solid arrows in Fig. 14b. It is in fact a single path: $a \rightarrow e \rightarrow d \rightarrow c \rightarrow b$ while all the other arcs are just forward arcs and therefore can be simply removed (leading to an empty G_c.) Obviously, the reachability over this kind of graphs can be done in just one single checking by using the intervals created over such a spanning tree (which is simply a path.)

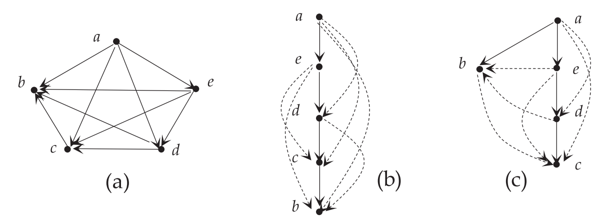

Fig. 14. Illustration for spanning trees.

Searching the graph in a different way, we may find a different spanning tree as shown by the solid arrows in Fig. 14c, for which we have three forward arcs: $a \rightarrow c$, $a \rightarrow d$, and $e \rightarrow c$, as well as three cross arcs: $e \rightarrow b$, $d \rightarrow b$, and $c \rightarrow b$. So the corresponding G_c cannot be *empty* and for evaluating reachability queries some more checks have to be performed.

Clearly, what we want is to find a spanning tree so that G_c is minimized. But, how to find such a spanning tree?

In the following, we address this issue.

Let G be a DAG. Let $\Im(G)$ be the family including all the spanning trees of G. For $T \in \Im(G)$, denote by f_T and c_T the number of forward arcs and cross arcs, respectively.

Intuitively, the larger f_T is, the smaller c_T and then the size of G_c. So our optimization problem is to find a T such that f_T with respect to it is maximum. Unfortunately, there are exponentially many spanning trees for a given DAG. Thus, it is unlikely to find an optimal one in polynomial time. In fact, the problem itself is *NP*-complete. But we can devise a linear-time algorithm to find a spanning tree of G with fewer cross arcs than a traditional depth-first search.

5.2.1 NP-Completeness

We first prove the *NP*-completeness of the problem.

Let P be a path in T. Let u, v be two nodes on P. We call the forward arc from u to v an attached arc of P. Obviously, to maximize f_T, we need to maximize the number of attached arcs of each path in T. However, even the problem to find a spanning tree, which contains a path with the maximal number of attached arcs, is difficult. We will show that even this easier problem is *NP*-complete by itself. For this purpose, we define the following decision problem:

Input: A DAG G and a positive integer $k \leq n$.

Question: Is there a spanning tree T such that it contains a path P of length q with the number of attached arcs of P equal to $(q - 1)(q - 2)/2$.

We refer to this problem as a *maximum attachment* problem.

Proposition 4. The problem to find a maximum attachment is *NP*-complete.

Proof. We can design an algorithm to generate all spanning trees (forests) T of G and check each T to see whether it has a path with the maximum attachment. Since the number of such T's is bounded by $O((n - 1)!)$, the problem is in *NP*.

Next, we reduce the basic *NP*-complete problem *satisfiability* [9] to the maximum attachment problem. To this end, we consider an instance of *satisfiability* with a collection of clauses $C = \{c_1, \ldots, c_x\}$. Each c_i is of the form $c_{i1} \vee c_{i2} \vee \ldots \vee c_{ix_i}$, where each c_{ij} ($1 \leq j \leq x_i$) is a literal. For C, we construct a DAG G as follows.

1. Generate an undirected graph G', whose nodes are pairs of integers $[i, j]$, for $1 \leq i \leq x$ and $1 \leq j \leq x_i$. A

node $[i, j]$ is connected to another node $[k, l]$ if both of the following hold:

- $i \neq k$, and
- $c_{ij} \neq \neg c_{kl}$.

2. Explore G' in depth-first manner to change it to G as below:
   - If an edge (u, v) in G' is explored from u to v, create an arc $u \rightarrow v$ in G.
   - In G, reverse the direction of any back arc. (Then, the resulting G must be a DAG.)

It is easy to see that G can be constructed in polynomial time. Furthermore, if there exists a satisfying assignment of Boolean values for C there must be a spanning tree of G containing a path P of length q such that the number of the attached arcs of P equal to $(q-1)(q-2)/2$. It is because if C is satisfiable, there must be a clique of size q in G', which is a subset of nodes S such that if $u, v \in S$ then (u, v) in G'. Exploring the clique in *DFS* and then reverse any back arc, we will get a path of length q with the number of the attached arcs equal to $(q-1)(q-2)/2$.

Now we assume that T is a spanning tree of G, which contains a path P of length q, and the number of the attached arcs is equal to $(q-1)(q-2)/2$. Then, we assign a value to the variable in each literal $\boldsymbol{\chi}$ corresponding to a node on P such that $\boldsymbol{\chi}$ is *true* while a value to the variable in any other literal $\boldsymbol{\chi}'$ (not corresponding to any node on P) such that $\boldsymbol{\chi}'$ is *false*. Then, C evaluates to *true* under such an assignment since each node represents a literal appearing in a clause different from any clause represented by others, and for each two literals represented by two nodes on the path, their values are definitely not negative to each other.

5.2.2 A Top-Down Algorithm

From the above discussion, we can see that it is not possible for us to find an 'optimal' spanning tree for a given G in polynomial time. But we still want to find a relatively *good* solution to the problem in time linear in the number of arcs in G. In the following, we present an algorithm to explore G top-down, which is able to find a spanning tree T with more forward arcs than a traditional *DFS* (depth-first search). The main idea behind this algorithm is to recognize a kind of "triangles" as illustrated in Fig. 15a, during a *DFS* search.

In Fig. 15a, assume that node v is the current node along a path from u to v, and w is one of v's children, but has been visited before (along an arc from u to w). We can remove the tree arc $u \rightarrow w$ and make $v \rightarrow w$ a tree arc. Then, $u \rightarrow w$ is changed to a forward arc as illustrated in Fig. 15b.

In order to find such kind of transformations, we maintain a Boolean array H such that $H[v] = 1$ indicates that node v is on the current path during the depth-first search.

