IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 5, 2020, accepted February 17, 2020, date of publication February 21, 2020, date of current version March 3, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2975650

One Edge at a Time: A Novel Approach Towards Efficient Transitive Reduction Computation on DAGs

XIAN TANG[1], JUNFENG ZHOU[2], YAXIAN QIU[2], XIANG LIU[1], YUNYU SHI[1], AND JINGWEN ZHAO[1]

[1]School of Electronic and Electrical Engineering, Shanghai University of Engineering Science, Shanghai 201620, China
[2]School of Computer Science and Technology, Donghua University, Shanghai 201620, China

Corresponding author: Junfeng Zhou (zhoujf@dhu.edu.cn)

This work was supported in part by the Natural Science Foundation of China under Grant 61472339 and Grant 61303040.

ABSTRACT Given a directed acyclic graph (*DAG*) G, G's transitive reduction (*TR*) G^{tr} is the unique *DAG* satisfying that G^{tr} has the minimum number of edges and has the same transitive closure (*TC*) as G. *TR* computation has been extensively studied during the past decades and was used in many applications, where the main problem is how to compute *TR* efficiently for large graphs. However, existing approaches have either large space complexity or higher time complexity, which makes them cannot compute *TR* efficiently on large dense graphs. We propose a novel approach for *TR* computation, which takes every single edge as the basic processing unit, and utilizes existing reachability algorithms to test whether it is redundant or not. In this way, we avoid the costly graph traversal operation of existing approaches. We identify the performance bottleneck and propose a set of heuristics to sort edges, such that to reduce the average processing cost of each edge. We show by experimental results that our approach works much better than all the existing approaches, and can be faster than the state-of-the-art approach by more than two orders of magnitude on large dense graphs.

INDEX TERMS Graph data management, directed acyclic graph, transitive reduction.

I. INTRODUCTION

Transitive reduction (*TR*) is a classical problem in graph theory. Given a directed acyclic graph (*DAG*) $G = (V, E)$, where V is the set of nodes and E the set of edges, G's *TR* is $G^{tr} = (V, E^{tr})$, where E^{tr} is the set of edges of G^{tr}, which is the unique *DAG* that has the least number of edges and same transitive closure (*TC*) as that of G [1]. Assume that for $\forall u, v \in V$, $u \rightsquigarrow v$($u \not\rightsquigarrow v$) denotes that there exists at least (does not exist) one directed path from u to v, i.e., u can (cannot) reach v. Considering reachability relationship, both G and G^{tr} satisfy that either $u \rightsquigarrow v$ or $u \not\rightsquigarrow v$. For example, Figure 1(a) is a *DAG* excerpted from the interaction network of Kyoto Encyclopedia of Genes and Genomes,[1] its *TR* is shown in Figure 1(b). The red dotted arrows in Figure 1(a) are redundant edges. Here, we say an edge $e = (u, v)$ is redundant if u can reach v through other nodes, which means that removing this edge from G does not change the *TC* of G. *TR* computation is to find and delete all redundant edges from the given *DAG* G, such that to get the unique *DAG* G^{tr}.

TR computation was one of the hot research issues during the past decades [2]–[13] and was used extensively in many applications [14]–[22] to simplify the computation or analysis, such as *TC* computation, reachability, citation network, social network, bioinformatics network and temporal network, etc. For example, by computing the *TR* of citation network, we may reveal the real cross-domain impact of a paper, patent or court judgement, which cannot be observed by the previous approach [14]. For another example, in [23], the authors proposed to compress a *DAG* based on equivalence relationship between nodes, the time complexity is as high as $O(|V|(|V| + |E|))$ and cannot scale to large graphs. After computing *TR*, however, we can simplify the equivalence relationship [12], [13]. Given the *TR* of the input

