

Graph Indexing for Efficient Evaluation of Label-constrained Reachability Queries

YANGJUN CHEN and GAGANDEEP SINGH, Dept. Applied Computer Science,
University of Winnipeg, Canada

Given a directed edge labeled graph G , to check whether vertex v is reachable from vertex u under a label set S is to know if there is a path from u to v whose edge labels across the path are a subset of S . Such a query is referred to as a *label-constrained reachability (LCR)* query. In this article, we present a new approach to store a compressed transitive closure of G in the form of intervals over spanning trees (forests). The basic idea is to associate each vertex v with two sequences of some other vertices: one is used to check reachability from v to any other vertex, by using intervals, while the other is used to check reachability to v from any other vertex. We will show that such sequences are in general much shorter than the number of vertices in G . Extensive experiments have been conducted, which demonstrates that our method is much better than all the previous methods for this problem in all the important aspects, including index construction times, index sizes, and query times.

CCS Concepts: • **Theory of computation** → **Shortest paths**;

Additional Key Words and Phrases: Labeled directed graphs, Label constraint reachability, Tree labeling, Recursive graph decomposition, Spanning trees

ACM Reference format:

Yangjun Chen and Gagandeep Singh. 2021. Graph Indexing for Efficient Evaluation of Label-constrained Reachability Queries. *ACM Trans. Database Syst.* 46, 2, Article 8 (May 2021), 50 pages.
<https://doi.org/10.1145/3451159>

1 INTRODUCTION

Graph reachability has received much attention in recent years in the graph database research community. One of the important research problems is the so-called **Label-Constrained Reachability (LCR)** over graphs. Given two vertices u and v in an edge labeled directed graph G and a label constraint set S , an *LCR* query asks if there is a path from u to v such that all edge labels on the path are a subset of S . As an example, consider a social network, where each vertex v represents a person and two persons are linked by an edge if they are related in some way, over which we may ask, for instance, whether u is a remote relative of v . Then, we will check whether u is reachable from v through a path with each edge on it labeled only with those relationships like *parent-of*, *brother-of*, *sister-of*, *uncle-of*, and so on. As a second example, we consider an

Authors' addresses: Y. Chen, Dept. Applied Computer Science, University of Winnipeg, 515 Portage Ave., Winnipeg, Manitoba, Canada R3B 2E9; email: yc9579@gmail.com; G. Singh, 60 Osborne st. Winnipeg, Manitoba, Canada R3L 03C; email: gagandeep.singh5133@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

0362-5915/2021/05-ART8 \$15.00

<https://doi.org/10.1145/3451159>

application in forensics and assume that a detective may want to investigate an individual who is related to a known criminal through some relationships, such as *money laundry*, *human trafficking*, and so on. Then, the reachability is checked under a set of labels, representing different criminal activities. For the third example, let us have a look at a metabolic network that is also an edge labeled graph with each vertex representing a compound. Two compounds are connected by an edge if one can be transformed into another through a certain chemical reaction controlled by a certain enzyme. Here, a basic question is whether two compounds can be active through a pathway under a set of enzymes. Other practical applications such as semantic web, security networks, citation, and tracing the transmission of infectious diseases all need to take edge labels into account. In fact, *LCR* queries also appear as an important fragment of the language of regular path queries [4, 5, 7, 31], which are essentially reachability queries constrained by regular expressions. Indeed, formulated in terms of regular path queries, *LCR* is equivalent to the problem of determining whether or not there is a path in G from u to v such that the concatenation of edge labels along the path forms a string in the language denoted by a regular expression $(a_1 \cup \dots \cup a_n)^*$, where a_i is taken from a certain alphabet ($i = 1, \dots, n$), \cup is disjunction, and $*$ is the Kleene star. *LCR* and, more generally, regular path queries are supported in practical graph query languages such as SPARQL 1.1 (<http://www.w3.org/TR/sparql11-query/>), PGQL (pgql-lang.org), and openCypher (<http://www.opencypher.org>).

According to the research on the large SPARQL query logs [40], *LCR* queries are a vast majority of path related queries in the practical workload.

The research on efficient solutions for *LCR* queries was initiated in the work of Jin et al. [13] with several recent follow-up studies [3, 16, 20, 22, 32]. However, they need either too much time to build up indexes, such as the methods reported in References [22, 32], or too much time to answer a query, such as in Reference [20], not scaling well to larger graphs that are common in contemporary applications. References [3] and [16] mainly target the computation of shortest paths.

In this article, we address this issue and present a new indexing method to solve the problem. Generally, we recognize a kind of graph decomposition and edge classification, by which, with respect to a spanning tree (forest) T of G , all the edges in G are divided into four disjoint groups: *tree edges*, *forward edges*, *cross edges*, and *back edges* [10] and handled in different ways to avoid constructing **transitive closures (TC)** or *partial TCs* [13], both of which require a great amount of generation time and a very large storage space.

Concretely, our method works as follows:

- G will be decomposed into a series of k spanning trees (forests) T_0, \dots, T_{k-1} (for some $k \leq n$). Then, a series of tree-like subgraphs T_0', \dots, T_{k-1}' will be created. If G is a directed acyclic graph (*DAG* for short), then each T_i' ($i = 1, \dots, k - 1$) is constructed by adding to T_i , the forward edges with respect to T_i . If G contains cycles, then both the corresponding forward and back edges will be added to T_i to form T_i' .
- Accordingly, each vertex v will be associated with two (uniquely determined) sequences: $X = x_0, \dots, x_r$ and $Y = y_0, \dots, y_r$ with $0 \leq r \leq k - 1$, $x_0 = y_0 = v$, and $x_j, y_j \in T_j$ ($0 \leq j \leq r$). For $0 < i \leq r$, x_i is the *dominant vertex* of x_{i-1} , used to check reachability from x_{i-1} to any other vertex through cross edges while y_i is the *transferring vertex* of y_{i-1} , used to check reachability to y_{i-1} from any other vertex through cross edges. So, X is called a *from-sequence* while Y is called a *to-sequence*.
- To check whether v is reachable from another vertex u , also associated with a from-sequence $W = w_0, \dots, w_s$ and a to-sequence $Z = z_0, \dots, z_s$ with $0 \leq s \leq k - 1$, we will find whether there exists an j ($0 \leq j \leq \min\{r, s\}$) such that $u = w_0 \rightsquigarrow \dots \rightsquigarrow w_{j-1} \rightsquigarrow w_j$, $y_j \rightsquigarrow y_{j-1} \rightsquigarrow \dots \rightsquigarrow y_0 = v$, and y_j is reachable from w_j through a path in T_j' . Here, $w_i \rightsquigarrow w_{i+1}$ ($y_{i+1} \rightsquigarrow y_i$) for $0 \leq i \leq j - 1$ represents that w_{i+1} is reachable from w_i (y_i is reachable from y_{i+1}) in T_i' .

- In this way, a query q (to check whether $u \rightsquigarrow v$ under S) will be decomposed into a series of subqueries q_0, \dots, q_l ($0 \leq l \leq j$), defined as below:
 - For $i = 0$, q_0 is to check whether $w_0 \rightsquigarrow y_0$ under S in T_0' . If it is the case, then return *true*. Otherwise, continue to check q_1 .
 - For $i > 0$, q_i is evaluated in three steps:
 - (a) Check whether $w_i \rightsquigarrow w_{i+1}$ under S in T_i' . If it is not the case, return *false*. Otherwise, go to (b).
 - (b) Check whether $y_{i+1} \rightsquigarrow y_i$ under S in T_i' . If it is not the case, then return *false*. Otherwise, go to (c).
 - (c) Check whether $w_i \rightsquigarrow y_i$ under S in T_i' . If it is the case, then return *true*. Otherwise, continue to check q_{i+1} if $i < j - 1$, or return *false* if $i = j$.

In addition, we also associate each vertex v with an extra pair of integers (s_v, t_v) , working as a filter, which are in fact two topological numbers, described in Reference [29]. If v is reachable from another node u , associated with (s_u, t_u) , then we must have $s_v \leq s_u$ and $t_v \leq t_u$. Thus, $s_v \not\leq s_u$ or $t_v \not\leq t_u$ indicates a negation, and then neither the scanning of the to- and from-sequences nor the checking of labels is needed to negatively answer a label constraint reachability query from u to v .

In this way, a query can be evaluated very efficiently, since the reachability within each T_i' ($i = 1, \dots, k - 1$) can be checked very quickly by using the index structure built for T_i' . We will discuss this index structure in great detail.

In summary, the index construction time for DAGS can be reduced to $O(\sum_{i=0}^{k-1} (m_i + \chi_i |\Sigma| + \chi_i h_i))$ while the index space to $O(\sum_{i=0}^{k-1} (|T_i| + \chi_i |\Sigma| + \chi_i h_i))$, where m_i is $|T_i'|$ plus the number of all the corresponding cross edges with respect to T_i , χ_i , and h_i are, respectively, the number of all forward edges and the maximum number of forward edges attached to a path in T_i , and Σ is the set containing all the edge labels of G . (We say, a forward edge $s \rightarrow t$ is attached to a path p if both s and t appear on p .) The query time is bounded by $O(\sum_{i=0}^{k-1} (h_i^2 + h_i |\Sigma|))$. For cyclic graphs, the index construction time and the index space are bounded by $O(\sum_{i=0}^{k-1} (m_i + b_i + \chi_i |\Sigma| + \chi_i h_i))$ and $O(\sum_{i=0}^{k-1} (|T_i| + b_i + \chi_i |\Sigma| + \chi_i h_i))$, respectively, where b_i is the maximum number of all those back edges $s \rightarrow t$ such that their end vertices t are on a same path in T_i' . The query time is bounded by $O(\sum_{i=0}^{k-1} b_i (h_i^2 + h_i |\Sigma|))$.

In general, $k \leq n$. However, in our experiments, for all the tested real data graphs, $k \ll n$.

The organization of the rest of this article is as follows: We summarize the notations and symbols used throughout this article in Section 2. In Section 3, we review the related work. In Section 4, we define some important concepts related to graphs, edge labeled graphs, as well as *LCR* queries to provide a discussion background. Section 5 is devoted to the description of our method for DAGs, while Section 6 is for graphs containing cycles. In Section 7, we discuss all the important techniques used by the processes described in Sections 5 and 6. In Section 8, we report the experiment results. A short conclusion is set forth in Section 9.

2 NOTATIONS

Throughout the article, a lot of symbols and notations are used, which are summarized in Table 1 for reference.

3 RELATED WORK

In the past several decades, much research on the evaluation of reachability queries has been made and many algorithms have been proposed. Roughly speaking, all of them can be divided into two categories: *reachability without label constraints* and *reachability with label constraints*. By the former, we will check whether two vertices are connected through a path in a directed

Table 1. Symbols and Notations

G	a directed graph
$LCR(u, v, S, G)$	a query to check whether v is reachable from u under label set S in G
$u \rightsquigarrow v$	representing that vertex v is reachable from vertex u through a path in G
T	a spanning tree T plus all the forward edges
T_c	a skeleton tree, formed by removing some vertices from T
G_c	a summary graph, formed by removing some edges from T and adding some new edges
T'	a spanning tree plus all the forward edges and back edges
$l(e)$	label associated with an edge e
$L(p)$	path label: all labels cross p
p_v	a tree path (a path in T) from the root to vertex v
p_{uv}	a segment from u to v on a tree path p
$A(p)$	a multi-set representing all the labels on p
$T[v]$	the subtree rooted at v in T
$[\alpha, \gamma, \beta]$	a triplet associated with each vertex v , where α is v 's preorder number; $\beta - 1$ is equal to the largest preorder number among all the vertices in $T[v]$; and γ is a set containing the multi-labels of all the root-to- v paths in T that can be represented very efficiently.
τ	a quadruple of the form $[s, t, A(p_{st}), x]$, representing a forward edge from vertex s to vertex t , attached to a tree path p_{st} and labeled with x
$V_{c-start}$	all the start vertices of cross edges
V_{c-end}	all the end vertices of cross edges
$V_{f-start}$	all the start vertices of forward edges
V_{f-end}	all the end vertices of forward edges
V_{fs}	all those start vertices s of forward edges $s \rightarrow t$, where $t \in V_{c-start}$ or t is an ancestor of some vertex in $V_{c-start}$ with respect to T .
V_{fe}	all those end vertices t of forward edges $s \rightarrow t$, where $s \in V_{c-end}$ or s is a descendant of some vertex in V_{c-end} with respect to T .
V_{LCA}	all those vertices with each being a <i>lowest common ancestor</i> (<i>LCA</i> for short) of more than one vertex in $V_{c-start} \cup V_{fs}$, which are not related by the ancestor/descendant relationship in T
v^{\rightarrow}	dominant vertex of v , used to check reachabilty from v to some other vertices in G_c .
v^{\leftarrow}	transferring vertex of v , used to check reachabilty to v from some other vertices in G_c .
E_{f-c}	set of forward edges to be inserted into G_c
ω_v	combined sequence of dominant and transferring vertices, associated with v
\mathfrak{w}_v	label sequence, associated with v
$V_{b-start}$	all the start vertices of back edges
V_{b-end}	all the end vertices of back edges
V_{bs}	all those start vertices u of back edges $u \rightarrow v$, where $v \in V_{c-start}$ or v is an ancestor of some vertex in $V_{c-start}$ with respect to T .
V_{be}	all those end vertices v of back edges $u \rightarrow v$, where $u \in V_{c-end}$ or u is a descendant of some vertex in V_{c-end} with respect to T .
T'	a spanning tree T plus all the forward edges and all back edges
G_c'	a summary graph, formed by removing some edges from T' and adding some new edges
E_{b-c}	set of back edges to be inserted into G_c'

graph [1, 4–6, 11–14, 23, 26, 27, 43–47]. By the latter, we will not only check whether a vertex is reachable from another vertex in a directed graph, but also through a certain path whose labels fall into a given set of labels [13, 16, 20, 22, 32].

–*reachability without label constraints*

Let $G(V, E)$ be a directed graph. The reflexive, transitive closure of G is a digraph $G^* = (V, E^*)$, where $v \rightarrow u \in E^*$ iff there is a path from v to u in G . Obviously, if a transitive closure is physically stored, then the checking of the ancestor-descendant relationship can be done in a constant time. However, the materialization of a whole transitive closure is very space-consuming. Therefore, it is desired to find a way to compress transitive closures, but without sacrificing too much query time.

Chain decomposition methods. In Reference [11], Jagadish suggested a method to decompose a DAG into vertex-disjoint chains. On a chain, if vertex v appears above vertex u , then there is a path from v to u in G . Then, each vertex 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 $O(\mu n)$ space overhead and $O(1)$ query time, where μ is the number of chains. However, to find a set of chains for a graph, Jagadish's algorithm requires $O(n^3)$ time (see page 566 in Reference [11]). In addition, the number μ of the produced chains is normally much larger than the minimal number of chains. In the worst case, μ is $O(n)$.

The method discussed in Reference [5] greatly improves Jagadish's method. It needs only $O(n^2 + \omega^{1.5}n)$ time to decompose a DAG into a minimum set of vertex-disjoint chains, where ω represents G 's width, defined to be the size of a largest vertex subset U of G such that for any pair of vertices $u, v \in U$ there does not exist a path from u to v or from v to u . Its space overhead is $O(\omega n)$ and its query time is bounded by a constant. In Reference [4, 6], the concept of the so-called general spanning tree is introduced, in which each edge corresponds to a path in G . Based on this data structure, the real space requirement becomes smaller than $O(\omega n)$, but the query time increases to $\log \omega$.

Interval-based methods. In Reference [1], Agrawal et al. proposed a method based on interval labeling. This method first figures out a spanning tree T and assigns to each vertex 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 vertices in $T[v]$, the subtree rooted at v). Another vertex 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. [58]. In a next step, each vertex v in G will be assigned a sequence $s(v)$ of intervals such that another vertex u in G with interval (x, y) is a descendant of v (with respect to G) iff there exists an interval (a, b) in $s(v)$ such that $a \leq y < b$. The time and space complexities are bounded by $O(\lambda m)$ and $O(\lambda n)$, respectively, where $m = |E|$ and λ is the number of the leaf vertices in T . The querying time is bounded by $O(\log \lambda)$. In the worst case, $\lambda = O(n)$.

The method discussed in Reference [21] 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 vertex v a dual label: $[a_v, b_v)$ and (x_v, y_v, z_v) . In addition, a $t \times t$ matrix \mathbf{N} (called a *TLC* matrix) is maintained, where t is the number of non-tree edges (edges not appearing in T). Another vertex 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 $O(n + m + t^3)$ time. The query time is $O(1)$. As a variant of *Dual-I*, one can also store \mathbf{N} as a tree (called a *TLC* search tree), which can reduce the space overhead from a practical viewpoint, but increases the query time to $\log t$. This scheme is referred to as *Dual-II*.

2-hop labeling. The method proposed by Cohen et al. [8] 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 , then there must exist $(h_1, v) \in H$ and $(h_2, u) \in H$ and one of the paths between v and u is the concatenation $h_1 h_2$. Using this method to label a graph, the worst space overhead is still in the order of $O(n^2)$. 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 Reference [8], by which each vertex v is assigned two labels, $C_{in}(v)$ and $C_{out}(v)$, where $C_{in}(v)$ contains a set of vertices that can reach v , and $C_{out}(v)$ contains a set of vertices reachable from v . Then, a vertex u is reachable from vertex v if $C_{in}(u) \cap C_{out}(v) \neq \phi$ (empty set). Using this method, the overall label size is increased to $O(\sqrt{m} \log n)$. In addition, a reachability query takes $O(\sqrt{m})$ time, because the average size of each label is above $O(\sqrt{m})$. The time for generating labels is $O(n^4)$. The 2-hop labeling is improved by the so-called 3-hop labeling [61] and path-hop labeling [9]. The path-hop labeling is slightly better than the 3-hop labeling with its indexing time and index size bounded by $O(nm)$ and $O(\lambda n)$, respectively. Its query time is in the order of $O(\log^2 \lambda)$.

Path-tree decomposition. Recently, Jin et al. [14] discussed a new method, by which a DAG G is decomposed into a set of vertex-disjoint paths. Then, a weighted directed graph G_w (called path-graph in Reference [15]) is constructed, in which each vertex represents a path and there is an edge (i, j) if on path i there is a vertex connected to a vertex on path j . The weight associated with (i, j) is the number of such connections. Then, find a maximum spanning tree T_w (called path-tree) of G_w and label the vertices in T_w with intervals as done in the method proposed by Agrawal et al. The space complexity of this method is $O(\lambda n)$. The query time and the labeling time are bounded by $O(\log^2 \lambda)$ and $O(\lambda m)$, respectively (see the analysis of Reference [14]). As mentioned above, λ 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].

GRAIL. The method proposed by Yildirim et al. [23] is a lightweight indexing structure. It traverses G for several times to create an interval sequence for each vertex, used as a filter as follows: Let $L_u = L_u^1, \dots, L_u^K$ and $L_v = L_v^1, \dots, L_v^K$ be the interval sequences of u and v , respectively. If there exists i ($i \in \{1, \dots, k\}$) such that $L_u^i \not\subseteq L_v^i$, then u is definitely not a descendant of v . But if for all $i \in \{1, \dots, k\}$ $L_u^i \subseteq L_v^i$, then 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 $O(k(n + m))$. If k is chosen as a constant, then the index size is proportional to $O(n)$ and can be established very fast. But in the worst case, the query time is $O(m)$ as if no index is established. The method discussed in Reference [29] is similar to *GRAIL*, but each vertex v is associated with a single pair of integers (x, y) . If v is reachable from another vertex u , associated with (x', y') , we must have $x \leq x'$ and $y \leq y'$. Thus, $x \not\leq x'$ or $y \not\leq y'$ indicates a negative answer, and then no traversal of G is needed to further check the reachability from u to v .

SCARAB. In Reference [27], a different method is discussed, in which a deducted TC over a subset V^* of vertices, 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 vertices (where δ is a pre-determined constant), i.e., their distance is greater than δ , then there must exist two vertices 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 Reference [27], which is based on the set-cover algorithm [59] and needs $\Omega(\sum_{v \in V} (N_\delta(v) + E_\delta(v)))$ time, where $N_\delta(v)$ and $E_\delta(v)$ denote the vertices and the edges, respectively, in v 's forward δ -neighbourhood. In the worst case, its size is $\Omega(nd^\delta)$, where d is the maximum out-degree of a vertex 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 $O(n + m + |V^*|^2)$. The query time is bounded by $O(d^{\lceil \delta/2 \rceil} + d^{2\delta} \log |V^*|)$.

This method is further improved by Jin et al. [31]. 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 vertex hierarchy is defined as $V_0 = V \supset V_1 \supset V_2 \supset \dots \supset V_h$, with corresponding edge sets $E_0, E_1, E_2, \dots, E_h$, 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 $0 < i \leq h$. Its theoretical labeling time is slightly better than *SCARAB*, since G_i is constructed from G_{i-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 vertex makes up a layer, but with very high labeling time $\Omega(nl(n+m)L)$, where L is the maximal labeling size. Also, its index size and query time are comparable to *SCARAB* [27].

Independent-permutation. The method discussed in References [43, 47] is a hash-based approach. The main idea is to associate each vertex v with two sets $out(v)$ and $in(v)$. $out(v)$ is the entire set of vertices that u can reach including u itself while $in(v)$ is the entire set of vertices in which every vertex can reach u including u itself. To check $u \sim v$, it will be checked if u can reach all the vertices that v can reach and all vertices that can reach u can also reach v , respectively, denoted as $out(v) \subseteq out(u)$ and $in(v) \subseteq in(u)$. However, the set-containment checking $out(v) \subseteq out(u)$ is done by checking $out(v) \not\subseteq out(u)$, instead. To speed up the operation, the so-called Bloom-filtering [47] is used, by which the hash functions are utilized to reduce the space requirements. The main disadvantage of this method is the possible false positives due to the use of hash functions. In many cases, the search of G is needed to answer a query.

PWAH. The method discussed in Reference [26] works in two phases. In the first phase, a deducted transitive closure of G will be created using a method described in Reference [41], and then for each vertex a bit vector is used to represent all those vertices reachable from it. In the second phase, each of such vectors will be compressed using the so-called *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 vector has to be partially decompressed.

–*reachability with label constraints*

LCR queries are mainly discussed in References [13, 16, 20, 22, 32]. Although many strategies are available for evaluating reachability queries without edge labels, such as those described above, as well as those via regular paths [51, 52], none of them can be easily modified or extended for *LCR* queries.

Jin et al. The first algorithm for this purpose was proposed in Reference [13], by which the whole transitive closure of a graph G is divided into a spanning tree (forest) T and a partial transitive closure NT , defined according to a kind of path classification:

- P_s contains all the paths whose start edge is a tree edge (an edge in T).
- P_e contains all the paths whose end edge is a tree edge.
- P_n – contains all the paths whose start and end edges are both non-tree edges.

Accordingly, three sets of path labels between two vertices u and v can be defined. (A path label is all the edge labels across a certain path.)

$$\begin{aligned} M_s(u, v) &= \{L(p_{uv}) | p_{uv} \in P_s\} \\ M_e(u, v) &= \{L(p_{uv}) | p_{uv} \in P_e\} \\ NT(u, v) &= \{L(p_{uv}) | p_{uv} \in P_n\} - M_s(u, v) - M_e(u, v) \end{aligned}$$

where p_{uv} stands for a path from u to v , and $L(p_{uv})$ for the path label of p_{uv} .

Then, NT is defined to be all $NT(u, v)$'s for $(u, v) \in V \times V$.