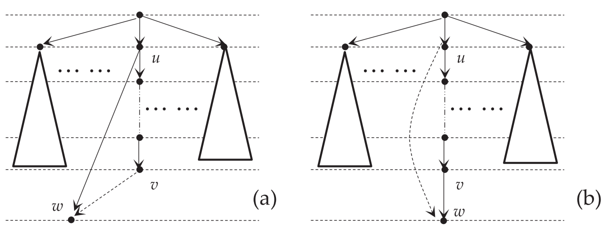

Fig. 15. Illustration for "triangles" encountered during a DFS.

Otherwise, $H[v] = 0$. By the current path, we mean the path from the root to the currently encountered node. Let v be the currently encountered node. Let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = v$ be the current path. Then, when we check a child w, we will first examine whether w has already been accessed some time earlier. If it the case, we will call a function: *triangle*(v, u, w), where u is the parent of w in T. If $H[u] = 1$, *triangle*(v, u, w), returns *true*; otherwise, *false*.

In the following algorithm, two data structures are also used:

S – a stack to control the depth-first search;

$C_T(v)$ – a list containing all the children of v in T.

In addition, for simplicity, we assume that G is a rooted graph.

Algorithm 3. *mDFS*(G)

```
begin
 1.  Each entry of H is set 0; add root to T;
 2.  push(root, S); mark root; H[root]: = 1;
 3.  while (S ≠ φ) do {
 4.    v: = pop(S);
 5.    for each child w do {
 6.      if w is marked then {
 7.        Let u be the parent of w in T;
 8.        if triangle(v, u, w) = true then {
 9.          remove w from C_T(u);
10           add w to C_T(v);
11.        }
12.      }
13.      else {add w to C_T(v);
14.          push(w, S); mark w; B[w]: = 1;}
15.    }
16.    B[v]: = 0;
17.  }
end
```

The above algorithm works almost in the same way as *DFS*. The only difference consists in the use of array H. Besides, each accessed node is marked. At an iteration of the *while*-loop (lines 3 - 16), the current node v is popped out from stack S. Then, each of it's children w will be accessed in turn. If w has already been visited before, *triangle*(v, u, w) will be executed (see line 8), where u is the parent of w in T, part of the spanning tree constructed up to now. If *triangle* (v, u, w) returns *true*, we must have $H[u] = 1$ and we will remove w from $C_T(u)$ and add it to $C_T(v)$. Note that w is not pushed into stack S and thus will not be accessed once again. If w is not marked, it will be added to $C_T(v)$, pushed into S, and marked (see lines 13 - 14). Finally, we will set H $[w] = 1$ since w becomes the current node (see line 14). After all the nodes in $T[v]$ are accessed, a backtracking happens and we will reset $H[v]$ to 0 since it does not belong to the current path anymore (see line 16).

The time complexity of the algorithm is obviously $O(e)$.

Example 5. Consider the graph shown in Fig. 13a again. If we use the traditional depth-first search to explore the graph, we may create a spanning tree as shown by the solid arrows in Fig. 13c.

But if we use *mDFS* to explore G, a series of triangle transformations will be performed as illustrated in Fig. 16,

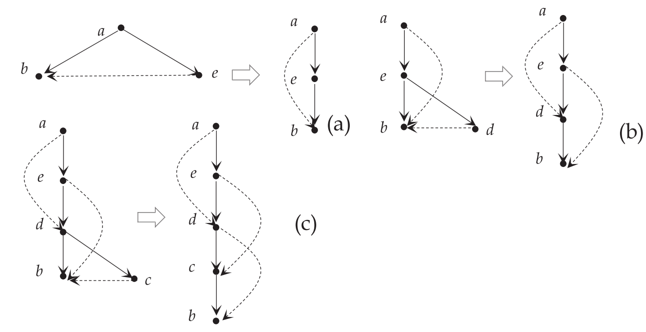

Fig. 16. Illustration for triangle transformation.

leading to the spanning tree shown by the solid arrows in Fig. 14b.

In Fig. 16a, we show part of a spanning tree containing two arcs: $a \rightarrow b$ and $a \rightarrow e$. When we meet b again (along arc $e \rightarrow b$), a transformation will be carried out as shown in Fig. 16a. In the subsequent steps, we may meet another two triangles of the forms as shown in Fig. 16b and 16c, respectively, which will also be transformed in turn.

Proposition 5. Let T and T' be spanning trees created by exploring G using *DFS* and *mDFS*, respectively. Then, $f_T \leq f_{T'}$.

Proof. Let $\Delta_{u,w,v}$ be a triangle met during the *DFS*, in which $u \rightarrow w$ is a tree arc, $v \rightarrow w$ is a cross arc, and there is a tree path from u to v. As discussed above, *mDFS* will transform this triangle to make $v \rightarrow w$ a tree arc and $u \rightarrow w$ a forward arc. By this transformation, any forward arc from u or an ancestor of u to w or a descendant of w in $T[w]$ is still a forward arc with respect to T'. This shows that $f_T \leq f_{T'}$.

The time spent for doing a transformation is bounded by a constant. Thus, the time complexity of *mDFS*() is still bounded by $O(n + e)$.

5.3 About the Value of *k*

By Proposition 3, we know that G_c is strictly smaller than G and thus $k \leq n$.

In the following, to show the effectiveness of Algorithm *mDFS*(), we will make a probabilistic analysis of G_c (produced by using *mDFS*()) to show that the expected number of arcs in G_c is bounded by $O(n^{1.5})$.

Let u be a node in G. With respect to u, for any node v, we define a random variable $\boldsymbol{\xi}_v$ as below:

$$\xi v = \begin{cases} 1 & \text{if } u \rightarrow v \in G_c \\ 0 & \text{Otherwise.} \end{cases} \tag{3}$$

Then, the expected out-degree of node u in G_c is $E(\sum_v \xi_v) = \sum_v E(\xi_v)$where $E(\boldsymbol{\mu}_v)$ represents the mathematical expectation of $\boldsymbol{\mu}_v$.

We can estimate $\sum_v E(\xi_v)$ as follows.

If $\boldsymbol{\xi}_v = 1$, then $u \rightarrow v \in G_c$ and for any $w \in G$, either $u \rightarrow w \notin G_c$, or $w \rightarrow v \notin G_c$. Otherwise, we would have a triangle Δ_{uwv} and *mDFS*() would change $u \rightarrow v$ to a forward arc (and then remove it.) For the same reason, if $\boldsymbol{\xi}_v = 1$, we will definitely not have any path of length > 2 from u to v.

