DAG Reduction: Fast Answering Reachability Queries

Junfeng Zhou[†,‡], Shijie Zhou[‡], Jeffrey Xu Yu[§], Hao Wei[§], Ziyang Chen[♭], Xian Tang[‡]
[†,‡,♭]{zhoujf,shijie,zychen,txianz}@ysu.edu.cn, [§]{yu,hwei}@se.cuhk.edu.hk

ABSTRACT

Answering reachability queries is one of the fundamental graph operations. The existing approaches build indexes and answer reachability queries on a directed acyclic graph (*DAG*) G, which is constructed by coalescing each strongly connected component of the given directed graph $\mathcal{G}$ into a node of G. Considering that G can still be large to be processed efficiently, there are studies to further reduce G to a smaller graph. However, these approaches suffer from either inefficiency in answering reachability queries, or cannot scale to large graphs.

In this paper, we study *DAG* reduction to accelerate reachability query processing, which reduces the size of G by computing transitive reduction (*TR*) followed by computing equivalence reduction (*ER*). For *TR*, we propose a bottom-up algorithm, namely *buTR*, which removes from G all redundant edges to get the unique smallest *DAG* G^t satisfying that G^t has the same transitive closure as that of G. For *ER*, we propose a divide-and-conquer algorithm, namely *linear-ER*. Given the result G^t of *TR*, *linear-ER* gets a smaller *DAG* G^ε in linear time based on equivalence relationship between nodes in G. Our *DAG* reduction approaches (*TR* and *ER*) significantly improve the cost of time and space, and can be scaled to large graphs. We confirm the efficiency of our approaches by extensive experimental studies for *TR*, *ER*, and reachability query processing using 20 real datasets.

1. INTRODUCTION

Given a directed graph $\mathcal{G}$, a reachability query $u? \rightsquigarrow v$ asks whether a node v is reachable from a node u. Answering reachability queries is one of the fundamental graph operations and has been extensively studied [1, 5, 12–16, 18, 21, 23–25, 27–30]. Its applications include social networks, biological networks, the Semantic Web, ontology, transportation networks, program workflows, etc. Due to its importance and the emergence of large graphs, it is still a challenging task for reachability queries to be answered faster with less index size and index construction time offline.

Observing that two nodes can reach each other in a strongly connected component (*SCC*) and can be identified in linear time w.r.t. the size of $\mathcal{G}$ [20], the existing methods focus on answering reachability queries on a directed acyclic graph (*DAG*) $G = (V, E)$ by coalescing *SCC*s of $\mathcal{G}$ into nodes of G, where $V(E)$ is the set of nodes (edges) of G. The size of G becomes smaller, but still can be large to be processed efficiently. To address this problem, there are studies to further reduce G to a smaller graph for reachability query processing. However, these approaches suffer from either inefficiency in answering reachability queries, or cannot scale to large graphs. In [12], Jin et al. proposed a *SCARAB* framework, which exacts, from a *DAG* G, a smaller "reachability backbone" G^b carrying the major reachability relationship. It is shown in [12] that existing algorithms can scale to large graphs based on *SCARAB*. However, the cost behind the scalability is large index size and more index construction time. The query performance is improved only for a few algorithms, such as the *GRAIL* algorithm [29], and degenerates for others due to its expensive search strategy. In [9], Fan et al. studied equivalence reduction (*ER*), where two nodes u and v are equivalent in a *DAG* G if (a) they can reach/be-reached-by the same set of nodes and (b) they cannot reach each other. The result of *ER* over G is a smaller graph G^e by replacing each set of equivalent nodes of G with a representative node in G^e. After *ER*, reachability queries can be processed more efficiently. However, the $\mathsf{compress_R}$ [9] algorithm on *ER* computation cannot scale to large graphs due to its high space complexity $O(|V|^2)$ and high time complexity $O(|V|(|V| + |E|))$.

Considering that reachability queries can be processed more efficiently after *ER*, but $\mathsf{compress_R}$ cannot scale to large graphs by directly computing *ER* from G, in this paper, we study *DAG* reduction to accelerate reachability query processing, which gets the result of *ER* by first computing *TR*. We show that given the result of *TR*, *ER* computation can be largely simplified by our newly proposed algorithms. However, *TR* computation itself is a non-trivial problem. Existing algorithms on *TR* computation, such as the naive *DFS* and *PTR* [19], cannot scale to large graphs either, due to their high space and time complexities. To address this problem, we further propose efficient algorithm on *TR* computation, such that both *TR* and *ER* computation can be scaled to large graphs. Our main contributions are as follows.

- For *TR*, we propose a new algorithm *buTR*, which first identifies from G a set of nodes from which all the redundant edges can be safely deleted. The result is a smaller graph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$. Then, we process nodes of G' in a bottom-up fashion to find all the remaining redundant edges. The time complexity of *buTR* is $O(|V| + |E| + d \triangle |V'|)$ where $d = |E|/|V|$ is the average degree of G, $\triangle$ is the average number of visited nodes for each processed node of V' in computing, and the space complexity is linear in $O(|V|)$. By *TR*, we obtain the unique smallest *DAG* G^t, which has the same transitive closure as that of G.
- For *ER*, we first show that equivalence relationship of two nodes on G can be determined by their neighbor nodes in $G^t = (V, E^t)$, rather than by all the set of nodes that can reach/be-reached-by the two nodes as $\mathsf{compress_R}$ does, and we have an algorithm *Sort-ER* to get *ER* G^ε of G^t with time complexity $O(|E^t| \log |V|)$. We further show that the equivalence relationship is a partition of V, and each

SIGMOD'17, May 14-19, 2017, Raleigh, NC, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05. . . $15.00
DOI: http://dx.doi.org/10.1145/3035918.3035927

node u is a unique constraint that their in-neighbors (out-neighbors) take it as their common out-neighbour (in-neighbor), we propose an algorithm *linear-ER* to get G^ε of G^t in $O(|V| + |E^t|)$ time.

- We conduct extensive experimental study. The experimental results show that our *TR* and *ER* approaches are much more efficient than existing ones and can be scaled to large graphs, and based on the result of *DAG* reduction G^ε, reachability queries can be answered faster, with less index sizes and index construction time.

2. PRELIMINARIES & THE PROBLEM

We model a graph as a directed graph $\mathcal{G}$, and focus on the *DAG* representation of $\mathcal{G}$, denoted as $G = (V, E)$, where V is the set of nodes and E the set of edges. Here, a node in G represents a strongly connected component (*SCC*) of $\mathcal{G}$, and an edge in G represents the edge from an *SCC* S_i to another *SCC* S_j if there is an edge from a node in S_i to a node in S_j. G can be constructed from $\mathcal{G}$ in linear time [20]. A reachability query over $\mathcal{G}$ can be answered using G, such that u can reach v over $\mathcal{G}$ iff u's *SCC* can reach v's *SCC* over G.

We use $in_G(u) = \{v|(v, u) \in E\}$ to denote the set of in-neighbor nodes of u, and $out_G(u) = \{v|(u, v) \in E\}$ the set of out-neighbor nodes of u. We define $in^*_G(u)$ as the set of nodes in G that can reach u where $u \notin in^*_G(u)$, and $out^*_G(u)$ the set of nodes in G that u can reach where $u \notin out^*_G(u)$. We call $in_G(u)/out_G(u)/in^*_G(u)/out^*_G(u)$ as u's *graph* parents/children/ ancestors/descendants w.r.t. a *DAG* G. In a similar way, we call $in_T(u)/out_T(u)/in^*_T(u)/out^*_T(u)$ as u's *tree* parent/children/ ancestors/descendants w.r.t. a tree T, respectively. We also call $out^*_G(u) \bigcup \{u\}$ the transitive closure of u, and denote it as *TC*(u). Given a *DAG* $G = (V, E)$, we use $X = \{1, 2, ..., |V|\}$ to denote a topological order (topo-order) of G, which can be got by a topological sorting on G. A topological sorting of G is a mapping $t : V \to X$, such that $\forall(u, v) \in E$, we have $t_u < t_v$, where $t_u(t_v)$ is the topo-order of $u(v)$ w.r.t. X. A topo-order X of G can be got in linear time $O(|V| + |E|)$ [19]. We show important notations in Table 1 for ease of reference.

Transitive Reduction (*TR*): Given a *DAG* $G = (V, E)$ and edge $(u, v) \in E$, we say (u, v) is redundant, if there exists a node w, such that u can reach v through w. The *TR* of G is the *unique smallest DAG* $G^t = (V, E^t)$ *without* redundant edges and has the same transitive closure (*TC*) as that of G [2]. E.g., G^t in Fig. 1(b) is the *TR* of G in Fig. 1(a), and all dashed edges in Fig. 1 (a) are redundant edges.

Equivalence Reduction (*ER*): Given a *DAG* $G = (V, E)$, two nodes u and $v(u \neq v)$ are said equivalent to each other on G, denoted as $u \equiv v$, iff $in^*_G(u) = in^*_G(v) \wedge out^*_G(u) = out^*_G(v)$. The *ER* of G is a *DAG* $G^e = (V^e, E^e)$, where a node $v_e \in V^e$ represents a set S_{v_e} of equivalent nodes that are equivalent to v in G, and an edge $(u_e, v_e) \in E^e$ represents the edge from a node of S_{u_e} to a node of S_{v_e} in G. Note that two nodes in the same set of equivalent nodes cannot reach each other due to that G is a *DAG*, and given a query $u? \rightsquigarrow v$, if $u \not\equiv v$, we can answer it by testing $u_e? \rightsquigarrow v_e$ over G^e.

Problem Statement: Given a *DAG* $G = (V, E)$, we study *DAG* reduction, which is to find the *smallest DAG*, G^ε, by *TR* and *ER*, where "smallest" means that G^ε has the same *TC* as that of G^e, but without redundant edges, i.e., G^ε is the *TR* of G^e. E.g., given G in Fig. 1(a), G^ε in Fig. 1(c) is the result of *DAG* reduction. As a comparison, the *ER* G^e of G may contain edges such as (v_3, v_{13}), (v_5, v_8), etc.

3. RELATED WORK

Existing algorithms working on G to answer reachability queries can be divided into two categories: (1) Label-Only and (2) Online-Search. By Label-Only, $u? \rightsquigarrow v$ can be answered by comparing labels of u and v. By Online-Search, $u? \rightsquigarrow v$ is answered by *DFS* at run-time, when it cannot be answered by labels of u and v.

Table 1: Table of notations

Notation	Description
$G = (V, E)$	a *DAG* with a node set V and an edge set E
$G^t = (V, E^t)$	G's *TR* with a node set V and an edge set $E^t \subseteq E$
$G^\varepsilon = (V^\varepsilon, E^\varepsilon)$	G^t's *ER* with a node set $V^\varepsilon \subseteq V$ and an edge set $E^\varepsilon \subseteq E^t$
X	a topo-order of a *DAG* G
t_v	node v's topo-order in X
T_X	the *LPM* tree w.r.t. a topo-order X
$\mathcal{T}_G$	a po-tree denoting the processing order of nodes in a *DAG* G
$in_G(v)(in_T(v))$	the set of graph parents (tree parent) of node v in a *DAG* G(tree T)
$in^*_G(v)(in^*_T(v))$	the set of graph (tree) ancestors of node v in a *DAG* G(tree T)
$out_G(v)(out_T(v))$	the set of graph (tree) children of node v in a *DAG* G(tree T)
$out^*_G(v)(out^*_T(v))$	the set of graph (tree) descendants of node v in a *DAG* G(tree T)

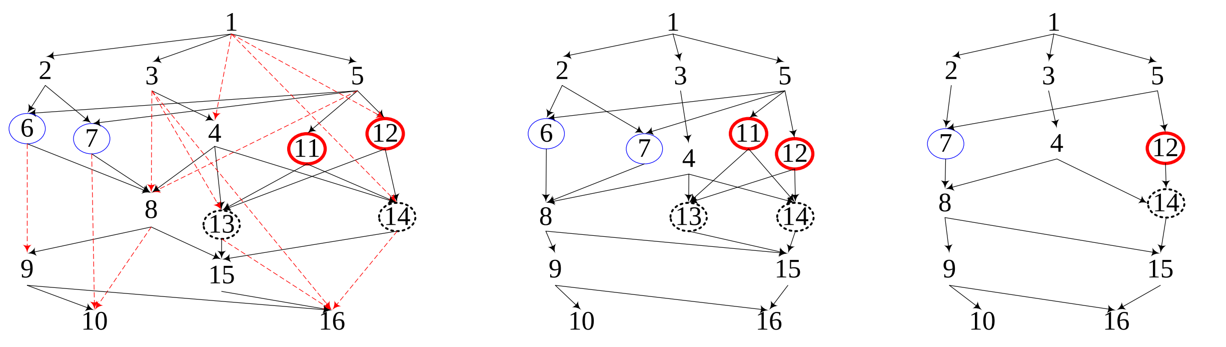
The Label-Only methods [1, 5, 13–16, 23, 27] focus on compressing *TC* to get a smaller index size for fast query processing. The recent work includes *TF* [5], *DL* [14], and *PLL* [27]. *TF* [5] folds the given *DAG* recursively based on topological level to reduce the cost of 2-hop computation. *DL* [14] and *PLL* [27] share the same idea of computing 2-hop label. Given all nodes in a certain order, the construction of *DL* and *PLL* labels is enumerating each node with a forward *BFS* and a backward *BFS* to add u to labels of nodes that u can reach and nodes that can reach u. During each *BFS*, an early stop condition is adopted to accelerate the computation and reduce the index size.

The Online-Search methods [18,21,24,25,28,29] answer $u? \rightsquigarrow v$ by performing *DFS* from u at run-time if needed. The recent work includes *GRAIL* [28,29], *FERRARI* [18], *FELINE* [24], and *IP*$^+$ [25]. All these methods use additional pruning strategies to facilitate query answering, such as comparing topological level of u and v [18, 24, 25, 28, 29], comparing topo-order [18], and comparing interval of u and v over a spanning tree [24].

Besides, there are studies focusing on reducing G to a smaller *DAG* to accelerate reachability query processing, including (1) *SCARAB* Framework, (2) transitive reduction and (3) equivalence reduction.

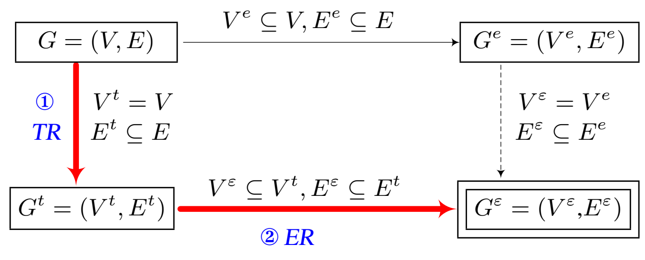
SCARAB **Framework** is studied in [12], which extracts from *DAG* G a "reachability backbone" G^b carrying the major reachability relationship of G. For each node $u \in G$, it maintains, in G^b, a set of local neighbor nodes $S_{in}(u)(S_{out}(u))$ that can reach (be reached by) u. Given a query $u? \rightsquigarrow v$, *SCARAB* returns the final answers in two cases: (Case 1) *Local-Search*: *SCARAB* performs bidirectional *BFS* search from u and v to check whether u can reach v. If the answer is FALSE, it answers the query by case 2. (Case 2) *Reachability-Join-Test*: *SCARAB* returns the final answer by testing $\bigvee_{u' \in S_{out}(u), v' \in S_{in}(v)}$ $u'? \rightsquigarrow v'$ using anyone of existing methods. It is shown in [12] that *SCARAB* can scale to large real graphs. The cost behind its scalability is large index size and more index construction time. Moreover, the query performance is improved only for a few algorithms, such as the *GRAIL* algorithm [29], for other algorithms, the query performance degenerates due to the expensive *Local-Search* in Case 1, and the need of testing $|S_{out}(u)| \times |S_{in}(v)|$ queries to answer $u? \rightsquigarrow v$ in Case 2.

Transitive Reduction has been extensively studied [2, 10, 19, 22, 26]. Compared with G, as *TC* of G^t equals that of G, the given reachability query on G can be answered using G^t directly. We discuss the complexities. First, for the time complexity to be measured as a function of the number of nodes in G, Aho et al. in [2] proved that transitive reduction and transitive closure have the same complexity using matrix multiplication, and the fastest known algorithm [26] takes time $O(|V|^{2.3727})$ with space $O(|V|^2)$, which is unacceptable when processing large graphs with limited memory, therefore is not considered for comparison in our experiment. Even though there exist algorithms [10, 22] achieving linear-time complexity to get G^t with the assumption that G is N-free, a linear-time recognition algorithm



(a) G (b) G^t (c) G^ε

Figure 1: *DAG* reduction: a *DAG* G (a), the *DAG* G^t from G by *TR* (b), and the reduced *DAG* G^ε by *ER* from G^t (c). Nodes are denoted by their topo-orders.