Denote by $\text{succ}(v)$ all the successors of v in T and $\text{pred}(v)$ all the predecessors of v in T . All the path labels between u and v can be represented as

$$M(u, v) = NT(u, v) \cup (\{T(u, u') | u' \in \text{succ}(u)\} \odot \{NT(u', v')\} \odot \{T(v', v) | v' \in \text{pred}(v)\})$$

where $T(u, u')$ represents a path label from u to u' in T , and \odot operator joins two sets of sets, such as $\{s_1, s_2\} \odot \{s_1', s_2'\} = \{s_1 \cup s_1', s_1 \cup s_2', s_2 \cup s_1', s_2 \cup s_2'\}$.

To mitigate the computational complexity of NT s to some extent, Jin et al. have also used two additional techniques. By the first one, $NT(u, v)$ is reduced based on a simple fact that if a path label L from u to v is a superset of another path label L' also from u to v , then L can be removed from $NT(u, v)$ without affecting the correctness of LCR queries, since if $L \subseteq$ some label constraint S , then we must also have $L' \subseteq S$. Thus, $NT(u, v)$ contains only non-comparable elements in the power set of Σ and its size is bounded by $\binom{|\Sigma|}{|\Sigma|/2}$. This bound can be easily observed and in Combinatorics is often referred to as the Sperner's theorem [2]. In the worst case, this pruning process requires $O(n^3 2^{|\Sigma|})$ time [13]. The second technique is based on a different observation that different spanning trees (forests) will lead to different NT s. To find a best spanning tree (forest) to minimize the size of an NT , a *weight* $w(e)$ for each edge e is introduced, defined to be proportional to the number of path labels that can be removed from the NT if e appears in T . With $w(e)$'s, any algorithm for finding a maximum spanning tree can be used for this purpose.

However, the cost for finding $w(e)$'s is prohibitively high. For this reason, Jin et al. proposed a *sampling* method to compute a single-source transitive closure for each sampling vertex, which enables them to develop a heuristics to find $w(e)$'s. The time required for this process is bounded by $O(km \binom{|\Sigma|}{|\Sigma|/2})$ with $O(n^2 \binom{|\Sigma|}{|\Sigma|/2})$ space required for storing indexes, where κ is the number of *sampling seeds* (vertices) for each of which a single-source transitive closure is created [13]. The query time is bounded by $O(\log \kappa + \sum_{i=0}^{k-1} |NT(u_i, v_i)|)$, where k is the maximum number of pairs (u_i, v_i) such that u_i is reachable from u (under a label constraint set S) and v_i is reachable from v_i (under S) through a tree path in T [13].

Zou et al. The method described above has been improved by Zou et al. [22, 32]. In their algorithm, an interesting concept of *distance* was introduced, by which the distance of two vertices u , v is defined to be the minimum number of distinct edge labels among all the paths from u to v . Based on this concept, they designed a Dijkstra-like algorithm to generate single-source transitive closures, by which redundant path labels, which would be created by Jin et al. [13] can be avoided. The reason for this is as follows: Let p_1, p_2 be two paths both going from u to v . Assume that $L(p_2) \subseteq L(p_1)$. Then, the distance of p_1 must be larger than p_2 and the Dijkstra's algorithm will ignore p_1 and only explore the shorter path p_2 .

Another improvement of Reference [32] consists in the isolation of SCCs. For each SCC containing vertices v_1, \dots, v_k for some k , a bipartite graph is created, which contains two sets of vertices $V_1 = \{v_1^1, \dots, v_k^1\}$, $V_2 = \{v_1^2, \dots, v_k^2\}$, and there is an edge $v_1^1 \rightarrow v_j^2$ for each pair v_i, v_j in the SCC, associated with the path labels of all the paths from v_i to v_j . In this way, many redundant path labels can be removed.

For very large graphs, Zou et al. [32] also proposed a graph partitioning strategy, by which G is divided into several components and for each of them an index as described above will be constructed. Hence, for evaluating an LCR query, the use of indexes and graph searching have to be hybridized in some way.

Although much redundant work is removed by Zou et al. [32], the theoretical computational complexities of their methods remain the same as the algorithm of Jin et al. [13]. This can be easily seen by considering a graph with each edge differently labeled. In such a graph, no path label is redundant.

Valstar et al. Recently, Valstar et al. has proposed a method [20] that is slightly different from Jin's. By this method, a set of vertices, referred to as *landmarks*, is first randomly selected, and for each of them a single-source transitive closure is constructed. Then, two more data structures are added:

- For each non-landmark vertex v , a set of pairs (v', L) with v' being a landmark will be built, where L is a set of path labels from v to v' .
- For each landmark vertex u , a set of pair (H, L) will be established, where H contains a subset of landmarks each reachable from u through a path labeled with L .

According to Reference [20], the index construction time and size are, respectively, bounded by $O(\lambda(n(\log n + 2^{|\Sigma|)} + m)2^{|\Sigma|})$, and $O(\lambda n 2^{|\Sigma|})$, where λ is the number of the chosen landmarks. However, in the worst case, the query time is bounded by $O(m)$, since for a *false* query almost the whole graph needs to be searched if the landmarks in the index cannot be used.

Hassan et al. The method discussed in Reference [16] is to find a shortest path p from a vertex v to another vertex u such that all edge labels on p are a subset of S , which is a more general problem than *LCR*. The main idea behind it is to run the Dijkstra's algorithm against an index structure described below.

- By the index, graph G is partitioned into $|\Sigma|$ portions such that each of them contains only the edges of the same label.
- A vertex u in a portion P is called a *bridge* if it has at least one outgoing edge with a label different from P . Then, between each u and a bridge vertex v (within P) a short cut edge e is produced. The weight of e is defined to be the sum of all the edge weights on a shortest path from u to v (within P).

To check whether v is reachable from u under S , the subgraph composed of all the shortcuts in all those portions with labels appearing in S will be traversed using the Dijkstra's algorithm.

The indexing time and the index size are bounded by $O(|\Sigma|n'^3)$ and $O(m + |\Sigma|n'd)$, respectively, where n' is the largest number of vertices in a portion and d is the largest vertex out-degree in G . According to Reference [16], its query time is bounded by $O(|S| m' + |S|n' \log n')$, where m' is the number of edges in a largest portion. (The method proposed in Reference [39] is another algorithm for finding a shortest path under S . But according to the experiments reported in Reference [16], this method's query time is up to four orders of magnitude worse than the algorithm discussed in Reference [16]).

Regular path queries (RPQ). This kind of queries is essentially reachability queries, but constrained by regular expressions [51], and therefore is also more general than *LCR*. In fact, *LCR* is just a subset of *RPQ*, equivalent to the problem of determining whether or not there is a path in G from u to v such that the edge labels along the path make up a string $\in (a_1 \cup \dots \cup a_n)^*$, where each a_i ($i = 1, \dots, n$) is a label taken from a given set of symbols. To the best of our knowledge, current state-of-the-art systems (such as SPARQL engines) rely on variations of BFS with no index being used [55] or use indexes that cannot be effectively applied for *LCR* queries, since they handle different query types (see References [56, 57]). Besides, the current techniques on regular path query evaluation on RDF database systems is impractical on graphs with more than a few thousand edges [38]. However, modern applications require to process queries on a graph that is multiple orders of magnitude larger.

In addition to the above methods, there are some approximate approaches discussed in the literature, such as the algorithm proposed by Bonchi et al. [3] to compute approximately shortest paths between two vertices, and the method described by Dumbrava et al. in Reference [42] to give approximate answers to *LCR* queries, as well as the strategy discussed in Reference [52] for

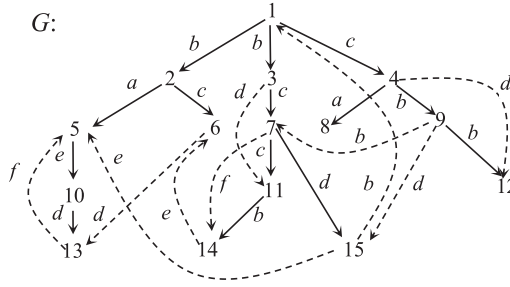


Fig. 1. A running example.

approximate regular-simple-path reachability. None of these methods can be modified to an efficient approach to produce exact answers to *LCR*.

In our approach, however, we avoid constructing large *TCs* or partial *TCs* by decomposing a graph into a series of subgraphs to keep and transfer reachability information through common edges among them. More importantly, this enables us to build a very concise index structure by which each vertex v is associated with two sequences used to check reachability from v , as well as to v , respectively. The length of each sequence is much shorter than n , therefore requiring much less space for storing indexes than all the existing methods. However, unlike *Valstar's* [20] and *Hassan's* [16], by which only part of vertices is indexed and G needs to be searched when the index cannot be used, our method indexes all vertices and no search of G is needed in any case.

4 BASIC DEFINITION

In this section, we give all the basic definitions that are required for the subsequent discussion. First, we restate the edge labeled directed graph, which was first described in Reference [13].

Definition 4.1 (Edge Labeled, Directed Graphs [13]). An edge labeled, directed graph is a quadruple $G = \langle V, E, \Sigma, l \rangle$, where V is a finite set of vertices, Σ is a set of labels, $E \subseteq V \times V$ is a finite set of edges, and $l: E \rightarrow \Sigma$ is a labeling function that assigns each edge $e \in E$ a label in Σ , denoted as $l(e)$.

As an example, consider Figure 1, in which we show a typical edge labeled directed graph (where solid and dashed edges are with respect to the edge classification to be discussed later in this section). As in Reference [13], we will use integers to represent vertices and letters for edge labels. In addition, in the following discussion, we will simply refer to an edge labeled directed graph as a graph, since we will not touch any other kind of graphs.

Given two vertices u and v in a graph G , a path p from u to v is represented as a sequence $p = (u_1, e_1; u_2, e_2; \dots; u_k)$ for some k , where $u_1 = u$, $u_k = v$, and for each $i \in \{1, \dots, k-1\}$ $e_i = u_i \rightarrow u_{i+1}$. In the case that $k = 1$, we have $u = v$ and consider $e_0 = u \rightarrow u$ as a virtual edge labeled with an empty symbol.

Based on the edge labels, the label of a path can be defined as follows:

Definition 4.2 (Path Labels). Let $G = \langle V, E, \Sigma, l \rangle$ be a graph. Let $p = (u_1, e_1; u_2, e_2; \dots; u_k)$ be a path from u to v . Then, the label of p is a set of labels $L(p) = \{l(e_1)\} \cup \{l(e_2)\} \cup \dots \cup \{l(e_{k-1})\}$.

For example, in Figure 1, for path $p = 1 \xrightarrow{b} 2 \xrightarrow{a} 5 \xrightarrow{e} 10 \xrightarrow{d} 13$, we have $L(p) = \{a, b, d, e\}$.

Definition 4.3 (Problem Definition [14]). Given two vertices u and v in G and a label set S . Vertex v is reachable from vertex u under S if there is a path p from u to v and $L(p) \subseteq S$.

The query defined above is denoted as $LCR(u, v, S, G)$, or $LCR(u, v, S)$ if G is clear from the context. For example, with respect to G shown in Figure 1, $LCR(1, 13, \{b, c, d, e\})$ asks if vertex

13 is reachable from vertex 1 under label constraint $\{b, c, d, e\}$. Since there is a path p from 1 to 13 such that $L(p) = \{b, c, d\} \subset \{b, c, d, e\}$, the query evaluates to *true*. However, $LCR(1, 13, \{a, c\})$ evaluates to *false*, since although vertex 13 is reachable from vertex 1, we cannot find any path satisfying the corresponding label constraint. In fact, infinitely many paths connecting 1 and 13 can be recognized due to the cycle: $5 \xrightarrow{e} 10 \xrightarrow{d} 13 \xrightarrow{f} 5$. However, none of them have a path label in $\{a, c\}$.

Definition 4.4 (Edge Classification [10]). By a spanning tree (forest) T of G , we mean a subgraph of G , which is a tree (forest) and covers all the vertices of G . With respect to T , all the edges in G can be classified into four groups:

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

For example, by exploring G in Figure 1 in the depth-first manner, we can find a spanning tree T as shown by the solid edges. With respect to T , we have $E_{cross} = \{6 \xrightarrow{d} 13, 14 \xrightarrow{e} 6, 15 \xrightarrow{e} 5, 9 \xrightarrow{b} 7, 9 \xrightarrow{d} 15\}$, $E_{forward} = \{3 \xrightarrow{d} 11, 7 \xrightarrow{f} 14, 4 \xrightarrow{d} 12\}$, and $E_{back} = \{13 \xrightarrow{f} 5, 15 \xrightarrow{b} 1\}$. All such cross, forward, and back edges together are referred to as non-tree edges and represented as dashed edges in Figure 1. Finally, we point out that in a DAG, we definitely have no back edges, since a back edge implies a cycle.

In addition, 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 connecting it to the root of every tree in T with an edge labeled with an empty symbol. Therefore, we will not distinguish spanning trees and spanning forests and always assume that there is a virtual root if what is found is a spanning forest.

5 LCR QUERIES OVER DAGs

In this section, we present our algorithm to evaluate LCR queries over DAGs, by which a DAG will be recursively decomposed and accordingly a query will be transformed into a series of subqueries with each being able to be evaluated efficiently.

We first sketch the overall idea of graph decomposition in Section 5.1. Then, we discuss how to evaluate reachability queries over the decomposed two parts in Section 5.2 and Section 5.3, respectively. In Section 5.4, we discuss how the recursive graph decomposition can be carried out. The analysis of time and space complexities and a proof of correctness are presented in Section 5.5.

5.1 Overall Idea

Let T be a spanning tree of G . Denote $T \cup E_{forward}$ by \mathbf{T} . By the first *decomposition* of G , we will establish \mathbf{T} from G and form a *summary* graph G_c containing the reachability information through the cross edges and transform the query $LCR(u, v, S, G)$ to a subquery that checks reachability over \mathbf{T} (referred to as a \mathbf{T} -checking) followed by another subquery checking reachability over G_c (referred to as a G_c -checking). Specifically, we will first check whether v is reachable from u under S in \mathbf{T} . If it is the case, then return *true*. Otherwise, we will continue to check whether v is reachable from u under S in G_c . When doing this, G_c itself will be further decomposed, leading to an elegant recursive strategy. As an example, consider the DAG shown in Figure 2(a), which is

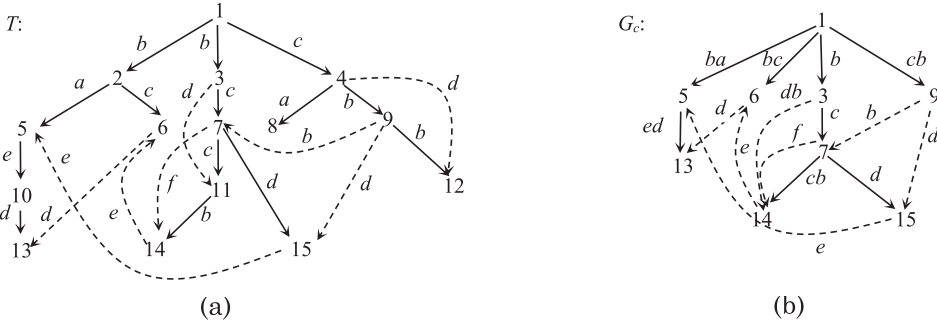


Fig. 2. A spanning tree and the corresponding G_c .

obtained by eliminating all the back edges from the graph shown in Figure 1. It can be decomposed into a spanning tree T shown by the solid edges in Figure 2(a) plus the relevant forward edges ($3 \xrightarrow{d} 11$, $7 \xrightarrow{f} 14$, $4 \xrightarrow{d} 12$), and a G_c shown in Figure 2(b). Here, we notice that G_c is not a proper subgraph of G , but with some edges changed to transfer information of reachability. However, we will use the word “decomposition” to refer to the transformation of G into T and G_c without causing confusion.

Assume that we want to know whether vertex 10 is reachable from vertex 4 under $\{b, d, e\}$. First, we will check whether 10 is reachable from 4 under $\{b, d, e\}$ in T . Since it is not the case, we need to check G_c . For this, we will find vertex 9 and vertex 5 in G_c and check their reachability. It is because

if vertex 4 reaches any vertex in G through a path going through some cross edges, it must go through vertex 9; and if any vertex in G reaches vertex 10 through a path going through some cross edges, it must go through vertex 5.

Thus, the following three-step checking will be carried out:

- (i) $4 \sim 9$ under $\{b, d, e\}$ in T ?
- (ii) $9 \sim 5$ under $\{b, d, e\}$ in G_c ?
- (iii) $5 \sim 10$ under $\{b, d, e\}$ in T ?

Each of the above checks returns *true*. So we know that vertex 10 is reachable from vertex 4 under $\{b, d, e\}$.

The motivation to decompose a graph in such a way is that the transitive closure of T can be very effectively compressed while the corresponding queries can be very efficiently evaluated.

In the following, we will first discuss the T -checking, and then how G_c can be constructed, as well as how the G_c -checking can be recursively conducted. For simplicity, here we will not discuss how to select spanning trees to increase the number of forward edges (and then decrease the number of cross edges), which will eventually lead to fewer recursive graph decompositions. We shift this part of discussion to Section 7.1.

5.2 T -checking

For doing a T -checking efficiently, a kind of tree labeling needs to be designed.

5.2.1 Tree Labeling. Let p be a Path in T . We will use $A(p)$ to represent a multi-set of the form $\{a_1^{j_1}, \dots, a_k^{j_k}\}$, or simply $a_1^{j_1} \dots a_k^{j_k}$ with each $j_i > 0$ ($i = 1, \dots, k$), containing all the edge labels on p , where each a_i ($1 \leq i \leq k$) is a label appearing on p and j_i is the number of edges on p , which are labeled with a_i . For example, for $p = 1 \rightarrow 3 \rightarrow 7 \rightarrow 11 \rightarrow 14$ in Figure 1, we have $A(p) = b^2 c^2$.

1: [0, -, 15]	9: [13, {cb}, 15]
2: [1, {b}, 6]	10: [3, {bae}, 5]
3: [6, {b}, 11]	11: [8, {bc ² , bd}, 11]
4: [11, {c}, 15]	12: [14, {cb ² , cd}, 15]
5: [2, {ba}, 5]	13: [4, {baed}, 5]
6: [5, {bc}, 6]	14: [9, {b ² d, bcf, b ² c ² }, 10]
7: [7, {bc}, 11]	15: [10, {bcd}, 11]
8: [12, {ca}, 13]	

Fig. 3. Tree encoding for the vertices in T shown in Fig. 2(a).

First, for computing the labels over a subpath, we need the following concept:

Definition 5.1 (Difference of Multi-Sets). Let B and C be two multi-sets. The difference between B and C , denoted as $B - C$, is also a multi-set of the form $a_1^{j_1} \dots a_k^{j_k}$ with each $j_i > 0$ such that for each $i \in \{1, \dots, k\}$ $a_i \in B$, but $a_i \notin C$; or there exist integers x, y such that $a_i^x \in B$, $a_i^y \in C$, and $x - y = j_i$.

Let p' be a subpath of p . Denote by $p \setminus p'$ the remaining part of p after p' is cut off from it. We can use $A(p) - A(p')$ to represent the remaining multi-set of p after p' is discarded. For instance, for a subpath p' (of p) = $1 \rightarrow 3 \rightarrow 7$, we have $A(p') = bc$. Then, $A(p) - A(p') = b^2c^2 - bc = b^{(2-1)}c^{(2-1)} = bc$, which is all the edge labels over $p'' = p \setminus p' = 7 \rightarrow 11 \rightarrow 14$.

Obviously, if $A(p) = a_1^{j_1} \dots a_k^{j_k}$ with each $j_i > 0$, then we must have $L(p) = \{a_1, \dots, a_k\}$. Different from $L(p)$, $A(p)$ is referred to as a *multi-label* of p . Without causing confusion, we will interchangeably use $A(p) \subseteq S$ or $L(p) \subseteq S$ to represent the containment of all the labels on p in S .

Now, consider $LCR(u, v, S, G)$. By the T -checking, we will evaluate $LCR(u, v, S, T)$. For this purpose, we will associate each vertex v in T with a triplet: $[\alpha, \gamma, \beta]$, where α is v 's preorder number, which is created when searching T in preorder; $\beta - 1$ is equal to the largest preorder number among all the vertices in $T[v]$, and γ is a set containing the multi-labels of all the root-to- v paths in T . For example, with respect to the spanning tree shown by the solid arrows in Figure 2(a), vertex 6 is labeled with $[5, \{bc\}, 6]$, and vertex 12 is labeled with $[14, \{cb^2, cd\}, 15]$ (see Figure 3). In the same way, we can check all the other triples in Figure 3. In addition, we call $[\alpha, \beta]$ an interval of v .

Let u, v be two vertices on a tree path in T and u is above v (i.e., u is an ancestor of v with respect to T). We will use p_u to stand for the tree path from the root to u and p_{uv} for the tree path from u to v . Obviously, $A(p_{uv}) = A(p_v) - A(p_u)$. If there exists a forward edge $e = s \rightarrow t$ attached to p_{uv} , then we say, p_{st} (also, any of its subpaths) is covered by e . Replacing p_{st} with e , we will get another path whose multi-label can be obtained by replacing the multi-label of p_{st} by the label on e .

Using γ_{uv} to represent all those multi-labels with each representing a path label from u to v , we have the following lemma:

LEMMA 5.1. *Let u and v be two vertices in T associated with $[\alpha_u, \gamma_u, \beta_u]$ and $[\alpha_v, \gamma_v, \beta_v]$, respectively, and S be a label set. Vertex v is reachable from vertex u under S if the following two conditions are satisfied:*

1. $[\alpha_u, \beta_u] \supseteq [\alpha_v, \beta_v]$, and
2. there exists a multi-label S' in γ_{uv} such that $S' \subseteq S$.

PROOF. If $[\alpha_u, \beta_u] \supseteq [\alpha_v, \beta_v]$, then there must be a path from u to v in T . So, v is reachable from u in T . If $S' \subseteq S$, then there must be a path p from u to v such that $A(p) = S' \subseteq S$, where p is either the tree path from u to v , or a path formed by replacing some edges on the tree path from u to v with the corresponding covering forward edges. \square

For example, to check whether 14 is reachable from 3 in T (i.e., T plus all the forward edges shown in Figure 2(a)) under $\{d, b, e\}$, we will first check whether $[\alpha_3, \beta_3] = [6, 11] \supseteq [\alpha_{14}, \beta_{14}] = [9, 10]$. Since it is the case, we will further check whether there is a multi-label in $\gamma_{3,14} = \{bd, cf, bc^2\}$, which is a subset of $\{d, b, c\}$. Since it is also *true*, the answer to the query is *yes*.

However, the reverse of Lemma 5.1 is not always *true*. That is, Lemma 5.1 is only a sufficient condition, not necessary for reachability, since we may have reachability through cross edges. Therefore, in the case that a T -checking returns *false*, the corresponding G_c -checking should be conducted. For instance, even though $[\alpha_4, \beta_4] = [11, 15] \not\supseteq [\alpha_{14}, \beta_{14}] = [9, 10]$, 14 is reachable from 4, but through some cross edges in G_c .

According to Lemma 5.1, we give the following algorithm T -checking() to do a T -checking:

ALGORITHM 1: T -checking(u, v, S, T)

begin

1. **if** $u = v$ **then** return *true*;
2. **if** $[\alpha_v, \beta_v] \not\supseteq [\alpha_u, \beta_u]$ **then** return *false*;
3. check all multi-labels from u to v by using γ_{uv} ;

end

However, in the above algorithm, line 3 should be further specified on how to compute γ_{uv} , as well as how to check multi-labels in γ_{uv} . In the following, we will present two methods to do this task.

5.2.2 T -search Based Method. By the first method, we view T itself as a storage of all γ 's. To evaluate $LCR(u, v, S, T)$, we will explore T to find a path p , which is made up of some edges on p_{uv} and some forward edges attached to p_{uv} , such that $A(p) \subseteq S$.

Specifically, this can be done as follows.