Denote by p the probability that an arc appearing in G_c.

Denote by $\boldsymbol{\gamma} = t(u, v)$ the number of nodes between u and v in a topological order of G_c. Then, we can see that the probability that G_c has an arc $u \rightarrow v$, but does not contain any triangle Δ_{uwv} is

$$p(1-p^2)^{\gamma}(1-p^3)(1-p^3)^{\binom{\gamma}{2}} \cdots (1-p^{\gamma+1})^{\binom{\gamma}{\gamma}}. \tag{4}$$

Thus, we have

$$E(\xi_v) \leq p(1-p^2)^{\gamma}. \tag{5}$$

Therefore, we can show that the expected out-degree of a node in G_c is $\leq (1 - (1 - p^2)^{n+1})/p \leq \sqrt{n} + 1/\sqrt{n}$. (See [33] for a detailed discussion.) Then, the expected number of arcs in G_c is bounded by

$$n \times \sqrt{n} = n^{1.5} \tag{6}$$

We also notice that *mFDs*() reduces not only the number of cross arcs, but also the number of nodes in G_c since in G_c any nodes $\notin V_{critical} \cup V_{start} \cup V_{end}$ will be discarded. To see this, let us have a look at Fig. 15a once again. After the tree arc $u \rightarrow w$ is transformed to a forward arc (see Fig. 15b), we can not only discard it, but also node w if $w \notin V_{critical} \cup V_{start} \cup V_{end}$.

Assume that the average number of the removed nodes by each graph deduction is $\boldsymbol{\delta}$ and we stop the graph decomposition process whenever we meet a deducted graph with less than n arcs since in this case we may have a tree (with high probability). We need to solve the following inequality to estimate the value of k:

$$(n - k\delta)^{1.5} \leq n. \tag{7}$$

So, we have

$$k \geq (n - n^{2/3})/\delta. \tag{8}$$

To estimate $\boldsymbol{\delta}$, we further assume that $|E| = O(n^2)$. Then, the expected number of removed arcs is in the order of

$$O(n^2 - n^{1.5}) = O((n - \sqrt{n})n).$$

From this, we infer that the number of eliminated nodes is in the order of $O(n - \sqrt{n})$ since the out-degree of any node in G is bounded by $O(n)$.

However, the number of arcs of a graph is normally smaller than n^2, and for $n \geq 4$, we always have $n - \sqrt{n} \geq \sqrt{n}$. So, we set $\boldsymbol{\delta} = \sqrt{n}$. Then, from (8), we get

$$k \geq (n - n^{2/3})/\sqrt{n} = \sqrt{n} - \sqrt[6]{n}. \tag{9}$$

This shows that the expected value of k is around $\sqrt{n}$.

Finally, we point out that the above analysis is suitable only for very dense graphs. In practice, however, the number of arcs of a graph is normally $\ll O(n^2)$; and k should be much smaller than this expected value. In fact, in our experiments, for all the tested graphs, k is much smaller than $\sqrt{n}$.

6. EXPERIMENTS

In this section, we report the test results.

We conducted our experiments on a Linux machine with 128GB of memory and a 2.9GHz 64-core processor. The programs are compiled using Microsoft virtual C++ compiler version 6.0, running standalone.

6.1 Tested Methods

In the experiments, we have tested altogether 12 methods:

- *DFS* (depth-first search)
- *Tree encoding* by Agrawal et al. (*TE* for short) [1],
- *Chain decomposition* by Chen et al. (*CD* for short) [5],
- *Dual-II* by Wang et al. [24],
- *Path-tree* by Jin et al. (*PTree* for short) []11,
- *PWAH* by Schaik et al. [28],
- *GRAIL* by Yildirim et al. [25],
- *SCARAB* by Jin et al. [29],
- *HL* by Jin and Wang [34]
- *FELINE* by Velosol et al. [31],
- *BFL* by Su et al. [35], and
- Recursive DAG decomposition (discussed in this paper, *RDD* for short).

All their theoretical computational complexities are listed in Table 1 (in Section 3).

Among all these methods, the source code of *CD*, *PTree*, *SCARAB*, *PWAH*, *GRAIL*, and *FELINE* are either downloaded from authors' websites or provided by the authors directly while all the other methods are implemented by ourselves.

All the tests are organized into two groups on real data and synthetical data, respectively.

6.2 Query Generation

First of all, we distinguish between *positive* and *negative* queries. A positive query evaluates to *true* while a negative query evaluates to *false*. For partial index based methods, however, we need to further differentiate two kinds of negative queries. By the first kind of negative queries, the answers can be determined by checking indexes. By the second kind of negative queries, the answers are determined only after the whole G is searched. Besides, for the creation of positive queries, the trivial cases that the checked nodes v and u are not far away from each other should be carefully avoided. To this end, we use a function $dis(u, v)$ to compute the 'distance' between u and v, defined to be the number of nodes visited by the *BFS* (breadth-first search) from u to v.

For a fair comparison, we have designed a procedure to create queries for each dataset G, which takes altogether six parameters:

G – dataset;
q – number of queries to be generated;
r_1 – rate of positive queries;
r_2 – rate of negative queries of the first kind;
r_3 – rate of negative queries of the second kind;
strategy – name of the method to be tested.

In this algorithm, we use three sets Q_t, $Q_{f\text{-}1}$, and $Q_{f\text{-}2}$ to store generated positive queries, negative queries of the first kind, and the second kind, respectively; and use r to accommodate the threshold over $dis(u, v)$ for any positive queries.

In addition, a hash function $h(u, v)$ is used to create a hash value for each produced pair of nodes (u, v) (representing a query: $u \Rightarrow v$?) and store it in a hash table to avoid generating repeated queries. However, for simplicity, this technique detail is not presented in the algorithm.

For the tested 100000 queries, we set $r_1 = 40\%$, $r_2 = 30\%$, and $r_3 = 30\%$.