$G = (V, E)$
$V^e \subseteq V, E^e \subseteq E$
$G^e = (V^e, E^e)$
① TR
$V^t = V$
$E^t \subseteq E$
$V^\varepsilon = V^e$
$E^\varepsilon \subseteq E^e$
$G^t = (V^t, E^t)$
$V^\varepsilon \subseteq V^t, E^\varepsilon \subseteq E^t$
$G^\varepsilon = (V^\varepsilon, E^\varepsilon)$
② ER

Figure 2: Relationships between different graphs.

for N-free graphs is still an open problem, and G is not N-free in practice. Second, for the time complexity to be measured by the number of nodes/edges in G, the naive method is by depth-first search (*DFS*) or breadth-first search (*BFS*) in $O(|V||E|)$. Simon proposed a path-decomposition based transitive reduction (*PTR*) algorithm [19] to get the *TR* of a *DAG* G. Let k be the number of paths got from G by *PTR*, the time complexity of *PTR* is $O(|E| + k|V| + k|E^t|)$, and the space complexity is $O(k|V|)$. In practice, k is large that approaches $|V|$, which makes *PTR* cannot scale to large graphs. The problem of using G^t lies in the higher space and time cost in *TR* computation.

Equivalence Reduction is studied in [9] for reachability query processing, which reduces the given *DAG* G to get a smaller *DAG* G^e based on equivalence relationship. The $\mathsf{compress_R}$ algorithm [9] works as follows to get G^e. For each node $u \in V$, it first finds u's graph ancestors (descendants) by backward (forward) *BFS* with cost $O(|V| + |E|)$. Second, it identifies all the sets of equivalent nodes. Then, it replaces each set by one of its node to get the compressed graph G^e. For $\mathsf{compress_R}$, the time complexity is $O(|V|(|V| + |E|))$ and the space complexity is $O(|V|^2)$. Based on G^e, for a given query $u? \rightsquigarrow v$, if $u \equiv v$ then $u \not\rightsquigarrow v$. Otherwise, we answer $u? \rightsquigarrow v$ by testing $u_e? \rightsquigarrow v_e$ on G^e using any of existing methods, where $u_e(v_e)$ is the node in G^e denoting the set of nodes equivalent to $u(v)$ in G.

Compared with *SCARAB*, only one query needs to be tested over the compressed graph of *ER* for the given query on G. Compared with *TR*, *ER* computation removes from G not only edges, but also nodes. Usually in practice, reachability queries can be answered more efficiently after *ER*. However, the high space and time cost makes $\mathsf{compress_R}$ difficult to be scaled to large graphs for *ER* computation.

4. AN OVERVIEW ON DAG REDUCTION

Given a *DAG* G, Fig. 2 shows the relationships between G and its *ER* G^e, *TR* G^t and its *ER* G^ε. The output of $\mathsf{compress_R}$ [9] is G^e. The number of redundant edges of G^e depends on the insertion order of the edges when constructing G^e. In the worst case, G^e is the *TC* of G^ε. Compared with G^e by $\mathsf{compress_R}$, the result of our *DAG* reduction is G^ε, which is the *TR* of G^e without redundant edges. The benefit is that G^ε has the *minimum* storage representation w.r.t. the property that *TC* of G^ε equals that of G^e, thus analysis and visualization are more easier to be done [8]. Given a *DAG* G, although it has unique *TR* G^t, its *ER* without redundant edges may not be unique. This is because that each node v in G^ε represents a set P of equivalent nodes in G and v can be any node of P. All the *ERs* are isomorphic due to that all nodes of P are equivalent to each other.

4.1 Processing Strategy and Challenges

One way to get the result of *DAG* reduction G^ε is to first get G^e by $\mathsf{compress_R}$, then get G^ε by any one of existing algorithms on *TR* computation. However, $\mathsf{compress_R}$ is unscalable for *ER* computation due to its large time complexity $O(|V|(|V| + |E|))$ and space complexity $O(|V|^2)$. In brief, to check whether two nodes u and v are equivalent to each other, $\mathsf{compress_R}$ first finds the graph *ancestors* and *descendants* of each node by traversing from u and v, respectively. Second, $\mathsf{compress_R}$ checks whether u's graph ancestors and descendants are same as that of v. E.g., for G in Fig. 1(a), to check whether v_6 an v_7 are equivalent to each other, $\mathsf{compress_R}$ needs to first traverse from v_6 to find its graph ancestors $\{v_1, v_2, v_5\}$ and descendants $\{v_8, v_9, v_{10}, v_{15}, v_{16}\}$, respectively. Then, $\mathsf{compress_R}$ processes v_7 in the same way. With such results, $\mathsf{compress_R}$ takes the two nodes as equivalent ones by first comparing their graph ancestors then comparing their graph descendants.

To reduce the space and time cost of $\mathsf{compress_R}$, a natural question to ask is whether there exists a way such that the equivalence relationship of two nodes can be transformed from comparing the whole set of graph ancestors and descendants to comparing a small subsets of nodes, which is confirmed by the following lemma.

Lemma 4.1: *Let $A_i(u) \subseteq in^*_G(u)$ $(D_i(u) \subseteq out^*_G(u))$ be the subset containing all the nodes that can reach (be reached by) u through shortest paths with at most $i (i \geq 1)$ edges. Then, $\forall u, v \in V, \forall i \geq 1$, if G has no redundant edges, Eq. (1) and Eq. (2) hold.* □

$$A_i(u) = A_i(v) \Leftrightarrow in^*_G(u) = in^*_G(v) \quad (1)$$
$$D_i(u) = D_i(v) \Leftrightarrow out^*_G(u) = out^*_G(v) \quad (2)$$

Hereafter, all proofs can be found from Appendix A. Based on Lemma 4.1, we can get the result of *DAG* reduction by first *TR*, then *ER*, which is shown by the *bold* arrows in Fig. 2. E.g., for the *DAG* G in Fig. 1(a), we do not afford expensive cost to first get G^e by $\mathsf{compress_R}$. Instead, we first get G^t shown in Fig. 1(b), which does not contain redundant edges, then get the result of *DAG* reduction G^ε (Fig. 1(c)) from G^t. Since $A_1(u) = in_{G^t}(u)$ and $D_1(u) = out_{G^t}(u)$ are the smallest subsets to make Eq. (1) and Eq. (2) hold, we use them for *ER* computation. Compared with $\mathsf{compress_R}$, the benefits of computing *ER* based on G^t are twofold: (1) we significantly reduce the space from storing all graph ancestors and descendants for each

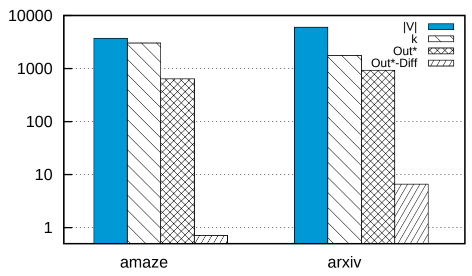
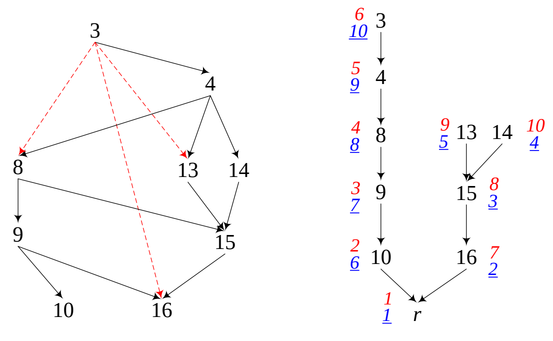


Figure 3: Statistics of a dense graph (`arxiv`) and a sparse graph (`amaze`), where Out* denotes $|out^*_G(\cdot)|$, Out*-Diff denotes the average value of $|out^*_G(u) - out^*_G(v_{\max})|$ for all nodes $u \in V$. $\forall u \in V$, $v_{\max}$ is the estimated graph child of u with the largest set of graph descendants.



(a) G' (b) $\mathcal{T}_{G'}$

Figure 4: The reduced graph G' (a) and its po-tree $\mathcal{T}_{G'}$ (b).

node by compress$_\mathsf{R}$ to graph parents and children; (2) we significantly reduce the time from comparing graph *ancestors and descendants* to comparing graph *parents and children*. E.g., given G^t in Fig. 1(b), both v_6 and v_7 have *two* graph parents and *one* graph child, which are less than their *three* graph ancestors and *five* descendants.

Even though we can get *ER* G^ε of G^t without affording the much more expensive time and space cost as compress$_\mathsf{R}$ does, it makes sense only if we can get the *TR* G^t of G first.

We discuss *PTR* [19]. *PTR* computes *TR* by first decomposing G into k paths, such that *TC*(u) can be represented by at most k nodes, where each one belongs to a different path. After that, *PTR* processes nodes of G in descending order w.r.t. a topo-order X, by which it knows *TC*$(v)(v \in out_G(u))$ when processing u. In detail, for every node u, it updates *TC*(u) using *TC*(v), where (u, v) is not a redundant edge. During processing, as it needs to remember *TC*(u) for every $u \in V$, *PTR* has space complexity $O(k|V|)$ and time complexity $O(|E| + k|V| + k|E^t|)$. Fig. 3 shows the statistics for two real graphs, one is `amaze`, the other is `arxiv` (see Appendix C for detailed description). `amaze` is a sparse graph with average degree $d = 0.97$, `arxiv` is a dense graph with $d = 11$. From Fig. 3 we know that for `amaze`, $k = 0.81|V|$, and for `arxiv`, $k = 0.29|V|$. The large k for *PTR* makes it unscalable in practice with limited memory size.

To reduce the space complexity of *PTR*, an alternative is *DFS* which has space complexity $O(|V|)$ due to the fact that it visits the set of all reachable nodes from each node on the fly. *DFS* randomly picks, in each iteration, a node u and visits all nodes of $out^*_G(u)$ to find redundant edges from u. Let $|out^*_G(\cdot)|$ be the *average* number of visited nodes for all nodes, the time complexity of *DFS* is $O(d|out^*_G(\cdot)||V|)$, where $d = \frac{|E|}{|V|}$ is the average degree of G. The efficiency of *DFS* is affected by two factors: (1) the number of processed nodes, for *DFS*, it is $|V|$, and (2) the average traversing cost, for *DFS*, it is $d|out^*_G(\cdot)|$, which is mainly dominated by the average number of visited nodes $|out^*_G(\cdot)|$ with the given *DAG* G. As shown in Fig. 3, for `amaze`, $|out^*_G(\cdot)| = 0.17|V|$, and for `arxiv`, $|out^*_G(\cdot)| = 0.15|V|$, which means that in practice, *DFS* may be inefficient due to that $|out^*_G(\cdot)|$ could be comparable to $|V|$.

As indicated by the time complexity of *DFS*, there are two critical problems we need to solve to achieve efficient *TR* computation: (1) designing efficient algorithm to identify all redundant edges from some nodes in linear time, such that to reduce the number of nodes that cannot be processed in linear time, and (2) designing efficient algorithm to reduce the traversing cost of remaining nodes. For the first problem, we propose a new spanning tree, namely *LPM* tree, then utilize the positional relationships between nodes to identify redundant edges in linear time. For the second problem, we propose new heuristics to estimate the number of reachable nodes for every node in linear time, then process nodes in a *bottom-up* fashion. For each node u, we only visit nodes of $out^*_G(u) \setminus out^*_G(v)$ on the fly, where v is the graph child node of u with the largest number of reachable nodes. In this way, the traversing cost is largely reduced with $O(|V|)$ space.

Our *DAG-Reduction* algorithm is shown in Algorithm 1 to obtain the *DAG* reduction G^ε for a given *DAG* G, which is done by first calling Algorithm 4 (Section 5) in line 1 to get the *TR* G^t, then calling Algorithm 5 (Section 6) in line 2 to return the final result G^ε. In the following discussion, we will first show the basic idea of our *TR* and *ER* algorithms in Section 4.2 and Section 4.3, then discuss more details of *TR* and *ER* computation in Section 5 and Section 6, respectively.

Algorithm 1: *DAG-Reduction* (G)

```
1  compute the TR $G^t$ of $G$ (Algorithm 4)
2  compute the ER $G^\varepsilon$ of $G^t$ (Algorithm 5)
```

4.2 Basic Idea of TR Computation

The basic idea of our method on *TR* computation is to reduce (1) the number of processed nodes, and (2) the average traversing cost. Compared with *DFS*, we do not need to process as many as $|V|$ nodes with average traversing cost as high as $d|out^*_G(\cdot)|$. Compared with *PTR*, the space complexity of our method is still $O(|V|)$.

We discuss how to reduce the number of processed nodes. The main idea is to first find, in linear time, a set of nodes called RRNs satisfying that (1) the redundant edges from RRNs are all removed and (2) the redundant edges from any non-RRNs can be identified without visiting RRNs. After that, *DFS* processes only non-RRNs to find the remaining redundant edges, which equals reducing the number of processed nodes. E.g., for G in Fig. 1(a), we find that $v_1, v_2, v_5, v_6, v_7, v_{11}$ and v_{12} are RRNs. After that, we remove these nodes and get a smaller graph G' in Fig. 4(a). Compared with G, $G' = (V', E')$ contains less nodes to be processed next.

We discuss how to reduce the average traversing cost. Let u be a graph parent of v, the main idea is based on the fact that $out^*_G(v) \subset out^*_G(u)$ reduces the traversing cost of u, if $out^*_G(v)$ is maintained. Obviously, to make the traversing cost *minimal* by maintaining the largest $out^*_G(v)$, we need to know the exact number of graph descendants for every node. However, knowing the exact size of $out^*_G(u)$ for all nodes $u \in V$ is non-trivial, it equals computing *TC* of G. We will show shortly that by our newly proposed heuristics, we can estimate the number of reachable nodes for every node in linear time, such that to make $out^*_G(u) \setminus out^*_G(v)$ as small as possible. Based on such estimation we construct a spanning tree, denoted as po-tree $\mathcal{T}_{G'}$, indicating the $\underline{p}$rocessing $\underline{o}$rder for nodes of G'. The po-tree $\mathcal{T}_{G'}$ is constructed by inserting each node u as a *tree* child of v, where v has the largest number of graph descendants in u's graph children. E.g., the po-tree of G' in Fig. 4(a) is given in Fig. 4(b). By the po-tree $\mathcal{T}_{G'}$, we process nodes of G' in a *bottom-up* fashion. After processing a node v, the next

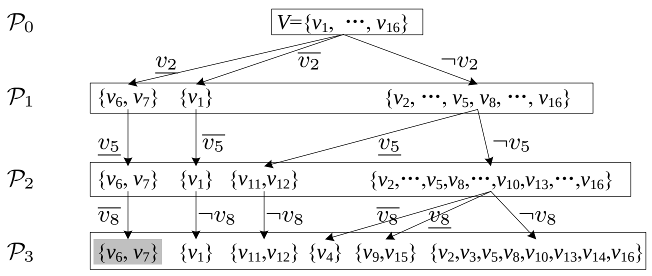


Figure 5: Illustration of the divide-and-conquer method, where $\underline{v_i}$ ($\overline{v_i}$) means that all nodes in the set under v_i are its graph children (parents), $\neg v_i$ means that all nodes in the set under v_i are neither graph parents nor children of v_i.

node to be processed is one of v's unprocessed graph parent u (u is a tree child of v in $\mathcal{T}_{G'}$). The efficiency of our algorithm is achieved as follows. During processing u, we do not need to visit all nodes of $out^*_G(u)$. Instead, we only visit nodes of $out^*_G(u) \setminus out^*_G(v)$ on the fly. Compared with *PTR*, we do not need to afford $O(k|V|)$ space to remember *TC* (u) for all nodes $u \in V$, and compared with *DFS*, the traversing cost of each node u can be reduced accordingly due to the fact that $|out^*_G(u) \setminus out^*_G(v)| \leq |out^*_G(u)|$. As shown in Fig. 3, for each processed node u, by avoiding visiting nodes that are reachable from its graph child v, we can reduce the *average* number of visited nodes from 639 by *DFS* to 0.71 for `amaze`, and reduce the *average* number of visited nodes from 928 by *DFS* to 6.6 for `arxiv`.

4.3 Basic Idea of *ER* Computation

We first give a sorting algorithm for a given G^t. Here, to compare if two nodes are equivalent, the algorithm relies on the sorting of all nodes by comparing their graph ancestors/descendants to speed up the process. However, it cannot be done in linear time.