- (1) Search T starting from u in the depth-first manner.
- (2) For each encountered edge $u' \xrightarrow{l} v'$ (a tree edge or a forward edge), we will check whether $[\alpha_{v'}, \beta_{v'}] \supseteq [\alpha_v, \beta_v]$. If it is not the case, then the containment of l in S will not be checked and the subgraph rooted at v' will not be further explored. Otherwise, we distinguish between two cases:
 - (i) $l \in S$. In this case, if $v' = v$, then return *true*; otherwise, continue to explore the subgraph rooted at v' in T .
 - (ii) $l \notin S$. We will check another edge going out from u' , which has not yet been visited. If such an edge does not exist, then backtrack to explore the edges leaving the vertex from which u' was discovered.
- (3) This process continues until we find a path satisfying the condition, or all the edges going out of u have been visited. In the former case, return *true*. In the latter case, return *false*.

This is a very simple process, but needs to search part of T (concretely, a tree path from u to v in T and some forward edges attached to it). For very large graphs, it can be time-consuming. To mitigate this problem to some extent, we organize all the forward edges into a graph, called a *compatible graph*, and replace the search of T with the search of such a graph, which will be discussed in the next subsection in great detail.

5.2.3 Compatible Graph-based Method. Let v be a vertex in T . Let $e_i = s_i \xrightarrow{x} t_i$ ($i = 1, \dots, l$ for some l) be all the forward edges attached to p_v . We will use a compact data structure (accompanied with a simple procedure to do replacements of subpaths by forward

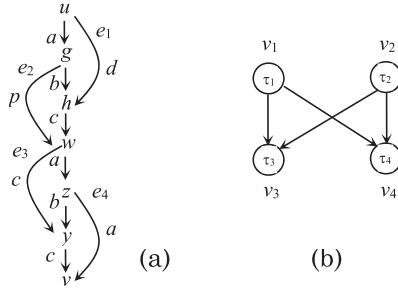


Fig. 4. Illustration for compatible graphs.

edges) to represent γ_v in $[\alpha_v, \gamma_v, \beta_v]$:

$$\langle A(p_v); \tau_1, \dots, \tau_l; \lambda_v \rangle,$$

where each τ_i ($i \in \{1, \dots, l\}$) is a quadruple of the form $[s_i, t_i, A(p_{s_i t_i}), x]$ corresponding to a forward edge e_i and λ_v is a multi-label (appearing on p_v) such that each label in it is not covered by any forward edge attached to p_v , and therefore cannot be “replaced away” by using forward edges. That is, λ_v must appear on any path from the root to v in T .

For example, in the spanning tree T shown in Figure 2(a), we have $p_{12} = 1 \xrightarrow{c} 4 \xrightarrow{b} 9 \xrightarrow{b} 12$, $A(p_{12}) = b^2c$, and then $\gamma_{12} = \langle b^2c; [4, 12, b^2, d]; \{c\} \rangle$. Here, we note that $\lambda_{12} = \{c\}$ contains a label not appearing on the segment from 4 to 12 in T , which is covered by the unique forward edge $4 \xrightarrow{d} 12$ attached to p_{12} .

Using this data structure, any path label in T can be dynamically produced.

To show how this works, we need to define another two concepts.

Definition 5.2 (Replacement). Let v be a vertex in T , and $s \xrightarrow{x} t$ be a forward edge attached to p_v . A replacement of p_{st} with $\tau = [s, t, A(p_{st}), x]$ on p_v , denoted as $p_v \circ \tau$, is a multi-set, equal to $(A(p_v) - A(p_{st})) \cup \{x\}$.

For example, for p_{12} and $\tau = [4, 12, b^2, d]$ shown above, we have $p_{12} \circ \tau = cd$. This is $A(p)$ for another path p from vertex 1 to 12.

Definition 5.3 (Compatibility of τ 's). Let $\tau_1 = [s, t, A(p_{st}), x_1]$, and $\tau_2 = [s', t', A(p_{s't'}), x_2]$ be two quadruples in a γ_v . We say τ_1 and τ_2 are compatible if p_{st} and $p_{s't'}$ are not edge-overlapped.

For illustration, consider the tree path shown in Figure 4(a) and the four attached forward edges e_1, e_2, e_3 , and e_4 . Denote by τ_i the quadruple built for e_i ($i = 1, \dots, 4$). Then, τ_1 and τ_3 are compatible while τ_1 and τ_2 not. So, for a set of compatible quadruples τ_1, \dots, τ_j corresponding to j forward edges attached to p_v , $p_v \circ \tau_1 \circ \dots \circ \tau_j$ must be equal to $A(p)$ for some p in T .

Assume that for a given query $LCR(u, v, S, T)$ we have $[\alpha_u, \beta_u] \supseteq [\alpha_v, \beta_v]$, but $L(p_{uv}) \not\subseteq S$. In this case, we will first check $\lambda_{uv} = \lambda_v - \lambda_u$. If λ_{uv} contains any label $\notin S$, then return *false*, since such a label definitely appears on any path from u to v and thus the query cannot be satisfied. To see this, consider $LCR(1, 12, \{d, f\}, T)$ with T being a tree shown by the solid edges in Figure 2(a) plus all the forward edges. Obviously, we have $[\alpha_1, \beta_1] = [0, 15] \supseteq [\alpha_{12}, \beta_{12}] = [14, 15]$. But $L(p_{1,12}) = \{b, c\} \not\subseteq S = \{d, f\}$. We will then check $\lambda_{1,12} = \lambda_{12} - \lambda_1 = \{c\} - \emptyset = \{c\}$. Since $c \notin S = \{d, f\}$, return *false*. It is because $1 \xrightarrow{c} 4$ is not covered by any forward edge attached to $p_{1,12}$ and therefore cannot be replaced, which implies that a path label (from 1 to 12) coverable by S can never be found. In the opposite, however, for the path shown in Figure 4(a), $\lambda_v = \emptyset \subseteq$ any set S , and hence

the corresponding replacements should be always searched (since a replacement may lead to a satisfying answer).

Let $\gamma_u = \langle A(p_u); \tau_1, \dots, \tau_k; \lambda_u \rangle$ and $\gamma_v = \langle A(p_v); \tau_1', \dots, \tau_l'; \lambda_v \rangle$. The general working process begins to calculate $A_{uv} = A(p_v) - A(p_u)$ first. If $A_{uv} \not\subseteq S$, then we will check whether $\lambda_{uv} = \lambda_v - \lambda_u \subseteq S$. If it is not the case, then return *false*. Otherwise, we will try to find a set of compatible quadruples: $\tau_{i_1}, \dots, \tau_{i_f} (0 \leq f \leq l)$ ($0 \leq f \leq l$) attached to p_{uv} such that $p_{uv} \circ \tau_{i_1} \circ \dots \circ \tau_{i_f} \subseteq S$. This process can be expedited by organizing all τ_i 's in γ_v into a graph, called a *compatible graph*, as defined below.

First, we use $\tau.s, \tau.t, \tau.A, \tau.x$ to refer to the four elements in τ , respectively.

Definition 5.4 (Compatible Graphs). A compatible graph C_v for $\gamma_v = \langle A(p_v); \tau_1, \dots, \tau_l; \lambda_v \rangle$ is a graph, in which each vertex represents a τ_i in γ_v . There is an edge $\tau_i \rightarrow \tau_j$ if (1) τ_i and τ_j are compatible, (2) $\tau_i.s$ is an ancestor of $\tau_j.s$, (3) between τ_i and τ_j is there no other τ , which is compatible to both.

According to this definition, for any edge $\tau \rightarrow \tau'$ in a compatible graph, we definitely have no path of length at least two (edges) from τ to τ' . Otherwise, condition (3) is violated.

As an example, see the path (segment) p_{uv} shown in Figure 4(a) and all the attached forward edges, for which we will construct a compatible graph as shown in Figure 4(b). In this graph, each τ_i corresponds to a forward edge e_i in Figure 4(a). In terms of the above description, we can design an algorithm to explore C_v to find a set: τ_1, \dots, τ_j (for some j) along a path such that $p_{uv} \circ \tau_1 \circ \dots \circ \tau_j \subseteq S$. For instance, to evaluate $LCR(u, v, \{a, c, d\}, T)$, where T is shown in Figure 4(a), we need to explore a path from v_1 to v_4 in Figure 4(b) to find τ_1 and τ_4 such that $p_{uv} \circ \tau_1 \circ \tau_4 = a^2 c^2 b^2 \circ [u, h, ab, d] \circ [z, v, bc, a] = a^2 cd \subseteq S = \{a, c, d\}$.

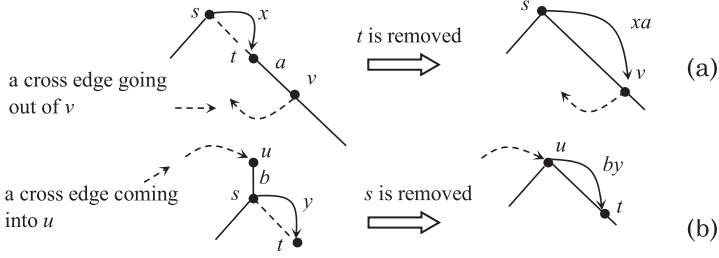
In Section 7.2, we will discuss in great detail how to explore C_v efficiently to find a replacement. Also, how to create a general compatible graph, denoted C_T , for all forward edges in T , instead of separated C_v 's.

5.3 G_c -checking

In this subsection, we discuss the G_c -checking, which will be invoked if the corresponding T -checking fails. As will be seen later, it is much more complicated than the T -checking, but can be done very efficiently. First, we show how G_c is constructed. Then, how a G_c -checking is made to complete the evaluation of an LCR query.

5.3.1 Subgraph G_c . It is a difficult task to construct G_c efficiently, since we have to figure out what edges should be added to G_c to transfer reachability. For this purpose, a subtle classification of vertices in T needs to be carefully conducted:

- $V_{c-start}$ - all the start vertices of cross edges.
- V_{c-end} - all the end vertices of cross edges.
- $V_{f-start}$ - all the start vertices of forward edges.
- V_{f-end} - all the end vertices of forward edges.
- V_{fs} - all those start vertices s of forward edges $s \rightarrow t$, where either $t \in V_{c-start}$ or t is an ancestor of some vertex in $V_{c-start}$ with respect to T . (See the left part of Figure 5(a) for illustration.)
- V_{fe} - all those end vertices t of forward edges $s \rightarrow t$, where either $s \in V_{c-end}$ or s is a descendant of some vertex in V_{c-end} with respect to T . (See the left part of Figure 5(b) for illustration.)
- V_{LCA} - all those vertices with each being a **lowest common ancestor (LCA)** of more than one vertex in $V_{c-start} \cup V_{fs}$, which are not related by the ancestor/descendant relationship in T .

Fig. 5. Illustration for V_{fs} and V_{fe} .

In the above classification of vertices, the first four classes are quite straightforward. To see what is V_{fs} , let us have a look at Figure 5(a) (left part), in which we can see a forward edge $s \rightarrow t$ (labeled with x) with t being an ancestor of a vertex $v \in V_{c-start}$. Then, s is a vertex belonging to V_{fs} .

Similarly, to see what is V_{fe} , let us see Figure 5(b) (left part). Here, we can find another forward edge $s \rightarrow t$ (labeled with y) with s being a descendant of a vertex $u \in V_{c-end}$. Then, t is a vertex belonging to V_{fe} .

The reason for recognizing these vertices is that both kinds of forward edges need to be kept in G_c to check reachability through cross edges, but with edge labels somehow changed (to be discussed below).

Finally, the understanding of V_{LCA} is also straightforward. As an example, let us have a close look at Figure 2(a) again, for which we have

$$\begin{aligned}
 V_{c-start} &= \{6, 14, 15, 9\} \\
 V_{c-end} &= \{5, 13, 6, 7, 15\} \\
 V_{f-start} &= \{3, 7, 4\} \\
 V_{f-end} &= \{11, 12, 14\} \\
 V_{fs} &= \{3, 7\} \\
 V_{fe} &= \{14\}, \text{ and} \\
 V_{LCA} &= \{1, 7\}
 \end{aligned}$$

Especially, we notice that vertex 1 is the LCA of $\{6, 3, 9\} \subset V_{c-start} \cup V_{fs}$ while vertex 7 is the LCA of $\{14, 15\} \subset V_{c-start}$. We need to recognize this sort of vertices, since they can be used as “connecting” points to transfer information on reachability through cross edges.

By using a linear time algorithm for finding all LCA s, we can recognize all these subsets in $O(m)$ time. We will discuss this algorithm in great detail in Section 7.3.

Now, we begin to discuss the construction of G_c . First, let us denote $V_c = V_{LCA} \cup V_{c-start} \cup V_{c-end} \cup V_{fs} \cup V_{fe}$. We define a tree (forest) structure T_c (as part of G_c), called a *skeleton tree* of G (with respect to T), which contains all the vertices in V_c for the following reasons:

- $V_{c-start}$ and V_{c-end} are included to keep information on reachability through cross edges;
- V_{fs} and V_{fe} are included to keep information on reachability through forward edges; and
- V_{LCA} is included as “connecting” points between T and G_c .

In T_c , there is an edge from u to v if and only if there is a path p from u to v in T and p contains no other vertices in these subsets except u and v themselves. In Figure 6, we show a T_c built for the graph shown in Figure 2(a). So, an edge $u \rightarrow v$ in T_c may correspond to a path in T , labeled with $L(p_{uv})$. (See edge $1 \xrightarrow{ba} 5$ in Figure 6 for illustration.)

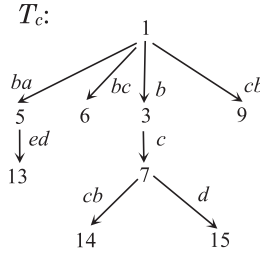


Fig. 6. A skeleton tree.

To construct G_c (see Figure 2(b) for illustration), we still need another two concepts to recognize all those forward edges that have to be kept in G_c . In addition, they also play an important role for transferring reachability from T to G_c .

Let v be a vertex in T . We denote by $D(v)$ all those vertices in $T[v]$, which also appear in $V_{LCA} \cup V_{c-start} \cup V_{fs}$. We consider a vertex $v' \in D(v)$, which has no ancestors in $D(v)$. Then, any other vertex in $D(v)$ must be a descendant of v' . Otherwise, assume that $u \in D(v)$ is not a descendant of v' . Then, the LCA of v' and u should be an ancestor of v' and this is also in $D(v)$. It is a contradiction. This observation motivates the following concept:

Definition 5.5 (Dominant Vertex). Let v be a vertex in T . Let $V' = V_{LCA} \cup V_{c-start} \cup V_{fs}$. A vertex $u \in V'$ is called a dominant vertex of v , denoted as v^{\rightarrow} , if one of the following two conditions is satisfied:

- u is a vertex closest to v in $T[v]$ among all vertices in V' if $v \notin V'$, or
- u is v itself if $v \in V'$.

If such a vertex does not exist, then v^{\rightarrow} is set to be the special symbol \perp .

Definition 5.6 (Transferring Vertex). Let v be a vertex in T . Let $V'' = V_{c-end} \cup V_{fe}$. A vertex $u \in V''$ is called a transferring vertex of v , denoted as v^{\leftarrow} , if one of the following two conditions is satisfied:

- u is the *lowest ancestor* of v in T among all vertices in V'' if $v \notin V''$, or
- u is v itself if $v \in V''$.

If such a vertex does not exist, then v^{\leftarrow} is set to be \perp .

As an example, consider the graph shown in Figure 2(a) again, in which 11^{\rightarrow} is vertex 14, since 14 is a vertex closest to 11 in $T[11]$ among all the vertices in $V_{LCA} \cup V_{c-start} \cup V_{fs}$. In addition, $11^{\leftarrow} = 7$, since 7 is the lowest ancestor of 11 in T among all vertices in $V_{c-end} \cup V_{fe}$. In a similar way, we find that $7^{\rightarrow} = 7^{\leftarrow} = 7$. The motivation of dominant vertices is that v^{\rightarrow} dominates all those vertices in $V_{c-start} \cup V_{fs}$, which appear in $T[v]$. That is, such vertices must also appear in $T[v^{\rightarrow}]$. Thus, if any vertex is reachable from v through a cross edge, then it must be through v^{\rightarrow} . In the opposite, if v is reachable from a certain vertex through a cross edge, then it must be through v^{\leftarrow} . So, it is referred to as a transferring vertex.

In general, any forward edge $s \xrightarrow{x} t$ in T satisfying one of the following two conditions will be kept or replaced with a new edge while all the other forward edges will be simply removed.

- (i) $t \in V_{c-start}$ or is an ancestor of some vertex in $V_{c-start}$, or
- (ii) $t \in V_{fe}$.

If (i) is satisfied, then we distinguish between two cases: if $t \in V_c$, then $s \xrightarrow{x} t$ will be simply kept; if $t \notin V_c$, then $s \xrightarrow{x} t$ will be replaced with a new edge $s \rightarrow t'$, labeled with $xL(p_{tt'})$, where t' is the dominant vertex of t . For instance, for forward edge $3 \xrightarrow{d} 11$ in T shown in Figure 2(a) a new forward edge $3 \xrightarrow{db} 14$ will be generated as shown in Figure 2(b).

If (ii) is satisfied, then we also distinguish between two cases: if $s \in V_c$, then $s \xrightarrow{x} t$ is kept; if $s \notin V_c$, then $s \xrightarrow{x} t$ will be replaced with $s' \rightarrow t$, labeled with $L(p_{s's})x$, where s' is the transferring vertex of s . In Figure 2(b), the new forward edge $7 \xrightarrow{f} 14$ is the same as the original one in T shown in Figure 2(a), since vertex 7 itself is in $V_{c-end} = \{5, 13, 6, 7, 15\}$.

The forward edge $4 \xrightarrow{d} 12$ will be simply eliminated, since 12 is neither a vertex in $V_{c-start}$, nor an ancestor of some vertices in $V_{c-start}$, and also $12 \notin V_{fe} = \{14\}$, i.e., either of the above two conditions is not met.

Denote by E_{f-c} the set of all such new forward edges. G_c is constructed as

$$G_c = T_c \dot{E} \cup E_{cross} \cup E_{f-c}. \quad (1)$$

So, the graph given in Figure 2(b) is the G_c with respect to G and T shown in Figure 2(a), whose size is significantly reduced.

5.3.2 G_c -checking by Using “Connecting” Vertices. Having specified the construction of G_c , we are now ready to discuss how a G_c -checking can be carried out.

The following lemma is critical to this task:

LEMMA 5.2. *Assume that vertex u is not an ancestor of vertex v in T , but v is reachable from u via some cross edges in G . Then, any way v is reached from u must be through u^{\rightarrow} and v^{\leftarrow} .*

PROOF. According to Definition 5.5, u^{\rightarrow} is closest to u in $T[u]$ among all vertices in $V' = V_{LCA} \cup V_{c-start} \cup V_{fs}$. According to Definition 5.6, v^{\leftarrow} is the lowest ancestor of v in T among all vertices in $V'' = V_{c-end} \cup V_{fe}$. It indicates that any path from u to v through some cross edges must go through u^{\rightarrow} and v^{\leftarrow} .

According to the above discussion, we give the following algorithm to do the G_c -checking:

ALGORITHM 2: G_c -checking(u, v, S, G_c)

begin

1. **if** T -checking(u, u^{\rightarrow}, S, T) **then**
2. **{if** T -checking(v^{\leftarrow}, v, S, T) **then**
3. **{if** $LCR(u^{\rightarrow}, v^{\leftarrow}, S, G_c)$ **then** return *true*;**}**
4. return *false*;

end

The above algorithm is a three-step computation. In step (1), we first check whether u^{\rightarrow} is reachable from u under S in T . If not successful, then return *false*. Otherwise, we go to step (2), in which we will check whether v is reachable from v^{\leftarrow} under S in T . Again, depending on whether it fails or not, return *false* or go to step (3). In step (3), we will check whether v^{\leftarrow} is reachable from u^{\rightarrow} under S in G_c .

Now, to evaluate $LCR(u, v, S, G)$, we need to associate each $v \in G$ with a tuple $\langle x, y, z \rangle$:

- $x = [\alpha, \gamma, \beta]$, a triplet created by labeling the vertices in T (see Section 5.2);
- $y = v^{\rightarrow}$; and
- $z = v^{\leftarrow}$.

In $\langle x, y, z \rangle$, x is used for T -checking while y and z are for G_c -checking.

The following proposition is easy to prove:

PROPOSITION 5.1. *Let u and v be two vertices in G , labeled $([\alpha_u, \gamma_u, \beta_u], y_u, z_u)$ and $([\alpha_v, \gamma_v, \beta_v], y_v, z_v)$, respectively. Vertex v is reachable from u under a label set S if one of the following conditions holds:*

- (1) $[\alpha_u, \beta_u] \supseteq [\alpha_v, \beta_v]$ and there exists a multi-label S' in γ_{uv} such that $S' \subseteq S$, or
- (2) v is reachable from z_v under S in \mathbf{T} , z_v is reachable from y_u under S in G_c , and y_u is reachable from u under S in \mathbf{T} .

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

- (1). According to Lemma 5.1, v is reachable from u under S in \mathbf{T} if (i) holds.
- (2). According to Lemma 5.2, v is reachable from u under S in G_c if (ii) holds. □

By using the \mathbf{T} -checking and G_c -checking, the general process to evaluate $LCR(u, v, S, G)$ can be easily described as below, in which we use $G = \mathbf{T} \oplus G_c$ to represent the decomposition of G described above.

ALGORITHM 3: $LCR(u, v, S, G)$

begin

1. let $G = \mathbf{T} \oplus G_c$;
2. **if** \mathbf{T} -checking(u, v, S, \mathbf{T}) **then** return *true*
3. **else** return G_c -checking(u, v, S, G_c);

end

In the above algorithm, we first make a \mathbf{T} -checking to see whether v is reachable from u under S in \mathbf{T} (see line 2). If it is the case, then the task is done. Otherwise, we have to make a G_c -checking. Especially, in the third step the G_c -checking has a recursive call to algorithm 2 that in turn calls Algorithm 3 again, so we have mutual recursion here. This implies a recursive graph decomposition to divide G into a series of spanning trees.

5.4 Recursive DAG Decomposition

From the above discussion, we can see that G_c itself can be further decomposed, leading to a recursive decomposition of G . The only difference is that in G_c an edge may correspond to a path in \mathbf{T} and therefore labeled with the corresponding path label. So, the reachability checking over G_c can be done in the same way as described in Section 5.2 and Section 5.3.

Let G_0 be a DAG. We will use $T_0, E_{cross}^0, E_{forward}^0$ to represent one of its spanning trees, the corresponding set of cross edges and forward edges, respectively. Then, we have

$$\begin{aligned} T_0 &= T_0 \cup E_{forward}^0 \\ G_c^0 &= T_c^0 \cup E_{cross}^0 \cup E_{f-c}^0 \end{aligned} \quad (2)$$

where T_c^0 is the skeleton tree for G_0 , and E_{f-c}^0 the corresponding set of new forward edges. Denote $G_{i+1} = G_c^i$ for $i \geq 0$. The recursive decomposition of G_0 can be represented by the following equations:

$$\begin{cases} T_i = T_i \cup E_{forward}^i \\ G_{i+1} = T_c^i \cup E_{cross}^i \cup E_{f-c}^i \end{cases} \quad (3)$$

where T_c^i is the skeleton tree for G_i for $i = 0, 1, \dots, k-1$ for some k . Note that each T_i ($i \in \{0, \dots, k-1\}$) is a tree-like graph.