Algorithm 4. *GenerationQ*(G, q, r_1, r_2, r_3, *strategy*)

begin
1. Q_t: $= \boldsymbol{\phi}$; $Q_{f\text{-}1}$: $= \boldsymbol{\phi}$; $Q_{f\text{-}2}$: $= \boldsymbol{\phi}$; r: $= 20\% \times |G|$;
2. **while** $|Q_t| + |Q_{f\text{-}1}| + |Q_{f\text{-}2}| \leq q$ **do** {
3. choose two random nodes u and v from G;
4. run *BFS* to find whether $u \Rightarrow v$;
5. **if** $u \Rightarrow v$ **then** {**if** $dis(u, v) > r$ and $|Q_t| < q \times r_1$;
6 **then** Q_t: $= Q_t \cup \{(u, v);\}$
6. **else** {**if** *strategy* is full index-based and $|Q_{f\text{-}1}| < q \times r_2$;
7. **then** $Q_{f\text{-}1}$: $= Q_{f\text{-}1} \cup \{(u, v)\}$
8. **else** {if $u \not\Rightarrow v$ can be checked by using the index and $|Q_{f\text{-}1}| < q \times r_2$;
9. **then** $Q_{f\text{-}1}$: $= Q_{f\text{-}1} \cup \{(u, v)\}$;}
10. **else if** $Q_{f\text{-}1}| < q \times r_2$
11. **then** $Q_{f\text{-}2}$: $= Q_{f\text{-}2} \cup \{(u, v)\}$;
12. }

end

TABLE 3
Small Real Datasets

dataset	$\lvert V\rvert$	$\lvert E\rvert$	Avg. deg.	$\lvert V^*\rvert$	$\lvert E^*\rvert$
AgroCyc	13969	17694	1.27	12684	13408
Amaze	11877	28700	2.41	3710	3734
Anthra	13736	17307	1.25	12499	13104
Ecoo	13800	17308	1.25	12620	13350
arXiv	6000	66707	11.12	6000	66707
Human	40051	43879	1.09	38811	39576
Kegg	14271	35170	2.46	3617	3908
Mtbrv	10697	13922	1.31	9602	10245
Nasa	5704	7939	1.39	5605	7735
go	6973	13361	1.92	6973	13361
VchoCyc	10694	14, 207	1.32	9491	10143
PubMed	9000	40028	4.48	9000	40028
Yago	9000	42392	4.71	9000	40028
Xmark	6483	7954	1.23	6080	7072

TABLE 4
Large Real Datasets

dataset	$\lvert V\rvert$	$\lvert E\rvert$	Avg. deg.	$\lvert V^*\rvert$	$\lvert E^*\rvert$
Successor	1095062	1145304	1.06	542235	564890
Pagelinks	137830	2949220	21.39	47242	48435
Interproc	3532298	4716476	1.33	353748	431599
Uniprot22m	1595444	1595442	0.99	1595444	1595442
Uniprot100m	16087295	16087293	0.99	16087295	16087293
Uniprot150m	25037600	25037598	1.00	25037600	25037598
cit-Patents	3774768	16518947	4.37	3774768	16518947
citeseerx	6540399	15011, 259	2.29	6540399	16518,94
go_uniprot	6967956	34770235	4.99	6967956	34770235

TABLE 5
Query Time Over Small Graphs

dataset	DFS (ms)	TE (ms)	CD (ms)	DUAL-II (ms)	PTree (ms)	PWAH (ms)	GRAIL (ms)	SCARAB (ms)	HL (ms)	FELINE (ms)	BFL (ms)	RDD (μs)
AgroCyc	23.69	3.31	2.03	2.40	1.35	1.57	19.38	2.64	4.3	25.78	38.11	31
Amaze	41,53	5.0	3.43	3.67	2.01	3.89	25.76	4.05	2.9	30.45	43.67	6
Anthra	21.4	2.67	1.07	1.78	1.36	1.39	17.16	2.57	3.9	19.08	21.79	6
ecoo	27.75	2.82	2.85	2.7	1.3	1.46	25.63	2.62	4.4	26.03	34.12	35
arXiv	416,23	6.09	4.23	4.32	4.36	30.46	165.82	136.61	101.8	206.32	178.06	15
human	399.0	29.54	18.74	30.2	14.91	67.11	252.77	201.45	152.5	245.96	267.56	45
kegg	133,52	3.27	1.65	3.01	1.96	4.97	100.14	6.99	3.4	111.09	98.23	37
mtbrv	30.31	3.33	1.74	2.03	1.3	1.5	21.94	2.51	5.1	26.71	26.34	23
nasa	20.94	2.0	1.45	2.09	1.66	4.18	17.94	3.22	4.1	19.01	21.89	9
go	19.0	13.02	12.35	43.0	2.44	5.69	14.4	3.78	3.21	16.01	26.45	15
vchocyc	30.83	2.11	1.31	1.65	1.34	6.69	20.67	2.49	3.8	24.33	34.45	26
pubmed	234.15	8.78	6.70	10.05	3.34	37.11	136.35	8.31	2.9	167.21	156.41	22
yago	173.75	5.93	3.32	6.78	2.88	6.69	73.75	4.24	3.21	83.24	78.37	24
xmark	16.4	2.01	1.56	1.98	1.77	13.34	12.08	8.85	6.5	14.35	20.67	17

TABLE 6
Query Time Over Large Graphs

dataset	DFS (ms)	TE (ms)	CD (ms)	DUAL-II (ms)	PTree (ms)	PWAH (ms)	GRAIL (ms)	SCARAB (ms)	HL (ms)	FELINE (ms)	BFL (ms)	RDD (μs)
Successor	243.55	31.95	26.78	-	-	35.86	89.51	20.54	22.87	15.72	57.21	102
Pagelinks	387.51	36.77	30.51	-	23.88	43.0	72.0	18.9	10.07	70.25	43.78	157
Interproc	782.01	121.03	51.23	-	-	167.93	136.71	98.67	46.73	100.93	98.25	301
Uniprot22m	261.24	23.12	14.78	-	9.1	21.9	40.5	25.6	5.9	35.7	100.56	310
Uniprot100m	390.99	25.66	20.45	-	-	28.3	53.0	33.6	7.5	46.03	57.32	463
Uniprot150m	437.97	27.91	23.47	-	-	29.1	56.6	33.0	10.7	49.32	49.07	452
cit-Patents	1588.81	112.34	67.88	-	-	176.3	501.5	517.2	-	479.27	543.98	655
citeseerx	7412.43	40.97	25.65	-	-	39.8	2585.6	719.1	23.7	2341.58	210.67	2001
go_uniprot	252.83	32.15	25.32	-	-	52.5	47.6	29.8	12.0	41.65	78.45	2516

6.3 Tests on Real Data

In Tables 3 and 4, we show a collection of small and large real data, respectively, which have been used as the standard benchmarks in the recent studies on reachability indexes [10], [17], [18], [24], [25], [28], [29], [31], [34]. In the tables, for each graph, besides the numbers of nodes and arcs in the original graphs, the numbers of nodes and arcs after each SCC is coalesced to a single node, represented respectively by V^* and E^*, are also given.