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

[1]http://www.genome.jp/kegg/

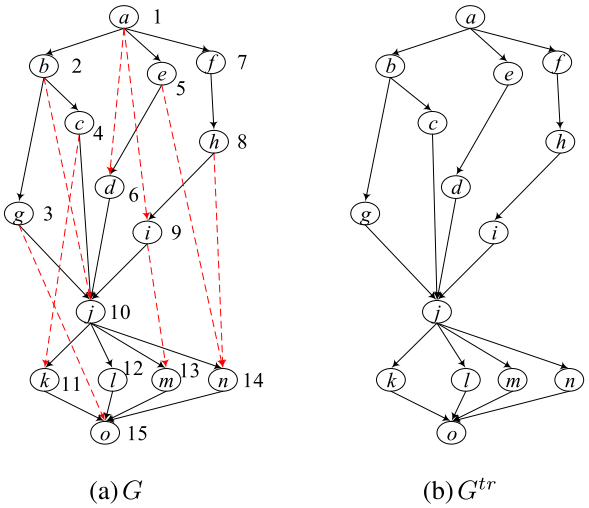


FIGURE 1. ***DAG G* and its *TR* G^{tr}, where the integer on the right of each node *v* is *v*'s topo-order.**

DAG, the time complexity of performing the compression based on simplified equivalence relationship can be reduced to $O(|V| + |E|)$.

However, for existing approaches, the cost of *TR* computation is high in both time and space. When considering only the number of nodes, the best approach that computes *TR* using matrix multiplication [7] has time complexity $O(|V|^{2.3727})$ and space complexity $O(|V|^2)$, therefore cannot scale to large graphs. Considering this problem, researcher proposed several approaches for *TR* computation based on graph traversal [2], [3], [5], [11]–[13]. These approaches usually have smaller space complexity, therefore can be used to process larger graphs. The naive approach is *DFS*, which processes each node v separately to find all redundant edges starting from v. Although space complexity is $O(|V|)$, *DFS* suffers from the highest time complexity $O(|V|(|V|+|E|))$. To reduce the time complexity, *PTR* [3] first decomposes the given graph into k paths, then processes all nodes in descending topological order to compute the *TR*. The time complexity is $O(|E| + k|V| + k|E^{tr}|)$. Compared with the approach that uses matrix multiplication [7], the space complexity is reduced to $O(k|V|)$. In practice, k could be as large as $|V|$, which makes the space complexity of *PTR* degenerate to $O(|V|^2)$ and cannot scale to large graphs. The most recent approach is *buTR* [12], [13], which identifies the overlap between *TC*s of nodes, and makes improvements by avoiding processing the overlapping repeatedly. The time complexity of *buTR* is $O(|V| + |E| + d\triangle|V|))$, where $d = |E|/|V|$ is the average degree and $\triangle$ is the average number of visited nodes for each processed node in computing, i.e., $\triangle$ is the average size of non-overlapping part, or the average size of *TC* difference. As $d|V| = |E|$, the average cost of processing each edge by *buTR* is $\triangle$. Even though *buTR* was shown to be much more efficient than previous approaches [12], [13], its performance is dominated by the size of the non-overlapping part, i.e., $\triangle$, which is small only when the given graph is sparse. When the given graph becomes dense, the size of non-overlapping part, i.e., $\triangle$, increases dramatically, and the performance of *buTR* degenerates significantly.

Considering the above problems, we propose to compute *TR* in a completely reverse direction. Different from existing approaches that compute *TR* based on graph traversal, the basic idea of our approach is computing *TR* by processing each edge separately without graph traversal. We make the following contributions.

1) We propose a general framework that takes existing algorithms for reachability queries answering as a plugin to avoid expensive graph traversal operation when computing *TR*. The space complexity is $O(|V|)$, and the time complexity is $O(d\theta|E|)$, where θ is the average cost of answering a reachability query by the underlying reachability algorithm.
2) We identify that the performance bottleneck of our approach lies in the calling times of the underlying algorithm to answer reachability queries. We then propose several heuristics to significantly reduce the calling times of the underlying reachability algorithm, such that our approach can scale to large and dense graphs.
3) We conduct rich experiments on both real and synthetic datasets. The experimental results show that compared with existing approaches, our algorithm works much better on both sparse and dense graphs, and can scale to large graphs.