To make further improvement, we propose a new linear divide-and-conquer algorithm, which takes initially all nodes in V as possible equivalent ones, then repeatedly divides this set into smaller ones satisfying that nodes in different sets are *definitely inequivalent*, while nodes in the same set are *possible equivalent*. All the sets form a partition of V. We show the idea using Fig. 5. Let $\mathcal{P}_0$ be the first partition. In each iteration, we randomly pick a node v_i, and use it to divide some sets in partition $\mathcal{P}_{i-1}$ into more subsets to get $\mathcal{P}_i$. In other words, a set $P \in \mathcal{P}_{i-1}$ will be divided by v_i into at most three disjoint subsets in $\mathcal{P}_i$, where the first set P_1 contains nodes that are graph children of v_i, denoted by $\underline{v_i}$ in Fig. 5, the second set P_2 contains nodes that are graph parents of v_i, denoted by $\overline{v_i}$, and the third set P_3 contains nodes that are neither graph parents nor children of v_i, denoted by $\neg v_i$. If $\exists P_i = \emptyset (i \in [1, 3])$, P is divided into two or even one set in $\mathcal{P}_i$. After processing all nodes, we get $\mathcal{P}_{|V|}$ containing all sets of equivalent nodes.

For example, given G^t in Fig. 1(b), $\mathcal{P}_0 = \{V\}$ initially. Assume that the first randomly selected node is v_2, the second is v_5 and the third is v_8. We first process v_2, which divides V into three sets to get $\mathcal{P}_1 = \{P_{11}, P_{12}, P_{13}\}$, where all nodes in $P_{11} = \{v_6, v_7\}$ are graph children of v_2, the single node in $P_{12} = \{v_1\}$ is a graph parent of v_2, and all nodes in $P_{13} = \{v_2, ..., v_5, v_8, ..., v_{16}\}$ are neither v_2's graph parents nor children. We then process v_5 based on $\mathcal{P}_1$ to get $\mathcal{P}_2 = \{P_{21}, P_{22}, P_{23}, P_{24}\}$. As all nodes in P_{11} are graph children of v_5, $P_{21} = P_{11}$. $P_{22} = P_{12}$ due to that v_1 is a graph parent of v_5. $P_{13} \in \mathcal{P}_1$ is divided into two sets, $P_{23} = \{v_{11}, v_{12}\}$ and $P_{24} = \{v_2, ..., v_5, v_8, ..., v_{10}, v_{13}, ..., v_{16}\}$, where nodes in P_{23} are graph children of v_5, and nodes in P_{24} are neither v_5's graph parents nor children. After processing v_8, we get $\mathcal{P}_3$ containing 6 sets. For each node v_i, the sets in leaf nodes of the tree in Fig. 5 in computing form the partition $\mathcal{P}_i (i \in [1, |V|])$. For each set $P \in \mathcal{P}_i$, we can find, from edges on the path between P and the root of the tree in Fig. 5, the set of

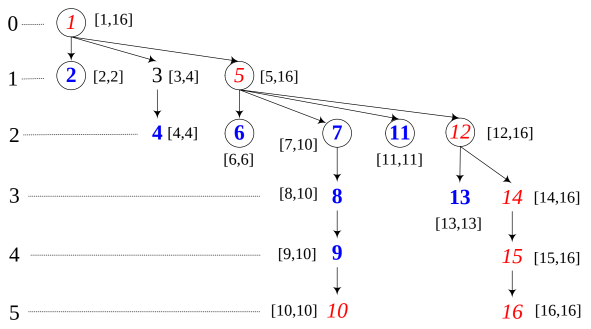


Figure 6: The *LPM* tree T_X of G generated from X. The integers on the left of T_X are topological levels, i.e., the length of the longest path ending at a node.

processed graph parents and children for all nodes of P. As each set of $\mathcal{P}_i$ has a distinct path to the tree root denoting a unique set of graph parents and children, two nodes from different sets of $\mathcal{P}_i$ are definitely inequivalent. The following processing is similar. After processing all nodes, we get $\mathcal{P}_{|V|}$. As each set of $\mathcal{P}_{|V|}$ cannot be further divided into smaller ones, all nodes in the same set of $\mathcal{P}_{|V|}$ are definitely equivalent to each other, i.e., $\mathcal{P}_{|V|}$ contains all sets of equivalent nodes. E.g., since v_6 and v_7 have the same set of graph parents and children, i.e., v_2, v_5 and v_8, shown on the path from the root to $\{v_6, v_7\}$ in $\mathcal{P}_3$ in Fig. 5, and there is no other nodes that take v_6 or v_7 as their graph parents or children according to G^t in Fig. 1 (b), we know that $\{v_6, v_7\} \in \mathcal{P}_{|V|}$, thus v_6 and v_7 are equivalent to each other.

5. TRANSITIVE REDUCTION

We discuss in this section more details on the optimizations for *TR*, including $(O1)$ marking nodes to reduce the size of the processed graph, and $(O2)$ estimating the number of graph descendants to construct a po-tree to reduce the average traversing cost.

5.1 O1: Marking Nodes of G

Given a node u in a *DAG* G, it is not possible to know whether an edge from u is redundant or not by scanning only u's graph children. The aim of this optimization is, based on a spanning tree, to identify a set of nodes in linear time $O(|V| + |E|)$ satisfying that all redundant edges from them are correctly identified, such that to get a smaller graph G' to be processed next. However, not any spanning tree is appropriate. Given a node u in a spanning tree T of G, if v is a tree descendant but *not* a tree child of u, then we can safely say that the non-tree edge (u, v) is a redundant edge, if it exists. If v is a tree child of u, then we *cannot* tell whether edge (u, v) is redundant or not easily.

5.1.1 The LPM Tree and the Marked Nodes

Given a topo-order X of *DAG* G, the *LPM* tree T_X is a spanning tree of G, where the incoming edge to a node v in T_X is from its *last* graph *parent* u, which has the *maximum* topo-order among v's graph parents in X. As an example, given G in Fig. 1(a) with its topo-order X, the *LPM* tree T_X is shown in Fig. 6. We have the following result.

Property 5.1: *Each edge of* T_X *is* not *a redundant edge.* □

As a comparison, the *DFS*/*BFS*-based spanning tree does not have this property. E.g., for G in Fig. 1(a), edge (v_3, v_{13}) may be an edge of both a *DFS* or *BFS*-based spanning tree. According to Property 5.1, we *only* need to focus on *non*-tree edges to find redundant ones.

Definition 5.1: (**Complete Node,** CN) Given a node $u \in V$, we say u is a CN of T_X, if $\forall v \in out^*_G(u), v \in out^*_{T_X}(u)$. □

Here, intuitively, if u is a CN, then u's graph descendants are its tree descendants, thus all edges from u pointing to nodes that are not u's tree child nodes in T_X are redundant edges. E.g., the CNs of T_X in Fig. 6 are $v_1, v_5, v_{10}, v_{12}, v_{14}, v_{15}$ and v_{16} according to Definition 5.1. Consider v_1, $out_G(v_1) = \{v_2, v_3, v_4, v_5, v_{12}, v_{14}\} \subseteq out^*_{T_X}(v_1)$. Since v_4 is a tree descendant but not a tree child of v_1, the edge (v_1, v_4) is redundant. Similarly, (v_1, v_{12}) and (v_1, v_{14}) are redundant edges.

Further, if u is not a CN, we may still have a chance to find all redundant edges from u. The main idea is to find every node u such that all redundant edges from u can be identified by the positional relationship between nodes of u's graph children. E.g., v_6 is not a CN and it has two graph children, v_8 and v_9. And we know that edge (v_6, v_9) is redundant due to that v_9 is a tree child of v_8 in T_X. We divide u's graph children into two disjoint sets, S_1 and S_2, satisfying that $out_G(u) = S_1 \bigcup S_2$, where S_1 contains nodes that are u's tree descendants in T_X, and $S_2 = out_G(u) \setminus out^*_{T_X}(u)$. As nodes of S_1 are u's tree descendants, we can easily find redundant edges between u and nodes of S_1 based on T_X and Property 5.1. We use Definition 5.2 to find redundant edges between u and nodes of S_2.

Definition 5.2: (**Reducible Node,** RN) $\forall u \in V$, let l_u be the topological level of u, i.e., the length of the longest path ending at u, and $l_{\min}(u) = \min\{l_v | v \in out_{T_X}(u)\}$ if $out_{T_X}(u) \neq \emptyset$, otherwise $l_{\min}(u) = \infty$. We say u is an RN of T_X, if $out_G(u) \setminus out^*_{T_X}(u)$ can be represented by C_1 and C_2 satisfying the following conditions:

1. $out_G(u) \setminus out^*_{T_X}(u) = C_1 \cup C_2$ and $C_1 \cap C_2 = \emptyset$,
2. $\forall v \in C_1, l_v \leq l_{\min}(u)$,
3. $\forall v, w \in C_1, t_v < t_w \Rightarrow l_v \geq l_w$,
4. $\forall w \in C_2, \exists v \in C_1$, such that $w \in out^*_{T_X}(v)$. □

In Definition 5.2, the first condition further divides S_2 into two disjoint sets C_1 and C_2. The second condition guarantees that there does not exist paths from nodes of $out_{T_X}(u)$ to nodes of C_1. The third condition guarantees that no edge exists between two nodes of C_1. By the second and third condition, we know that all edges from u to nodes of C_1 are *not* redundant edges. The fourth condition guarantees that any edge from u to a node of C_2 is a redundant edge. Therefore, if u is an RN, it means that we can find C_1 and C_2, such that all redundant edges from u can be correctly identified. It is worth noting that if u is a CN, then u is also an RN, and in this case, $out_G(u) \setminus out^*_{T_X}(u) = \emptyset$. E.g., all nodes but v_3 in Fig. 6 are RNs according to Definition 5.2. Consider v_3. Since $out_{T_X}(v_3) = \{v_4\}$, we have $l_{\min}(v_3) = l_{v_4} = 2$. Since $out_G(v_3) \setminus out^*_{T_X}(v_3) = \{v_8, v_{13}, v_{16}\}$, and none of the three nodes satisfies the second condition of Definition 5.2, v_3 is not an RN.

Definition 5.3: (**Removable** RN, RRN) Given an RN u, u is an RRN, if $in^*_G(u) = \emptyset$, or $\forall v \in in^*_G(u)$, v is an RN. □

Intuitively, given an RRN u, all redundant edges from each of u's graph ancestors have been correctly identified, thus we do not need to find redundant edges from any of u's graph ancestors again, and for any non-RN v, finding redundant edges from v will not visit u. Therefore, RRNs are useless for processing non-RNs and can be safely removed. E.g., v_3 in Fig. 6 is not an RRN since it is not an RN of T_X according to Definition 5.3. Even though v_4 is an RN of T_X, it is not an RRN, since $v_3 \in in^*_G(v_4)$ is not an RN of T_X. The 7 RRNs found in T_X are the circled nodes in Fig. 6, i.e., $v_1, v_2, v_5, v_6, v_7, v_{11}$ and v_{12}. After removing the 7 RRNs, the reduced G' is shown in Fig. 4(a).

5.1.2 The Algorithm

As shown by Algorithm 2, we first find all RRNs, then find all CNs from non-RRNs. If a node v is both a CN and RRN, it will be marked as an RRN due to that v is useless for post processing.

Consider RRN. Definition 5.3 implies that we need to check whether every node of $in^*_G(v)$ is an RN, in order to know whether v is an RRN. We give Lemma 5.1 to show that we only need to visit nodes of $in_G(v)$.

Lemma 5.1: *Given an* RN v, *v is an* RRN *iff $in_G(v) = \emptyset$, or every node of $in_G(v)$ is an* RRN. □

According to Lemma 5.1, to know whether v is an RRN, we first need to know whether it is an RN. To know whether v is an RN, we need to know whether all redundant edges from v can be correctly identified according to Definition 5.2. Given $w \in out_G(v)$, if edge (v, w) is redundant, there must exist a node $x \in out_G(v)$ satisfying $t_v < t_x < t_w \wedge x \rightsquigarrow w$. Here, $x \rightsquigarrow w$ is determined by their positional relationship in T_X. And the problem becomes $\forall w \in out_G(v)$, whether $\exists x \in out_G(v)(x \neq w)$, such that x is a tree ancestor of w. We use *DT*-order to make the cost of determining whether v is an RN *minimal* by visiting all nodes of $out_G(v)$ only *once*.

The *DT*-order: A *DT*-order is a *DFS*-based topo-order which visits all nodes of G in *DFS* way under the restriction of topological sorting, i.e., a node can be visited only if all its graph ancestors have been visited. E.g., the topo-order for nodes in Fig. 1(a) is a *DT*-order of G.

Lemma 5.2: *If the LPM tree T_X is generated based on a DT-order X of G, then X is also a DFS-order of T_X.* □

According to Lemma 5.2, visiting nodes of an *LPM* tree in *DFS*-order X equals visiting nodes of G in ascending topo-order X. We assign each node an interval $I_u = [s, e]$ to facilitate checking the ancestor-descendant relationship for nodes in T_X, where $I_u.s = t_u$, and $I_u.e$ is the maximum *DT*-order of u's tree descendants. $I_u \subset I_v$ means that v is a tree ancestor of u. The interval of each node in T_X is shown in Fig. 6. Note that we cannot have Lemma 5.2 if the given topo-order is not a *DT*-order, and the interval used in [11, 24, 29] does not have any relationship with topo-order.

Algorithm 2: *markCNRRN* $(G = (V, E))$

```
1  construct $T_X$
2  check whether $\forall v \in V$ is an RN by calling isRN($v, T_X$)
3  check whether $\forall v \in V$ is an RRN according to Lemma 5.1 in
   *ascending DT*-order
4  check whether every non-RRN is a CN in *descending DT*-order
5  **return** $G$ after removing RRNs

**Function isRN**($v, T_X$)
6  $I_x \leftarrow [0, 0]$; $l_{\min}(v) \leftarrow \infty$
7  **for each** ($w \in out_G(v)$ in ascending *DT*-order $X$ of $G$) **do**
8      **if** ($I_w \subset I_v$) **then**
9          **if** ($l_{\min}(v) > l_w$) **then** $l_{\min}(v) \leftarrow l_w$
10         **if** ($I_w \not\subset I_x$) **then** $I_x \leftarrow I_w$
11         **else** delete edge $(v, w)$
12     **else**
13         **if** ($I_w \not\subset I_x$) **then**
14             **if** ($l_{\min}(v) < l_w$) **then return** FALSE
15             **else** $I_x \leftarrow I_w$; $l_{\min}(v) \leftarrow l_w$
16         **else** delete edge $(v, w)$
17 **return** TRUE
```

In Algorithm 2, isRN() is used to check whether a node is an RN, which processes v's graph children in *ascending DT*-order X (line 7) to determine whether v is an RN by visiting its graph children only once. In isRN(), w is the current processed node, x is the last node processed before w satisfying that edge (v, x) is not redundant, $l_{\min}(v)$ denotes the smallest topological level for nodes processed before w. To know whether w has a tree ancestor in $out_G(v)$, we only need to test whether w is a tree descendant of x. If v is an RN, we know whether it is an RRN by visiting all its graph parents only once according to Lemma 5.1.

Consider CN. Let $x_u = \max \arg_v \{t_v | v \in out^*_G(u)\}$ be, among u's graph descendants, the one with the largest topo-order. We process all non-RRNs in descending order w.r.t. *DT*-order X. For each graph parent node u of the current processed node v, we update x_u using x_v. When processing u, we know u is a CN iff x_u is a tree descendant of u, which can be determined by comparing their intervals.

After identifying RRNs and CNs, we remove all RRNs from G and return G in line 5 as the reduced graph G' to be processed next.

Example 5.1: Given G in Fig. 1(a), Algorithm 2 first constructs the *LPM* tree as shown in Fig. 6, then marks all nodes but v_3 as RNs. After that, Algorithm 2 finds 7 RRNs, i.e., $v_1, v_2, v_5, v_6, v_7, v_{11}$ and v_{12}. Finally, it finds all CNs from non-RRNs, and returns G without the 7 RRNs as the reduced graph G' shown in Fig. 4(a). □

5.1.3 Analysis

Theorem 5.1: *Given an LPM tree T_X of G, Algorithm 2 correctly finds all* RRN*s and* CN*s.* □

With *DT*-order we have the *LPM* tree constructed in linear time $O(|V| + |E|)$ (line 1). Given a node v, isRN() visits v's graph children once (line 2), and v's graph parents are also visited once to determine whether it is an RRN (line 3). Therefore, to find all RRNs, Algorithm 2 visits $\sum_{v\in V}(|in_G(v)| + |out_G(v)|) = 2 \times |E|$ edges. For CN, we need to visit graph parents of every node once.Therefore, the time complexity of Algorithm 2 is $O(|V| + |E|)$.