In Tables 5 and 6, the query time of our method is given in microseconds while for all the other methods the query time is given in millisecond due to the fact that our method is several orders of magnitude better than the others.

First, when compared with the other full index based methods, the size of our indexes is quite small, i.e., the *A*- and *B*-sequences associated with a node by our method is very short. To see this, let us have a look at Tables 6 and 7, from which we can see that after the first three steps of recursive graph decomposition almost all graphs, except arXiv, PubMet, and Yago, are quickly shrunk to a very small graph. Even for arXiv, PubMet, and Yago, the size of the graphs are also significantly reduced.

For a residue graph (i.e., the remaining graph after several steps of graph deduction and decomposition), we can establish a matrix M as discussed in [5], [6] if it becomes small. Therefore, the time complexity for evaluating a query should be O(k) plus the time to access an entry in M, where k is the maximum length of *A*- and *B*-sequences.

Since k is often quite small, our method works better than the others.

In addition, although the theoretical query time of the method *CD* (chain decomposition [5]) is O(1) (better than all the other tested methods), its index size is very large and normally cannot be completely kept in main memory, which leads to longer query time than expected. It is comparable to the others, but with more indexing time.

Secondly, when compared with all the partial index based methods, our method has the following two advantages:

1. For a negative query of the first kind, our method has almost the same performance as a partial index method, by using the two topological numbers associated with each node, working as a filter to answer negatively this part of queries [31].
2. For a positive query or a negative query of the second kind, the running time of our method is always bounded by O(k), no search of G at all. But by each partial index based method, such as *GRAIL*, *HL*, *FELINE*, and *BFL*, the whole graph or a large part of the graph has to be searched since for such queries their indexes are almost useless.

In Figs. 17 and 18, we show the indexing time and index sizes for all the tested small graphs. From these, we

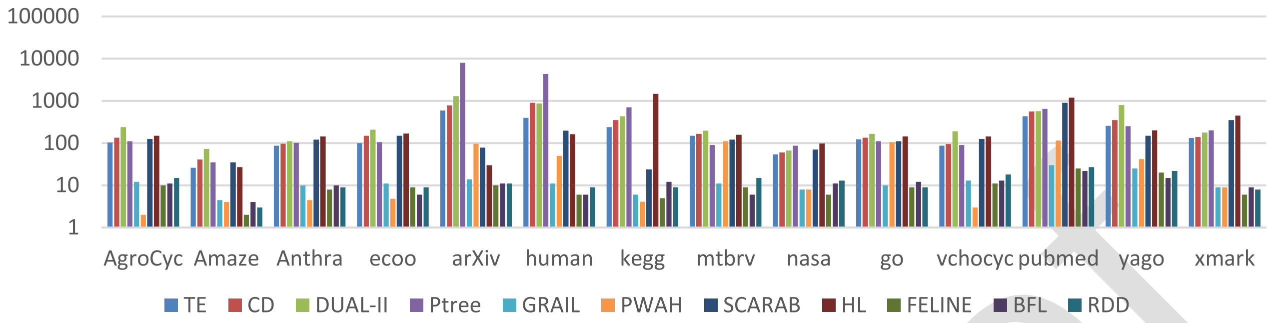

Fig. 17. Indexing time (ms) for small graphs.

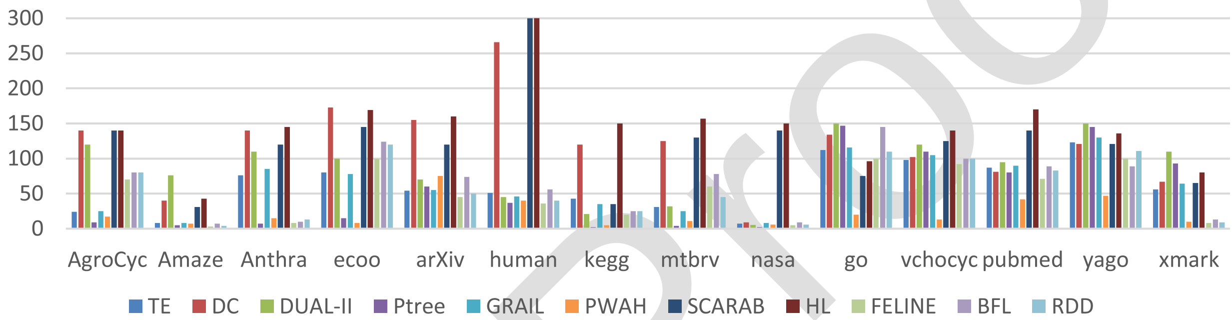

Fig. 18. Index space (× 1000 bytes) for small graphs.

TABLE 7
Small Graph Deduction Process With Modified DFS

dataset		Level 1	Level 2	Level 3	Level 4	Level 5
AgroCyc	$\|V\|$	12684	46	24	6	0
	$\|E\|$	13408	57	25	4	0
Amaze	$\|V\|$	1422	1369	1259	1208	1206
	$\|E\|$	1447	1394	1316	1315	1314
Anthra	$\|V\|$	361	340	337	335	334
	$\|E\|$	645	618	614	611	610
Ecoo	$\|V\|$	434	403	388	384	383
	$\|E\|$	835	797	764	759	758
arXiv	$\|V\|$	5108	3384	2832	2736	2090
	$\|E\|$	58243	18631	17833	17646	5456
Human	$\|V\|$	412	397	394	392	391
	$\|E\|$	720	704	701	698	697
Kegg	$\|V\|$	1230	1208	1188	1186	1185
	$\|E\|$	1348	1325	1301	1299	1298
Mtbrv	$\|V\|$	364	341	337	334	333
	$\|E\|$	681	651	644	641	640
Nasa	$\|V\|$	381	204	88	20	0
	$\|E\|$	425	256	78	19	0
go	$\|V\|$	2346	1296	1067	943	417
	$\|E\|$	4765	2355	2122	1920	808
VchoCyc	$\|V\|$	386	370	351	345	344
	$\|E\|$	731	665	665	658	657
PubMed	$\|V\|$	2656	2515	2334	2143	2093
	$\|E\|$	9514	9359	7072	6778	4404
Yago	$\|V\|$	5357	5226	4917	4415	4001
	$\|E\|$	33647	24168	14389	11147	9328
Xmark	$\|V\|$	385	178	160	142	135
	$\|E\|$	444	257	226	198	190