The remain of this paper is organized as follows. In Section II, we discuss the preliminaries and related work. In Section III, we discuss the baseline approach for *TR* computation, and discuss two optimizations in Section IV. We show the detailed experimental results in Section V, and conclude our work in Section VI.

II. BACKGROUND AND RELATED WORK

A. PRELIMINARIES

Given a directed acyclic graph (*DAG*) $G = (V, E)$, where V is the set of nodes and E the set of edges, we use $in_G(u) = \{v|(v, u) \in E\}$ to denote the set of in-neighbors of u in G, and $out_G(u) = \{v|(u, v) \in E\}$ the set of out-neighbors of u. We use $in^*_G(u)$ to denote 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 use $X = \{1, 2, \ldots, |V|\}$ to denote a topological order (topo-order) of G, which can be got by a topological sorting (topo-sorting) on G. A topo-sorting of G is a mapping $t : V \rightarrow X$, 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 $O(|V| + |E|)$ [3].

The transitive closure (*TC*) of G is $G^* = (V, E^*)$, where $E^* = \{(u, v)|u, v \in V,\ v \in out^*_G(u)\}$. For simplicity, we use $TC(v)$ to denote $out^*_G(v)$, which we call as the transitive closure of v. G's *TR* $G^{tr} = (V, E^{tr})$ is the unique *DAG* [1] that has least number of edges and the same *TC* $G^* = (V, E^*)$ with G, satisfying $E^{tr} \subseteq E \subseteq E^*$. Given an edge $e = (u, v)$, if u can reach v through other nodes, we say e is redundant.

TABLE 1. Table of notations.

Notation	Description
$G = (V, E)$	a *DAG* with a node set V and an edge set E
$G^{tr} = (V, E^{tr})$	G's *TR* with a node set V and an edge set $E^{tr} \subseteq E$
X	a topo-order of a *DAG* G
t_v	node v's topo-order in X
$in_G(v)$	the set of in-neighbors of v in a *DAG* G
$in^*_G(v)$	the set of nodes that can reach v in a *DAG* G
$out_G(v)$	the set of out-neighbors of v in a *DAG* G
$out^*_G(v)$	the set of nodes v can reach, or v's *TC*, or $TC(v)$

Therefore, each edge in $E \setminus E^{tr}$ is a redundant edge, and each edge in E^{tr} is not a redundant edge. We show important notations in Table 1 for ease of reference.

Problem Statement: Given a *DAG* G, return its *TR* G^{tr}.

B. RELATED WORK

Existing approaches on *TR* computation can be generally divided into two categories: (1) matrix multiplication [1], [7] and (2) *DAG* traversal [2], [3], [5], [11]–[13]. We discuss the details below.

1) MATRIX MULTIPLICATION

In [1], the authors proposed algorithms for *TR* computation based matrix multiplication, and proved that both *TR* and *TC* computation share the same time complexity.

Following this research direction, many works made improvements on the time complexity of *TR* computation, they try to make it close to $O(|V|^2)$. The best known algorithm using matrix multiplication is *CWO* [7], which made improvements on *CW* [24]. The time and space complexities of *CWO* are $O(|V|^{2.3727})$ and $O(|V|^2)$, respectively.

Obviously, given limited memory size, these algorithms cannot scale to large graphs, due to higher time and space complexities.

2) *DAG* TRAVERSAL

The naive approach to compute *TR* by *DAG* traversal will process each node v separately. For the node v processed in each iteration, we perform depth-first-search (*DFS*) or breadth-first-search (*BFS*) from v to visit nodes in $out^*_G(v)$, such that to find all redundant edges starting from v. After performing *DFS*/*BFS* on all nodes, we get the *TR* G^{tr}. We call the two approaches *DFS* and *BFS*, respectively. Even though *DFS* and *BFS* reduce the space complexity to $O(|V|)$, they suffer from higher time complexity $O(|V|(|V| + |E|))$, which makes them cannot scale to large graphs either.