5.2 O2: Estimating # of Graph Descendants

We process each node based on one of its graph child to reduce the traversing cost. Consider G' in Fig. 4(a). If we process v_3 after v_{16}, we need to visit all nodes in $out^*_G(v_3) \setminus out^*_G(v_{16})$, which contains 8 nodes. As a comparison, if we process v_3 after v_4, we need to visit nodes in $out^*_G(v_3) \setminus out^*_G(v_4)$, which contains only 1 node. To minimize $out^*_G(u) \setminus out^*_G(v)$, where v is a graph child of u, we need to know the exact size of $out^*_G(u)$ for each node u. However, knowing the exact number of graph descendants for all nodes is non-trivial, since it needs to compute *TC* of the given *DAG*.

Suppose that u is the graph parent of v, [30] proposed heuristics to estimate the lower and upper bounds (denoted as lb and ub, respectively) of $out^*_G(u)$. The lower bound of u is obtained by summing up the contributions of u's graph children, where each graph child v contributes $\frac{1}{|in_G(v)|}$ of its lower bound to u. If $|out_G(u)| = 1$ and $|in_G(v)| > 1$, the lower bound of u may be less than that of v, which may result in v is not the one wanted for u. On the other hand, the upper bound of u is the sum of the upper bounds of u's graph children. As many nodes share the same set of graph descendants, the upper bound of u may be much larger than the exact result, which cannot help us to select the appropriate v for u. [6] proposed to estimate the number of graph descendants for all nodes by performing k random permutations, to guarantee the difference between the estimated size and the accurate size is bounded with certain probability. The larger the k, the better the estimated results. As the cost of each random permutation is $O(|V| + |E|)$, the overall cost is $O(k(|V| + |E|))$. Even though we can get a better estimation, the larger k value may result in inefficiency for transitive reduction.

Here, with our *LPM* tree T_X, we can get an estimation in linear time $O(|V| + |E|)$, to significantly accelerate *TR* computation. Let $C_u \subseteq out_{G'}(u)$ be the set of u's graph children that do not have tree ancestors in $out_{G'}(u)$ w.r.t. T_X[1], $\widetilde{N}(u)$ the estimated size of $out^*_G(u)$, and $v_{\max}$ the node with the largest estimated number of graph descendants in C_u. We give two heuristics to estimate the sizes.

(H1) Using the sum of sub-tree sizes as the estimation. We take $|C_u| + \sum_{v\in C_u} |out^*_{T_X}(v)|$ as the lower bound of $|out^*_G(u)|$.

(H2) Using $v_{\max}$ to estimate $\widetilde{N}(u) = |C_u| + \widetilde{N}(v_{\max})$.

Example 5.2: For **H1**, consider v_{14} in Fig. 4(a), which is a CN of T_X in Fig. 6. $C_{v_{14}} = \{v_{15}\}$, we know that $|C_{v_{14}}| + |out^*_{T_X}(v_{15})| = 2 = |out^*_G(v_{14})|$. However, some leaf nodes of the *LPM* tree may have graph descendants in the given *DAG*. If some nodes of C_u are leaf nodes of the *LPM* tree, then the estimated results may be far from accurate. E.g., for v_3 in Fig. 4(a), if the three redundant edges from v_3 do not exist, then $C_{v_3} = \{v_4\}$, and we have that $|C_{v_3}| + |out^*_{T_X}(v_4)| = 1 < |out^*_G(v_3)| = 8$.

For **H2**, consider v_4 in Fig. 4(a). $C_{v_4} = \{v_8, v_{13}, v_{14}\}$. Suppose that $\widetilde{N}(v_8) = 4$ and $\widetilde{N}(v_{13}) = \widetilde{N}(v_{14}) = 2$. With **H2**, we can get the estimated size of v_4, i.e., $\widetilde{N}(v_4) = |C_{v_4}| + \widetilde{N}(v_8) = 3 + 4 = 7$, which is the accurate result. However, when the set of subtrees of **H1** have similar sizes, or when most edges from u to nodes of C_u are redundant edges, **H2** may get results smaller or larger than that of **H1**, or even larger than $|out^*_G(u)|$. In practice, the result of **H2** is smaller than the accurate result, because each redundant edge $(u, v)(v \neq v_{\max})$ can make $\widetilde{N}(u)$ increased by one, but it may make $\widetilde{N}(u)$ decreased by $|out^*_{T_X}(v)|$. □

As can be seen above, **H1** and **H2** are complementary to each other. When one gets a smaller result, the other usually gets a larger value. Also **H1** gets a lower bound of $|out^*_G(u)|$ and **H2** gets a result that is usually smaller than the accurate result. We take the larger value of **H1** and **H2** as the estimated result. By summarizing the above description, we estimate the approximate size of $out^*_G(u)$ based on Eq. (3).

$$\widetilde{N}(u) = \begin{cases} |out^*_{T_X}(u)|, & u \text{ is a CN and } |out^*_{T_X}(u)| \geq \widetilde{N}(v_{\max}) \\ \widetilde{N}(v_{\max}), & u \text{ is a CN and } |out^*_{T_X}(u)| < \widetilde{N}(v_{\max}) \\ \max\{|C_u| + \widetilde{N}(v_{\max}), \\ \quad |C_u| + \sum\limits_{v\in C_u} |out^*_{T_X}(v)|\}, & \text{otherwise} \end{cases} \quad (3)$$

There are three cases in Eq. (3). (Case-1) u is a CN and $|out^*_{T_X}(u)| \geq \widetilde{N}(v_{\max})$, we take $|out^*_{T_X}(u)|$ as the accurate result, which can be got in $O(1)$ time, i.e., $|out^*_G(u)| = I_u.e - I_u.s$. (Case-2) u is a CN but $|out^*_{T_X}(u)| < \widetilde{N}(v_{\max})$, we take $\widetilde{N}(v_{\max})$ as the estimated result, to guarantee that $\forall v \in out_{G'}(u)$, $\widetilde{N}(u) \geq \widetilde{N}(v)$. (Case-3) u is not a CN, we take the larger value of $\{|C_u| + \widetilde{N}(v_{\max}), |C_u| + \sum_{v\in C_u} |out^*_{T_X}(v)|\}$ as the estimated result, which guarantees that $\widetilde{N}(u)$ is not smaller than the lower bound.

Consider G' in Fig. 4(a). $|out^*_G(v_3)| = 8$, the estimated results by our method is 11. For all other nodes, our estimated results are the same as the accurate results. As a comparison, since v_3 has many graph descendants taking v_{15} as their graph descendants, the ub [30] method will get inaccurate results by counting estimated results of v_{15} several times. For v_3, the estimated result of ub is 16. On the contrary, lb [30] is also inaccurate for the similar reason. [6] estimates the results based on k random permutations, its accuracy depends on the value of k, the larger the value the more accurate results it can get. However, a larger value for k means unaffordable cost for estimation.

[1] C_u is defined without topological levels, while C_1 in Definition 5.2 is defined with topological levels. E.g., for node v_3 in Fig. 6, $C_{v_3} = \{v_4, v_8, v_{13}, v_{16}\}$, while $C_1 = \emptyset$ for v_3.

We use Algorithm 3 to make the estimation and generate the po-tree, which processes all nodes in descending *DT*-order, such that when processing a node u, we have the estimated values for u's graph children. For each node u, we use Eq. (3) to estimate the number of graph descendants, then insert u into the po-tree $\mathcal{T}_{G'}$ as a tree child of $v_{\max}$.
$\underline{o}$rder for nodes of G'. After processing all nodes, we get the estimated values for all nodes and the po-tree $\mathcal{T}_{G'}$ as well. The po-tree of G' in Fig. 4(a) is given in Fig. 4(b). For each node, we visit its child nodes only once, thus the time complexity of Algorithm 3 is $O(|V| + |E|)$.

Algorithm 3: *genPoTree* $(G' = (V', E'))$

```
1 initialize the po-tree $\mathcal{T}_{G'}$ with a single root node $r$
2 for each ($u \in V'$ in descending DT-order $X$ of $G'$) do
3     if ($out_{G'}(u) = \emptyset$) then $v_{\max} \leftarrow r$
4     else $v_{\max} \leftarrow \text{maxarg}_{v \in out_{G'}(u)} \{\widetilde{N}(v)\}$
5     compute the value of $\widetilde{N}(u)$ using Eq. (3)
6     insert $u$ as a tree child of $v_{\max}$ in $\mathcal{T}_{G'}$
7 return $\mathcal{T}_{G'}$
```

5.3 The Algorithm for *TR*

5.3.1 The Processing Strategy

As shown by Algorithm 4, *buTR* first outputs a smaller graph G' in line 1. In line 2, it generates a po-tree $\mathcal{T}_{G'}$ based on Eq. (3). Then, it processes nodes of G' in a bottom-up fashion. After processing a node v, the next node to be processed is one of v's unprocessed graph parent u (u is a tree child of v in $\mathcal{T}_{G'}$). We divide nodes of $out^*_G(u)$ into two sets, $out^*_G(v)$ and $out^*_G(u) \setminus out^*_G(v)$, and process them separately.

Processing Nodes of $out^*_G(v)$ (line 10): We use a *flag* for each node to denote whether it belongs to $out^*_G(v)$ or not. We explain our idea using Fig. 7. Here, assume that all nodes of $out^*_G(v)$, i.e., S_1, in Fig. 7 are graph descendants of v. When processing u_1, we know that edge (u_1, y_4) is redundant, since $(u_1, v) \in E$ and $y_4 \in out^*_G(v)$.

Processing Nodes of $out^*_G(u) \setminus out^*_G(v)$ (lines 11-15): We first pass u as another flag to each of u's graph child $y \in out_{G'}(u) \setminus out^*_G(v)$ (line 11), then mark all nodes of $out^*_G(u) \setminus out^*_G(v)$ by either *DFS* or *BFS*, indicating that they are graph descendants of u (lines 12-15). During *DFS*/*BFS*, edge (u, y) is redundant if we encounter a node $y \in out_{G'}(u) \setminus out^*_G(v)$ marked by u. Reconsider Fig. 7. When processing u_1, we pass u_1 to y_1 and y_2. When we encounter y_2, we observe u_1 by *DFS*/*BFS* from y_1. Thus (u_1, y_2) is redundant due to that u_1 can reach y_2 through y_1.

5.3.2 Avoiding the Rollback Operation

Consider Fig. 7, where $y_3 \notin out^*_G(v)$. After processing u_1, if we set the flag value of y_3 as TRUE denoting that $y_3 \in out^*_G(u_1)$, then when we precede to u_2, we may wrongly take (u_2, y_3) as a redundant edge due to (1) we process u_2 based on v, and (2) y_3 is marked as TRUE before processing u_2 indicating that $y_3 \in out^*_G(v)$. To avoid such a problem, we need to remember S_2, and rollback its status to FALSE before preceding to u_2, which is to visit S_2 again. We use *DT*-order and the *reverse* *DT*-order to avoid the rollback operation.

The Reverse *DT*-order: Recall that a *DT*-order visits all nodes of G in *DFS* way under the restriction that a node can be visited only if all its graph ancestors have been visited. Given a *DT*-order X of G, we have its *reverse* *DT*-order, denoted as $\overline{X}$, which can be got in $O(|V| + |E|)$ time by visiting nodes of G in *DFS* way in the *reverse* order of X under the restriction of topological sorting[2].

[2]The two topo-orders used in *FELINE* [24] are not *DT*-orders, and the cost of getting the second one is $O(|V| \log |V| + |E|)$.

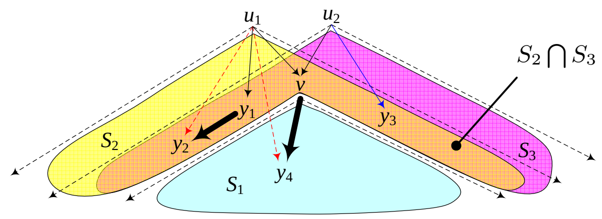


Figure 7: Relationship of transitive closures of different nodes, where bold (thin) arrows denote paths (edges), $out^*_G(v) = S_1$, $out^*_G(u_1) = S_1 \cup S_2$, $out^*_G(u_2) = S_1 \cup S_3$.

This is done as follows based on a stack S. For all the nodes without incoming edges, we push them into S in *ascending* *DT*-order X. When a node u is popped out from S, we assign u its $t_{\overline{u}}$ in $\overline{X}$, which is equal to the order it is popped out from S. After that, we push u's graph children that take u as their unique graph parent into S in *ascending* *DT*-order X, and remove u and all its outgoing edges. Such operation is performed repeatedly until S becomes empty. E.g., given the graph $\mathcal{T}_{G'}$ in Fig. 4(b) and the *DT*-order Z denoted as the red italic integers (topological sorting is performed from tree ancestor to descendants), the *reverse* *DT*-order $\overline{Z}$ is denoted as the *underlined* integer beside each node. And we have the following result.

Lemma 5.3: *Given DT-orders Z and $\overline{Z}$ of a tree T, let $t_u(t_{\overline{u}})$ be the topo-order of node u in $Z(\overline{Z})$, then nodes u and v do not have ancestor-descendant relationship iff $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$.* □

Lemma 5.4: *Let $N_1(N_2)$ be the number of visited nodes and edges for all nodes of $\mathcal{T}_{G'}$ processed in ascending DT-order $Z(\overline{Z})$ by* **buTR**, *then $N_1 = N_2$.* □

Our solution to avoid the rollback operation is based on Lemma 5.3 and Lemma 5.4. With Lemma 5.4, we can process nodes of $\mathcal{T}_{G'}$ in ascending order w.r.t. any *DT*-order of $\mathcal{T}_{G'}$. Assume that u_1 and u_2 do not have ancestor-descendant relationship in $\mathcal{T}_{G'}$, and (u_1, v) and (u_2, v) are two edges in $\mathcal{T}_{G'}$. With Lemma 5.3, if $t_{\overline{u_1}} < t_{\overline{u_2}}$, then $t_{u_1} > t_{u_2}$. Therefore, we process nodes of $\mathcal{T}_{G'}$ in *ascending* *DT*-order $\overline{Z}$, but mark nodes with their *DT*-orders in Z. We process u_1 and mark nodes of $out^*_G(u_1) \setminus out^*_G(v)$ with t_{u_1}. Then, when processing u_2, we can directly mark nodes of $out^*_G(u_2) \setminus out^*_G(v)$ with t_{u_2}. In this way, we guarantee that a node, if it needs to be marked more than once, is marked with values in *descending* order, such that we know which node set it belongs to, and therefore avoid the rollback operation on nodes of $out^*_G(u) \setminus out^*_G(v)$, i.e., we visit each node of $out^*_G(u) \setminus out^*_G(v)$ only once, instead of twice. We explain it using an example.