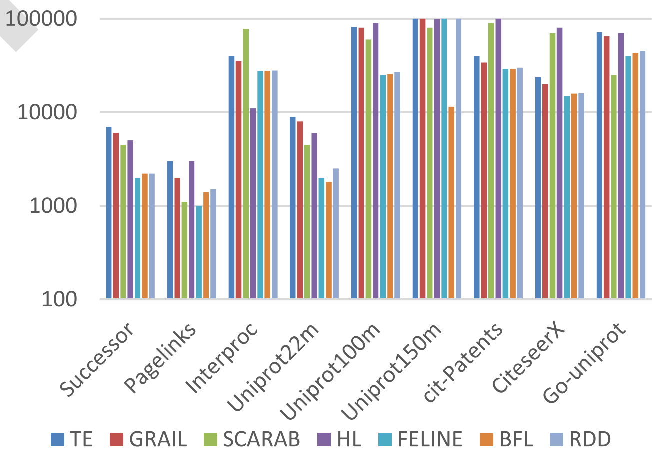

Fig. 19. Indexing time (ms) for large graphs.

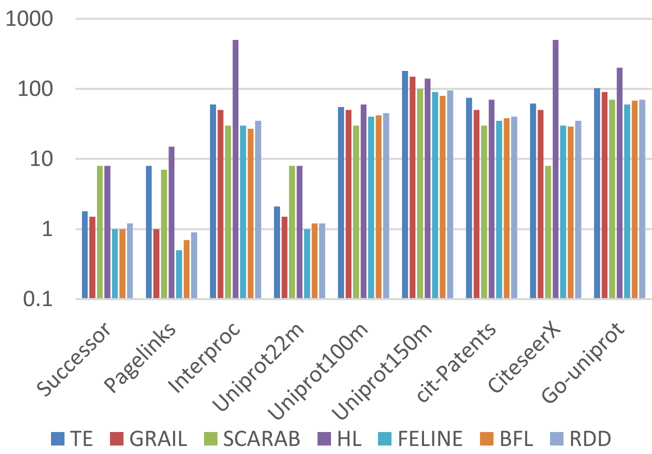

Fig. 20. Index size (× Mbytes) for large graphs.

TABLE 8
Large Graph Deduction Process With Modified DFS

dataset		Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9	Level 10
Successor	$\|V\|$	13442	1739	1102	794	642	0	0	0	0	0
	$\|E\|$	16845	1846	957	795	641	0	0	0	0	0
Pagelinks	$\|V\|$	17285	17242	17189	17118	17070	15981	15940	15932	15784	15617
	$\|E\|$	28918	27869	26195	25668	24357	23874	22830	22801	22789	22356
Interproc	$\|V\|$	34442	29018	21117	17888	15810	15808	15800	15606	15406	14806
	$\|E\|$	64736	55057	44379	30003	24042	24031	24021	23926	23026	22031
Uniprot22m	$\|V\|$	19668	14948	12085	10311	9143	6979	0	0	0	0
	$\|E\|$	16260	12849	10749	9412	8512	6497	0	0	0	0
Uniprot100m	$\|V\|$	198408	151456	115615	88255	67370	51427	39257	29967	22876	17462
	$\|E\|$	164159	130284	103400	82063	65129	54274	43075	34186	27132	21533
Uniprot150m	$\|V\|$	309106	234214	180164	137530	105792	80145	61650	47061	35924	27215
	$\|E\|$	255485	204388	127742	100584	79200	82880	72304	56436	40981	36016
cit-Patents	$\|V\|$	37007	27211	18766	14325	11019	7346	6121	5.655	5013	5010
	$\|E\|$	226289	113147	89797	68547	52325	41528	13842	10253	7826	7811
citeseerx	$\|V\|$	64121	48576	37950	29192	22629	18103	13510	10638	10003	9068
	$\|E\|$	148626	87427	69386	52966	42037	31138	17299	13307	10300	10042
go_uniprot	$\|V\|$	66999	51537	40263	30272	25227	18549	14721	11238	8711	6599
	$\|E\|$	463603	264919	189227	150181	115523	60802	33778	16889	12991	9842

TABLE 9
Large Graph Deduction Process With Traditional DFS

dataset		Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9	Level 10
Successor	$\|V\|$	93056	72675	67154	30004	9875	8903	7695	7654	7611	6789
	$\|E\|$	205723	141552	14268	74502	16221	16764	16124	15800	14235	13678
Pagelinks	$\|V\|$	92341	83634	67264	55004	51786	45432	42111	41257	41112	39342
	$\|E\|$	281918	257869	202195	118564	101278	96657	90981	89766	88453	76546
Uniprot22m	$\|V\|$	121452	86732	65724	54980	45333	41777	39667	30665	0	0
	$\|E\|$	136666	100312	87256	67005	55886	51432	42367	30664	0	0

can see that our method is just a little bit worse than *FELINE*, by which only one search of graphs is conducted, and each node in a graph is associated with a pair of integers. By our method, besides the generation of two topological numbers for each node, G and its deduced graphs will be searched up to 5 or more than 5 times and each node will then be attached with two sequences each containing 10 or more integers. Therefore, our method is in general worse than *FELINE*. Notice that by *GRAIL*, each graph is also searched exactly 5 times as ours. However, by our method, except for the first time of the graph search, a quite smaller graph will be navigated by any of the subsequent graph searches. Thus, our method is generally comparable to *GRAIL*.

In addition, by *SCARAB*, a *GRAIL*-like index is built over the backbone of a graph. This is much smaller than the original graphs. However, some more time is spent for handling relationships between the nodes outside and inside of backbones. Thus, the total indexing time and the index size of *SCARAB* are higher than *GRAIL*.