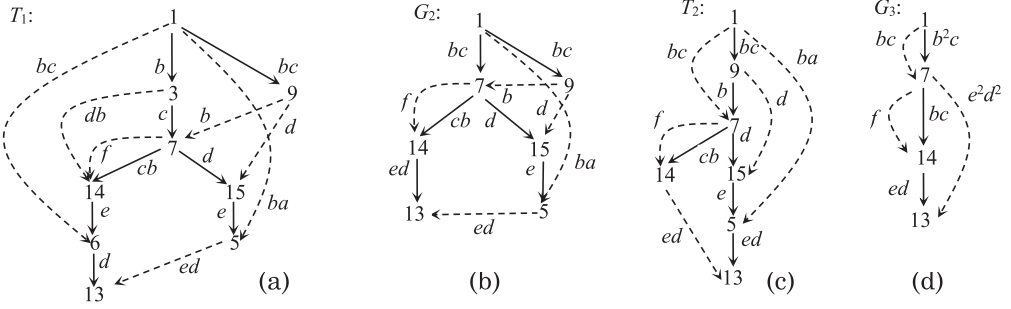


Fig. 7. Recursive DAG decomposition.

The following example helps for illustration:

Example 5.1. Denote by G_0 the graph shown in Figure 2. Denote by T_0 the spanning tree represented by the solid edges in the graph. With respect to T_0 (in Figure 2), both E_{cross}^0 and $E_{forward}^0$ are shown by the dashed edges in the same figure. T_c^0 is shown in Figure 6. Then, $G_1 = T_c^0 \cup E_{cross}^0 \cup E_{f-c}^0$ is a graph as shown in Figure 2(b).

A spanning tree T_1 of G_1 is shown by the solid edges in Figure 7(a). With respect to T_1 , we have $V_{c-start}^1 = \{5, 9\}$, $V_{c-end}^1 = \{7, 13, 15\}$, $V_{f-start}^1 = \{1, 3, 7\}$, $V_{f-end}^1 = \{5, 6, 14\}$, $V_{fs}^1 = \{1\}$, $V_{fe}^1 = \{14\}$, and $V_{LCA}^1 = \{1\}$.

Thus, T_c^1 is constructed as shown by the solid edges in Figure 7(b). Adding $E_{cross}^1 = \{9 \xrightarrow{b} 7, 9 \xrightarrow{b} 15, 5 \xrightarrow{ed} 13\}$ and $E_{f-c}^1 = \{1 \xrightarrow{ba} 5, 7 \xrightarrow{f} 14\}$ to T_c^1 , we get $G_2 = T_c^1 \cup E_{cross}^1 \cup E_{f-c}^1$. Note that forward edge $1 \xrightarrow{bc} 6$ is simply removed, since 6 is neither in $V_{c-start}^1$ nor an ancestor of any vertex in $V_{c-start}^1$, and also $6 \notin V_{fe}^1 = \{14\}$, showing that it will not be involved in reachability through cross edges and thus deleted from G_2 . (See the conditions for keeping forward edges given in Subsection 5.3.1.) For the same reason, $3 \xrightarrow{db} 14$ is also removed.

One of the spanning trees of G_2 is shown by the solid edges in Figure 7(c), denoted T_2 . With respect to T_2 , we have $V_{c-start}^2 = \{14\}$, $V_{c-end}^2 = \{13\}$, $V_{f-start}^2 = \{1, 7, 9\}$, $V_{f-end}^2 = \{5, 7, 14, 15\}$, $V_{fs}^2 = \{1, 7\}$, $V_{fe}^2 = \emptyset$, and $V_{LCA}^2 = \{1\}$. Then, we are able to construct T_c^2 . Recognizing E_{cross}^2 and E_{f-c}^2 as described above, we can build G_3 as shown in Figure 7(d). (Note that forward edge $9 \xrightarrow{d} 15$ and $1 \xrightarrow{ba} 5$ in Figure 7(c) are removed for the same reason as $1 \xrightarrow{bc} 6$ was taken away from G_2 .)

Assume that the spanning tree found for G_3 is a tree shown by the solid arrows in Figure 7(d), with respect to which we have no cross edges. Then, when constructing T_c^3 , all the forward edges can be simply ignored and the graph will not be further decomposed any more.

From the above discussion, each vertex v in G_i ($0 \leq i \leq k-1$) will be associated with a triplet $[\alpha_v^i, \gamma_v^i, \beta_v^i]$ and a pair of “connecting” vertices $(v_i^{\rightarrow}, v_i^{\leftarrow})$. Therefore, as a whole data structure, each vertex v in G will be associated with two sequences:

- $\omega_v = [\alpha_v^0, \gamma_v^0, \beta_v^0] \dots, [\alpha_v^{k-1}, \gamma_v^{k-1}, \beta_v^{k-1}]$, and
- $\omega_v = (v_0^{\rightarrow}, v_0^{\leftarrow}), \dots, (v_{(k-1)}^{\rightarrow}, v_{(k-1)}^{\leftarrow})$

where v_i^{\rightarrow} ($i \geq 0$) stands for its dominant in T_i while v_i^{\leftarrow} for its transferring vertex in T_i .

Table 2. \mathfrak{w}_v -sequences

v	T_0	T_1	T_2	T_3
1	[0, -, 15]	[0, -, 9]	[0, -, 7]	[0, -, 4]
2	[1, {b}, 6]			
3	[6, {b}, 11]	[1, {b}, 8]		
4	[11, {c}, 15]			
5	[2, {bd}, 5]	[7, {bc ² d, ba}, 8]	[5, {c ² b ² d, bc ² d, ba}, 7]	
6	[5, {be}, 6]	[4, {b ² c ² e, bc, b ² d ² e, bcef}, 6]		
7	[7, {bc}, 11]	[2, {bc}, 8]	[2, {cb, b ² c}, 7]	[1, {cb, b ² c}, 4]
8	[12, {ca}, 13]			
9	[12, {cb}, 15]	[8, {cb}, 9]	[1, {cb}, 7]	
10	[3, {bae}, 5]			
11	[8, {bc ² , bd}, 11]			
12	[14, {cb ² , cd}, 15]			
13	[4, {baed}, 5]	[5, {b ² c ² de, bcd, b ² d ² e, bcd}, 6]	[6, {c ² b ² d ² e, bc ² d ² e, bade}, 7]	[3, {b ³ c ² de, b ² cfde, b ² c ² d ² , b ² c ² de, bcfed, bce ² d ² }, 4]
14	[9, {b ² d, bcf, b ² c ² }, 10]	[3, {b ² c ² , b ² d, bcf}, 6]	[3, {c ² b ³ , c ² b ² , cb ² f, bcf}, 4]	[2, {c ² b ³ , c ² b ² , cb ² f, cbf}, 4]
15	[10, {bcd}, 15]	[6, {bcd}, 8]	[4, {b ² cd, bcd}, 7]	

To evaluate a query $LCR(u, v, S, G)$, we will repeatedly perform the following steps until we find that v is reachable from u under S or the relevant data structure is used up.

Initially, $i = 0$.

- (1) Use $[\alpha_u^i, \gamma_v^i, \beta_v^i]$ and $[\alpha_u^i, \gamma_v^i, \beta_v^i]$ to check whether v is reachable from u under S in T_i . If it is the case, then return *true*. Otherwise, go to (2).
- (2) $x = u^-$. Use $[\alpha_x^i, \gamma_u^i, \beta_u^i]$ and $[\alpha_x^i, \gamma_x^i, \beta_x^i]$ to check whether x is reachable from u under S in T_i . If it is not the case, then return *false*. Otherwise, go to (3).
- (3) $y = v^-$. Use $[\alpha_y^i, \gamma_y^i, \beta_y^i]$ and $[\alpha_v^i, \gamma_v^i, \beta_v^i]$ to check whether v is reachable from y under S in T_i . If it is not the case, then return *false*. Otherwise, go to (4).
- (4) Use $[\alpha_x^i, \gamma_x^i, \beta_x^i]$ and $[\alpha_y^i, \gamma_y^i, \beta_y^i]$ to check whether y is reachable from x under S in T_{i+1} . If it is the case, then return *true*. Otherwise, $u := x, v := y, i := i + 1$. If $i < k$, then go to (1); otherwise, return *false*.

Example 5.2. Along with the graph decomposition shown in Example 5.1, two sequences for every vertex will be created as shown in Table 2 and Table 3, respectively. In Table 2, for ease of understanding, we show path labels in a naive way, which should, however, be stored by using the compatible graph-based method or the T -search-based method discussed in Section 5.2 for efficiency.

In the second column of Table 2, we show the labels for the vertices in T_0 (i.e., the spanning tree T_0 of $G_0 = G$ shown in Figure 2(a), plus the forward edges with respect to T_0). Each node in T_0 is associated with a dominant and a transferring vertex, which are shown in the second column in Table 3. These vertices are used to check reachability in $G_1 = G_c^0$, which is shown in Figure 2(b). For G_1 , we will generate T_1 . It is T_1 , plus the relevant forward edges shown in Figure 7(a). The labels of the vertices in T_1 are shown in the third column of Table 2. Their dominant and transferring

Table 3. ω_v -sequences

v	T_0	T_1	T_2
1	$\langle 1, \perp \rangle$	$\langle 1, \perp \rangle$	$\langle 1, \perp \rangle$
2	$\langle 6, \perp \rangle$		
3	$\langle 3, \perp \rangle$	$\langle 5, \perp \rangle$	
4	$\langle 9, \perp \rangle$		
5	$\langle \perp, 5 \rangle$	$\langle \perp, 15 \rangle$	$\langle \perp, \perp \rangle$
6	$\langle 6, 6 \rangle$	$\langle \perp, 7 \rangle$	
7	$\langle 7, 7 \rangle$	$\langle 5, 7 \rangle$	$\langle 7, \perp \rangle$
8	$\langle \perp, \perp \rangle$		
9	$\langle 9, \perp \rangle$	$\langle 9, \perp \rangle$	$\langle 7, \perp \rangle$
10	$\langle \perp, 5 \rangle$		
11	$\langle 14, 7 \rangle$		
12	$\langle \perp, \perp \rangle$		
13	$\langle \perp, 13 \rangle$	$\langle \perp, 13 \rangle$	$\langle \perp, 13 \rangle$
14	$\langle 14, 7 \rangle$	$\langle \perp, 4 \rangle$	$\langle 14, \perp \rangle$
15	$\langle 15, 7 \rangle$	$\langle 5, 15 \rangle$	$\langle \perp, \perp \rangle$

vertices are shown in the third column of Table 3. In the same way, we can generate the remaining parts of Table 2 and Table 3. The corresponding summary graphs and spanning trees are shown in Figures 7(b), (c), and (d), respectively.

Now, we trace the evaluation of a query $LCR(4, 13, S, G)$, where G is the graph shown in Figure 2(a) and $S = \{b, d, e\}$, to demonstrate how the data structures are utilized.

- Step 1: Evaluate $LCR(4, 13, S, T_0)$ by using $[\alpha_4^0, \gamma_4^0, \beta_4^0] = [11, \{c\}, 15]$ and $[\alpha_{13}^0, \gamma_{13}^0, \beta_{13}^0] = [4, \{bd^2e\}, 5]$ (see ω_4 and ω_{13} in Table 2). It returns *false*.
- Step 2: Find $4^\rightarrow = 9$ and $13^\leftarrow = 13$ with respect to T_0 . (See ω_4 and ω_{13} in Table 3.) So, we will first evaluate $LCR(4, 4^\rightarrow, S, T_0) = LCR(4, 9, S, T_0)$ by using $[\alpha_4^0, \gamma_4^0, \beta_4^0] = [11, \{c\}, 15]$ and $[\alpha_9^0, \gamma_9^0, \beta_9^0] = [12, cb, 15]$. It returns *true*. Then, we evaluate $LCR(4^\rightarrow, 13^\leftarrow, S, T_1) = LCR(9, 13, S, T_1)$ by using $[\alpha_9^1, \gamma_9^1, \beta_9^1] = [8, \{cb\}, 9]$ and $[\alpha_{13}^1, \gamma_{13}^1, \beta_{13}^1] = [5, \{b^2c^2de, bcd, b^2d^2e, bcdef\}, 6]$. It returns *false*. (9 and 13 are not on a same tree path in T_1 .)
- Step 3: Find $9^\rightarrow = 9$ and $13^\leftarrow = 13$ with respect to T_1 . Since $9^\rightarrow = 9$ and $13^\leftarrow = 13$, we need only to evaluate $LCR(9^\rightarrow, 13^\leftarrow, S, T_2) = LCR(9, 13, S, T_2)$ by using $[\alpha_9^2, \gamma_9^2, \beta_9^2] = [1, \{cb\}, 7]$ and $[\alpha_{13}^2, \gamma_{13}^2, \beta_{13}^2] = [6, \{c^2b^2d^2e, bc^2d^2e, bade\}, 7]$. It also returns *false*. (9 and 13 are on a same tree path in T_2 ; but any path label on a path from 9 to 13 contains $c \notin S$ or $a \notin S$.)
- Step 4: Find $9^\rightarrow = 7$ and $13^\leftarrow = 13$ with respect to T_2 . We will first evaluate $LCR(9^\rightarrow, 9^\rightarrow, S, T_2) = LCR(9, 7, S, T_2)$ by using $[\alpha_9^2, \gamma_9^2, \beta_9^2] = [1, cb, 7]$ and $[\alpha_7^2, \gamma_7^2, \beta_7^2] = [2, \{cb, b^2c\}, 7]$. It returns *true*. Then, we continue to check $LCR(13^\leftarrow, 13, S, T_3) = LCR(13, 13, S, T_3)$. It trivially evaluates to *true*. Next, we check $LCR(9^\rightarrow, 13^\leftarrow, S, T_3) = LCR(7, 13, S, T_3)$ by using $[\alpha_7^3, \gamma_7^3, \beta_7^3] = [1, \{cb, cb^2\}, 4]$ and $[\alpha_{13}^3, \gamma_{13}^3, \beta_{13}^3] = [3, \{b^3c^2de, b^2c^2de, b^2c^2d^2e, b^2c^2de, bc^2d^2e, bce^2d^2\}, 4]$. It returns *true*. So, the query $LCR(9, 13, S, G)$ evaluates to *true*. (7 and 13 are on a same tree path in T_3 and there is a path from 7 to 13 with the path label $= \{e, d\} \subseteq S$.)

From the above discussion, we can see that the *to*- and *from*-sequences of v mentioned in Section 1 consists in ω_v 's while ω_v 's are mainly used to facilitate the T -checking.

Formally, the from-sequence of v is a sequence calculated when evaluating a query, as shown below:

$$v, v_0^{\rightarrow}, (v_0^{\rightarrow})_1^{\rightarrow}, \dots (\dots ((v_0^{\rightarrow})_1^{\rightarrow})) \dots)_l^{\rightarrow} \text{ for some } l \leq k-1.$$

The to-sequence of v is similar:

$$v, v_0^{\leftarrow}, (v_0^{\leftarrow})_1^{\leftarrow}, \dots (\dots ((v_0^{\leftarrow})_1^{\leftarrow})) \dots)_l^{\leftarrow}.$$

Notice that these two kinds of sequences are implicitly associated with each vertex in G and accessed when executing a query $LCR(u, v, S, G)$.

5.5 Time Complexity and Correctness

From the above discussion, the computational complexities of our method can be easily observed. Denote by m_i , χ_i , and h_i the numbers of edges in G_i , the number of forward edges with respect to T_i , and the maximum number of forward edges attached to a path in T_i , respectively. Then, we have

- time for index construction: $t_1 = O(\sum_{i=0}^{k-1} (m_i + \chi_i |\Sigma| + \chi_i h_i))$,
- space for storing index: $t_2 = O(\sum_{i=0}^{k-1} (|T_i| + \chi_i |\Sigma| + \chi_i h_i))$,
- time for evaluating a query: $t_3 = O(\sum_{i=0}^{k-1} (h_i^2 + h_i |\Sigma|))$.

To analyze t_1 , we should note that the index construction for T_i mainly consists of two parts. The first part is to search G_i to find T_i , as well as the corresponding forward and cross edges. Its cost is bounded by $O(m_i)$. The second part is the cost for constructing the compatible graphs C_v 's for every vertex v in G_i . We can organize all of them into a global graph C_i , in which each vertex is a quadruple (see Section 5.2.3) and therefore its construction requires $O(|\Sigma|)$ time. In addition, the number of edges in C_i is bounded by $O(\chi_i h_i)$, since each vertex in C_i has at most $O(h_i)$ parents. Thus, the time for constructing the index for T_i (i.e., C_i and the intervals of the vertices in T_i) is bounded by $O(|m_i| + \chi_i |\Sigma| + \chi_i h_i)$ and t_1 is $O(\sum_{i=0}^{k-1} (m_i + \chi_i |\Sigma| + \chi_i h_i))$.

Accordingly, the index size generated for G_i is bounded by $O(|T_i| + \chi_i |\Sigma| + \chi_i h_i)$, since besides the global compatible graph, $|T_i|$ intervals need to be stored. So, t_2 is $O(\sum_{i=0}^{k-1} (|T_i| + \chi_i |\Sigma| + \chi_i h_i))$. Next, we have t_3 , since for each C_i we only explore part of it, corresponding to the forward edges attached to a certain path in T_i . That means, we will only access at most h_i vertices and h_i^2 edges in C_i . So, its cost for evaluating a subquery against T_i is bounded by $O(h_i^2 + h_i |\Sigma|)$ and t_3 is $O(\sum_{i=0}^{k-1} (h_i^2 + h_i |\Sigma|))$.

However, in general, the number of edges in a C_i is considerably smaller than $O(\chi_i h_i)$. It is because in a compatible graph for each edge $\tau \rightarrow \tau'$ we have no path containing two or more edges connecting τ and τ' . According to Mehlhorn (see Reference [60], pp. 9–11), the expected outdegree of a vertex in a random graph with such a property is bounded by $O(\sqrt{h_i})$. Thus, the expected number of edges in a compatible graph is $\chi_i \sqrt{h_i}$, not $\chi_i h_i$. Finally, we notice that the recursive depth k will greatly impact the computational complexities, but we shift the discussion on this to Section 7.1.

To prove the correctness of the algorithm, we first introduce a new concept.

Definition 5.7 (Index). Let G be an edge labeled graph $\langle V, E, \Sigma, l \rangle$. Denote by ind_G an index over G , including all ω_v 's and ω_v 's for the vertices v in G , as well as compatible graphs C_i ($i = 1, \dots, k-1$), built as described in Sections 5.2–5.4. Define $ind_G(v)$ to be a set of pairs (u, L) such that u can be found through ind_G as a vertex reachable from v , and L is a set of labels that can be formed by using the replacement operations over all the tree paths each connecting a vertex in the *from*-sequence of v to the corresponding vertex in the *to*-sequence of u .

We say that $ind_G(v)$ is complete over G if, for any path labeled L' from v to u , we have $(u, L) \in ind_G(v)$ for some $L \subseteq L'$. We say that $ind_G(v)$ is sound over G if, for any $(w, L) \in ind_G(v)$, there is a path labeled L from v to w . In the following, we prove that our index is both complete and sound.

LEMMA 5.3. *Let T be a spanning tree of G and \mathbf{T} be a graph obtained by adding to T all the forward edges with respect to T . Denote by $ind_{\mathbf{T}}$ an index over \mathbf{T} . Then, $ind_{\mathbf{T}}(v)$ is complete and sound over \mathbf{T} for any $v \in \mathbf{T}$.*

PROOF (COMPLETENESS). Let p be a path from v to u in \mathbf{T} . Then, p is a tree path or a path obtained by replacing some segments of the tree path p_u with the corresponding covering forward edges. We consider $[\alpha_u, \gamma_u, \beta_u]$ and $[\alpha_v, \gamma_v, \beta_v]$. They are part of $ind_{\mathbf{T}}$. Then, by applying the replacement operations to p_{vu} , using some τ_i 's in $\gamma_u = \langle A(p_u); \tau_1, \dots, \tau_l; \lambda_u \rangle$ (for some l) and $A(p_{vu}) = A(p_u) - A(p_v)$ (which is in $\gamma_v = \langle A(p_v); \tau_1', \dots, \tau_h'; \lambda_v \rangle$ for some h), we can generate $ind_{\mathbf{T}}(v)$, which contains $(u, L(p))$. So, $ind_{\mathbf{T}}(v)$ is complete.

Soundness. Assume that $(u, L) \in ind_{\mathbf{T}}(v)$. That means, by applying some τ_i 's in $\gamma_u = \langle A(p_u); \tau_1, \dots, \tau_i; \lambda_u \rangle$ to $A(p_{vu}) = A(p_u) - A(p_v)$, we can get L . This shows that L must be a path label over a path from v to u in \mathbf{T} , since both γ_u and γ_v are created in terms of the labels over all the paths from v to u in \mathbf{T} .

PROPOSITION 5.2. *Let G be an edge labeled DAG. Then, $ind_G(v)$ is complete and sound for any $v \in G$.*

PROOF (COMPLETENESS). We prove the completeness of $ind_G(v)$ by induction on k , the depth of recursive decompositions of G . (Here, by the depth of recursive decomposition of G , we mean when we will stop the recursive decomposition.)

Basic step: When $k = 0$, G itself is a tree-like graph \mathbf{T}_0 , which is a spanning tree T plus all the forward edges with respect to T . According to Lemma 5.3, $ind_G(v)$ is complete for each $v \in \mathbf{T}_0$. When $k = 1$, G is decomposed into \mathbf{T}_0 and \mathbf{T}_1 and any v is associated with two labels: $[\alpha_v^1, \gamma_v^1, \beta_v^1]$, $[\alpha_v^2, \gamma_v^2, \beta_v^2]$; and a pair $\langle v^{\rightarrow}, v^{\leftarrow} \rangle$. Assume that there exists a path p from v to u in G . If p does not go through any cross edge with respect to \mathbf{T}_0 , then the completeness of $ind_G(v)$ holds according to Lemma 5.3. If p contains some cross edges, then, according to Lemma 5.2, there exist two paths in \mathbf{T}_0 : p_1 goes from v to v^{\rightarrow} , p_2 from u^{\leftarrow} to u , and a path in \mathbf{T}_1 : p_3 goes from v^{\rightarrow} to u^{\leftarrow} such that p is the concatenation of p_1 , p_2 , and p_3 . Obviously, $L(p_1) \cup L(p_2) \cup L(p_3) = L(p)$. Applying Lemma 5.3, respectively, to these three subpaths, we can see that the completeness of $ind_G(v)$ holds.

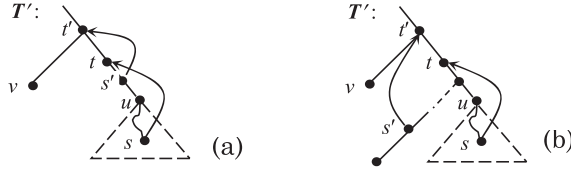
Induction step: Assume that when $k = l \geq 0$ $ind_G(v)$ is complete for each $v \in G$. That is, for any path p from v to another vertex u , we have $(u, L') \in ind_G(v)$ with $L' \subseteq L(p)$. That means there exists $j \leq l$ such that

- $v \rightsquigarrow x_1 \rightsquigarrow \dots \rightsquigarrow x_j$,
- $x_j \rightsquigarrow z_j$ in \mathbf{T}_j , where \mathbf{T}_j is the spanning tree of G_j plus the forward edges with respect to \mathbf{T}_j ,
- $z_j \rightsquigarrow \dots \rightsquigarrow z_1 \rightsquigarrow u$, and
- all the labels on the relevant paths make up a subset of $L(p)$, where x_1, \dots, x_j are the first j vertices in the *from*-sequence associated with v ; and z_1, \dots, z_j are the first j vertices in the *to*-sequence associated with u .

Now, we consider the case of $k = l + 1$. We need to distinguish between two cases:

- (1) p contains only the edges in \mathbf{T}_0 .
- (2) p contains some edges not in \mathbf{T}_0 .

In case (1), according to Lemma 5.3, the completeness of $ind_G(v)$ holds.

Fig. 8. Illustration for b -chains.