To improve the time complexity for *TR* computation, *GK* [11] processes all nodes in descending topo-order. For each node v, it quickly computes $TC(v)$ by the transitive closures of v's out-neighbors, which is used to quickly find redundant edges. The time complexity is $O(|V||E^{tr}|)$, which is smaller than $O(|V||E|)$. However, as *GK* needs to use bit vector to maintain the *TC* of each node, its space complexity is as high as $O(|V|^2)$ and cannot scale to large graphs.

In [3], the authors proposed a path decomposition based approach, namely *PTR*, for *TR* computation. The basic idea is to first divide the given *DAG* into k paths, then process all nodes in descending topo-order, during which it represents each node's *TC* by at most k nodes. For each processed node v, *PTR* visits v's child nodes in ascending topo-order. It finds redundant edges starting from v based on the help of *TC*s of v's child nodes, and computes v's *TC* incrementally. Assume that the cost of checking whether an edge is redundant is $O(1)$. For each non-redundant edge starting from v, *PTR* needs $O(k)$ time to update the *TC* of v. Therefore, the time complexity of *PTR* is $O(|E| + k|V| + k|E^{tr}|)$. For each node v, *PTR* needs k nodes to maintain v's *TC*, thus its space complexity is $O(k|V|)$. In practice, k could be as large as $|V|$, thus the space complexity degenerates to $O(|V|^2)$, which still makes *PTR* cannot scale to large graphs.

The most recent algorithm on *TR* computation is *buTR* [12], [13], which makes improvements by avoiding the processing of the overlapping between *TC*s of nodes. The idea is based on the fact that if $u \rightsquigarrow v$, then $out^*_G(u) \supset out^*_G(v)$. Therefore, if we first process v and remember $out^*_G(v)$, when processing u, we do not need to visit nodes in $out^*_G(v)$. That is, *buTR* avoids processing the overlapping part $out^*_G(v)$. When processing u, *buTR* only visits the non-overlapping part $out^*_G(u) \setminus out^*_G(v)$. *buTR* processes all nodes in a bottom-up manner. Its time and space complexities are $O(|V| + |E| + d\triangle|V|)$ and $O(|V|)$, respectively. For each node, $\triangle$ is the average size of the non-overlapping part, i.e., $|out^*_G(u) \setminus out^*_G(v)|$. It was shown in [12], [13] that *buTR* is much more efficient than other existing approaches and can scale to large graphs for *TR* computation. However, this is only true for sparse graphs where $\triangle$ is small. For graphs where $\triangle$ becomes large, such as dense graphs, *buTR* is not efficient anymore.

Summarization: Table 2 shows the comparison of several algorithms on *TR* computation, from which we know that for existing algorithms, *buTR* has the best time and space complexities. As $d|V| = |E|$, the time complexity of *buTR* can be represented as $O(\triangle|E|)$. Obviously, the performance of *buTR* is dominated by the average size of the non-overlapping part $\triangle$, which denotes the average processing cost of each edge. Compared with *buTR*, our approach *TR-O* belongs to neither "Matrix", nor "Traversal". As it processes each edge independently without graph traversal, we put it into a new category "Edge". As shown in Table 2, the processing cost of each edge for our approach *TR-O* in the "worst" case is $\theta d^S_{\max}$. From Table 2 it is difficult to tell which one is better, as it shows the average cost for *buTR* and "worst" cost for *TR-O*$^+$, we will show in the experiment that the "average" processing cost of *TR-O*$^+$ is much less than that of *buTR*, especially when the graph becomes dense.