For the large graphs, we only show the results of seven strategies in Figs. 19 and 20, *SCARAB*, *GRAIL*, *HL*, *FELINE*, *BFL*, *TE* and *RDD* while all the other methods fail to handle any of them, or one or two of them. From these two figures,

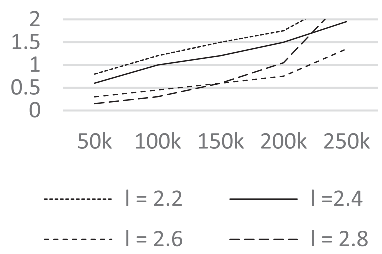

Fig. 21. Query time (μs) for SF graphs.

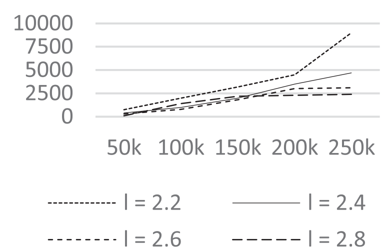

Fig. 22. Indexing time (ms) for SF graphs.

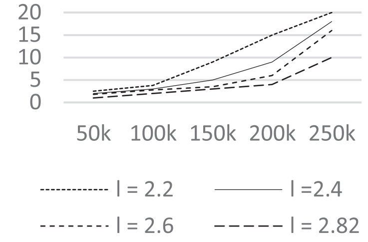

Fig. 23. Index size (Mbytes) for SF graphs.

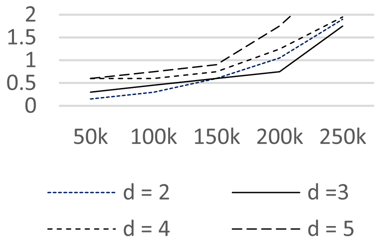

Fig. 24. Query time (μs) for ER graphs.

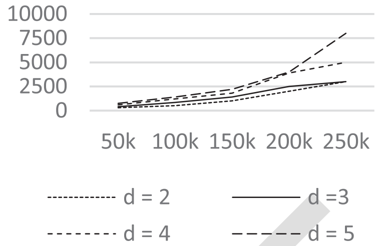

Fig. 25. Indexing time (ms) for ER graphs.

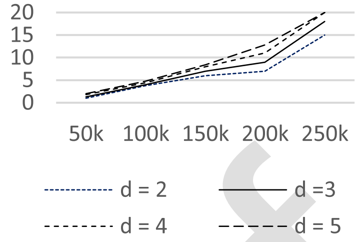

Fig. 26. Index size (Mbytes) for ER graphs.

TABLE 10
Query Time on SF Graphs (ms)

	TE	GRAIL	SCARAB	FELINE	HL	BFL
$l = 2.2$	41.57	79.83	27.09	65.09	77.31	51.75
$l = 2.4$	29.12	72.34	23.80	72.78	69.45	47.76
$l = 2.6$	24.65	61.67	19.45	51.72	56.98	41.43
$l = 2.8$	23.56	57.98	16.05	64.07	51.65	35.74

TABLE 11
Query Time on ER Graphs (ms)

	TE	GRAIL	SCARAB	FELINE	HL	BFL
$d = 2$	27.65	41.34	16.87	47.23	40.54	29.12
$d = 3$	35,12	49.34	21.43	49.13	45.87	45.87
$d = 4$	40,37	58.67	24.09	62.34	67.89	49.11
$d = 5$	46,87	76.38	31.24	69.56	75.67	56.77

we can see that the same analysis for the small graphs can be applied to the large ones.

Finally, we show the graph deduction process with the traditional *DFS* being used in Table 9. In comparison with Table 8, we can see that the graph deduction is much slower by using the traditional *DFS* than by using *mDFS*.

6.4 Tests on Synthetical Data

Two types of synthetical datasets are used in our experiments. One is the Erdos Renyi Model (ER). It is a random graph of $|V|$ vertices and $|E|$ edges. To create sets of arcs, we randomly select node u and node v from the corresponding node sets. When creating arcs, it is guaranteed that an edge is not repeatedly generated. This method simulates many real-world problems and may contains many large *SCC*s. The second dataset is the Scale FreeModel (SF). It is another random graph satisfying the power law distribution of node outdegrees d: $P(d) = \alpha d^{-l}$, where α and l are two constants. This dataset is created by the graph generator *gengraphwin* (http://fabien.viger.free.fr/liafa/generation/). For the SF graphs, we fix α to 1. To study the scalability in these two kinds of graphs, we vary graph size from 5k to 250K vertices. For ER graphs, we also vary node degree d from 2 to 5; and for SF graphs, we change l from 2.2 to 2.8. Thus, the largest graph generated may have more than one million arcs. Note that the smaller l is, the denser the corresponding graph. The goal of this test is to understand the impact of graph density on performance, for both different synthetical graph generation models. We expect that all the parameters for all the tested methods will increase as the graphs become denser.

It is because the number of possible paths to explore between nodes, as well as the index sizes, increases with density. Figs. 21, 22, and 23 summarize the results for the ER graphs and Figs. 24, 25, ands 26 for the SF graphs.

In addition, for the purpose of comparison, we have also tested some other methods on the two synthetical graphs. The results are summarized in two tables: Table 10 shows the query time of some strategies on an SF graph while Table 11 shows the query time on an ER graph. Both the SF and ER graphs contain 250k nodes. Comparing these two tables respectively with Figs. 21 and 24, we can see that all of them are much worse than ours.

7. 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 $k \geq 1$), which enables us to associate two sequences with each node u in G, denoted as A-sequence and B-sequence, respectively. The A-sequence is utilized to check reachability from u to any other node while the B-sequence is for checking the reachability from any other node to u. In this way, the query time can be reduced to O(k) and the space requirement to O(kn), where k is the number of the decomposed spanning trees. Theoretically, we have $k \leq \sqrt{n}$, where n is the number of nodes in G.

Extensive experiments are conducted to test different strategies over different kinds of graphs and real 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

[1] A. B. Agrawal and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *Proc. ACM SIGMOD Intl. Conf. Manage. Data*, 1989, pp. 253–262.

[2] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computation of reachability labeling for large graphs," in *Proc Int. Conf. Extending Database Technol.*, 2006, pp. 961–979.

[3] N. H. Cohen, "Type-extension tests can be performed in constant time," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 626–629, 1991.