Example 5.3: Consider v_{14} and v_{13} in Fig. 4(b), where $t_{\overline{v_{14}}} = 4 < t_{\overline{v_{13}}} = 5$ and $t_{v_{14}} = 10 > t_{v_{13}} = 9$. Assume that edges (v_{13}, v_{10}) and (v_{14}, v_9) also exist in Fig. 4(a). We process v_{14} before v_{13}. When processing v_{14}, we mark nodes of $out^*_G(v_{14}) \setminus out^*_G(v_{15}) = \{v_9, v_{10}, v_{15}\}$ with $t_{v_{14}} = 10$. At this point, we know that nodes with flag values ≤ 10 are v_{14}'s graph descendants, among which those with flag values $= 10$ (v_9, v_{10} and v_{15}) belong to $out^*_G(v_{14}) \setminus out^*_G(v_{15})$, and those with flag values < 10 belong to $out^*_G(v_{15})$. Next, we process v_{13}, we mark nodes of $out^*_G(v_{13}) \setminus out^*_G(v_{15}) = \{v_{10}, v_{15}\}$ with $t_{v_3} = 9$. Here, we know nodes with flag values ≤ 9 are v_{13}'s graph descendants, among which those with flag values $= 9$ belong to $out^*_G(v_{13}) \setminus out^*_G(v_{15})$, and those with flag values < 9 belong to $out^*_G(v_{15})$. As a result, nodes in $out^*_G(v_{14}) \setminus out^*_G(v_{15})$ and $out^*_G(v_{13}) \setminus out^*_G(v_{15})$ are visited only once. □

5.3.3 Analysis

Theorem 5.2: *Given a DAG G, Algorithm 4 correctly identifies all redundant edges.* □

Algorithm 4: *buTR* $(G = (V, E))$

```
1  $G' \leftarrow$ markCNRRN$(G)$
2  $\mathcal{T}_{G'} \leftarrow$ genPoTree$(G')$
3  for each $(u \in out_{\mathcal{T}_{G'}}(r)$ in descending DT-order $Z)$ do
4      processTreeChild$(u, r)$
5  return $G$ after removing redundant edges
Procedure processTreeChild$(u, v)$
6  delRdtEdge$(u, v)$
7  for each $(x \in out_{\mathcal{T}_{G'}}(u)$ in descending DT-order $Z)$ do
8      processTreeChild$(x, u)$
Procedure delRdtEdge$(u, v)$  /*$\forall x \in V$, $flag[x]=\infty \wedge edge[x]$=-1 initially*/
9  for each $(w \in out_{G'}(u))$ do
10     if $(flag[w] \leq t_v)$ then delete edge $(u, w)$
11     else $flag[w] \leftarrow t_u$; $edge[w] \leftarrow u$
12 for each $(w \in out_{G'}(u) \setminus (\{v\} \bigcup out^*_G(v)))$ do
13     visit nodes of $out^*_G(u) \setminus out^*_G(v)$ by DFS/BFS from $w$
14     set the flag value of each visited node $x$ as $t_u$
15     delete edge $(u, x)$, if $edge[x] = u (x \neq w)$
```

For each node u in the po-tree $\mathcal{T}_{G'}$, the number of visited nodes for u is $\triangle_{u,v_{\max}} = |out^*_G(u) \setminus out^*_G(v_{\max})|$. Here, $v_{\max}$ has the largest number of graph descendants among u's graph children, and $v_{\max}$ is the unique parent of u in $\mathcal{T}_{G'}$. The cost of processing u is $d \times \triangle_{u,v_{\max}}$. Let $\triangle = \frac{\sum_{u \in V'} \triangle_{u,v_{\max}}}{|V'|}$ be the average number of visited nodes of processing all nodes V', the cost of processing all nodes of V' is $O(d \triangle |V'|)$. Since the time complexity of Algorithm 2 and Algorithm 3 is $O(|V| + |E|)$, the *time* complexity of *buTR* is $O(|V| + |E| + d \triangle |V'|)$.

During the processing, we need to maintain an *LPM* tree and po-tree, and for each node of G, we need to maintain 5 variables, the *space* complexity of *buTR* is $O(|V|)$.

6. EQUIVALENCE REDUCTION

Given the output G^t of *buTR*, we show in this section how to get the *ER* G^ε of G^t. By first sorting the adjacency lists (such as by topo-order), we have a total order of all nodes by comparing their graph parents and children. After that, each set of equivalent nodes can be clustered together by either one of existing sorting algorithms, which we call as *Sort-ER* without giving more details. We mainly discuss the linear algorithm *linear-ER*.

Definition 6.1: (**Partial Equivalence** $\equiv_S$) Given the *TR* $G^t = (V, E^t)$ of a *DAG* G and a subset $S \subseteq V$, we say two nodes u and v are equivalent to each other on G^t w.r.t. S if they have the same set of graph parents and children in S, and is denoted as $u \equiv_S v$. □

Definition 6.1 defines a relaxed equivalence relationship for all nodes of G^t, which considers only graph parents and children in a subset of V. E.g., given G^t in Fig. 1(b), if $S = \{v_8\}$, then v_8's graph parents, i.e., $\{v_4, v_6, v_7\}$, form an equivalent set due to the three nodes have the same graph child v_8. v_8's graph children, i.e., $\{v_9, v_{15}\}$, form another equivalent set due to that the two nodes have the same graph parent v_8, and all other nodes form the third equivalent set due to that they do not have graph parents and children in S.

The partial equivalence relationship $\equiv_S$ also defines, for V, a partition $\mathcal{P}$ satisfying (1) $\mathcal{P}$ does not contain the empty set, (2) the union of the sets in $\mathcal{P}$ is equal to V ($\mathcal{P}$ covers V), and (3) the intersection of any two distinct sets in $\mathcal{P}$ is empty. Obviously, nodes in the same set of $\mathcal{P}$ are possible equivalent, while nodes in different sets of $\mathcal{P}$ are definitely inequivalent.

Algorithm 5: *linear-ER* $(G^t = (V, E^t))$

```
1  $\mathcal{P}_0 \leftarrow \{V\}$
2  for each $(i \in [1, |V|]$ in ascending order) do
3      $\mathcal{P}_i \leftarrow$ refine$(\mathcal{P}_{i-1}, v_i)$
4  replace each set of $\mathcal{P}_{|V|}$ by one of its nodes to get $G^\varepsilon$
Function refine$(\mathcal{P}_{i-1}, v_i)$     /*$S = in_{G^t}(v_i) \bigcup out_{G^t}(v_i)$*/
5  generate a partition $\mathcal{S}$ for $S$, of which two nodes in the same set
     of $\mathcal{P}_{i-1}$ belong to the same set of $\mathcal{S}$, otherwise, they belong to
     different sets of $\mathcal{S}$
6  for each $(w \in S)$ do
7      remove $w$ from the set of $\mathcal{P}_{i-1}$ it belongs to
8  $\mathcal{P}_{i-1} \leftarrow \mathcal{P}_{i-1} \bigcup \mathcal{S}$
9  return $\mathcal{P}_{i-1}$
```

Lemma 6.1: *Let S_i, S_j be subsets of V contain i and j nodes respectively, $\mathcal{P}_i(\mathcal{P}_j)$ the partition of V corresponding to the partial equivalence relationship $\equiv_{S_i}$ $(\equiv_{S_j})$, then $S_i \subset S_j \Rightarrow \mathcal{P}_j \preceq \mathcal{P}_i$, where $\mathcal{P}_j \preceq \mathcal{P}_i$ denotes that every element of $\mathcal{P}_j$ is a subset of some element of $\mathcal{P}_i$.* □

Let $S_i = \emptyset$ if $i = 0$, otherwise $S_i = S_{i-1} \cup \{v_i\}$, we have $\emptyset = S_0 \subset S_1 \subset S_2 \subset ... \subset S_{|V|} = V$. Based on Lemma 6.1, we have Eq. (4).

$$\mathcal{P}_{|V|} \preceq \mathcal{P}_{|V|-1} \preceq ... \preceq \mathcal{P}_0 = \{V\} \tag{4}$$

Theorem 6.1: *Given $G^t = (V, E^t)$, partition $\mathcal{P}_{|V|}$ contains all sets of equivalent nodes w.r.t. G.* □

According to Theorem 6.1, $\mathcal{P}_{|V|}$ is the result we want to get. Based on Eq. (4), we have Algorithm 5, which visits the graph parents and children of each node only *once* to find all sets of equivalent nodes. After that, each set of $\mathcal{P}_{|V|}$ is replaced by one of its nodes to get the compressed graph G^ε.

Analysis: We first discuss the complexity of *Sort-ER*. Let $f_u = |in_{G^t}(u)| + |out_{G^t}(u)|$ and assume that *Sort-ER* is implemented based on the mergesort algorithm [17], thus the *space* complexity is $O(|V|)$. To get the final sorted results, mergesort needs to loop $\log |V|$ times. In the i^{th} loop, mergesort merges $|V|/2^{i-1}$ sorted runs into $|V|/2^i$ sorted runs. For two nodes u and v, the cost of comparison is $\min\{f_u, f_v\}$. In each loop, as mergesort compares at most $|V|$ different pairs of nodes, the cost of each loop is bounded by $2|E^t|$, thus the time complexity of sorting all nodes to identify all sets of equivalent nodes is $O(|E^t| \log |V|)$. As the complexity of removing equivalent nodes and their outgoing edges is $O(|V| + |E^t|)$, therefore, the *time* complexity of *Sort-ER* is $O(|E^t| \log |V|)$.

We then discuss the complexity of *linear-ER*. As shown in Algorithm 5, we need to remember the set number for each node, thus the *space* complexity of *linear-ER* is $O(|V|)$. Consider the time complexity. We process all nodes in lines 2-3. In each iteration, we process one node v_i in line 3 by calling Function refine(), which visits v_i's graph parents and children once to get the new partition $\mathcal{P}_i$, thus the cost of processing all nodes of G^t is $\sum_{v_i \in V}(|out_{G^t}(v_i)| + |in_{G^t}(v_i)|) = 2|E^t|$. Since the cost of line 4 is $|V| + |E^t|$, the *time* complexity of *linear-ER* is $O(|V| + |E^t|)$.

By combining Algorithm 4 and Algorithm 5 together, we know that the *space* complexity of the *DAG-Reduction* algorithm is $O(|V|)$, and the *time* complexity is $O(|V| + |E| + d \triangle |V'|)$.

7. EXPERIMENT

We test three groups of algorithms: First, for *TR*, it includes *PTR* [19], *DFS*, and our *buTR*. Second, for *ER*, it includes our *Sort-ER* and

Table 2: Statistics of datasets, where $d = |E|/|V|$ is the average degree of G, $|out^*_G(\cdot)|$ is the average number of reachable nodes for nodes of G, $r_n(r_e)$ is the ratio of the number of nodes (edges) in G', G^t, and G^ε over that of G, respectively.

Dataset	G				G'		G^t	G^ε	
	$\|V\|$	$\|E\|$	d	$\|out^*_G(\cdot)\|$	$r_n\%$	$r_e\%$	$r_e\%$	$r_n\%$	$r_e\%$
amaze	3,710	3,600	0.97	639	57.5	63.5	94.0	29.8	31.4
kegg	3,617	3,908	1.08	729	65.1	64.6	93.8	37.6	35.7
xmark	6,080	7,025	1.16	88	66.8	65.1	99.0	55.8	57.0
citeseer	10,720	44,258	4.13	39	86.9	85.3	51.8	84.9	46.1
pubmed	9,000	40,028	4.45	58	92.1	94.7	67.5	76.7	62.0
arxiv	6,000	66,707	11.12	928	97.8	91.5	20.0	97.9	19.7
email	231,000	223,004	0.97	11,698	9.2	10.1	96.9	14.7	8.3
unip150m	25,037,600	25,037,598	1.00	1.6	0.0	0.0	100.0	25.6	25.6
wiki	2,281,879	2,311,570	1.01	18,522	98.8	98.3	98.7	1.4	1.3
LJ	971,232	1,024,140	1.05	206,903	61.1	59.3	95.1	11.1	10.8
web	371,764	517,805	1.39	55,055	60.1	60.8	79.8	30.5	24.9
05Patent	1,671,488	3,303,789	1.98	7.7	83.3	84.7	90.1	80.3	78.9
citeseerx	6,540,401	15,011,260	2.30	15,510	89.1	93.1	74.4	39.7	46.4
dbpedia	3,365,623	7,989,191	2.37	83,658	76.1	81.2	59.2	50.5	31.7
govwild	8,022,880	23,652,610	2.95	561	100.0	99.9	93.7	69.0	82.5
Patent	3,774,768	16,518,947	4.38	1,544	96.5	96.0	71.6	91.2	68.9
go-unip	6,967,956	34,769,339	4.99	26	79.4	91.4	67.2	2.1	2.3
10go-unip	469,526	3,476,397	7.40	39	93.7	96.9	58.7	16.5	11.5
twitter	18,121,168	18,359,487	1.01	1,346,820	83.0	83.4	90.9	1.7	1.8
web-uk	22,753,644	38,184,039	1.68	3,417,930	64.9	61.0	66.8	15.9	14.8

linear-ER with input G^t. We also compare *DAG-Reduction* with compress$_\mathsf{R}$ [9] for the input G, where *DAG-Reduction* is to get G^t by calling *buTR* followed by identifying G^ε by calling *linear-ER*. Third, for reachability query processing, we select five state-of-the-art algorithms, including *GRAIL* [29] (abbreviated as *GRL*[3]), *FELINE* [24] (abbreviated as *FL*), *IP*$^+$ [25][4], *PLL* [27] and *TF* [5]. Besides, we also make comparison between *DAG* reduction and the reachability backbone [12] in Appendix B. We test the reachability algorithms using random reachability query workloads. Here, a random workload is generated by sampling node pairs with the same probability. The query time is the running time of a total of 1,000,000 reachability queries.

We obtained the source code of all existing algorithms for reachability query processing from the authors, and implemented all other algorithms using C++ and compiled by G++ 4.6.3. All experiments were run on a PC with AMD Athlon(tm) II X2 250 3.0 GHz CPU, 16 GB memory, and Ubuntu 12.04.4 Linux OS. For algorithms that run $\geq$ 24 hours or exceed the memory limit (16GB), we will show their results as "–" in the tables.

Table 2 shows the statistics of 20 real datasets used in our experiments. We give detailed description of these datasets in Appendix C.

7.1 Transitive Reduction (*TR*)

In this part, we first report the comparison between our algorithm and existing ones on *TR*, then show the impact of the optimizations and the impact of different *TC* estimating methods.

Comparison on *TR*: Table 3 shows the running time of different *TR* algorithms, where k is the size of path decomposition of *PTR*. For *buTR*, Step1 and Step2 denote *markCNRRN* and the operation after *markCNRRN*, respectively.

From Table 3 we know that *DFS* is greatly affected by the size of the average transitive closure $|out^*_G(\cdot)|$ (refer to the 5th column in Table 2). When $|out^*_G(\cdot)|$ increases, such as for twitter and web-uk, *DFS* fails to get the result in limited time. *buTR* outperforms *DFS* on most datasets, because c_{avg} for *buTR* is very small. For the amaze dataset, c_{avg} of *DFS* is 649.3, while c_{avg} of *buTR* is 0.26. Regarding *buTR*, Step1 may need more time than Step2 even though Step1 has linear time complexity, and Step2 needs more time than Step1 when

Table 3: Comparison of running time for *TR* (ms).

Dataset	*PTR*$(k/\|V\|)$	*DFS* (c_{avg})	*buTR*		
			Step1	Step2	*buTR* (c_{avg})
amaze	38 (0.81)	22 (649.3)	0.51	0.22	***0.73*** (0.26)
kegg	33 (0.75)	24 (743.1)	0.40	0.23	***0.63*** (0.27)
xmark	38 (0.71)	5.30 (99.1)	0.83	0.40	***1.23*** (2.01)
citeseer	121 (0.54)	***8.42*** (57.5)	4.47	3.96	8.43 (12.98)
pubmed	73 (0.62)	8.15 (93.1)	3.38	3.54	***6.92*** (26.19)
arxiv	67 (0.29)	196 (4,301.1)	4.07	5.51	***9.58*** (89.52)
email	90,817 (0.93)	14,100 (11,824.2)	37	11	***48*** (0.23)
unip150m	-	***1,706*** (2.4)	6,598	1,599	8,197 (0)
wiki	-	301,304 (18,636.4)	363	178	***541*** (0.02)
LJ	-	1,661,850 (210,734.0)	190	63	***253*** (0.10)
web	-	298,438 (63,400.3)	128	88	***216*** (1.02)
05Patent	1,244,350 (0.74)	***1,079*** (9.8)	2,015	1,751	3,766 (4.79)
citeseerx	-	3,951,890 (21,522.5)	6,242	11,266	***17,508*** (27.81)
dbpedia	-	2,428,480 (88,209.0)	2,310	2,244	***4,554*** (3.39)
govwild	-	76,369 (854.8)	4,799	4,842	***9,641*** (9.63)
Patent	482,117 (0.13)	579,911 (2,347.6)	10,158	277,135	***287,293*** (925.71)
go-unip	-	***3,661*** (40.2)	5,074	6,085	11,159 (21.89)
10go-unip	395,541 (0.84)	***383*** (62.0)	461	567	1,028 (31.64)
twitter	-	-	4,006	1,833	***5,839*** (0.03)
web-uk	-	-	4,815	3,105	***7,920*** (3.51)

Table 4: Running time (ms) of *buTR* with different optimizations.

Dataset	*buTR*-B(c_{avg})	*buTR*-O1(c_{avg})	*buTR* (c_{avg})
amaze	1.91 (29)	2.02 (35)	***0.73*** (0.26)
kegg	2.46 (38)	1.80 (26)	***0.63*** (0.27)
xmark	4.72 (40)	2.59 (18)	***1.23*** (2.01)
citeseer	14.10 (54)	18.61 (50)	***8.43*** (13)
pubmed	15.43 (100)	19.74 (103)	***6.92*** (26)
arxiv	188 (2,544)	205.19 (2,248)	***9.58*** (90)
email	662 (230)	147 (375)	***48*** (0.23)
unip150m	***4,582*** (1)	8,123 (0)	8,197 (0)
wiki	36,512 (981)	17,405 (447)	***541*** (0.02)
LJ	206,129 (11,189)	79,070 (6,658)	***253*** (0.10)
web	72,702 (6,784)	32,382 (4,211)	***216*** (1.02)
05Patent	***1,311*** (8)	3,750 (8)	3,766 (4.79)
citeseerx	7,471,010 (16,828)	7,996,991 (17,710)	***17,508*** (28)
dbpedia	1,512,640 (23,776)	1,114,804 (20,843)	***4,554*** (3.39)
govwild	137,841 (771)	164,472 (770)	***9,641*** (9.63)
Patent	822,854 (1,932)	945,138 (1,940)	***287,293*** (926)
go-unip	***6,993*** (31)	12,801 (37)	11,159 (22)
10go-unip	***693*** (50)	1,275 (53)	1,028 (32)
twitter	-	8,853,434 (28,800)	***5,839*** (0.03)
web-uk	-	-	***7,920*** (3.51)

both c_{avg} and the size of G' become large. *PTR* suffers from long time and large space due to the path-decomposition, and it works efficient only when the number of paths k is small. E.g., for the arxiv dataset, the ratio of $\frac{k}{|V|}$ is 0.29, and *PTR* is more efficient than *DFS*. But for the email dataset, $\frac{k}{|V|} = 0.93$, *DFS* outperforms *PTR*. When the given graph becomes large, *PTR* fails to get the result in limited time due to large space consumption.