In case (2), we consider v^{\rightarrow} and u^{\leftarrow} and notice that p must go through both these two vertices. Then, we denote by p_1 the subpath from v to v^{\rightarrow} , p_2 the subpath from v^{\rightarrow} to u^{\leftarrow} , and p_3 the subpath from u^{\leftarrow} to u . Notice that p_2 appears in $G' = G_c^0$, for which the recursive depth is l . (It is because for G the recursive depth is $l + 1$ and the recursive depth for G' is then one less than that for G .) According to the induction hypothesis, $ind_G(w)$ is complete for each $w \in G'$. Thus, $ind_G(v^{\rightarrow})$ is complete. Then, there exists $L' \subseteq L(p_2)$ such that $(u^{\leftarrow}, L') \in ind_G(v^{\rightarrow})$. From this, we can see that $(u, L(p_1) \cup L(p_2) \cup L') \in ind_G(v)$ and $(L(p_1) \cup L(p_2) \cup L') \subseteq L(p) = (L(p_1) \cup L(p_2) \cup L(p_3))$. This shows the completeness in case (2).

Soundness. The proof of the soundness can also be done by induction as above.

6 LCR QUERIES OVER CYCLIC GRAPHS

Let G be a cyclic graph, i.e., a graph containing cycles. Let T be a spanning tree (forest) of it. As with DAGs, we will decompose G into two components T' and G_c' , where $T' = T \cup E_{back} = T \cup E_{forward} \cup E_{back}$ ($= G \setminus E_{cross}$), and G_c' is a subgraph to be defined below. Accordingly, we will transform an LCR query to a T' -checking and a G_c' -checking.

6.1 T' -checking

Now, for doing T' -checking $LCR(u, v, S, T')$, we need some new concepts. The first of them is the so-called *back edge chains*.

Definition 6.1 (Back Edge Chain). A sequence of back edges $s_1 \xrightarrow{x_1} t_1, \dots, s_l \xrightarrow{x_l} t_l$ ($l > 0$) with t_{i+1} being an ancestor of t_i for $i \in \{1, \dots, l-1\}$ is called a *back edge chain* (b -chain for short) if $t_i = s_{i+1}$ or t_i is an ancestor of s_{i+1} .

See Figure 8(a) for illustration, in which a b -chain starting from s and containing two back edges: $s \rightarrow t$ and $s' \rightarrow t'$ is demonstrated. In Figure 8(b), we illustrate another b -chain also containing two back edges $s \rightarrow t$ and $s' \rightarrow t'$, but s and s' are not on a same path.

Definition 6.2 (b-chain Path). Let $B = s_1 \xrightarrow{x_1} t_1, \dots, s_l \xrightarrow{x_l} t_l$ be a b -chain. A b -chain path with respect to B is a set of subpaths connected by all the back edges $s_i \xrightarrow{x_i} t_i$ ($i = 1, \dots, l$) in B , each subpath is from t_{i+1} to s_i ($i = 1, \dots, l-1$) in T .

For example, in Figure 8(b), we can see a b -chain path, which is made up of back edge $s \rightarrow t$, path from t to s' in T , and back edge $s' \rightarrow t'$.

Then, to see whether v is reachable from u under S through back edges, we will check whether there exists a b -chain path P starting from a back edge $s \xrightarrow{x} t$ and ending at another back edge $s \xrightarrow{x} t$ such that

- (i) $s \in T[u]$,
- (ii) there is a path p from u to s in T with $L(p) \subseteq S$,
- (iii) there is a path p' from t' to v in T with $L(p') \subseteq S$, and
- (iv) all the labels on P fall in S .

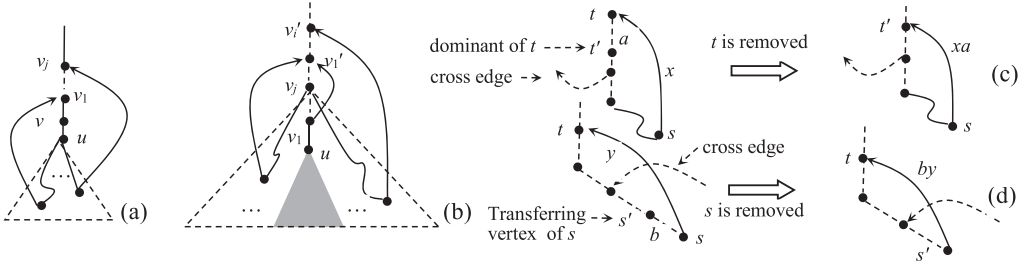


Fig. 9. Illustration for the execution of $b\text{-path}()$ and sets V_{bs} and V_{be} .

To describe a process to evaluate $LCR(u, v, S, \mathbf{T}')$, we recognize a set of vertices $\{v_1, \dots, v_l\}$ (for some $l \geq 0$) in T , called a *transit set* with respect to u with the following conditions being satisfied:

- (1) Each v_i ($i = 1, \dots, l$) is an ancestor of u in T .
- (2) For each v_i , there exists a back edge $s \xrightarrow{x} t$ with $t = v_i$, $x \in S$, and there is also a path p in T from u to s such that $L(p) \subseteq S$.

Obviously, should there be a path p' from a v_i ($i \in \{1, \dots, l\}$) to v in T such that $L(p') \subseteq S$, then $LCR(u, v, S, \mathbf{T}')$ returns *true*. Otherwise, we will continue to figure out a transit set V_j with respect to each $v_j \in \{v_1, \dots, v_l\}$. Then, for each $v' \in V_1 \cup \dots \cup V_l$, we will check whether there exists a path p'' from a v' to v in T such that $L(p'') \subseteq S$. We repeat this process until we find a *b-chain* path with all its labels in S or end up with an empty transit set. In the former case, $LCR(u, v, S, \mathbf{T}')$ returns *true* while in the latter case $LCR(u, v, S, \mathbf{T}')$ returns *false*.

In terms of the above discussion, we give the following recursive algorithm to evaluate $LCR(u, v, S, \mathbf{T}')$:

ALGORITHM 4: $b\text{-path}(u, v, S, A)$ (*Initially, $u = \{u\}$, $A = \mathbf{T}'$.*)

begin

1. let $u = \{v_1, \dots, v_l\}$; let $v' \in u$ be the ancestor of all the other v_i 's in u ;
2. **if** there exists i such that $\mathbf{T}\text{-checking}(v_i, v, S, A \setminus E_{back}) = \text{true}$ **then** return *true*;
3. **else** {**for** each $v_j \in u$ **do** {figure out the transit set V_j with respect to v_j };
4. $u := -V_1 \cup \dots \cup V_l$;
5. **if** $u \neq \emptyset$ **then** return $b\text{-path}(u, v, S, A \setminus \{\text{back edges in } A[v']\})$ **else** return *false*;

end

In the above algorithm, u represents a transit set. Initially, u is set to be $\{u\}$ and A to \mathbf{T}' . First, in line 2, we will check whether there exists $v_i \in u = \{v_1, \dots, v_l\}$ such that $\mathbf{T}\text{-checking}(v_i, v, S, A \setminus E_{back})$ evaluates to *true*. If it is the case, then the algorithm returns *true*. Otherwise, we will find a transit set with respect to each $v_j \in u$ (see line 3) and make a recursive call (see line 5), where special attention should be paid to the new values for u and A . u is set to be $V_1 \cup \dots \cup V_l$ while A is reduced to $A \setminus \{\text{back edges in } \mathbf{T}'[v']\}$ to avoid repeated access of back edges.

In Figure 9(a), we illustrate the first execution of $b\text{-path}(u, v, S, A)$ for the case of $v \sim u$, by which $u = \{u\}$ and $A = \mathbf{T}'$. Then, $LCR(u, v, S, \mathbf{T}' \setminus E_{back})$ (in line 2) definitely returns *false*. Hence, line 3 will be executed, generating a new transit set $\{v_1, \dots, v_j\}$ (see the back edges in Figure 9(a).) Figure 9(b) is the illustration of the first recursive call (i.e., the second execution: $b\text{-path}(\{v_1, \dots, v_j\}, v, S, \mathbf{T}' \setminus \{\text{back edge in } \mathbf{T}'[u]\})$ invoked in line 5, by which the back edges in $\mathbf{T}'[u]$ (marked gray) are removed to avoid visiting them once again.

By using the above algorithm, $\mathbf{T}\text{-checking}(u, v, S, \mathbf{T}')$ can be described as below.

ALGORITHM 5: T' -checking(u, v, S, T')**begin**

1. $\mathbf{u} := \{u\}; A := T'$;
2. return $b\text{-path}(\mathbf{u}, v, S, A)$;

end

Example 6.1. Consider $LCR(15, 13, \{a, b, d, e\}, G)$, where G is the graph shown in Figure 1. Then, in the execution of $b\text{-path}()$, we will first make a T -checking by calling $T\text{-checking}(15, 13, \{a, b, d, e\}, T)$, where $T = T' \setminus E_{back}$ (see line 2). Since $[\alpha_{15}, \beta_{15}] = [10, 11] \not\subseteq [\alpha_{13}, \beta_{13}] = [4, 5]$, this T -checking will definitely return *false* and thus line 3 will be executed, where $\mathbf{u} = \{15\}$, by which along the back edge $15 \xrightarrow{b} 1$, we will find the transit set $V = \{1\}$. In line 4, \mathbf{u} will be set to be $V = \{1\}$. Then, by the recursive call $b\text{-path}(\{1\}, 13, \{a, b, d, e\}, T' \setminus \{\text{back edges in } T'[15]\})$ in line 5, $T\text{-checking}(1, 13, \{a, b, d, e\}, T)$ will be invoked (see line 2), which returns *true*.

The main cost of a recursive execution of Algorithm $b\text{-path}()$ consists in the execution of T -checking(v_i, v, S, T) (in line 2) for each v_i in $\mathbf{u} = \{v_1, \dots, v_k\}$. To estimate its value, two facts should be remarked:

- (a) Each v_i in \mathbf{u} (except the initial value u) is the end vertex of a back edge and all v_i 's must be on the tree path from the root to u .
- (b) The transit set used by any other recursive call of Algorithm $b\text{-path}()$ must have the same property as \mathbf{u} described in (a).

Denote by b the number of all those back edges $s \rightarrow t$ with t appearing on the tree path from the root to u in T' . Then, the running time of $b\text{-path}()$ is bounded by $O(bh|\Sigma|)$, where h is the maximum number of all those forward edges attached to a tree path in T' (see Section 5.5).

Concerning the correctness of Algorithm $T'\text{-checking}(u, v, S, T')$, we have the following lemma:

LEMMA 6.1. *The answer returned by $T'\text{-checking}(u, v, S, T')$ is correct.*

PROOF. If there is a path P from a vertex u in \mathbf{u} to v under S , then we distinguish between two cases. In case (1), P is not through any back edges and the algorithm gives the correct answer. In case (2), P is through some back edges. Without loss of generality, assume that P uses $k > 0$ back edges. Then, after one iteration of the algorithm, we obtain a transit set \mathbf{u}' that contains at least one vertex such that there is a path under S from it to v , using at most $k - 1$ back edges. Moreover, every vertex in \mathbf{u}' is reachable from u with a path under S in T' . Thus, by a simple induction on the number of back edges, we can prove that the algorithm definitely returns the correct answer in case (2).

If there is no path from any vertex in \mathbf{u} to v under S , then for each u in \mathbf{u} its transit set \mathbf{u}' is ϕ , or any vertex in \mathbf{u}' can not reach v under S . Again, by an induction, but on the height of G , we are able to prove that the algorithm will return *false* in this case.

6.2 G_c' -checking

To construct G_c' , we are required to recognize four more subsets of vertices, besides the subsets discussed in Subsection 5.3.1:

- $V_{b\text{-start}}$ - all the start vertices of back edges.
- $V_{b\text{-end}}$ - all the end vertices of back edges.
- V_{bs} - all those start vertices s of back edges $s \rightarrow t$, where $t \in V_{c\text{-start}}$ or t is an ancestor of some vertex in $V_{c\text{-start}}$ with respect to T . (See the left part of Figure 9(c) for illustration.)

- V_{be} - all those end vertices t of back edges $s \rightarrow t$, where $s \in V_{c-end}$ or s is a descendant of some vertex in V_{c-end} with respect to T . (See the right part of Figure 9(d) for illustration.)

Furthermore, V_{LCA} , as well as V' (set of dominant vertices in Definition 5.5) and V'' (set of transferring vertices in Definition 5.6) needs to be redefined as follows:

- V_{LCA} - all those vertices with each being a lowest common ancestor of more than one vertex in $V_{c-start} \cup V_{fs} \cup V_{bs}$, which are not related by the ancestor/descendant relationship in T .
- $V' = V_{LCA} \cup V_{c-start} \cup V_{fs} \cup V_{bs}$.
- $V'' = V_{c-end} \cup V_{fe} \cup V_{be}$.

Using these notations, we can define v^{\rightarrow} and v^{\leftarrow} in the same way as for DAGs.

Again, as for forward edges, any back edge $s \xrightarrow{x} t$ in T' satisfying one of the following two conditions will be kept or replaced with a new edge while all the other back edges will be simply removed.

- $s \in V_{bs}$, or
- $s \in V_{c-end}$, or is a descendant of some vertex in V_{c-end} .

If (i) is satisfied, we distinguish between two cases: if $t \in V_c \cup V_{bs} \cup V_{be}$, then $s \xrightarrow{x} t$ will be kept; if $s \notin V_c \cup V_{bs} \cup V_{be}$, then $s \xrightarrow{x} t$ will be replaced with a new edge $s \rightarrow t'$, labeled with $xL(p_{t'})$, where t' is the dominant vertex of t (see Figure 9(c) for illustration).

If (ii) is satisfied, then we also distinguish between two cases: if $s \in V_c \cup V_{bs} \cup V_{be}$, then $s \xrightarrow{x} t$ will be kept; otherwise, $s \xrightarrow{x} t$ will be replaced with $s' \rightarrow t$, labeled with $xL(p_{s'})$, where s' is the transferring vertex of s (see Figure 9(d) for illustration).

Denote by E_{b-c} the set of all such new edges. G_c' is constructed as

$$G_c' = T_c \cup E_{cross} \cup E_{f-c} \cup E_{b-c}. \quad (4)$$

In terms of the above discussion, we give the following algorithm for doing G_c' -checking (Algorithm 6), in which we first make two T' -checkings: T' -checking ($u, u^{\rightarrow}, S, T'$) and T' -checking (v, v^{\leftarrow}, S, T'). If both of them return *true*, then we will call $LCR(u^{\rightarrow}, v^{\leftarrow}, S, G_c')$, in which G_c' -checking() will be recursively invoked (see Algorithm 7). Here, Algorithm 7 works like Algorithm 3. The only difference consists in that G is decomposed into $T' = T \cup E_{forward} \cup E_{back}$ and G_c' , i.e., $G = T' \oplus G_c'$. Recall that for DAGs $G = T \oplus G_c$.

ALGORITHM 6: G_c' -checking(u, v, S, G_c')

```

begin
1. if  $T'$ -checking( $u, u^{\rightarrow}, S, T'$ ) then
2.   {if  $T'$ -checking( $v, v^{\leftarrow}, S, T'$ ) then
3.     {if  $LCR(u^{\rightarrow}, v^{\leftarrow}, S, G_c')$  then return true;}}
4. return false;
end

```

ALGORITHM 7: $LCR(u, v, S, G)$

```

begin
1. let  $G = T' \oplus G_c'$ ;
2. if  $T'$ -checking( $u, v, S, T'$ ) then return true
3. else return  $G_c'$ -checking( $u, v, S, G_c'$ );
end

```

From the running time of T' -checking, the time complexity of G_c' -checking can be easily estimated. It is $O(\sum_{i=0}^{k-1} b_i(h_i^2 + h_i | \Sigma |))$, where b_i is the maximum number of all those back edges $s \rightarrow t$ with t appearing on a same tree path in T_i' .

In addition, more effort is required for the index construction to handle back edges, for which we need to maintain an extra tree structure for G_i , called a *be-tree* and denoted D_i . In D_i , each vertex s is for a set of back edges each with the same starting vertex s , and we have an edge from s to s' if s is an ancestor of s' in T_i' and there is no back edges emanating from any vertex on the tree path from s to s' (except for s to s' themselves). Its main purpose is to quickly figure out all the back edges $s \rightarrow t$ with s appearing in a certain subtree of T_i . Obviously, the size of D_i is smaller than $|T_i|$. Therefore, the index construction time and the index space are the same as for DAGs.

Also, based on Lemma 6.1, the following proposition can be established:

PROPOSITION 6.1. *Let G be a cyclic graph. Denote by ind_G an index over G , including all ϖ_v 's and ω_v 's for the vertices in G , all compatible graphs C_i , and all be-trees D_i ($i = 1, \dots, k - 1$). Then, ind_G is complete and sound over G .*

PROOF (SKETCH). The proposition is similar to Proposition 5.2. Hence, we just sketch its proof here. To show the completeness, however, we should explain that for any path p from v to another vertex u , we have $(u, L) \in ind_G(v)$ with $L \subseteq L(p)$. That means, there exists j such that

- $v \rightsquigarrow x_1 \rightsquigarrow \dots \rightsquigarrow x_j$,
- $x_j \rightsquigarrow z_j$ in T_j' , where T_j' is the spanning tree T_j of G_j plus the forward and back edges with respect to T_j ,
- $z_j \rightsquigarrow \dots \rightsquigarrow z_1 \rightsquigarrow u$, and
- all the labels on the relevant paths make up a subset of $L(p)$, where x_1, \dots, x_j are the first j vertices in the *from*-sequence associated with v ; and z_1, \dots, z_j are the first j vertices in the *to*-sequence associated with u .

According to Lemma 6.1, the T_j' -checking of $x_j \rightsquigarrow z_j$ in T_j' will return the correct answer. So, by induction on j , we can prove this proposition as for Proposition 5.2. \square

7 TECHNIQUE DETAILS

In the previous sections, the main process of our method is described. In this section, we will discuss three important techniques aforementioned to speed up the process: (i) how to find a better spanning tree in Section 7.1; (ii) how to explore a compatible tree efficiently in Section 7.2; finally, (iii) how to make the vertex classification in linear time in Section 7.3.

7.1 Finding Spanning Trees

For a given DAG $G(V, E)$, we can find different spanning trees by exploring G in different ways. Especially, for different spanning trees, the size of G_c can be different. Clearly, what we want is to find such a spanning tree that the number of edges in G_c is minimized. But, how to find such a spanning tree?

Let $\mathfrak{T}(G)$ be a family including all the spanning trees of G . For a spanning tree $T \in \mathfrak{T}(G)$, denote by $r_T(v)$ the number of the cross edges with respect to T , which come into v . We define

$$R(T) = \sum_{v \in V} r_T(v).$$

Intuitively, the smaller $R(T)$ is, the smaller the size of G_c . So, our optimization problem is to find a T such that $R(T)$ is minimum. Unfortunately, there are exponentially many spanning trees for a given DAG. So, it is unlikely to find an optimal one in polynomial time. In fact, it is *NP*-complete.

In the following discussion, we will first prove the *NP*-completeness of the problem. Then, we will present a top-down algorithm to find a spanning tree of G with fewer cross edges than the traditional **depth-first search (DFS)**.

7.1.1 NP-completeness. First, we notice that

$$R(T) = \sum_{v \in V} r_T(v),$$

where $f_T(v)$ is the number of the forward edges coming into v with respect to T . Thus, minimizing $R(T)$ is equivalent to maximizing

$$F(T) = \sum_{v \in V} f_T(v).$$

Therefore, to show the *NP*-completeness of minimizing $R(T)$, we can show the *NP*-completeness of maximizing $F(T)$.

To maximize $F(T)$, we need to maximize the number of the attached forward edges of each path in T .

Now, we consider a much easier problem to find a T such that it has a path with the maximal number of attached forward edges and show that even this problem is *NP*-complete. 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 k with the number of the attached forward edges of p equal to $(k-1)(k-2)/2$?

We call this problem a *maximum p -attachment* problem.

PROPOSITION 7.1. *The maximum p -attachment is NP-complete.*

PROOF. It is easy to see that the problem is in *NP*: An algorithm can generate all spanning trees T of G and check each T to see whether it has a maximum p -attachment.

The completeness for *NP* is shown by a reduction from the basic *NP*-complete problem SATISFIABILITY [10]. Let an instance of SATISFIABILITY be given by a collection of clauses $C = \{c_1, \dots, c_k\}$. Each c_i is of the form $x_{i1} \vee x_{i2} \vee \dots \vee x_{ik}$, where x_{ij} is a literal. We form a DAG in two steps:

- (1) Generate an undirected graph G' , whose vertices are pairs of integers $[i, j]$, for $1 \leq i \leq k$ and $1 \leq j \leq k_i$. A vertex $[i, j]$ is connected to another vertex $[k, l]$ if both of the following hold:
 - $i \neq k$, and
 - $x_{ij} \neq \neg x_{kl}$.
- (2) Explore G' in the depth-first manner to change it to a DAG G'' as follows:
 - If an edge (u, v) in G' is explored from u to v , then create an edge $u \rightarrow v$ in G'' .
 - In G'' , reverse the direction of any back edge. (Then, the resulting G'' must be a DAG.)
 Obviously, the DAG can be constructed in polynomial time.

Now, we claim that there is a satisfying truth assignment for C if and only if there is spanning tree containing a path p of length k such that the number of the attached edges of p is equal to $(k-1)(k-2)/2$. It is because if C is satisfiable, then there must be a *clique* of size k . Exploring the clique in the depth-first fashion and then reverse any back edge, we will get a path of length k with the number of the attached edges equal to $(k-1)(k-2)/2$.

Next, assume that T is a spanning tree of G'' , which contains a path p of length k with the number of the attached forward edges equal to $(k-1)(k-2)/2$. Assigning a value to the variable in each literal x corresponding to a vertex on p such that x is *true* while a value to the variable in any other

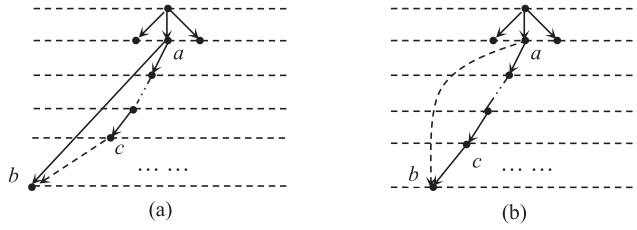


Fig. 10. Illustration for “triangles” encountered during a DFS.

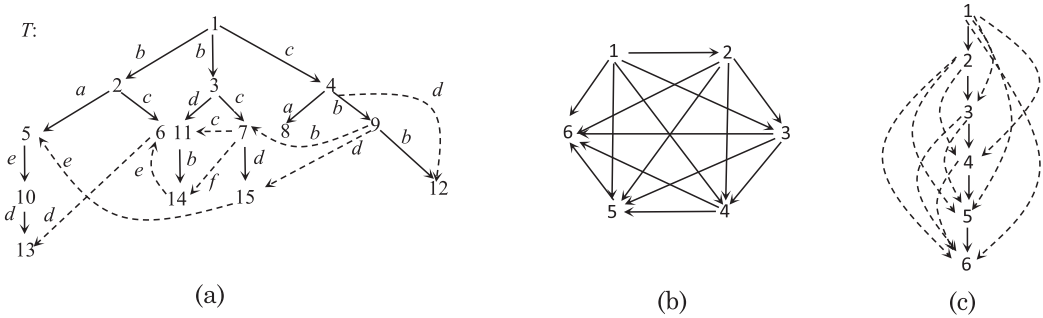


Fig. 11. A spanning tree and a dense graph.

literal y such that y is *false*, we get a satisfying truth assignment for C for the following reason: First, each vertex corresponds to a literal in a different clause. Second, for each pair of literals represented by two vertices on the path, they are not negation of each other. This completes the proof. \square

7.1.2 A Top-down Algorithm. In this subsection, we give our top-down algorithm to explore G , which is able to find a spanning tree with more forward edges than the traditional depth-first search. The main idea behind the algorithm is to recognize a kind of “triangles” as illustrated in Figure 10(a) during a search.