[4] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. Comput*, vol. 32, no. 5, pp. 1338–1355, 2003.

[5] Y. Chen and Y. B. Chen, "An efficient algorithm for answering graph reachability queries," in *Proc. IEEE 24th Int. Conf. Data Eng.*, 2008, pp. 892–901.

[6] Y. Chen and Y. B. Chen, "Core labeling: A new way to compress transitive closures," in *Proc. IEEE 4th Int. Conf. Signal-Image Technol. Internet-Based Syst.*, 2008, pp. 3–10.

[7] Y. Chen, "General spanning trees and reachability query evaluation," in *Proc. 2nd Can. Conf. Comput. Sci. Softw. Eng.*, 2009, pp. 243–252.

[8] Y. Chen and Y. B. Chen, "Decomposing DAGs into spanning trees: A new way to compress transitive closures," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 1007–1018.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York, NY, USA: W.H. Freeman &, 1990.

[10] H. V. Jagadish, "A compression technique to materialize transitive closure," *ACM Trans. Database Syst.*, vol. 15, no. 4, pp. 558–598, 1990.

[11] R. Jin, N. Ruan, Y. Xiang, and H. Wang, "Path-Tree: An efficient reachability indexing scheme for large directed graphs," *ACM Trans. Database Syst.*, vol. 1, pp. 1–52, 2011.

[12] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 595–608.

[13] D. E. Knuth, *The Art of Computer Programming*, Reading, U.K: Addison-Wesley, 1969.

[14] H. A. Kuno and E. A. Rundensteiner, "Incremental maintenance of materialized object-oriented views in multiview: Strategies and performance evaluation," *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 5, pp. 768–792, Sep./Oct. 1998.

[15] W. C. Lee and D. L. Lee, "Path dictionary: A new access method for query processing in object-oriented databases," *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 3, pp. 371–388, May/Jun. 1998.

[16] I. Munro, "Efficient determination of the transitive closure of directed graphs," *Inf. Process. Lett.*, vol. 1, no. 2, pp. 56–58, 1971.

[17] R. Schenkel, A. Theobald, and G. Weikum, "HOPI: An efficient connection index for complex XML document collections," in *Proc. Int. Conf. Extending Database Technol.*, 2004, pp. 237–255.

[18] R. Schenkel, A. Theobald, and G. Weikum, "Efficient creation and incrementation maintenance of HOPI index for complex XML document collection," in *Proc. Int. Conf. Extending Database Technol.*, 2006, pp. 237–255.

[19] M. A. Schubert and J. Taugher, "Determining type, part, colour, and time relationship," *Computer*, vol. 16, pp. 53–60, 1983.

[20] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Compt.*, vol. 1, no. 2, pp. 146–140, Jun. 1972.

[21] R. Tarjan, "Finding optimum branching," *Networks*, vol. 7, pp. 25–35, 1977.

[22] J. Teuhola, "Path signatures: A way to speed up recursion in relational databases," *IEEE Trans. Knowl. Data Eng.*, vol. 8, no. 3, pp. 446–454, Jun. 1996.

[23] M. Thorup, "Compact oracles for reachability and approximate distances in planar digraphs," *JACM*, vol. 51, pp. 993–1024, Nov. 2004.

[24] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *Proc. IEEE Int. Conf. Data Eng.*, 2006, pp. 75–75.

[25] H. Yildirim, V. Chaoji, and M. J. Zaki, "GRAIL: Scalable reachability index for large graphs," in *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 276–284, 2010.

[26] Y. Zibin and J. Gil, "Efficient subtyping tests with PQ-Encoding," in *Proc. ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang. Application*, 2001, pp. 96–107.

[27] H. S. Warren, "A modification of warshall's algorithm for the transitive closure of binary relations," *Commun. ACM*, vol. 18, pp. 218–220, Apr. 1975.

[28] S. J. van Schaik and O. de Moor, "A memory efficient reachability data structure through bit vector compression," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 913–924.

[29] R. Jin, N. Ruan, S. Dey, and J. X. Yu, "SCARAB: Scaling reachability computation on large graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 169–180.

[30] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *Proc. VLDB Endowment*, vol. 7, pp. 1–26, 2014.

[31] R. R. Veloso1, L. Cerf, W. Meira, Jr, and M. J. Zaki, "Reachability queries in very large graphs: A fast refined online search approach," in *Proc. 17th Int. Conf. Extending Database Technol.*, 2014, pp. 511–522.

[32] U. Feige, "A threshold of ln(n) for approximating set cover," *J. ACM*, vol. 45, no. 4, pp. 634–652, 1998.

[33] K. Mehlhorn, *Graph Algorithms and NP-Completeness*, New York, NY, USA: Springer, 1984.

[34] R. Jin and G. Wang, "Simple, fast, and scalable reachability oracle," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 1978–1989, 2013.

[35] J. Su, Q. Zhu, H. Wei, and J. X. Yu, "Reachability querying: Can it be even faster?," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 3, pp. 683–697, Mar. 2017.

[36] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum, "Ferrari: Flexible and efficient reachability range assignment for graph indexing," in *Proc. IEEE Int. Conf. Data Eng.*, 2013, pp. 1009–1020.

[37] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu, "TF-label: A topologicalfolding labeling scheme for reachability querying in a large graph," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2013, pp. 193–204.

[38] J. Cai and C. K. Poon, "Path-hop: Efficiently indexing large graphs for reachability queries," in *Proc. 19th ACM Int. Conf. Inf. Knowl. Manage.*, 2010, pp. 119–128.

[39] M. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin, "Lowest common ancestors in trees and directed acyclic graphs," *J. Algorithms*, vol. 57, no. 2, pp. 75–94, 2005.

Yangjun Chen received the PhD degree in computer science from the University of Kaiserslautern, Germany, in 1995. He is now a professor with the Department Applied Computer Science, University of Winnipeg, Canada. He has about 200 publications in Computer Science and Computer engineering.

Yibin Chen received the BS and master's degree from the Department of Electrical and Computer Engineering, University of Waterloo, and the Department of Electrical and Computer Engineering, University of Toronto, Canada, respectively. Now he is a software engineer.

Yifeng Zhang received the BS and master's degree from the Department of Applied Computer Science, University of Winnipeg, and the Department of Computing Science, University of Alberta, Canada, respectively. Now he is a software engineer.