The ratio of remained edges of G^t is shown in the 8th column in Table 2. The number of removed edges various with the given graph. For arxiv, more than 80% edges are removed. For govwild, only 6.3% edges are removed.

Impacts of the Optimizations: Table 4 shows the comparison of running time for *buTR*-B, *buTR*-O1 and *buTR*, where "B" denotes the baseline algorithm that processes nodes of G in a bottom-up fashion without any optimization, "O1" means that *buTR* first calls *markCNRRN* to get G', then processes nodes of G' as *buTR*-B does, *buTR* is Algorithm 4, which uses all optimizations. In Table 4, $c_{avg} = d\triangle$ denotes the average traversing cost for all nodes. c_{avg} for *buTR*-B is computed based on all nodes of G, and is computed based on all nodes of G' for the other two algorithms.

From Table 4 we know that *buTR* works much better than *buTR*-O1, and can be verified by the value of c_{avg}. The reason lies in that for *buTR*-O1, each node u is processed after one of its randomly selected

[3]*GRL* is the improved version of [28], and $k = 5$ for all datasets.

[4]The values of parameters are $k = 2, h = 2$, and $\mu = 100$.

Table 5: Running time (ms) of *buTR* using different estimating methods.

DataSet	ub	lb	kr ($k = 100$)	buTR
amaze	2.46	0.87	22.13	***0.73***
kegg	0.76	0.75	17.19	***0.63***
xmark	1.60	1.57	30.85	***1.23***
citeseer	13.25	13.44	84.20	***8.43***
pubmed	9.66	11.00	69.34	***6.92***
arxiv	39.27	48.68	58.66	***9.58***
email	56	55	2,049	***48***
unip150m	9,030	8,984	639,025	***8,197***
wiki	622	628	78,562	***541***
LJ	322	338	22,927	***253***
web	237	643	6,895	***216***
05Patent	4,100	4,179	65,902	***3,766***
citeseerx	18,508	507,810	347,280	***17,508***
dbpedia	4,900	13,373	88,323	***4,554***
govwild	10,657	10,968	375,403	***9,641***
Patent	***261,544***	444,341	442,395	287,293
go-unip	12,318	12,742	178,879	***11,159***
10go-unip	1,145	1,188	9,551	***1,028***
twitter	6,554	6,496	923,944	***5,839***
web-uk	12,972	825,701	990,805	***7,920***

graph child v, which may result in large size of $out^*_G(u) \setminus out^*_G(v)$, and a large value for c_{avg}. As a comparison, by constructing a good po-tree, the value of c_{avg} for *buTR* is small. E.g., the c_{avg} of *buTR*-O1 is 28,800 for the twitter dataset, while is 0.03 for *buTR*, and *buTR* is 1,516 times faster than *buTR*-O1. From Table 4 we know that *buTR*-O1 works at most 4.5 times faster than *buTR*-B on the email dataset, and can work successfully on the twitter dataset. For other datasets, the benefit of *buTR*-O1 is not obvious, or even beaten by *buTR*-B on some datasets. Even though, *markCNRRN* is necessary due to that it is not only used to reduce the size of G, but also get CNs for estimating the size of *TC* used by *buTR*.

From Table 4 we know that both *buTR* and *buTR*-O1 are beaten by *buTR*-B on datasets unip150m, 05Patent, go-unip and 10go-unip. The reasons lie in two aspects: (1) all the four datasets have small value for $|out^*_G(\cdot)|$ (see Table 2), therefore the traversing cost cannot be reduced significantly; (2) the cost of *markCNRRN* dominates the overall performance of both *buTR*-O1 and *buTR* for the four datasets, while *buTR*-B does not need to afford this cost.

Estimations in *buTR*: *buTR* estimates the size of $|out^*_G(\cdot)|$ for each node using Eq. (3) to construct a po-tree in order to reduce the traversing cost. We compare our method using Eq. (3) with lb (lower bound) and ub (upper bound) in [30] and kr (k random permutations) [6], where $k = 100$. Let $N(u) = |out^*_G(u)|$, and $\widetilde{N}(u)$ be the estimated result of $N(u)$, we use error rate $er(u) = \frac{|\widetilde{N}(u) - N(u)|}{N(u)}$ as a metrics to show the effectiveness of different estimating methods.

As shown in Fig. 8, for most graphs, our method is more accurate than existing methods, because by our method many nodes are with $er(u) \in [0, 0.2)$. kr [6] gets a better estimation on unip150m, go-unip and 10go-unip datasets, but is inefficient, since kr needs to traverse the given graph $k = 100$ times to get the estimation. As shown in Fig. 8 and Table 5, our estimating method is effective and efficient.

7.2 Equivalence Reduction (*ER*)

Table 6 shows the comparison of different algorithms on *ER*. First, given the input graph G^t, *linear-ER* is more efficient than *Sort-ER*. Second, given the input graph G, *DAG-Reduction* significantly outperforms compress$_R$, as ensured by the time complexity. Also, when the size of G and $|out^*_G(\cdot)|$ increases, compress$_R$ breaks down due to limited space (its space complexity is $O(|V|^2)$).

After *ER*, the ratios of the numbers of nodes and edges of G^ε are shown in the 9th and the 10th columns in Table 2, from which we know that the reduction ratios for all datasets vary significantly, this is because that the reduction ratio is determined by G itself. After getting $\mathcal{P}_{|V|}$ by Algorithm 5, each set of $\mathcal{P}_{|V|}$ will be replaced by one of its node to generate G^ε, and the reduction ratio w.r.t. nodes is $r_n = \frac{|\mathcal{P}_{|V|}|}{|V|}$. A small $|\mathcal{P}_{|V|}|$ means that each set of $\mathcal{P}_{|V|}$ contains a large number of nodes that are equivalent to each other on average.

Table 6: Comparison of running time for *ER* (ms).

Dataset	*Sort-ER*	*linear-ER*	compress$_R$	*DAG-Reduction*
amaze	1.48	***0.21***	146.33	***0.94***
kegg	0.44	***0.21***	162.34	***0.85***
xmark	1.23	***0.42***	37.04	***1.65***
citeseer	3.35	***1.40***	53.56	***9.83***
pubmed	2.55	***1.24***	52.65	***8.16***
arxiv	1.87	***0.77***	889.93	***10.35***
email	35.09	***16.01***	-	***64.35***
unip150m	12,457.10	***2,437.95***	27,717.90	***10,634.51***
wiki	196.42	***130.84***	-	***672.31***
LJ	144.37	***72.30***	-	***325.30***
web	149.61	***44.09***	-	***260.09***
05Patent	1,455.34	***879.42***	7,731.24	***4,645.65***
citeseerx	4,749.09	***2,707.66***	-	***20,214.56***
dbpedia	2,758.04	***855.35***	-	***5,409.35***
govwild	5,978.13	***2,404.77***	-	***12,046.18***
Patent	6,077.49	***4,593.92***	-	***291,886.92***
go-unip	8,452.43	***1,829.87***	53,496.90	***12,988.87***
10go-unip	465.57	***137.87***	4,504.15	***1,165.87***
twitter	1,976.76	***1,122.59***	-	***6,961.52***
web-uk	5,281.05	***1,654.51***	-	***9,574.59***

Table 7: Comparison of index sizes (MB).

Dataset	*GRL*	*GRL*$_*$	*FL*	*FL*$_*$	*IP*$^+$	*IP*$^+_*$	*PLL*	*PLL*$_*$	*TF*	*TF*$_*$
amaze	0.14	●0.06	0.13	●0.05	0.05	0.06	0.04	0.05	0.02	0.02
kegg	0.14	●0.07	0.12	●0.06	0.05	0.06	0.05	0.05	0.02	0.02
xmark	0.23	0.15	0.21	0.14	0.09	0.11	0.12	0.12	0.12	0.08
citeseer	0.41	0.39	0.37	0.35	0.14	0.21	0.28	0.31	0.83	0.52
pubmed	0.34	0.30	0.31	0.27	0.11	0.16	0.27	0.29	0.80	0.64
arxiv	0.23	0.25	0.21	0.22	0.11	0.13	0.35	0.40	14.66	●5.34
email	8.81	●2.17	7.93	●2.04	2.78	2.90	2.59	2.71	0.85	0.95
unip150m	955	●340	860	●316	299	348	318	328	132	140
wiki	87	●10	78	●10	43	26	26	26	9	9
LJ	37	●8	33	●7	15	12	11	12	4	4
web	14	●6	13	●5	6	5	5	5	3	2
05Patent	64	58	57	52	20	30	29	33	26	29
citeseerx	250	●124	225	114	87	98	36	54	1,523	●631
dbpedia	128	78	116	71	44	54	53	54	52	30
govwild	306	242	275	221	105	144	188	205	3,123	2,693
Patent	144	146	130	133	58	80	633	648	4,732	4,231
go-unip	266	●32	239	●32	80	81	251	●87	431	●40
10go-unip	18	●5	16	●4	5	6	21	●9	44	●9
twitter	691	●81	622	●79	316	211	202	209	70	71
web-uk	868	●225	781	●211	356	310	357	336	-	●324

7.3 Reachability Query Processing

Tables 7, 8 and 9 show the comparison of index sizes, index construction time and query time for existing reachability algorithms working on the input *DAG* G, as well as their counterparts on the result of *DAG* reduction G^ε with "$*$" as their subscript. For each algorithm, we use "●" to denote the best result is better than the worst one more than two times, we take others as comparable results.

Index Size: Table 7 shows that *DAG* reduction has a positive impact for all algorithms. E.g., for *GRL*, the index sizes based on G^ε are 11.4%, 12.1% and 11.7% to its counterparts on G for wiki, go-unip and twitter datasets, respectively. For *FL* algorithm, we have similar results as *GRL*. For *TF*, the index size on G^ε is 9.3% to that on G for the go-unip dataset. After *DAG* reduction, *TF* works successfully on the web-uk dataset. For *IP*$^+$, we have comparable results on all datasets.

Index Construction Time: We have shown that the result of *DAG* reduction, G^ε, can be got quickly, which is a one-time activity, and once G^ε has been obtained, it can be repeatedly used by different algorithms. Table 8 shows that, compared with G, all existing algorithms work

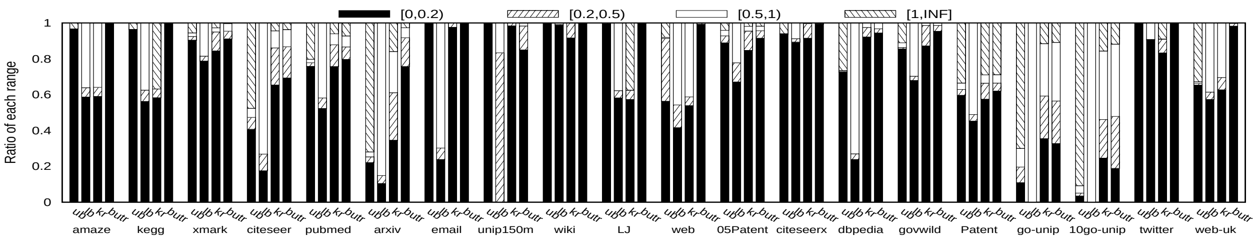


Figure 8: Comparison of the error rates for different estimating methods.

Table 8: Comparison of index construction time (ms).

Dataset	*GRL*	*GRL*$_*$	*FL*	*FL*$_*$	*IP*$^+$	*IP*$^+_*$	*PLL*	*PLL*$_*$	*TF*	*TF*$_*$
amaze	7.66	7.14	1.21	●0.60	1.50	1.20	4.46	●1.11	6.69	●2.44
kegg	7.47	6.84	1.34	0.70	2.80	1.29	1.16	0.96	10.29	9.58
xmark	13.69	13.87	2.45	1.73	●3.40	10.02	2.80	2.39	14.44	12.90
citeseer	51.62	44.02	9.69	7.58	12.40	10.11	14.80	14.05	101.03	●50.43
pubmed	36.73	35.84	6.57	5.15	10.00	10.00	12.76	12.36	72.19	48.49
arxiv	32.17	19.17	6.17	3.91	12.50	10.00	25.66	19.28	7,586	●614
email	1,063	931	110	●37	160	100	119	93	161	125
unip150m	192,236	162,680	18,040	●8,746	23,460	18,490	21,631	15,695	68,229	●31,366
wiki	12,578	12,665	1,006	●213	1,590	1,050	1,238	1,123	1,429	1,090
LJ	4,896	4,879	464	●169	710	490	594	484	923	761
web	1,755	1,627	250	●120	360	240	277	186	637	393
05Patent	9,805	10,548	2,132	1,958	2,870	2,660	3,082	2,770	7,667	6,855
citeseerx	34,038	44,421	6,435	4,516	10,430	7,880	4,074	3,386	101,029	●41,536
dbpedia	23,365	20,676	3,262	2,014	4,720	3,290	4,349	3,216	15,081	●7,194
govwild	39,443	50,204	7,013	6,163	10,810	10,100	13,914	13,292	143,272	114,950
Patent	39,088	34,477	8,395	7,482	12,770	10,590	245,345	242,475	253,020	197,125
go-unip	67,876	43,511	7,144	●805	11,230	●3,630	17,712	●3,926	67,524	●6,224
10go-unip	4,100	2,525	585	●140	840	●300	1,679	●540	6,293	●994
twitter	118,208	114,024	9,176	●1,852	15,380	9,810	11,150	8,789	18,785	13,289
web-uk	147,831	140,513	12,425	●4,646	21,610	15,190	21,944	16,471	-	●67,368

Table 9: Comparison of query time (ms).

Dataset	*GRL*	*GRL*$_*$	*FL*	*FL*$_*$	*IP*$^+$	*IP*$^+_*$	*PLL*	*PLL*$_*$	*TF*	*TF*$_*$
amaze	3,796	●254	41	42	20	18	29	20	22	11
kegg	4,720	●402	46	43	23	21	29	21	15	12
xmark	491	●243	298	220	40	34	46	47	32	26
citeseer	555	465	223	196	89	68	100	146	68	53
pubmed	550	425	169	128	62	51	86	102	64	53
arxiv	2,805	2,307	1,593	1,444	877	573	92	160	505	●248
email	53,024	●176	121	●37	70	●29	158	85	51	●14
unip150m	657	●236	26	20	11	13	120	74	46	41
wiki	1,744,266	●324	125	●14	47	●7	266	●43	77	●8
LJ	9,580,519	●42,942	250	●78	140	●44	243	●112	140	●41
web	662,571	●82,977	221	149	137	92	210	199	142	98
05Patent	527	532	90	91	29	29	205	212	74	74
citeseerx	64,769	45,120	771	502	210	147	259	241	146	96
dbpedia	212,503	●4,292	241	193	180	148	367	404	240	213
govwild	1,255	1,252	469	471	226	226	436	542	445	452
Patent	557	543	111	114	29	29	563	643	84	84
go-unip	848	665	128	143	65	63	267	189	85	102
10go-unip	728	552	145	102	70	51	223	170	87	78
twitter	-	●18,386	280	●28	160	●17	412	●87	203	●18
web-uk	-	●3,002,770	560	303	367	200	460	319	-	●175

more efficiently on G^ε. E.g., *FL* is 8.9 times faster on G^ε than on G for the go-unip dataset, *TF* is 12.4 times faster on G^ε than on G for the arxiv dataset.

Query Time on Random Workload: Similar to index size, Table 9 shows that query time can be improved significantly using *DAG* reduction for all algorithms. E.g., for *GRL*, the query time on G^ε are 301 and 5,384 times faster than that on G for email and wiki datasets. And more importantly, *GRL* can process all queries in limited time for twitter and web-uk datasets after *DAG* reduction. For *FL*, *IP*$^+$, *PLL* and *TF*, the query time on G^ε are 8.9, 6.7, 6.2 and 9.6 times faster than that on G for the wiki dataset, and are 10, 9.4, 4.7 and 11.3 times faster for the twitter dataset, respectively. And *TF* can process queries on the web-uk dataset after *DAG* reduction.