In Figure 10(a), assume that vertex c is the current vertex along a path from a to c , and b is one of c ’s children, but has been visited before (along the edge from a to b , as illustrated in the figure). We can remove the tree edge $a \rightarrow b$ and make $c \rightarrow b$ a tree edge. Then, $a \rightarrow b$ becomes a forward edge, as demonstrated in Figure 10(b). We call this process a *triangle transformation*. To do such a transformation efficiently, we arrange a Boolean array B such that $B[i] = 1$ indicates that vertex i is on the current path during the depth-first search. Otherwise, $B[i] = 0$. For simplicity, we assume that G is a rooted graph. Then, by the current path, we mean the path from the root to the currently encountered vertex. Let $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ be such a path. Assume that we are going to access one of v_k ’s children. At this moment, in B , all $B[v_j]$ ’s for $j = 1, \dots, k$ must be set to 1 while all the other entries must be 0.

In addition, three extra data structures are used:

- s – a stack to control the depth-first search;
- $c\text{-list}(v)$ – a list of all those children of v in G , which have not yet been visited.
- $Ch_T(v)$ – a list containing all the children of v in T .

In the above algorithm, the stack s is used to keep the current path. Then, for each vertex w in s , we have $B[w] = 1$. Let v be the vertex at the top of s (i.e., $\text{top}(s) = v$; see line 5). We will check the

ALGORITHM 8: $DFS-f(G)$ **begin**

1. Each entry of B is set 0;
2. for each v , store its children in $c-list(v)$;
3. push($root, s$); mark $root$; $B[root] := 1$;
4. **while** ($s \neq \phi$) **do** {
5. $v := top(s)$;
6. **while** $c-list(v) \neq \phi$ **do** {
7. let u be the first vertex in $c-list(v)$, chosen according to a heuristic if any;
8. **if** u is marked **then** {
9. let u' be the parent of u in T ;
10. **if** $B[u'] = 1$ **then** {remove u from $Ch_T(u')$; add u to $Ch_T(v)$;};
11. remove u from $c-list(v)$;};
12. **else** {add u to $Ch_T(v)$; push(u, s); mark u ; $B[u] := 1$; $v := u$;};
13. $w := pop(s)$; $B[w] := 0$;};

end

first element u in $c-list(v)$ (note that initially $c-list(v)$ contains all the children of v ; see line 2). Two cases need to be distinguished: u is marked (showing that v has been visited before), or not marked. If u is marked, then we will check whether its parent u' (in the spanning tree T created up to now) is on the current path by checking $B[u']$ (see lines 9–10). If it is the case, then a transformation will be conducted (see line 10). Otherwise, u is simply removed from $c-list(v)$ (see line 11). If u is not marked, then it will be added to T as one of v 's children (see line 12). Then, u is pushed into s and marked (see line 12). In a next step, one of u 's children will be visited (see the assignment statement: $v := u$ in line 12). We repeat this process until we meet a vertex v' with $c-list(v') = \phi$. In this case, the top element of s is popped out and the corresponding entry in B is set to 0 (see line 13).

Example 7.1. Consider the graph shown in Figure 11(a). If we use the traditional depth-first search to explore the graph, then we may create a spanning tree as shown by the solid arrows.

But if we use $DFS-f$ to explore G , then the triangle with three corners 3, 11, and 7 (see Figure 11(a)) will be recognized and transformed, leading to the spanning tree shown by the solid edges in Figure 2(a), which has two more forward edges than the spanning tree shown in Figure 11(a).

Concerning the correctness of the algorithm, we have the following proposition:

PROPOSITION 7.2. *Let T and T' be two spanning trees generated by a traditional depth-first search and a $DFS-f$ search of DAG G , respectively. Then, $F(T) \leq F(T')$.*

PROOF. Let $\Delta_{a,b,c}$ be a triangle met during the depth-first search. That is, $a \rightarrow b$ is a tree edge, $c \rightarrow b$ is a cross edge, and there is a tree path from a to c . By the $DFS-f()$, this triangle will be transformed such that $a \rightarrow b$ becomes a forward edge, and $c \rightarrow b$ a tree edge. More importantly, any forward edge from a or an ancestor of a to b or a descendant of b in $T[b]$ is still a forward edge with respect to T' . This shows that $F(T) \leq F(T')$. \square

The time for doing a transformation is bounded by a constant. Thus, the time complexity of $DFS-f()$ is still in $O(n + m)$.

7.1.3 About Recursive Depth k . Now, we are in position to discuss the recursive depth, i.e., the value of k . Intuitively, the sparser a graph is, the smaller the value of k . However, for a very dense graph, the value of k can be very small. To see this, let us consider a very dense graph shown in

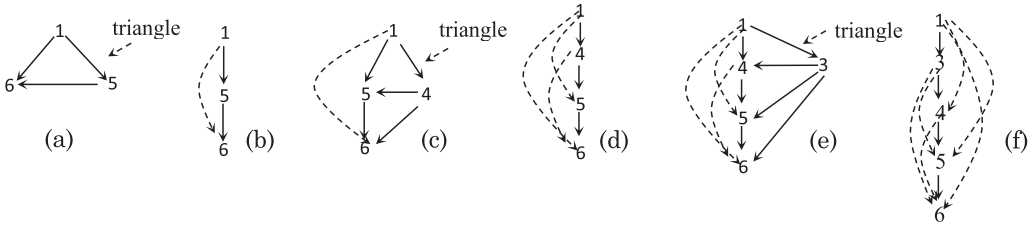
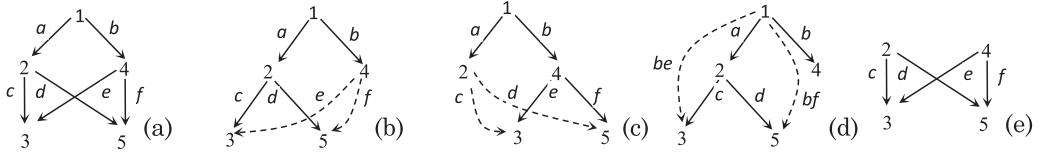
Fig. 12. Illustration of the execution of $DFS-f()$.

Fig. 13. Illustration of an unreducible graph.

Figure 11(b), for which we can find a spanning tree shown as solid edges in Figure 11(c). It is in fact a single path. The corresponding G_c is ϕ . Thus, for this graph k is equal to 1.

In Figure 12, we illustrate the execution of $DFS-f()$ when applied to this graph.

In Figure 12(a), we show the first triangle encountered during the execution of $DFS-f()$. By the triangle transformation, it will be changed to a graph as shown in Figure 12(b). In Figure 12(c), we show the second triangle encountered. Then, it will be changed to a graph as shown in Figure 12(d). The third triangle encountered is shown in Figure 12(e). It will be changed to a graph as shown in Figure 12(f). Continuing in this way, we will finally get the spanning tree shown in Figure 11(c).

However, there exist some DAGs G , whose G_c is G itself. As an example, consider the graph shown in Figure 13(a). One of the spanning trees T of this graph is shown by the solid edges in Figure 13(b). With respect to T , we have $V_{c-start} = \{4\}$, $V_{c-end} = \{3, 5\}$, $V_{LAC} = \{1, 2\}$, and $V_{f-start} = V_{f-end} = V_{fs} = V_{fe} = \phi$. Then, we can see that $V_{c-start} \cup V_{c-end} \cup V_{LAC} \cup V_{fs} \cup V_{fe} = \{1, 2, 3, 4, 5\}$, same as the vertex set of the original graph. Thus, its G_c must be identical to the original graph.

Figure 13(c) shows another spanning tree of the graph. Its G_c is also identical to G itself.

We have two ways to handle this situation. In the first way, we simply establish an index for G_c by using an existing method if G_c is small. In the second way, we slightly extend the strategy described in the previous sections. Assume that T is a forest. Let r_1, r_2, \dots, r_l (from left to right) be the roots of the subtrees in T . For each cross edge $v \xrightarrow{a} u$ with both v and u appearing in $T[r_1]$, we will add a forward edge e to T as follows: Denote by p the tree path from r_1 to v . e will be labeled with $L(p)a$ if no forward edge is attached to p , as illustrated in Figure 13(d), where corresponding to the two cross edges shown in Figure 13(b), two forward edges labeled, respectively, with “ be ” and “ bf ” are added to the tree shown in Figure 13(b). Should be there some forwards edges attached to p , e will be labeled with $C_v.a$, where C_v is the compatible graph associated with v and $C_s.a$ represents a set of path labels with each made up of a plus one of the path labels obtained by exploring C_v as described in Section 5.2. Then, we are able to remove r_1 and the edges incident to r_1 from G_c without loss of reachability information. It is because using $DFS-f(G)$, we definitely have no cross edge from a vertex in $T[r_1]$ to a vertex in any other $T[r_i]$ for $i > 1$ and the reachability from r_1 to any other vertex can be checked within $T[r_1]$. In Figure 13(e), we give the G_c of the graph shown in Figure 13(a).

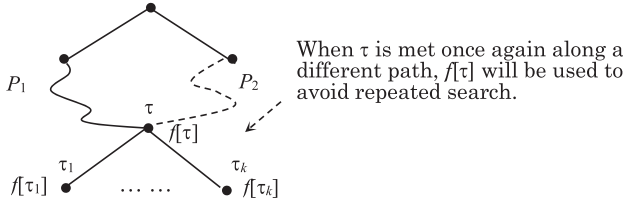


Fig. 14. Illustration for compatible graphs.

In this way, we always have $|G_C^{i+1}| \leq |G_C^i| - 1$ ($i = 0, \dots, k-2$), where $|G_C^i|$ stands for the number of vertices in G_C^i . Thus, we have $k \leq n$. In our experiments, for all the tested real data graphs G , we have $k \ll n$.

7.2 Searching Compatible Graphs

In this subsection, we discuss compatible graphs. First, how to explore a compatible graph C_v associated with a γ_v is described. Then, how to organize all C_v 's into a single global graph to save space.

7.2.1 Searching Compatible Graphs. To find a compatible set: τ_1, \dots, τ_j (for some j) along a path such that $p_{uv} \circ \tau_1 \circ \dots \circ \tau_j \subseteq S$, we need to search a C_v . For simplicity, however, we consider only a simple case that in A_{uv} , we have only a single label a^i such that $a \notin S$. But it can be easily extended to general cases of $a_1^{i_1}, \dots, a_k^{i_k}$ with each $a_l \notin S$ and $i_l \geq 1$ ($1 \leq l \leq k$). The algorithm to be given is in fact a depth-first search with a technique like *finishing timestamps* [10] being used to avoid repeated access of vertices. In the algorithm, the following notations will be used:

- $o(\tau, a)$ - an operator, which returns an integer l if $\tau.A$ contains a^l .
- $f[\tau]$ - a maximum number i such that a^i can be replaced by using the quadruples along a certain path in $C_v[\tau]$ (subtree rooted at τ in C_v). So, when we encounter a vertex τ once again, $f[\tau]$ can be simply used, which enables us to avoid searching $C_v[\tau]$ for a second time. (See Figure 14 for illustration, in which we show that if a vertex τ is met once again along a different path, we will utilize $f[\tau]$ to avoid a repeated visit of $C_v[\tau]$.) As will be seen in the following algorithm, $f[\tau]$ is created in a way like *finishing timestamps* during a depth-first search of a directed graph (see Reference [10], p. 540).

The whole working process consists of two procedures, named $CompExpl()$ and $R()$, respectively. In Algorithm $CompExpl(a^j, C_v, u, S)$, each vertex is initially considered as not marked (see line 1). Lines 2–5 check each vertex in C_v in turn and, when an unmarked vertex τ is found, visit it by calling $R(\tau, a, j, S)$ to explore part of $C_v[\tau]$.

ALGORITHM 9: $CompExpl(a^j, C_v, u, S)$

begin

1. Initially, each vertex in C_v is considered as unmarked.
2. **for** each vertex $\tau \in C_v$ **do** {
3. **if** τ is not marked and $\tau.s$ is a descendant of u in T **then**
4. {**if** $R(\tau, a, j, S) = true$ **then** return $true$;}
5. return $false$;

end

In each call $R(\tau, a, j, S)$, τ is first marked in line 1. Here, variable z is used to represent a^z , which we want to replace to satisfy the query. In line 2, we first check whether $\tau.x \in S$. If it is the case,

ALGORITHM 10: $R(\tau, a, j, S)$

```

begin
1. mark  $\tau$ ;  $z := j$ ;  $temp := 0$ ;
2. if  $\tau.x \in S$  then  $temp := o(\tau, a)$ ;
3.  $z := z - temp$ ;
4. if  $z = 0$  then return true;
5. for each  $\tau \rightarrow \tau'$  in  $C_v$  do /*access every child  $\tau'$  of  $\tau$ */
6.   if  $\tau'$  is not marked then
7.     {if  $R(\tau', a, z, S) = true$  then return true;}
8.     else {if  $f[\tau'] = z$  then return true;}
9.   if  $\tau$  is a leaf then  $f[\tau] := temp$ 
10.  else  $f[\tau] := temp + \max_{\tau \rightarrow \tau' \in C_v} \{f[\tau']\}$ ;
11. return false;
end

```

then the corresponding replacement can be conducted. Assume that $o(\tau, a) = l$. Then, a^l can be replaced and z will be decreased by l (see line 3.) If z becomes zero, then it shows that the whole a^l can be replaced and the algorithm returns *true* (see line 4). Otherwise, we will recursively explore all the subgraphs each rooted at a child of τ (see line 5). Lines 5–9 examine each vertex τ' adjacent to τ . If τ' is not marked (not yet visited), then we will recursively visit τ' if $\tau'.x \in S$. If τ' is marked, then we will check whether $f[\tau'] = z$ and return *true* if it is the case. In any case, $C_v[\tau']$ will not be repeatedly searched. Finally, after every edge leaving τ has been explored, lines 9–10 set value $f[\tau]$ for τ and returns *false* (see line 11), since in this case the checking of each subgraph $C_v[\tau']$ must return *false* (see lines 7 and 8). Special attention should also be paid to how $f[\tau]$ is calculated for each τ in lines 9 and 10. It is determined in a bottom-up way, i.e., it is computed based on the values for its children while the values for leaf vertices can always be directly calculated. As illustrated in Figure 14, when a τ is visited again along a different path, $f[\tau]$ can be used (see line 8) and $C_v[\tau]$ will not be repeatedly searched.

Example 7.2. Consider $LCR(u, v, S, T)$, where T is shown in Figure 4(a), and $S = \{a, b, d, p\}$. T is composed of a tree path p_{uv} and four forward edges e_1, e_2, e_3 , and e_4 . Obviously, we have $[\alpha_u, \beta_u] \supseteq [\alpha_v, \beta_v]$, $L(p_{uv}) - \{a, b, c\} \not\subset S$, and $\lambda_{uv} = \lambda_v - \lambda_u = \phi - \phi = \phi \subset S$. In this case, we will check $\gamma_v = \langle A(p_v); \tau_1, \tau_2, \tau_3, \tau_4; \emptyset \rangle$. (Since u is the root, $A(p_u)$ is trivially ϕ and need not be checked.) Here, we have

$$\begin{aligned}
 -A(p_v) &= \{a^2, b^2, c^2\}. \text{ (Since } c \notin S, c^2 \text{ should be replaced to satisfy the query.)} \\
 -\tau_1 &= [u, h, \{a, b\}, d], \tau_2 = [g, w, \{b, c\}, p], \tau_3 = [w, y, \{a, b\}, c], \tau_4 = [z, v, \{b, c\}, a].
 \end{aligned}$$

The compatible graph over them is shown in Figure 4(b). When exploring the graph, the following steps of computation will be carried out:

Step 1: τ_1 is visited by calling $R(\tau_1, c, 2, S)$. z is initialized to 2. Since $\tau_1.x = d \in S$, $temp$ is set to be $o(\tau_1, c) = 0$ (it is because $A(p_{uh}) = \{a, b\}$ does not contain c .) So z is not changed.

In the subsequent recursive calls, C_v will be continually explored.

Step 2: τ_3 is visited by calling $R(\tau_3, c, 2, S)$. z is initialized to 2. Since $\tau_3.x = c \notin S$, $temp = 0$ and z remains the same as before. Since τ_3 is a leaf, $f[\tau_3]$ is set to be the same as $temp = 0$.

Step 3: τ_4 is visited by calling $R(\tau_4, c, 2, S)$. Since $\tau_4.x = a \in S$, $temp$ is set to be $o(\tau_4, c) = 1$ and z is decreased to 1. Since τ_4 is a leaf, $f[\tau_4]$ is set to be the same as $temp = 1$.

Step 4: τ_2 is visited by calling $R(\tau_2, c, 2, S)$. Since $\tau_2.x = p \in S$ and $o(\tau_2, c) = 1$, z is decreased from 2 to 1. Both its children τ_3 and τ_4 are marked and will not be further accessed. But $f[\tau_4] = 1 = z$. So $R(\tau_2, c, 2, S)$ returns *true*.

From the above example, we can see that each vertex τ in C_v is visited at most once. Each time τ is met again, its $f[\tau]$ will be used to avoid repeated access.

For the general cases, we need to change $o(\tau, a)$ to $o(\tau, B)$ with B being an array of the form: $[a_1, \dots, a_k]$. Its return value is also an array of the form: $[j_1, \dots, j_k]$, indicating that $a_1^{j_1}, \dots, a_k^{j_k}$ can be replaced by τ .

In the following, we prove the correctness of Algorithm $R()$.

LEMMA 7.1. *Let $R(\tau_1, a, j_1, S) \rightarrow \dots \rightarrow R(\tau_k, a, j_k, S)$ be a chain of recursive calls with $j_1 \geq j_2 \geq \dots \geq j_k$ during the execution of $\text{CompExpl}(a^j, C_v, u, S)$. If $R(\tau_k, a, j_k, S)$ returns true, then each recursive call on the chain returns true. (Especially, if $R(\tau_k, a, j_k, S)$ does not invoke a further recursive call, then by the replacement $p_{uv} \circ \tau_1 \circ \dots \circ \tau_k, a_1^{j_1}$ will be removed.)*

PROOF. Consider $R(\tau_{k-1}, a, j_{k-1}, S)$ and $R(\tau_k, a, j_k, S)$. If j_k can be reduced to 0 by using τ_k , then $R(\tau_k, a, j_k, S)$ returns true (see line 4). Then, from line 7, we can see that $R(\tau_{k-1}, a, j_{k-1}, S)$ returns true. So, recursively, each $R(\tau_i, a, j_i, S)$ ($1 \leq i \leq k$) will return true. \square

LEMMA 7.2. *Let $R(\tau, a, l, S)$ be a call during the execution of $\text{CompExpl}(a^j, C_v, u, S)$. If $R(\tau, a, l, S)$ returns false, then any recursive call $R(\tau', a, l', S)$, invoked during the execution of $R(\tau, a, l, S)$ with τ' being a vertex in $C_v[\tau]$ and $l' \leq l$, will return false.*

PROOF. The lemma holds in terms of Lemma 7.1 and line 12 in Algorithm $R()$. \square

PROPOSITION 7.3. *Let T be a spanning tree (forest) of G . Let $\mathbf{T} = T \cup E_{\text{forward}}$. Let u and v be two vertices in G . Assume that a^j is the only label in $A(p_{uv})$ such that $a \notin S$. Then, Algorithm $\text{CompExpl}(a^j, C_v, u, S)$ returns true if a^j can be removed by using the corresponding forward edges. Otherwise, not.*

PROOF. Let e_1, \dots, e_k be a set of compatible forward edges attached to p_{uv} such that by using them to replace the corresponding segments, a^j can be removed. Then, the corresponding quadruples τ_1, \dots, τ_k must be on a path p in C_v , and p can be searched in one of two ways:

- (1) Along p , we will have a chain of recursive calls $R(\tau_1, a, j_1, S) \rightarrow \dots \rightarrow R(\tau_k, a, j_k, S)$ with $j_1 = j, j_1 \geq j_2 \geq \dots \geq j_k$, and $R(\tau_k, a, j_k, S)$ returns true. According to Lemma 7.1, $R(\tau_1, a, j_1, S)$ returns true.
- (2) Along a chain of recursive calls $p_1 = R(\tau_1, a, j_1, S) \rightarrow \dots \rightarrow R(\tau_l, a, j_l, S)$ with $j_1 = j, j_1 \geq j_2 \geq \dots \geq j_l$ such that there exists another chain p_2 starting from a recursive call $R(\tau_{l+1}, a, j_{l+1}, S)$ with its return value being false, but $f[\tau_{l+1}] = j_l - o(\tau_l, a)$. So, $R(\tau_{l+1}, a, j_{l+1}, S)$ will return true (see line 8 in Algorithm $R()$) and then $R(\tau_1, a, j_1, S)$ returns true. p is the concatenation of p_1 and p_2 . \square

If a^j cannot be replaced, then any recursive call returns false. According to Lemma 7.2, $R(\tau_1, a, j_1, S)$ returns false.

Since each vertex in C_v is accessed at most once, the running time of the algorithm is bounded by $O(h)$, where h is the number of all forward edges attached to p_{uv} .

7.2.2 About General Compatible Graphs. For efficiency, we can create a general compatible graph for all forward edges in \mathbf{T} , instead of a compatible graph for each single vertex v (or say, for each γ_v).

Definition 7.1 (General Compatibility Graph). A general compatible graph for a \mathbf{T} is a graph, $C_{\mathbf{T}}$, in which there is a vertex for each quadruple τ representing a forward edge in \mathbf{T} , and an edge $\tau \rightarrow \tau'$ if (1) τ_i and τ_j are compatible, (2) τ_i .s is an ancestor of τ_j .s, (3) between τ_i and τ_j is there no other τ , which is compatible to both.

Now, when looking for replacements to complete a T -checking $LCR(u, v, S, T)$, we will explore C_T , controlled by using intervals so that only the relevant part is visited. More exactly, the following property will be used: For any vertex (in C_T) representing a forward edge $s \rightarrow t$ attached to p_{uv} , we must have $[\alpha_u, \beta_u] \supseteq [\alpha_s, \beta_s]$ and $[\alpha_t, \beta_t] \supseteq [\alpha_v, \beta_v]$.

In addition, we can associate each vertex u in T with a set of pointers with each pointing to a vertex τ in C_T such that u is an ancestor of τ . s in T and there is not any other forward edge attached to the tree path from u to τ . s . Thus, such vertices can be used as the starting points to explore C_T when looking for replacements to satisfy reachability from vertex u to some other vertex v .

7.3 Classification of Vertices in T

We discuss now how to do the vertex classification with respect to a spanning tree T of G to construct G_c , but only focus on how to figure out the vertices in V_{LCA} , V_{fs} , and V_{fe} , since the vertices in $V_{c-start}$ and V_{c-end} , as well as in $V_{f-start}$ and V_{f-end} , can be easily recognized according to their definitions.

First, we mark all vertices in $V_{c-start} \cup V_{c-end} \cup V_{fe}$ as follows:

- Search T in the depth-first manner. In the process, keep a variable x , which contains the highest $c-end$ vertex on the current path.
- For each encountered vertex v , we will do the following checking:
- if it belongs to $V_{c-start} \cup V_{c-end}$, then mark it.
- In addition, if it belongs to V_{f-end} , then we will check the corresponding forward edge $s \rightarrow v$. If x is equal to s or an ancestor of s , then mark it, since it must be a vertex in V_{fe} .

Next, we will search T bottom-up and produce the skeleton tree T_c of T , containing only the vertices in $V_c = V_{LCA} \cup V_{c-start} \cup V_{c-end} \cup V_{fs} \cup V_{fe}$. Note that to recognize V_{LCA} , we can use the algorithm discussed in Reference [41]. However, besides V_{LCA} , we also need to recognize V_{fs} ($V_{c-start}$ and V_{fe} are already marked as described above). For this reason, we design a new procedure for this task.