Other TR Algorithms: Besides the above two kinds of approaches, there are many works that focus on other aspects of *TR* computation. For example, [6] studied the problem of approximate *TR* computation. Reference [10] studied *TR* computation in parallel. Reference [9] studied *TR* computation for dynamic graphs. References [2], [5]

TABLE 2. Comparison of algorithms for *TR* computation.

Algorithm	Time complexity	Space complexity	Category
CWO [7]	$O(\|V\|^{2.3727})$	$O(\|V\|^2)$	Matrix
DFS (*BFS*)	$O(\|V\|(\|V\|+\|E\|))$	$O(\|V\|)$	Traversal
GK [11]	$O(\|V\|\|E^{tr}\|)$	$O(\|V\|^2)$	
PTR [3]	$O(\|E\|+k\|V\|+k\|E^{tr}\|)$	$O(k\|V\|)$	
buTR [12]	$O(d\triangle\|V\|)$, or $O(\triangle\|E\|)$	$O(\|V\|)$	
TR-O$^+$	$O(\theta d^S_{\max}\|E\|)$	$O(\|V\|)$	Edge

proposed linear-time algorithm for *TR* computation based on the assumption that the input *DAG G* is *N*-free, however, a linear-time recognition algorithm for *N*-free graphs is still an open problem, and *G* is usually not *N*-free in practice [5].

Reachability Index: During the past decades, researchers have proposed many reachability indexes, which can be used in *TR* computation to help check the redundancy of each edge. Following [25], [26], the existing approaches are classified into two categories: *Label*−*Only* and *Label*+*G*. By *Label*−*Only*, it means that the index conveys the complete reachability information, and the given query $u? \rightsquigarrow v$ can be answered by comparing labels of u and v. By *Label*+*G*, it means that the index covers partial reachability information, and we may need to conduct *DFS*/*BFS* from u to check whether u can reach v, if we cannot get the result by comparing labels of u and v.

The *Label*−*Only* approaches try to compress *TC* to get a smaller index size to facilitate queries answering. The recent work includes *TF* [27], *DL* [28], *PLL* [29] and *TOL* [30]. The idea is to assign each node u a label $L_u = \{L_{out}(u), L_{in}(u)\}$, where $L_{out}(u)(L_{in}(u))$ is the out (in) label of u consisting of a set of nodes that can reach (be reached by) u. Then, $u? \rightsquigarrow v$ can be answered by computing the result of $L_{out}(u) \bigcap L_{in}(v)$. If $L_{out}(u) \bigcap L_{in}(v) \neq \emptyset$, then $u \rightsquigarrow v$, otherwise $u \not\rightsquigarrow v$.

The *Label*+*G* approaches assign each node u a label L_u that covers partial reachability information. The recent *Label*+*G* approaches include *GRAIL* [31], [32], *Yes-GRAIL* [33], *FERRARI-G* [34], *FELINE* [35], *IP*$^+$ [25] and *BFL*$^+$ [26]. Given a query $u? \rightsquigarrow v$, if comparing the labels of u and v cannot tell the result, *Label*+*G* algorithms need to perform *BFS*/*DFS* to get the final answer.

Here, the usability of a reachability algorithm for checking the redundancy of each edge lies in whether the reachability index can be efficiently constructed. If the reachability index cannot be constructed efficiently, it is meaningless to use it for *TR* computation. For existing reachability indexes, as *Label*−*Only* approaches need to compute the complete reachability information, they usually consume much longer time than *Label*+*G* approaches w.r.t. index construction. In this paper, we adopt *BFL*$^+$, which is a *Label*+*G* approach and the index can be constructed more efficiently than other *Label*+*G* approaches [26].

III. THE BASELINE ALGORITHM FOR *TR* COMPUTATION

According to the definition of *TR*, we need to identify all redundant edges to get the *TR*. Assume that we can correctly judge whether a given edge is redundant or not, the basic idea of our approach is directly based on the definition. That is, given a *DAG G*, we check whether each edge of *G* is redundant or not, and delete all redundant edges from *G* to return its *TR* G^{tr}.