By taking Tables 7 to 9 together, we know that *DAG* reduction makes significantly improvements on index sizes, index construction time and query time for all algorithms. More importantly, after *DAG* reduction, *GRL* and *TF* can work successfully on all datasets.

8. CONCLUSIONS

In this paper, we focus on *DAG* reduction which is to reduce G by first computing the transitive reduction *TR* followed by computing the equivalence reduction *ER*. With the newly proposed techniques, we show that we can significantly reduce the cost of *TR* compared with the existing *PTR* and *DFS* algorithms, and significantly reduce the cost of *ER* compared with the $\mathsf{compress}_\mathsf{R}$ algorithm. As an indication, among 20 real datasets being tested, for *TR*, *PTR* cannot complete in 10 datasets, *DFS* takes 1,661,850 ms for the LJ dataset whereas our *buTR* algorithm takes 253 ms; for *ER*, $\mathsf{compress}_\mathsf{R}$ cannot scale to 10 large datasets, whereas our *linear-ER* can efficiently compute all datasets. For reachability queries answering, we show that our *DAG* reduction can significantly improve the efficiency either by itself or by integrated with *SCARAB* using 20 real datasets.

9. ACKNOWLEDGMENTS

This work was partly supported by the grants from the Natural Science Foundation of China (No.: 61472339, 61303040, 61572421, 61272124), and Jeffrey Xu Yu was partly supported by the grant of the Research Grants Council of Hong Kong SAR, China, No. 14209314 and No. 14221716. The authors would like to thank Na Li and Qiang Ma for implementing algorithms on *ER* computation, and also would like to thank the anonymous referees for their insightful and valuable comments.

10. REFERENCES

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

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[3] P. Boldi, M. Santini, and S. Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, 2008.

[4] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *ICWSM*, 2010.

[5] J. Cheng, S. Huang, H. Wu, and A. W. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*, pages 193–204, 2013.

[6] E. Cohen. Estimating the size of the transitive closure in linear time. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 190–200, 1994.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *ACM-SIAM*, pages 937–946, 2002.

[8] V. Dubois and C. Bothorel. Transitive reduction for social network analysis and visualization. In *2005 IEEE / WIC / ACM International Conference on Web Intelligence*, pages 128–131, 2005.

[9] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[10] M. Habib, M. Morvan, and J. Rampon. On the calculation of transitive reduction - closure of orders. *Discrete Mathematics*, 111(1-3):289–303, 1993.

[11] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB*, pages 273–284, 2003.

[12] R. Jin, N. Ruan, S. Dey, and J. X. Yu. SCARAB: scaling reachability computation on large graphs. In *SIGMOD*, pages 169–180, 2012.

[13] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, 36(1):7, 2011.

[14] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *PVLDB*, 6(14):1978–1989, 2013.

[15] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.

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

[17] J. Katajainen and J. L. Träff. A meticulous analysis of mergesort programs. In *Algorithms and Complexity, Third Italian Conference, CIAC '97, Rome, Italy, March 12-14, 1997, Proceedings*, pages 217–228, 1997.

[18] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. FERRARI: flexible and efficient reachability range assignment for graph indexing. In *ICDE*, pages 1009–1020, 2013.

[19] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58:325–346, 1988.

[20] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[21] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.

[22] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. *SIAM J. Comput.*, 11(2):298–313, 1982.

[23] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, pages 913–924, 2011.

[24] R. R. Veloso, L. Cerf, W. M. Junior, and M. J. Zaki. Reachability queries in very large graphs: A fast refined online search approach. In *EDBT*, pages 511–522, 2014.

[25] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: An independent permutation labeling approach. *PVLDB*, 7(12):1191–1202, 2014.

[26] V. V. Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898, 2012.

[27] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pages 1601–1606, 2013.

[28] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.

[29] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *VLDB J.*, 21(4):509–534, 2012.

[30] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*, pages 1323–1334, 2014.

APPENDIX

A. PROOFS

Proof of Lemma 4.1: We prove this lemma from two aspects.

(1) If G has no redundant edges, $A_i(u) = A_i(v) \Rightarrow in^*_G(u) = in^*_G(v) \wedge D_i(u) = D_i(v) \Rightarrow out^*_G(u) = out^*_G(v)$.

$\forall i \geq 1$, as $A_i(u)$ contains *all* the the nodes that can reach u through shortest paths with at most i edges, we know that $A_i(u) \subseteq A_{i+1}(u)$ and $\forall w \in A_{i+1}(u) \setminus A_i(u)$, w can reach some node of $A_i(u)$ through *one* edge. Therefore, $A_i(u) = A_i(v) \Rightarrow A_{i+1}(u) = A_{i+1}(v)$. When $A_{i+1}(u) \setminus A_i(u) = \emptyset$, we have $A_i(u) = in^*_G(u)$. Therefore, if G has no redundant edges, $A_i(u) = A_i(v) \Rightarrow in^*_G(u) = in^*_G(v)$. Similarly, if G has no redundant edges, $D_i(u) = D_i(v) \Rightarrow out^*_G(u) = out^*_G(v)$.

(2) If G has no redundant edges, $in^*_G(u) = in^*_G(v) \Rightarrow A_i(u) = A_i(v) \wedge out^*_G(u) = out^*_G(v) \Rightarrow D_i(u) = D_i(v)$.

We assume that there exist $i \geq 1, u, v, w_i \in V$, such that when $in^*_G(u) = in^*_G(v), w_i \in A_i(u) \setminus A_i(v)$, i.e., $A_i(u) \neq A_i(v)$. If $i > 1$, $w_i \in A_i(u) \setminus A_i(v)$ means that there exists a node w_{i-1} satisfying that $(w_i, w_{i-1}) \in E$ and $w_{i-1} \in A_{i-1}(u) \setminus A_{i-1}(v)$. This is because that if $w_{i-1} \in A_{i-1}(v)$, then $w_i \in A_i(v)$. Thus, $A_i(u) \neq A_i(v) \Rightarrow A_{i-1}(u) \neq A_{i-1}(v)$. By induction, we know that $\forall i \geq 2$, there exists $w_1 \in A_1(u) \setminus A_1(v)$, i.e., $A_i(u) \neq A_i(v) \Rightarrow A_1(u) \neq A_1(v)$. Combining with the case of $i = 1$, this assumption means that $\forall i \geq 1$, there exists $w_1 \in V$, such that when $in^*_G(u) = in^*_G(v), w_1 \in A_1(u) \setminus A_1(v)$, i.e., $A_1(u) \neq A_1(v)$. Since $in^*_G(u) = in^*_G(v) \wedge w_1 \notin A_1(v)$, we know that $w_1 \in in^*_G(v)$, and there must exist at least one node $x \in A_1(v)$, such that w_1 can reach v through x, i.e., $w_1 \rightsquigarrow x \rightsquigarrow v$. By $in^*_G(u) = in^*_G(v) \wedge x \in in^*_G(v)$, we know that $x \in in^*_G(u)$, i.e., $x \rightsquigarrow u$. Thus, w_1 can reach u through x, and we know that (w_1, u) is redundant, which contradicts the assumption that G has no redundant edges. In the above assumption, we have the same result if $w_i \in A_i(v) \setminus A_i(u)$. Therefore, if G has no redundant edges, $in^*_G(u) = in^*_G(v) \Rightarrow A_i(u) = A_i(v)$. Similarly, if G has no redundant edges, we know that $out^*_G(u) = out^*_G(v) \Rightarrow D_i(u) = D_i(v)$.

Based on the above discussion, we know that $\forall u, v \in V, \forall i \geq 1$, if G has no redundant edges, then Eq. (1) and Eq. (2) hold. □

Proof of Property 5.1: Assume that there exists a redundant edge (x, y) in the *LPM* tree T_X, i.e., x is the tree parent of y in T_X. Since (x, y) is a redundant edge, x can reach y through at least one node w, such that $(w, y) \in E \wedge t_x < t_w < t_y$. According to the construction of *LPM* tree, y's tree parent should be w, rather than x, i.e., (x, y) should not be an edge of T_X, which contradicts the assumption. □

Proof of Lemma 5.1: We prove this lemma from two aspects.

(1) If $in_G(v) = \emptyset$, or every node of $in_G(v)$ is an RRN, then v is an RRN.

If $in_G(v) = \emptyset$, then $in^*_G(v) = \emptyset$. Thus, v is an RRN by Definition 5.3. If $in_G(v) \neq \emptyset$ and $\forall u \in in_G(v)$, u is an RRN, we know that all nodes of $in^*_G(u) \cup \{u\}$ are RNs according to Definition 5.3. Since $in^*_G(v) = \bigcup_{u \in in_G(v)} (in^*_G(u) \cup \{u\})$, all nodes of $in^*_G(v)$ are RNs. By Definition 5.3, v is an RRN.

(2) If v is an RRN, then $in_G(v) = \emptyset$, or every node of $in_G(v)$ is an RRN.

According to Definition 5.3, if v is an RRN, then $in^*_G(v) = \emptyset$, or $\forall u \in in^*_G(v)$, u is an RN. The first case means that $in_G(v) = \emptyset$. Consider the second case. $\forall u \in in_G(v)$, since $in^*_G(u) \bigcup \{u\} \subseteq in^*_G(v)$ and all nodes of $in^*_G(v)$ are RNs, we know that u is an RRN according to Definition 5.3. □

Proof of Lemma 5.2: First, given a *DAG* G, to get the *DT*-order X, the topological sorting can be done by (Step1) finding all the "start nodes" without incoming edges and pushing them into a stack S; (Step2) popping out a node v from S, assigning v its visiting order (*DT*-order) t_v, and pushing v's graph children which have no incoming edges into S after deleting edges starting from v; and (Step3) repeating Step2 until S becomes empty.

Second, we construct the *LPM* tree T_X during performing topological sorting on G. Let u be the last graph parent visited before v (v is pushed into S immediately after u is popped out from S). In Step2, after popping out a node v from S and assigning its *DT*-order t_v, v is inserted into T_X as a tree child of u (v is the t_v-th node inserted into T_X). After that, each of v's tree children is popped out from S and inserted into T_X recursively. Therefore, T_X is constructed recursively by inserting nodes into it in the ascending *DT*-order X.

Third, when we visit nodes of T_X in the ascending *DT*-order X, it means that after visiting a node v, we first visit each of its tree children recursively, which is a *DFS* visiting order for T_X. Therefore, if the *LPM* tree T_X is generated based on a *DT*-order X of G, then X is also a *DFS*-order of T_X. □

Proof of Theorem 5.1: We prove this theorem from two aspects.

(1) We prove the correctness for RRN.

The correctness of correctly identifying all RRNs is based on correctly identifying all RNs according to Lemma 5.1. We show the correctness of identifying all RNs from two aspects.

(1.1) isRN() correctly identifies whether a given node v is an RN. Function isRN() processes v's graph children in ascending *DT*-order X. When processing $w \in out_G(v)$, there are two cases: $w \in out^*_{T_X}(v)$ (line 8 holds) and $w \in out_G(v) \setminus out^*_{T_X}(v)$ (line 8 does not hold).

Consider the trivial case where $w \in out^*_{T_X}(v)$ processed in lines 9-11, i.e., w is a tree descendant of v. In this case, (v, w) is redundant if w is not a tree child of v (line 11); otherwise, (v, w) is not a redundant edge (Property 5.1), i.e., we can correctly find all redundant edges from v to its tree descendants given that v is an RN.

Consider the case where $w \in out_G(v) \setminus out^*_{T_X}(v)$ processed in lines 13-16, i.e., w is not a tree descendant of v. In this case, isRN() checks whether w belongs to C_2 in lines 13. If line 13 returns FALSE, it means that w belongs to C_2 (the fourth condition of Definition 5.2) and we delete the redundant edge (v, w) in line 16; otherwise, if line 13 returns TRUE, it means that $w \notin C_2$, we further check whether it belongs C_1 in line 14 (the second and the third conditions of Definition 5.2). In line 14, if $l_{\min}(v) < l_w$, it means that the second and third conditions do not hold, thus $w \notin C_1$. As a result, $C_1 \bigcup C_2 \neq out_G(v) \setminus out^*_{T_X}(v)$, we know that v is not an RN according to Definition 5.2 and isRN() returns FALSE in line 14. If $l_{\min}(v) \geq l_w$, it means that $w \in C_1$, and we continue to visit the next graph child of v. Finally, if isRN() returns TRUE in line 17, it means that $out_G(v) \setminus out^*_{T_X}(v)$ can be divided into two sets satisfying the four conditions, thus v is an RN by Definition 5.2, and we correctly delete all redundant edges in line 16.

Therefore, isRN() correctly identifies whether a given node v is an RN or not.

(1.2) All RNs are correctly identified by Algorithm 2.

As each given node can be correctly identified, all RNs can be correctly identified by calling isRN() to process all nodes.

Based on the above result, Algorithm 2 processes all nodes in ascending *DT*-order, such that when processing a node v, we know whether each one of its graph parents is an RRN or not, then we know whether v is an RRN or not by visiting v's graph parents only once according to Lemma 5.1. Thus, all RRNs are correctly identified.

(2) We prove the correctness for CN.

Let $x_u = \max \arg_v \{t_v | v \in out^*_G(u)\}$ be, among u's graph descendants, the one with the largest topo-order. Algorithm 2 processes nodes in descending *DT*-order to find all CNs from non-RRNs. For each processed node v, we update x_u for each of v's graph parent u using x_v. Therefore, when processing u, we know the correct value of x_u. Given a non-RRN u, if u is a tree ancestor of x_u in T_X, Algorithm 2 will mark u as a CN, otherwise not. We show the correctness of processing each non-RRN from two aspects.

(2.1) If u is a CN, then $x_u \in out^*_{T_X}(u)$.

By Definition 5.1, if u is a CN, then $\forall v \in out^*_G(u), v \in out^*_{T_X}(u)$. As $x_u \in out^*_G(u)$, we know $x_u \in out^*_{T_X}(u)$.

(2.2) If $x_u \in out^*_{T_X}(u)$, then u is a CN.

Let $I_u = [s, e]$ be the interval assigned to u to facilitate checking the ancestor-descendant relationship for nodes in T_X, where $I_u.s = t_u$, and $I_u.e$ is the maximum *DT*-order of u's tree descendants.

First, $\forall v \in out^*_G(u) \setminus \{x_u\}$, we have $t_u < t_v < t_{x_u}$.

Second, $x_u \in out^*_{T_X}(u)$ means that $t_{x_u} \leq I_u.e$.

As $I_u.s = t_u, \forall v \in out^*_G(u)$, we know $I_v \subset I_u$ according to Lemma 5.2, i.e., $\forall v \in out^*_G(u), v \in out^*_{T_X}(u)$. Thus, v is a CN according to Definition 5.1.

Therefore, given the *LPM* tree T_X, Algorithm 2 correctly identifies all RRNs and CNs. □

Proof of Lemma 5.3: We prove this lemma from two aspects.

(1) If u and v do not have ancestor-descendant relationship in T, then $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$.

In this case, u and v may be sibling nodes (Fig. 9(a)), or not (Fig. 9(b)-(d)), we show the correctness case by case.

(Case1) u and v are sibling nodes (Fig. 9(a)).

Assume that w is the tree parent of u and v. During the topological sorting of computing $\overline{Z}$ based on Z, after processing w (i.e., assigning its *DT*-order $t_{\overline{w}}$), both u and v become nodes without incoming edges, and u will be pushed into stack before v due to that $t_u < t_v$. Therefore, u is popped out from the stack (i.e., assign its *DT*-order $t_{\overline{u}}$) after v, that is, $t_{\overline{u}} > t_{\overline{v}}$, i.e., $t_u < t_v \Rightarrow t_{\overline{u}} > t_{\overline{v}}$. Similarly, if Z is computed based on $\overline{Z}$, we have $t_{\overline{u}} > t_{\overline{v}} \Rightarrow t_u < t_v$. Therefore, if u and v are sibling nodes, then $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$.

(Case2) u and v are not sibling nodes (Fig. 9(b)-(d)).
Let $w = lca(u, v)$ be the lowest common ancestor (LCA) of u and v in T, there are three sub-cases.