Initially, T_c is set to ϕ . Then, during a bottom-up traversal of T , not only the vertices in $V_{c-start} \cup V_{fe}$ (they are all marked) will be inserted into T_c , but the vertices in $V_{LCA} \cup V_{fs}$ will also be recognized and inserted into T_c .

For each vertex u , which has already been inserted into T_c , it will be associated with a Boolean value: $c(u)$ and two links: $l_1(u)$ and $l_2(u)$, described below.

- $c(u)$ is *true* if $T[u]$ contains a vertex $v \in V_{c-start}$. Otherwise, $c(u)$ is *false*.

During a bottom-up traversal of T , $c(u)$ can be computed as follows:

- If $u \in V_{c-start}$, then set a temporary Boolean variable σ to be *true*; otherwise, *false*.
- Let u_1, \dots, u_l be the children of u . Set $c(u) \leftarrow c(u_1) \vee \dots \vee c(u_l) \vee \sigma$.
- $l_1(u)$ is a link to a vertex inserted into T_c just before u , which is not a descendant of u in T .
- $l_2(u)$ is a link to its parent or one of its ancestors, whichever first inserted into T_c .

$l_2(u)$ can be created as below, during the construction of T_c .

Let u' be the vertex inserted just before u . If u' is a child (descendant) of u , then we will first create a link from u' to u , denoted as $l_2(u') = u$. Then, we will go along the l_1 -chain starting from u' : $u' \rightarrow l_1(u') \rightarrow l_1(l_1(u')) \rightarrow \dots \rightarrow$

$l_1^{(i)}(u') = u''$ for some i such that $l_1(u')$, $l_1(l_1(u'))$, \dots , $l_1^{(i-1)}(u')$ are all the children (descendants) of u , but $l_1^{(i)}(u')$ not, where $l_1^{(i)}(u')$ is just l_1 applied i times to u' . That is, we will go along the chain until we meet a vertex u'' that is not a child (descendant) of u . For each encountered vertex

v except u'' , set $l_2(v) \leftarrow u$. Finally, set $l_1(u) \leftarrow u''$ (i.e., change the l_1 -link of u to point to u'' .) We denote the whole process as

$$r \leftarrow \text{linkTrav}(u),$$

where r is the number of vertices $\in V_{LCA} \cup V_{c-start} \cup V_{fs}$, encountered during the navigation along the l_1 -chain. If $r > 1$, then u will be inserted into T_c no matter whether it is marked or not. Otherwise, u will not be inserted into T_c if it is not marked.

According to the above discussion, we design an efficient algorithm for recognizing all the vertices in V_c and store them in T_c .

ALGORITHM 11: $\text{find-}T_c(T)$

begin

1. $T_c \leftarrow \varnothing$ Mark any vertex in T , which belongs to $V_{c-start} \cup V_{c-end} \cup V_{fe}$. Establish $c(v)$ for all v
2. Explore T bottom-up. Let u be the currently encountered vertex. Do the following operations:
 - (a) If u is the first marked vertex encountered during the bottom-up searching, then simply insert u in T_c . Otherwise, do (b).
 - (b) Let u' be the vertex inserted into T_c just before u is met. Do (i) or (ii), depending on whether u is a marked vertex or not.
 - (i) If u is marked, then insert u into T_c
 - If u' is not a child (descendant) of u , then set $l_1(u) \leftarrow u'$ (i.e., set a l_1 -link from u to u').
 - If u' is a child (descendant) of u , then execute $r \leftarrow \text{linkTrav}(u)$.
 - (ii) If u is a non-marked vertex, then do the following:
 - If u' is not a child (descendant) of u , then u is ignored.
 - If u' is a child (descendant) of u then $r \leftarrow \text{linkTrav}(u)$. If $r > 1$, then mark u and insert u into T_c . Otherwise, u is ignored.
 - (c) If u is inserted in T_c , then check all those forward edges: $s_1 \rightarrow u, \dots, s_q \rightarrow u$, each with u being its end vertex in T and $c(u) = \text{true}$. Mark s_1, \dots, s_q (*It is because all these vertices must be in V_{fs} .*)

end

In the above algorithm, special attention should be paid to (2-b-ii) and (2-c). In (2-b-ii), we recognize all those vertices $\in V_{LCA}$ while in (2-c), we recognize all those vertices $\in V_{fs}$.

Example 7.3. Consider T shown in Figure 2(a) (i.e., the spanning tree shown by the solid edges plus the corresponding forward edges). By executing $\text{find-}T_c(T)$, we will first mark vertices 13, 5, 6, 14, 15, and 9. They are all $\in V_{c-start} \cup V_{c-end} \cup V_{fe}$. Thus, the first vertex inserted into T_c should be vertex 13 (see Figure 15(a)). In a next step, vertex 5 will be inserted into T_c and $l_2(13) = 5$ will be generated (see Figure 15(b)). In the third step, we will meet vertex 6. Since it is to the right of 6, a link $l_1(6) = 5$ will be created (see Figure 15(c)). When vertex 14 is encountered next, it will be inserted into T_c , as shown in Figure 16(d). Following this, we will meet vertex 15 (see Figure 15(e)). The next encountered vertex is vertex 7 (see Figure 15(f)). It is not marked, but the parent of vertex 15. So, the l_1 -link chain starting from vertex 15 will be searched to find another child (vertex 14) of vertex 7 along the chain. Here, a close attention should be paid to the replacement of $l_1(14) = 6$ with $l_1(7) = 6$, which enables us to easily find the lowest common ancestor of 6 and some other vertices from $V_{LCA} \cup V_{c-start} \cup V_{fs}$ if any. In the last three steps, we will meet vertices 3, 9, and 1. Among them, 3 and 9 are marked and will be inserted into T_c as shown in Figures 15(g) and (h), respectively. When vertex 1 is met, we will find all the four children of it in T_c along the l_1 -link chain starting from vertex 9. They are 5, 6, 3, and 9 with $5 \in V_{c-end}$, $6, 9 \in V_{c-start}$, and $3 \in V_{fs}$. Thus, $r = 4$ (see 2-b-ii) and vertex 1 will be added to T_c (see Figure 15(i)).

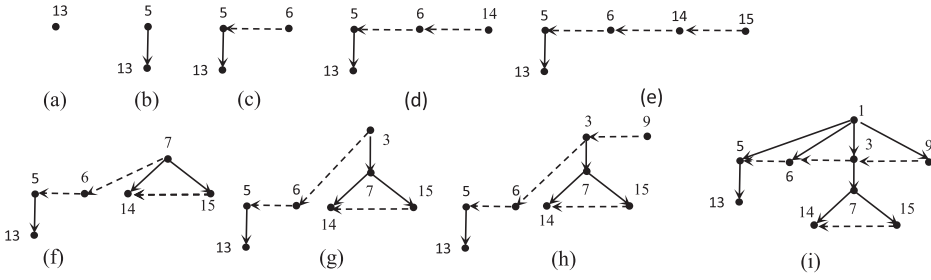


Fig. 15. A sample trace.

The T_c shown in Figure 15(i) is the same as the tree shown in Figure 6, from which G_c can be efficiently constructed, as illustrated in Figure 2(b). The algorithm for doing this requires only $O(m)$ time, as analyzed below:

- The time for executing line (1) is obviously bounded by $O(m)$.
- During the bottom-up search of T , each vertex in T is accessed at most two times: one access is done along the whole postorder of vertices, and another access is along a series of l_1 -links, by which for a vertex v $out-degree(v)$ edges will be visited. So, we have

$$\sum_{v \in V} out-degree(v) = m.$$

Concerning the correctness of this algorithm, we have the following proposition:

PROPOSITION 7.4 *Let $G = (V, E)$ be a DAG. Let T be a spanning tree (or a spanning forest) of G . Algorithm $find-T_c(T)$ generates T_c of G with respect to T correctly.*

PROOF. To show the correctness of the algorithm, we should prove the following: (1) each vertex in T_c is a vertex $\in V_c = V_{LCA} \cup V_{c-start} \cup V_{c-end} \cup V_{fs} \cup V_{fe}$; (2) any vertex not in T_c does not belong to V_c ; (3) for each edge $u \rightarrow v$ in T_c there is a path from u to v in T , which does not contain any vertex in T_c (except the two end points).

First, we prove (1) by induction on the height h of T_c . The height of a vertex v in T_c is defined to be the longest path from v to a leaf vertex in T_c .

Basis step. When $h = 0$, each leaf vertex in T_c is a vertex in $V_{c-start} \cup V_{c-end} \cup V_{fe}$. So it is correct.

Induction step. Assume that every vertex appearing at height $h = k$ in T_c is a vertex in V_c . We prove that every vertex v at height $k + 1$ in T_c is also a vertex in V_c . If v is marked, then it must be a vertex in $V_{c-start} \cup V_{c-end} \cup V_{fs} \cup V_{fe}$, the proposition holds. Assume that v is not marked. According to the algorithm, v has at least two children $\in V_{LCA} \cup V_{c-start} \cup V_{fs}$. Thus, $v \in V_{LCA}$.

To prove (2), we notice that (i) any vertex in $V_{c-start} \cup V_{c-end} \cup V_{fe}$ is marked (see line 1); (ii) any vertex in V_{fs} is marked before it is encountered (see line 2-c); and (iii) any unmarked vertex but inserted into T_c must belong to V_{LCA} (see line 2-b-ii). Finally, (3) can be seen from the fact that each l_2 -link corresponds to a path in T and such a path cannot contain any vertex in T_c (except the two end points), since the vertices in T are checked level-by-level bottom-up. \square

8 EXPERIMENTS

To show that our method does not only have a better theoretical computational complexity than the existing methods for this problem, but is also greatly better than them in practice, we have done a lot of tests on some real data and synthetic data.

Table 4. Real Datasets with Edge Labels

Datasets	$ V $	$ E $	$ \Sigma $	syn. label
robots	1.40k	2.90k	4	-
yeast	3.06k	13.3k	5	-
yago	5.00k	28.5k	66	-
advogato [35]	5.40k	51.0k	4	-
Youtube	15.0k	10.7M	5	-
BioGrid	64.0k	1.5M	7	-
yago2	16.4M	31.9M	97	-

Table 5. Real Datasets without Edge Labels

Datasets	$ V $	$ E $	$ \Sigma $	syn. label
epinions [35]	131k	840k	10	✓
webStanford [36]	251k	2.3M	10	✓
webGoogle [36]	875k	5.1M	10	✓
webBerkstan [36]	685k	7.0M	10	✓
socPokec [36]	1.6M	30M	10	✓
wikiLinks [35]	3.0M	102.0M	10	✓
citeseerX [35]	6.54M	150.1M	10	✓

In our experiments, we have altogether tested five different methods:

- (1) *BFS-based* [10] (*BFS for short*),
- (2) *Landmark-based* [20] (*LandM for short*),
- (3) *Zou’s method* [32] (*Zou’s for short*),
- (4) *Hassan’s method* [16] (*Hassan’s for short*), and
- (5) *Compatible-graph-based* discussed in this article (*CGB for short*).

The *LandM*, *Zou’s*, and *Hassan’s* methods are briefly described in Section 3. But special attention should be paid to the *Hassan’s*, since it was designed to find a shortest path p from a vertex v to another vertex u such that all edge labels on p fall in S . This is more general than *LCR* and requires in general much more query time.

The code of the landmark-based method is downloaded from <https://github.com/DeLaChance/LCR>, and the code of the *Hassan’s* method is from Reference [17]. All the other three methods are implemented by ourselves. The *Hassan’s* code is written in Java, and all the others in C++, running on a Linux machine with 128 GB of memory and a 2.9 GHz 64-core processor.

In the tests, both synthetic and real datasets are used, and queries are generated in a way controlled to avoid trivial cases that the checked vertices v and u are not far away from each other. Our goal is twofold:

- (i) to study how well our method performs on real data concerning the indexing time, index space, and query time.
- (ii) to examine the effects of different graph parameters on the performance, such as graph density, and the distribution of outdegrees of vertices, as well as the number of labels.

For this purpose, we will divide the experiments into two groups. In the first group, we test all the methods against real data. The second group is on synthetic data and further divided into three parts. In the first part, we compare our method with the *LandM* and the *Zou’s* on two kinds of synthetic data (the *uniform-data* and *Zipf-data* to be described below). In the second part, we fix the number of vertices of the synthetic graphs while vary the other parameters, such as the out-degree per vertex and label set size to study their effects. In the third part, we study the scalability of our method by increasing graph sizes.

As will be seen later, our method uniformly outperforms all the other tested approaches due to its greatly reduced index space and the short *to-* and *from-*sequences associated with each vertex.

8.1 Datasets

Table 4 provides an overview of the real datasets used in the experiments. These datasets are taken from various sources. Some are edge-labeled and some not. For those not edge-labeled, we assign labels to edges synthetically, which is indicated in the column “Synthetic labels.”

Table 6. Parameters for Synthetic Graphs

Size (# of nodes)	n
Edge labels	$\Sigma = \{1, 2, \dots\}$
Node types	not specified
Node type constraints	not specified
Probability distribution of out-degree	$P(d)$, where d represents the outdegree of a vertex.

All the graphs listed in Table 4 are with edge labels. Among them, except *BioGrid* [62], all the others *robots* [63], *yeast* [64], *yago* and *yago2* [65], *advogato* [66], and *Youtube* [36] are directed graphs. In addition, *yago* and *yago2* are two *RDF* data collections. In *BioGrid*, each undirected edge is replaced by directed edges to create a directed graph.

In Table 5, all the graphs are not edge labeled. For this reason, we will assign each edge in such graphs a number out of $\{1, 2, \dots, 9\}$, working as its label. All these datasets are taken from either SNAP [36] or KONECT [35] (<http://konect.cc/>), but come from a wide variety of application domains, including education (*robots*), Biology (*yeast*), *RDF* graphs (*yago*), media (*YouTube*), social network (*socPokec*), web network (*webStanford*, *webGoogle*, *webBerkstan*), Wikipedia (*wikiLinks*), and citation (*citeseerX*). Experimenting with them, we can observe the general behaviors of different strategies.

All the synthetic datasets are created by using the *gMark* [67]. According to the probability distribution of degrees of vertices, they can be categorized into two groups: data following *uniform distribution* ($P(d) = 1/(b - a + 1)$ ($d \in [a, b]$); and data following *Zipfian distribution* ($P(d) = \alpha d^{-l}$) [18], where d represents the out-degree of a vertex, and a , b , α , and l are four constants with $b > a$. For the uniform distribution, we take $a = 2$ and $b = 5$. For the Zipfian distribution, we fix α to 1, but set l to different values (either 2.2, 2.4, 2.6, or 2.8) to change the distribution of vertices' outdegrees. We notice that the larger l is, the smaller the number of edges in a graph. To study the scalability in these two kinds of graphs, we vary graph size from 100k to 400K vertices.

In Table 6, we show the main parameters used to determine the graph configuration [49].

8.2 Query Generation

By using the *gMark*, different kinds of *regular path queries* [51] can be created, including *LCRs*. However, by the *gMark*, we are not able to control two important properties of queries:

- (1) To avoid trivial cases that v and u are only few steps away for a query $LCR(u, v, S, G)$.
- (2) To determine whether $LCR(u, v, S, G)$ returns *true* or *false*.

For this reason, we have designed our own procedure to create queries for each dataset G (Algorithm 12) such that the “distance” between u and v , denoted as $dis(u, v)$, are properly controlled. Here, by “distance,” we mean the number of vertices visited by the *BFS* from u to v .

The above algorithm is used to generate both *true-* and *false-*queries, which takes a graph G and an integer l (to indicate the number of queries to be created) as inputs. To generate queries, we will randomly select l vertices u . For each of them, we will randomly select $\lceil l/100 \rceil$ vertices v different from u . Then, for each (u, v) , we will further create 10 subsets S of Σ and for each of them check whether $u \sim v$ under S . If it is the case, then add a query $LCR(u, v, S_k, G)$ to Q_t , which is used to accommodate positive queries; otherwise, add it to Q_f , which is used to accommodate negative queries. The cardinality of both Q_t and Q_f is bounded by l .

Table 7. Indexing Time and Size on Real Datasets

Datasets	CGB		LandM		Zou's		Hassan's	
	IT(s)	IS(M)	IT(s)	IS(M)	IT(s)	IS(M)	IT(s)	IS(M)
robots	0.8	0.1	0.1	5	7	1.457	6.02	3.2
yeast	2.1	0.5	0.41	6	14	3.0	13.08	11.23
yago	4.79	3.33	3.21	59	101	11.23	29.21	28.2
advogato	53	7.2	3	131	178	20.32	33.0	31.34
Youtube	1,699	92.2	2,831	300	713	245.15	727.11	693.1
BioGrid	142	12.8	50	1,302	3,007	1,011.1	203.35	73.2
yago2	1,749	1,076.08	5,3857	28,152	F	F	F	F
epinions	92	10.8	205	2,903	1,404	2,212	217.23	69.7
webStanford	510	28.6	935	7,806	1,710	6,766	640.37	231.28
webGoogle	701	153.6	5,887	33,931	5,056	22,130	979.02	502.15
webBerkstan	833	144.0	2,463	15,690	4,236	12,475	953.04	451.51
socPokec	1,612	512.3	9,762	75,698	F	F	2,302.71	2,307.23
wikiLinks	11,906	1,159	24,736	93,414	F	F	28,408.92	6,216.15
citeseerX	18,782	8,886	29,836	105,751	F	F	F	F

ALGORITHM 12: $queryGen(G, l)$ Input: G – a graph; l – size of query set;Output: Q_t – set of *true*-queries; Q_f – set of *false*-queries;**begin**

1. $Q_t := \phi; Q_f := \phi; l' := \lceil l/100 \rceil;$
2. **while** $|Q_t| \leq l$ or $|Q_t| \leq l$ **do** {
3. choose a random vertex u from G , and generate a random number $10 \leq r \leq n$;
4. **for** $j = 1$ to l' **do** {
5. choose a random vertex $v (\neq u)$ from G ; generate randomly 10 subsets of $\Sigma: S_1, \dots, S_{10}$;
6. **for** each S_k **do** {
7. run *BFS* to find whether $u \rightsquigarrow v$ under S_k ;
8. **if** it is the case and $dis(u, v) > r$ **then**
9. **if** $|Q_t| < l$ **then** $Q_t := Q_t \cup \{LCR(u, v, S_k)\}$;
10. **else if** $|Q_f| < l$ **then** $Q_f := Q_f \cup \{LCR(u, v, S_k)\}$;

end**8.3 Tests on Real Datasets**

In the first experiment, we compare the performance of our method with the *landM* [20], the *Zou's* [32], and the *Hassan's* on the real datasets described in Tables 4 and 5. We mainly report their respective **indexing times (IT)** and **index sizes (IS)**, as well as their **query times (QT)**. Indexing times and sizes are summarized in Table 7, which clearly show that ours uniformly outperforms the others; the *LandM*, the *Zou's*, and *Hassan's*. In the case of *Zou's*, we want to emphasize that it is only possible for small graphs to establish the indexes within the time limitation (four hours). For all the graphs with the number of edges close to 1M or above, it times out or collapses due to the system stack overflow. In addition, the indexing time of the *Hassan's* is comparable to ours, but better than all the other methods even though its index space is in general much larger than ours. However, the index space of the *Hassan's* is still much smaller than the *LandM* and *Zou's*. The reason for this is that by the *Hassan's*, not all the shortest paths are created as an index; instead,

Table 8. Breakdown of Indexing Time of CGB

	robots	yeast	yago	Advo- gato	You- tube	Bio- Grid	yago2	epinions	Web- Stanford	Web- Google	web- Berkstan	soc- Pokec	Wiki- Links	citeseerX
t_1 (s)	0.537	1.42	3.21	30.01	676.56	63.7	1025.77	64.57	207.52	347.98	367.34	689.43	4,236.23	6,987.56
t_2 (s)	0.263	0.68	1.58	22.99	1,022.4	28.3	723.23	27.43	302.48	353.02	465.66	922.57	7,669.77	11,794.44

Table 9. Query Time on Real Datasets

Datasets	<i>True query</i> (μ s)					<i>False query</i> (ms)				
	CGB	LandM	Zou's	BFS	Hassan's	CGB	LandM	Zou's	BFS	Hassan's
robots	0.13	20.12	8.90	17.77	3.19	0.01	0.06	0.02	0.70	0.12
yeast	0.31	45.81	23.23	56.78	6.73	0.09	0.11	0.06	3.23	0.13
yago	0.45	47.21	60.69	89.21	8.01	0.11	0.17	0.10	34.32	0.19
advogato	36.0	43.08	46.34	120.6	531.0	0.09	0.21	0.05	7.67	0.32
Youtube	8.50	21.10	25.80	3,220.6	2,324.04	8.50	8.19	3.07	38,88	10.9
BioGrid	11.9	23.71	35.23	1,754.3	420.05	1.9	11.95	3.45	29.0	16.4
yago2	71.5	292.73	F	16,547.7	F	22.31	121.64	F	269.8	F
epinions	20.8	52.10	65.77	106	989.8	1.1	1.58	1.02	2.31	1.7
webStanford	209.3	418.87	403.0	280	1,989.2	2.3	3.16	2.24	14.6	3.9
webGoogle	108.9	192.72	157.01	1,340	3,456.2	3.9	17.57	13.76	29.65	10.7
webBerkstan	116.4	139.52	146.78	995	3,743.0	7.4	20.78	13.66	27.6	30.0
socPokec	8.4	12.76	F	1,290	7,923.3	10.5	19.25	F	79.0	23.7
wikiLinks	31.4	49.06	F	3,120	12,432.2	11.4	25.42	F	101.9	45.9
citeseerX	46,7	67.98	F	5,769	F	13.02	209.16	F	363.56	F

graph G is partitioned into a set of *contracted paths* with each edge in it having the same label, plus some *bridge vertices* and *OtherHost Lists* [16], which can be used to speed up the navigation of G when evaluating a query. Such a data structure needs much less space than a transitive closure or a partial transitive closure.

Table 8 shows the breakdown of the indexing time of our method, which is mainly composed of two parts: the time t_1 for find spanning trees for G and the time t_2 for constructing compatible graphs.

In Table 9, we show the query times of our method and all the other four approaches. First, we notice that the query time for the *true*-queries is given in microseconds, while the query time for the *false*-queries is given in milliseconds. It is due to the huge difference between the query times of these two kinds of queries. From this, we can see that for *true*-queries, ours, the *LandM*, and the *Zou's* are slightly better than the *BFS* and the *Hassan's*. However, for *false*-queries, there is a big gap between the *BFS* and the others. The reason for this is that a *true*-query can stop after hitting its target, whereas by a *false*-query a large part of a graph has to be searched for the *BFS* to find that the source cannot reach to the target. In fact, the same analysis applies to the *LandM* even though it can be somehow better than the *BFS* in some cases. It is because by this method an index is constructed only for a small fraction of vertices and for most *false*-queries they are useless. To be clearer, consider graph *wikiLinks* with $n = 3.0M$. For it, the set of the chosen land marks will contain $1,250 + \sqrt{n} = 2,980$ vertices [20], which are only about 0.3% of all vertices. Then, for evaluating a query, the probability that the index is utilized is $(0.3 + 99.7 \times 0.3)/100 = 30.21\%$. That is, for a *false*-query, the probability that the *LandM* will search the whole *wikiLinks* is

Table 10. Maximum Length of *to*- and *from*-sequences of Vertices by CGB

	robots	yeast	yago	Advogato	YouTube	Bio-Grid	yago2	epinions	Web-Stanford	Web-Google	web-Berkstan	soc-Pokec	Wiki-Links	citeseerX
Max. Len.	24	60	82	91	233	396	546	96	107	76	167	301	423	388

Table 11. Average Length of Sequences Accessed and Average Number of Replacements when Evaluating a Query by CGB

	robots	yeast	yago	Advogato	YouTube	Bio-Grid	yago2	epinions	Web-Stanford	Web-Google	web-Berkstan	soc-Pokec	Wiki-Links	citeseerX
Ave. Num. interval	9	23	37	39	97	117	383	36	54	48	78	132	183	214
Ave. replacements	27	39	25	14	73	61	343	89	93	77	202	208	267	387

about 70%. So, for *false*-queries, the *LandM* is not much better than the *BFS*, and even worse than the *Hassan*'s. However, ours is still better than the *Hassan*'s although its index structure can be almost 5 times larger than ours. For small graphs, the *Zou*'s exhibits a great gain of the query time over both the *BFS* and the *LandM*, which shows that fully indexing a graph is beneficial, since in any case searching a large part of a graph can be avoided. Unfortunately, both its indexing time and index size are too large and cannot scale well on large graphs. In the opposite, by our method, the *to*-sequence and *from*-sequence of a vertex are quite short and when the *to*-sequence of the source, or the *from*-sequence of the target is exhausted, returns *false*. See Table 10 for some real tested lengths.

To show the reason why our index is much smaller than the others, in Table 10, we give the maximum lengths k of the *to*- and *from*-sequences associated with the vertices in different graphs by our method. Clearly, we can see that $k \ll n$. In Table 11, we show the average number of checked intervals and performed replacements by evaluating a query, from which we can observe the reason why our query time is much better than the others.

8.4 Tests on Synthetic Datasets

In this section, we report the tests on synthetic datasets. As mentioned before, this test is divided into three parts, which will be presented in Sections 8.4.1, 8.4.2, and 8.4.3, respectively.

8.4.1 Part I: Synthetic Graph Performance. In Part I, we compare the performance of our method with the *LandM* and the *Zou*'s on two groups of synthetic datasets created by using *gMark* [38]. For them, we choose $n = 250,000$ and $|\Sigma| = 10$. In the first group, the outdegree of vertices follows the uniform distribution, called *uniform-graphs*. Concretely, we will create four graphs, each with $P(d) = 1$ for a different $d \in \{2, 3, 4, 5\}$. Accordingly, the density of graphs is gradually increased. In the second group, the outdegree of vertices follows the Zipfian distribution ($P(d) = \alpha d^{-l}$), called *zipf-graphs*. In this graph, also four graphs are created, each with $\alpha = 1$, and $l = 2.2, 2.4, 2.6, 2.8$, respectively. Our goal here is to understand the impact of graph density on performance for both different synthetic 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 and indexes between vertices, as well as the number of minimal label sets, increase with density. Especially, for our method, the number of decomposed spanning trees will also be incremented. Table 12 and 13 summarize the results. From these, we can see that the index sizes of our method are much smaller than the *LandM* and *Zou*'s. It is because the indexes produced by

Table 12. Test Results on Uniform-graphs

d	CGB			LandM			Zou's		
	IT	IS	QT	IT	IS	QT	IT	IS	QT
2	10.2	17.2	1.1	170.3	45.02	75.2	745.1	372.3	54.4
3	20.1	25.6	1.4	219.3	67.5	89.3	901.4	350.7	73.0
4	31.2	29.1	2.3	438.9	80.3	219.8	1401.2	432.2	100.0
5	45.7	42.3	2.6	912.7	292.1	252.5	1695.1	481.1	127.2

For all the methods, IT is given in seconds, IS in Mbytes, and QT in milliseconds.

Table 13. Test Results on Zipf-graphs

l	CGB			LandM			Zou's		
	IT	IS	QT	IT	IS	QT	IT	IS	QT
2.8	17.03	6.9	0.12	13,3	107,5	90.7	1678.1	157.5	4.7
2.6	23.17	8.1	0.24	14,1	109.1	132.4	1710.7	166.3	7.4
2.4	26.2	11.05	0.47	14,7	112.3	200.2	1839.4	171.6	11.93
2.2	32.13	15	0.76	15.7	148.4	287.7	2120.7	183.8	15.02

For all the methods, IT is given in seconds, IS in Mbytes, and QT in milliseconds.

our method are highly compacted by using the multi-sets and the compatible graphs to represent path labels, which leads to a very short indexing time, since both the multi-sets and the compatible graphs can be constructed very fast. Our query time is also much better than *LandM* and *Zou's*. It includes a series of interval checks and a searching of several compatible graphs, whose cost is linear in the number of forward edges with respect to the found spanning trees.

In all cases, the *Zou's* method needs much more time than the *LandM* to establish an index. Its index sizes are also much larger than the *LandM's*. The reason for this is simple: The *Zou's* indexes are over all the vertices, while the *LandM's* only over part of vertices. This difference leads to a big difference between their query times. In many cases, especially for the Zipf-graphs, the *LandM's* query time can be 10 times higher than the *Zou's*.

8.4.2 Part II: Impact of Number of Edges and Label Set Size. We next test the performance of our method while varying the number of edges and labels using synthetic datasets. Again, we will use **uniform-graphs (UG)** and **Zipf-graphs (ZU)** of $n = 250,000$ with $|\Sigma|$ set to be different values: 8, 10, 12, 14, and 16. For *uniform-graphs*, $d = 2, 3, 4, 5$; and for *Zipf-graphs*, $\alpha = 1$ and $l = 2.2, 2.4, 2.6, 2.8$. Here, what we want is to better understand the impact of both the number of labels of a graph and the graph density on our method's performance. We expect that as graphs grow in either of these dimensions, indexing costs will increase, since the sizes of both multi-sets and compatible graphs will be enlarged. Especially, the number of recursive graph decompositions will also be increased as the density of graphs grows or more labels are attached to edges, leading to larger sequences associated with vertices as indexes.

Figures 16 and 17 show the indexing time, the index size, and the query time for the uniform-datasets and Zipf-datasets. When d is large, we observe that both the indexing time and index size rapidly grow as $|\Sigma|$ increases. However, when d is small, the growth of the indexing time and index size is slow. This shows that the size of multi-sets and compatibles increases exponentially in $|\Sigma|$ for large d and remains small when d is small, no matter what $|\Sigma|$ is.

The growths of the indexing time and index size are larger for the uniform-graphs than for the Zipf-graphs. This is because the uniform-graphs have a close uniformal out-degree distribution.

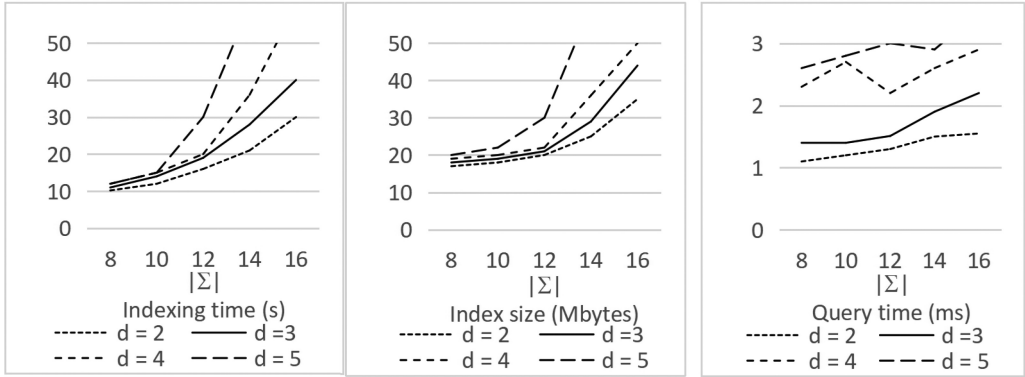


Fig. 16. Indexing time, index size, and average query times for uniform-graphs with $n = 250,000$, as a function of label set size $|\Sigma|$. The different curves indicate the vertex degree (either 2, 3, 4, or 5) of the datasets.

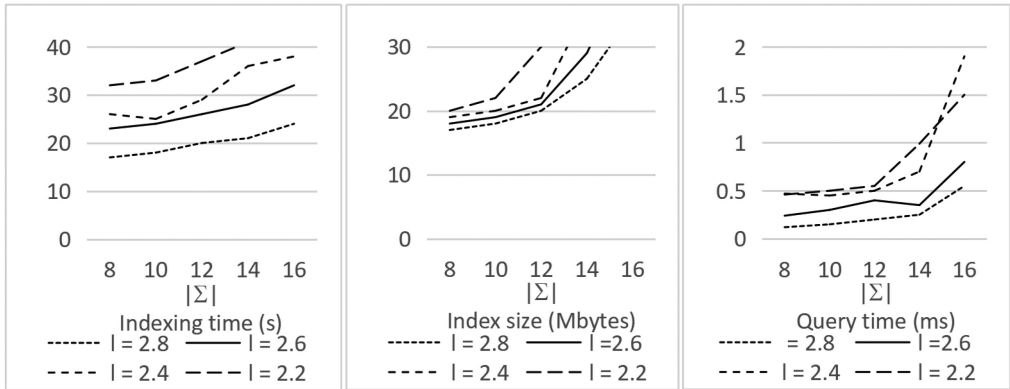


Fig. 17. Indexing time, index size, and average query times for Zipf-graphs with $n = 250,000$, as a function of label set size $|\Sigma|$. The different curves indicate the vertex degree (either 2, 3, 4, or 5) of the datasets.

In general, more paths connecting any two vertices imply larger multi-sets and larger compatible graphs.

Although the query time for the Zipf-graphs is in general lower than for the uniform-graphs, for small l values (then more vertices with larger out-degrees) their query time grows very fast as $|\Sigma|$ increases.

8.4.3 Part III: Impact of Graph Structures. Finally, we analyze the performance of our method on uniform- and Zipf-graphs as we vary the number of vertices $n \in \{1,000k, 1,500k, 2,000k, 2,500k, 3,000k, 3,500k, 4,000k\}$ while fix $|\Sigma| = 10$. For the uniform-graphs, $d = 5$; and for the Zipf-graphs, $l = 2.6$ (recall that $P(d) = \alpha d^{-l}$). Our goal here is to understand the scalability of our method.

Figure 18 shows the indexing time, index size, and the average query time. We can observe that all the three parameters for the uniform-graphs grow much faster than those for the Zipf-graphs. This can be explained by the fact that in a graph with a more uniform out-degree distribution, the average number of paths between any two vertices is higher than in a graph with more skewed out-degree distribution. If the number of paths increases between any two vertices, then so does the size of the corresponding compatible graph. Due to this effect, all the three parameters for the uniform-graphs increase faster.

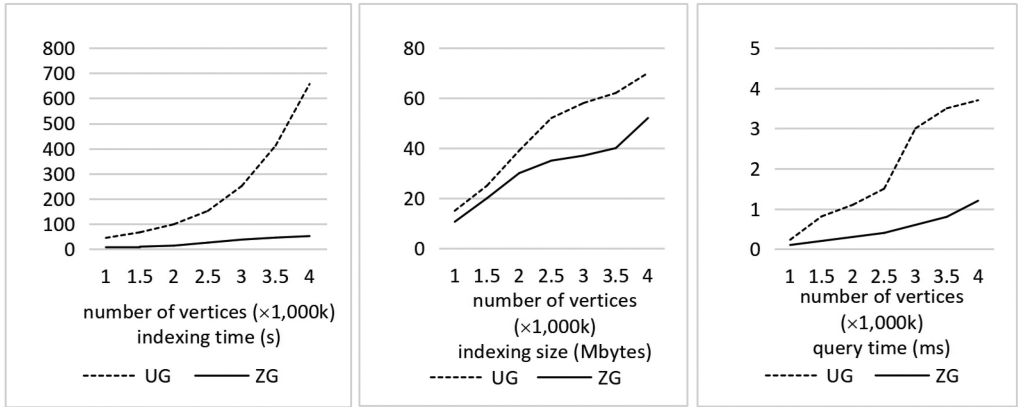


Fig. 18. Indexing time, index size, and average query times for uniform-graphs (UG) and Zipf-graphs (ZG), as a function of the number of vertices.

9 CONCLUSION

In this article, a new method to evaluate *LCR* queries is discussed. The main idea behind it is to decompose a graph G into a series of spanning trees T_0, \dots, T_{k-1} . Then, construct a series of tree-like subgraphs T_0', \dots, T_{k-1}' with each T_i' being T_i plus the corresponding forward and back edges. With respect to each T_i' , we recognize different kinds of vertices to evaluate queries and transfer reachability information efficiently. In this way, the index construction time and the index space can be, respectively, reduced to $O(\sum_{i=0}^{k-1} (m_i + \chi_i |\Sigma| + \chi_i h_i))$ and $O(\sum_{i=0}^{k-1} (|T_i| + \chi_i |\Sigma|) + \chi_i h_i)$ for DAGs, where m_i is $|T_i'|$ plus the number of all the corresponding cross edges with respect to T_i , χ_i and h_i are, respectively, the number of all forward edges and the maximum number of forward edges attached to a path in T_i ($i = 1, \dots, k - 1$), and Σ is the set containing all the edge labels of G . The query time is bounded by $O(\sum_{i=0}^{k-1} (h_i^2 + h_i |\Sigma|))$. For cyclic graphs, the index construction time and the index space are bounded by $O(\sum_{i=0}^{k-1} (m_i + b_i + \chi_i |\Sigma| + \chi_i h_i))$ and $O(\sum_{i=0}^{k-1} (|T_i| + b_i + \chi_i |\Sigma| + \chi_i h_i))$, respectively, where b_i is the maximum number of all those back edges $s \rightarrow t$ such that their end vertices t are on a same path in T_i' . The query time is bounded by $O(\sum_{i=0}^{k-1} b_i (h_i^2 + h_i |\Sigma|))$. Extensive experiments have been conducted that show our method is much better than all the existing methods in all the important aspects, including index construction times, index sizes, and query times.

ACKNOWLEDGMENT

The authors are grateful to the anonymous referees for their valuable comments.

REFERENCES

- [1] R. Agrawal, A. Borgida, and H. V. Jagadish. 1989. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the ACM SIGMOD International Conference*. 253–262.
- [2] T. Anderson. 1987. *Combinatorics of Finite Sets*. Clarendon Press, Oxford.
- [3] F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen. 2014. Distance oracles in edge-labeled graphs. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT'14)*. 547–558.
- [4] Y. Chen and Y. Chen. 2008. Core labeling: A new way to compress transitive closure. In *Proceedings of the International Conference on Signal Image Technology and Internet Based Systems*. 3–10.
- [5] Y. Chen and Y. Chen. 2008. An efficient algorithm for answering graph reachability queries. In *Proceedings of the 24th International Conference on Data Engineering*. IEEE, 893–902.
- [6] Y. Chen and Y. Chen. 2011. Decomposing DAGs into spanning trees: A new way to compress transitive closures. In *Proceedings of the 27th International Conference on Data Engineering*. IEEE, 1007–1018.

- [7] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. 2006. Fast computation of reachability labeling for large graphs. In *Proceedings of the International Conference on Extending Database Technology*. 26–31.
- [8] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.
- [9] J. Cai and C. K. Poon. 2010. Path-hop: Efficiently indexing large graphs for reachability queries. In *Proceedings of the Conference on Information and Knowledge Management (CIKM'10)*. 119–128.
- [10] T. H. Corman, C. E. Leieron, R. L. Rivest, and C. Stein. 2002. *Introduction to Algorithms*. McGraw Hill.
- [11] H. V. Jagadish. 1990. A compression technique to materialize transitive closure. *ACM Trans. Datab. Syst.* 15 (Dec. 1990), 558–598.
- [12] R. Jin, Y. Xiang, N. Ruan, and H. Wang. 2008. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 595–608.
- [13] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. 2010. Computing label-constraint reachability in graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 123–134.
- [14] R. Jin, N. Ruan, Y. Xiang, and H. Wang. 2011. Path-Tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Datab. Syst.* 36, 1 (2011), 1–52.
- [15] R. Jin, X. Yang, R. Ning, and F. David. 2009. 3hopp: A high compression indexing scheme for reachability query. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 813–826.
- [16] M. S. Hassan, W. G. Aref, and A. M. Aly. 2016. Graph indexing for shortest-path, finding over dynamic sub-graphs. In *Proceedings of the Special Interest Group on Management of Data Conference (SIGMOD'16)*.
- [17] I. Munro. 1971. Efficient determination of the transitive closure of directed graphs. *Inf. Proc. Lett.* 1, 2 (1971), 56–58, 1971.
- [18] M. E. J. Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemp. Phys.* 46, 5 (2005), 323–351.
- [19] M. Thorup. 2004. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM* 51, 6 (2004), 993–1024.
- [20] L. Valstar, G. Fletcher, and Y. Yoshida. 2017. Landmark indexing for evaluation of label-constrained reachability. In *Proceedings of the Special Interest Group on Management of Data Conference (SIGMOD'17)*. 14–19.
- [21] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. 2006. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22nd International Conference on Data Engineering*.
- [22] K. Xu et al. 2011. Answering label-constraint reachability in large graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*.
- [23] H. Yildirim, V. Chaoji, and M. J. Zaki. 2010. GRAIL: Scalable reachability index for large graphs. *Proc. VLDB Endow.* 3, 1 (2010) 276–284.
- [24] Y. Zibin and J. Gil. 2001. Efficient subtyping tests with PQ-encoding. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Application*. 96–107.
- [25] H. S. Warren. 1975. A modification of Warshall's algorithm for the transitive closure of binary relations. *Commun. ACM* 18, 4 (1975), 218–220.
- [26] S. J. van Schaik and O. de Moor. 2011. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the Special Interest Group on Management of Data Conference (SIGMOD '11)*. 913–924.
- [27] R. Jin, N. Ruan, S. Dey, and J. X. Yu. 2012. SCARAB: Scaling reachability computation on large graphs. In *Proceedings of the Special Interest Group on Management of Data Conference (SIGMOD'12)*. 20–24.
- [28] H. Wei, J. X. Yu, C. Lu, and R. Jin. 2014. Reachability querying: An independent permutation labeling approach. *Proc. VLDB Endow.* 7, 12 (2014).
- [29] R. R. Veloso, L. Cerf, W. Meira Jr, and M. J. Zaki. 2014. Reachability queries in very large graphs: A fast refined online search approach. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT'14)*. 511–522.
- [30] K. Mehlhorn. 1984. *Graph Algorithms and NP-completeness*. Springer-Verlag New York.
- [31] R. Jin and G. Wang. 2013. Simple, fast, and scalable reachability oracle. *Proc. VLDB Endow.* 6, 14 (2013).
- [32] L. Zou et al. 2014. Efficient processing of label-constraint reachability queries in large graphs. *Inf. Syst.* 40 (2014), 47–66.
- [33] A. Mendelzon and P. Wood. 1995. Finding regular simple paths in graph databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.
- [34] Z. Abul-Basher. Multiple-query optimization of regular path queries. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'17)*. IEEE, 1426–1430.
- [35] J. J. Kunegis. 2013. KONECT—The Koblenz network collection. In *Proceedings of the International Conference on World Wide Web Companion*. 1343–1350.
- [36] J. Leskovec and A. Krevl. 2019. SNAP datasets: Stanford large network dataset collection. Retrieved from <http://snap.stanford.edu/data>.

- [37] F. M. Suchanek, G. Kasneci, and G. Weikum. 2007. Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*. 697–706.
- [38] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. 2017. gMark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869.
- [39] M. N. Rice and V. J. Tsotras. 2010. Graph indexing of road networks for shortest path queries with label restrictions. *PVLDB* 4, 2 (2010), 69–80.
- [40] A. Bonifati, W. Martens, and T. Timm. 2017. An analytical study of large SPARQL query logs. *PVLDB* 11, 2 (2017), 149–161.
- [41] M. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. 2005. Lowest common ancestors in trees and directed acyclic graphs. *J. Algor.* 57, 2 (2005), 75–94.
- [42] S. Dumbrava, A. Bonifati, A. Diaz, and R. Vuillemot. 2019. Approximate evaluation of label-constrained reachability queries. *SUM* (2019) 250–265. arxiv.org/abs/1811.11561v1.
- [43] H. Wei, J. X. Yu, C. Lu, and R. Jin. 2014. Reachability querying: An independent permutation labeling approach. *Proc. of the VLDB Endow.* 7, 12 (2014).
- [44] J. Zhou, S. Zhou, J. X. Yu, H. Wei, Z. Chen, and X. Tang. 2017. DAG reduction: fast answering reachability queries. In *Proceedings of the Special Interest Group on Management of Data Conference (SIGMOD'17)*. 375–390.
- [45] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the Special Interest Group on Management of Data Conference (SIGMOD'14)*. 1323–1334.
- [46] J. Zhou et al. 2018. Accelerating reachability query processing based on DAG reduction. *J. VLDB* 27, 2 (2018), 271–296.
- [47] J. Su, Q. Zhu, H. Wei, and J. X. Yu. 2017. Reachability querying: Can it be even faster? *IEEE Trans. Knowl. Data Eng.* 29, 3 (2017), 683–697.
- [48] R. Angles, M. Arenas, P. Barcelo, A. Hogan, J. L. Reutter, and D. Vrgoc. 2016. Foundations of modern graph query languages. *CoRR*, abs/1610.06264 (2016).
- [49] P. Baeza. 2013. Querying graph databases. In *Proceedings of the Symposium on Principles of Database Systems*. 175–188.
- [50] C. Barrett, R. Jacob, and M. Marathe. 2000. Formal-language-constrained path problems. *SIAM J. Comput.* 30, 3 (2000), 809–837.
- [51] P. T. Wood. 2012. Query languages for graph databases. *ACM SIGMOD Rec.* 41, 1 (2012), 50–60.
- [52] S. Wadhwa, A. Prasad, S. Ranu, A. Bagchi, S. Bedathur. 2019. Efficiently answering regular simple path queries on large labeled networks. In *Proceedings of the Special Interest Group on Management of Data Conference (SIGMOD'19)*.
- [53] M. Chen, Y. Gu, Y. Bao, and G. Yu. 2014. Label and distance-constraint reachability queries in uncertain graphs. In *Proceedings of the International Conference on Database Systems for Advanced Applications*. 188–202.
- [54] A. Likhyan and S. Bedathur. 2013. Label constrained shortest path estimation. In *Proceedings of the Conference on Information and Knowledge Management (CIKM'13)*, 1177–1180.
- [55] N. Yakovets, P. Godfrey, and J. Gryz. 2016. Query planning for evaluating SPARQL property paths. In *Proceedings of the Special Interest Group on Management of Data Conference (SIGMOD'16)*, 1875–1889.
- [56] G. H. L. Fletcher, J. Peters, and A. Poulouvasilis. 2016. Efficient regular path query evaluation using path indexes. In *Proceedings of the International Conference on Extending Database Technology*. 636–639.
- [57] A. Gubichev et al. 2013. Sparqling Kleene: Fast property paths in RDF-3X. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems*.
- [58] M. A. Schubert and J. Taugher, Determining type, part, colour, and time relationship. *J. Computer* 16, 10 (special issue on Knowledge Representation), 53–60.
- [59] U. Feige. 1998. A threshold of $\ln(n)$ for approximating set cover. *J. ACM* 45, 4 (1998), 634–652.
- [60] K. Mehlhorn. 1986. *Graph Algorithm and NP-Completeness*, 2. Springer-Verlag.
- [61] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 2009. 3-HOP: A high-compression indexing scheme for reachability query. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 813–826.
- [62] *BioGrid*, Retrieved on 2018 from <http://thebiogrid.org>.
- [63] *robots*, Retrieved on 2018 from <http://tinyurl.com/gnexfof>.
- [64] *yeast*, Retrieved on 2018 from <http://vlado.fmf.uni-lj.si/pub/networks/data/>.
- [65] *yago and yago2*, Retrieved on 2018 from <http://www.mpi-inf.mpg.de/yago-naga/yago>.
- [66] *advogato*, Retrieved on 2018 from <http://networkrepository.com/soc-advogato.php>.
- [67] *gMark*, Retrieved on 2018 from <https://github.com/graphMark/gmark>.
- [68] T. Neumann and G. Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19 (2010), 91–113.

Received May 2020; revised February 2021; accepted February 2021