(Case2.1) u and v's tree ancestor v_a are sibling nodes (Fig. 9 (b)). Given Z, since v_a is a tree ancestor of v, we know $t_{v_a} < t_v$. According to Lemma 5.2, any node with *DT*-order between v_a and v are v_a's tree descendants. Since u and v_a are sibling nodes, given $t_u < t_v$, we know that $t_u < t_{v_a} < t_v$. When computing $\overline{Z}$, we have $t_{\overline{v_a}} < t_{\overline{v}} < t_{\overline{u}}$ as shown by Case1, i.e., $t_u < t_v \Rightarrow t_{\overline{u}} > t_{\overline{v}}$. Similarly, if Z is computed based on $\overline{Z}$, we have $t_{\overline{u}} > t_{\overline{v}} \Rightarrow t_u < t_v$.
Thus in this case, $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$.

(Case2.2) u's tree ancestor u_a and v are sibling nodes (Fig. 9 (c)). $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$ can be proved in the similar way as Case2.1.

(Case2.3) u_a and v_a are sibling nodes (Fig. 9 (d)). Since u_a and v_a are sibling nodes, given Z and $t_u < t_v$, we know that $t_{u_a} < t_u < t_{v_a} < t_v$ according to Lemma 5.2. When computing $\overline{Z}$, we have $t_{\overline{v_a}} < t_{\overline{v}} < t_{\overline{u_a}} < t_{\overline{u}}$, i.e., $t_u < t_v \Rightarrow t_{\overline{u}} > t_{\overline{v}}$. Similarly, if Z is computed based on $\overline{Z}$, we have $t_{\overline{u}} > t_{\overline{v}} \Rightarrow t_u < t_v$.
Thus in this case, $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$.

Therefore, if u and v do not have ancestor-descendant relationship in T, then $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$.

(2) If $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$ holds, then u and v do not have ancestor-descendant relationship in T.

Assume that u and v have ancestor-descendant relationship in T, which also consists of two cases.

(Case1) u is a tree ancestor of v.
In this case, we have that $u \rightsquigarrow v$, and for both Z and $\overline{Z}$, $t_u < t_v \wedge t_{\overline{u}} < t_{\overline{v}}$, i.e., $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$ does not hold.

(Case2) v is a tree ancestor of u.
Similar to Case1, we know $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$ does not hold.

Thus if $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$ holds, then u and v do not have ancestor-descendant relationship in T.

Therefore, nodes u and v do not have ancestor-descendant relationship iff $t_u < t_v \Leftrightarrow t_{\overline{u}} > t_{\overline{v}}$. □

Proof of Lemma 5.4: For each edge (u, v) of $\mathcal{T}_{G'}$, the cost of processing u, $c(u, v)$, is visiting nodes of $out^*_G(u) \setminus out^*_G(v)$ and the involved edges, and $c(u, v)$ does not change by switching from processing order in Z to $\overline{Z}$, because for any *DT*-order, edge (u, v) does not change, thus the number of processed nodes and edges does not change. Therefore, $N_1 = \sum_{i \in [1,|V'|-1]} c(u_i, v_i) = N_2$, where $|V'| - 1$ is the number of edges in $\mathcal{T}_{G'}$. □

Proof of Theorem 5.2: We prove this theorem from two aspects.

(1) Each edge deleted by Algorithm 4 is a redundant edge.

In *markCNRRN* (line 1 of Algorithm 4), each redundant edge is found based on tree relationship, i.e., edge (v, w) is redundant only if $\exists x \in out_G(v)$, such that x is a tree ancestor of w in T_X, which means that v can reach w through x, thus (v, w) is a redundant edge. After calling *markCNRRN*, we find redundant edges from each node

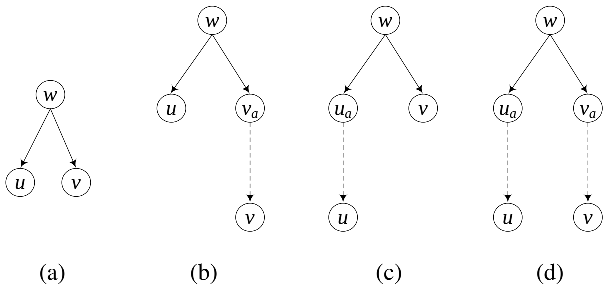


Figure 9: Illustration of the positional relationships where u and v do not have ancestor-descendant relationship in a tree. Each arrow (dashed arrow) denotes an edge (a path) between two nodes, (a) denotes that u and v are sibling nodes, and (b)-(d) denote u and v are not sibling nodes.

that is not an RRN in line 6 by calling Procedure delRdtEdge(). Let v be the tree parent of u in po-tree $\mathcal{T}_{G'}$ (v is a graph child of u in G), in delRdtEdge(), we identify redundant edges in lines 10 and 15, which correspond to the two cases that the redundant edges are from u to nodes of $out^*_G(v)$ and $out^*_G(u) \setminus out^*_G(v)$, respectively. If an edge (u, w) is deleted in line 10, it must be a redundant edge, due to that $w \in out^*_G(v) \wedge u \rightsquigarrow v$; otherwise if (u, w) is deleted in line 15, it must be a redundant edge due to that we first mark all of u's graph children using u, then encounter u at w when traversing from another node of u's graph children, i.e., there exists, for u, a graph child node $v(\neq w)$, such that u can reach w through v. Therefore, if we delete an edge, it must be a redundant edge.

(2) Algorithm 4 finds all redundant edges.

For all RRNs, *markCNRRN* correctly finds all redundant edges from them. After that, we process all nodes that are not RRNs. As shown in Algorithm 4, we process just one node u in each iteration, and only delete all redundant edges from u, every redundant edge from other nodes are not considered. As discussed above, for each redundant edge (u, w), either $w \in out^*_G(u) \setminus out^*_G(v)$ holds, or $w \in out^*_G(v)$ holds. And for both cases, we can correctly find all redundant edges from u in lines 10 and 15. Therefore, after processing all nodes, we correctly find all redundant edges. □

Proof of Lemma 6.1: Since $S_i(S_j)$ contains $i(j)$ nodes and $S_i \subset S_j$, we know $j > i$. Assume that $S_i = \{v_1, v_2, ..., v_i\}$, $S_j = \{v_1, v_2, ..., v_i, ..., v_j\}$, we can expand S_i to get S_j with $j - i$ steps by adding node $v_k(i < k \leq j)$ into S_i in the $(k - i)$th step to get a set S^{k-i}. After adding v_j into S_i in the $(j - i)$th step, we get $S^{j-i} = S_j$.

Let $S^0 = S_i$, we have $j - i + 1$ sets $S^0, S^1, S^2, ..., S^{j-i}$, which satisfy that $\forall x \in [1, j - i]$, $S^x \setminus S^{x-1} = \{v_{i+x}\}$. We use $\mathcal{P}^x$ to denote the partition of V corresponding to the partial equivalence relationship $\equiv_{S^x}$.

We first prove that $\forall x \in [1, j - i]$, $S^{x-1} \subset S^x \Rightarrow \mathcal{P}^x \preceq \mathcal{P}^{x-1}$. Given $\mathcal{P}^{x-1}$, the unique node $v_{i+x} \in S^x \setminus S^{x-1}$ divides each set $P \in \mathcal{P}^{x-1}$ into at most three disjoint subsets, where the first subset P_1 contains nodes that are graph children of v_{i+x}, the second subset P_2 contains nodes that are graph parents of v_{i+x}, and the third subset P_3 contains nodes that are neither graph parents nor children of v_{i+x}. The three subsets satisfy that $P_1 \bigcup P_2 \bigcup P_3 = P$. If $\exists P_i = \emptyset (i \in [1, 3])$, then P is divided into two or even one subset. After that, we get the partition $\mathcal{P}^x$ w.r.t. S^x, which satisfies that every set of $\mathcal{P}^x$ is a subset of some set of $\mathcal{P}^{x-1}$, i.e., $\mathcal{P}^x \preceq \mathcal{P}^{x-1}$.

Since $S^0 \subset S^1 \subset S^2 \subset ... \subset S^{j-i}$ and $\forall x \in [1, j - i]$, $S^x \setminus S^{x-1} = \{v_{i+x}\}$, we know that $\mathcal{P}^{j-i} \preceq \mathcal{P}^{j-i-1} \preceq ... \preceq \mathcal{P}^1 \preceq \mathcal{P}^0$. As $S_i = S^0$ and $S^{j-i} = S_j$, we know that $\mathcal{P}_j \preceq \mathcal{P}_i$.

Therefore, $S_i \subset S_j \Rightarrow \mathcal{P}_j \preceq \mathcal{P}_i$. □

Table 10: *DAG* reduction *vs* Backbone: *IP*$^+$.

Dataset	Index Size (MB)			Index Construction Time (ms)			Query Time (ms)		
	IP^+_*	IP^+_B	IP^+_{B*}	IP^+_*	IP^+_B	IP^+_{B*}	IP^+_*	IP^+_B	IP^+_{B*}
amaze	●0.055	0.16	0.064	●1.20	5.80	2.94	●18	59	37
kegg	●0.06	0.16	0.08	●1.29	15.43	3.49	●21	67	44
xmark	●0.11	0.26	0.17	●10.02	24.44	12.37	34.4	39	33.5
citeseer	●0.21	0.60	0.51	10	4.35	●1.66	68	107	84
pubmed	●0.16	0.49	0.40	10	●2.12	4.93	●51	96	107
arxiv	●0.13	0.46	0.33	10	11.20	●2.12	573	1,009	531
email	2.90	10	●2.34	100	1,156	●8	●29	174	41
unip150m	●348	1,069	386	18,490	●1,210	1,315	●13	121	44
wiki	26	97	●11	1,050	324,324	●61	●7	141	9
LJ	12	41	●9	490	47	●29	●44	325	86
web	●5.47	16	6.41	●240	9,309	7,842	●92	314	211
05Patent	●30	82	72	2,660	681	●537	●29	58	53
citeseerx	●98	305	152	7,880	1,752	●1,185	147	236	178
dbpedia	●54	161	90	●3,290	428,504	132,716	●148	372	273
govwild	●144	390	315	10,100	●1,613	1,713	226	355	351
Patent	●80	235	212	10,590	6,138	●4,724	●29	99	89
go-unip	81	400	●38	3,630	907	●558	●63	215	186
10go-unip	●6.05	29	6.18	300	61	●46	●51	205	139
twitter	211	771	●88	9,810	30,948	●648	●17	340	27
web-uk	310	995	●262	15,190	2,544	●1,733	●200	522	305

Table 11: *DAG* reduction *vs* Backbone: *TF*.

Dataset	Index Size (MB)			Index Construction Time (ms)			Query Time (ms)		
	TF_*	TF_B	TF_{B*}	TF_*	TF_B	TF_{B*}	TF_*	TF_B	TF_{B*}
amaze	●0.02	0.16	0.06	2	2	2	●11	54	29
kegg	●0.02	0.15	0.07	10	2.39	●2.15	●12	86	34
xmark	●0.08	0.27	0.18	13	7	7	26	37	34
citeseer	0.52	0.77	0.59	51	33	●22	●53	133	89
pubmed	0.64	0.67	0.55	48	24	●23	●53	117	79
arxiv	5.34	6.25	●2.06	614	190	●61	●248	540	298
email	●0.95	9.62	2.20	125	8	●7	●14	169	38
unip150m	●140	1,058	371	31,366	2,424	2,230	●41	1,300	49
wiki	●8.86	96	10	1,090	74	●47	●8	141	9
LJ	●4.21	41	8.29	761	46	●31	●41	329	85
web	●2.20	16	6.32	393	35	●26	●98	307	207
05Patent	●29	83	74	6,855	1,379	●1,214	74	56	76
citeseerx	631	824	●296	41,536	33,970	●9,823	96	156	127
dbpedia	●30	164	90	7,194	1,633	●840	213	364	276
govwild	2,693	464	●419	114,950	●6,033	6,529	452	348	369
Patent	4,231	1,742	●1,384	197,125	106,760	●76,510	84	96	101
go-unip	●40	434	48	6,224	4,089	●1,489	●102	220	205
10go-unip	8.61	32	●8.56	994	367	●253	●78	202	141
twitter	●71	766	82	13,289	1,337	●630	●18	343	21
web-uk	●324	-	407	67,368	-	●29,637	●175	-	305

Proof of Theorem 6.1: According to Definition 6.1, we know $u \equiv v \Leftrightarrow u \equiv_V v$. Since $\mathcal{P}_{|V|}$ is a partition of V, and two nodes in different sets of $\mathcal{P}_{|V|}$ are inequivalent, we only need to prove that two nodes in the same set of $\mathcal{P}_{|V|}$ are equivalent to each other.

As shown by Fig. 5, after processing all nodes of V, each leaf node of the tree is a set $P \in \mathcal{P}_{|V|}$. All nodes of P have the same set of graph parents and children, which are denoted as the set of nodes on the path p from the root to the leaf node P. Since all nodes of V are already processed after getting $\mathcal{P}_{|V|}$, P will not be further divided into smaller sets, i.e., all graph parents and children in the given graph G^t for nodes of P can be found from p. Therefore, all nodes in the same set of $\mathcal{P}_{|V|}$ are definitely equivalent to each other.

As $\mathcal{P}_{|V|}$ is a partition of V, i.e., $\bigcup_{P \in \mathcal{P}_{|V|}} P = |V|$, we know that $\mathcal{P}_{|V|}$ contains all sets of equivalent nodes w.r.t. G. □

B. *DAG* REDUCTION AND REACHABILITY BACKBONE

Both *DAG* reduction and reachability backbone (abbreviated as Backbone) [12] reduce the size of the given *DAG*. We show their impacts using *IP*$^+$ [25] and *TF* [5] as the representative of Online-Search and Label-Only methods, respectively, and use subscripts "$*$", "B" and "$B*$" to denote the version working on G^ε, the Backbone graph of G and the Backbone graph of G^ε, respectively. Tables 10 and 11 show the results of *IP*$^+$ and *TF*.

On one hand, using *DAG* reduction is a better choice to accelerate reachability query processing compared with Backbone. This is because, Backbone was proposed to tackle the scalability bottleneck for methods that cannot process large graphs, such as [7, 16]. It was shown in [12] that even though existing algorithms can scale to large graphs with Backbone graphs, the cost behind the scalability is large index size and more index construction time. The query performance may degenerate due to its expensive search strategy (see Section 3 and [12] for a detailed description).

On the other hand, from Tables 10 and 11 we know that if the Backbone graphs are generated based on the result of our *DAG* reduction G^ε, then compared with generating Backbone graphs from G, for both algorithms, the index size, index construction time and query time can be improved significantly for most datasets.

C. DATASETS

For the datasets listed in Table 2, amaze[5], kegg[5], xmark[5], email[6], wiki[6], LJ[6] and web[6] are directed graphs initially, we transformed them into *DAG*s by coalescing each strongly connected component into a node of *DAG*s. All other datasets are *DAG*s initially. These datasets are used in the recent works [5, 12, 14, 24, 25, 27, 28].

Among these datasets, the first six are small datasets and are downloaded from the same web page, amaze, kegg and xmark are from the `sigmod08` zip file, and pubmed and arxiv are from the `sigmod09` zip file. These small datasets are mainly used to make comparison between existing algorithms and our algorithms on *TR* and *ER*. amaze and kegg are metabolic networks, both have a central node that has a large in-degree and out-degree. xmark is an XML document, citeseer[5], pubmed[5] and arxiv[5] are all citation networks. The following 14 large datasets are mainly used for testing the performance of reachability query processing. email is a *DAG* transformed from directed graph email-EuAll, which is a email network from a EU research institution. unip150m[5] (uniprotenc_150m) is a *DAG* obtained from the RDF graph of UniProt[7], which contains many nodes without incoming edges and few nodes without outgoing edges. wiki is a *DAG* transformed from Wikipedia talk (communication) network wiki-Talk. LJ is a DAG of an online social network soc-LiveJournal1. web is a *DAG* of web graph web-Google. 05Patent[8](05cit-Patent), Patent[5] (cit-Patents) and citeseerx[5] are all citation networks with out-degree of non-leaf nodes ranging from 10 to 30. dbpedia[9] is the *DAG* of a knowledge graph. govwild[10] is a *DAG* transformed from a large RDF graph. go-unip[5] (go_uniprot) and 10go-unip[8] (10go-uniprot) are *DAG*s transformed from the joint graph of Gene Ontology terms with the annotations file from the UniProt. twitter[10] is a *DAG* transformed from a large-scale social network obtained from a crawl of twitter.com [4]. web-uk[10] is a *DAG* of a web graph dataset [3].

[5]https://code.google.com/archive/p/grail/downloads
[6]http://snap.stanford.edu/data/index.html
[7]http://www.uniprot.org/
[8]http://pan.baidu.com/s/1bpHkFJx
[9]http://pan.baidu.com/s/1c00Jq5E
[10]https://code.google.com/p/ferrari-index/downloads/list