Here, the key problem is how to check whether a given edge is redundant or not. Our approach is based on the following result.

Theorem 1: Given an edge $e = (u, v)$, we say e is a redundant edge iff there exists a node $w \in out_G(u)$, $w \neq v$, such that w can reach v. □

Proof 1: According to the definition of redundant edge, e is redundant iff there exists a node $x(\neq v)$, such that u can reach $x(x \in out^*_G(u))$ and x can reach v $(v \in out^*_G(x))$. We prove it from two aspects.

First, by $w \in out_G(u)$, we know that $w \in out^*_G(u)$. By w can reach v, we know that $v \in out^*_G(w)$. Therefore, if there exists a node $w \in out_G(u)$, $w \neq v$, and w can reach v, then e is redundant.

Second, if e is redundant, it means that there exists a node $x(\neq v)$, such that u can reach $x(x \in out^*_G(u))$ and x can reach v $(v \in out^*_G(x))$. As u can reach x, we know that there exists at least one node $w \in out_G(u)$, such that w can reach x. As x can reach v, we know that w can reach v. That is, if e is redundant, we know that there exists a node $w \in out_G(u)$, $w \neq v$, and w can reach v.

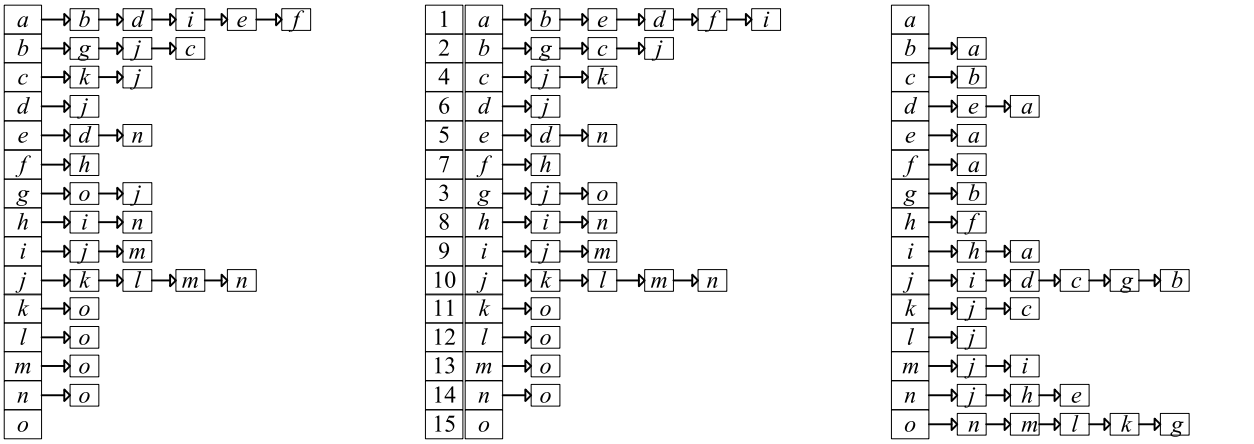
Therefore, e is a redundant edge iff there exists a node $w \in out_G(u)$, $w \neq v$, such that w can reach v. □

According to Theorem 1, we know that if we can correctly identify the reachability relationship between two nodes, then we can correctly tell whether a given edge is redundant or not. This can be done by directly calling either one of the existing reachability query algorithms.

Algorithm 1 *TR-B* $(G = (V, E))$

```
1  construct a certain reachability index RI for G
2  E ← sortEdge(G)
3  while (¬ isEmpty(E) do
4        (u, v) ← deQueue(E)  /*remove an edge from E*/
5        if ( isRedundant(u, v) = TRUE) then
6              delete (u, v) from G
7  Function sortEdge(G)
8  for each (u ∈ V) do
9        for each (v ∈ out_G(u)) do
10             enQueue(E, (u, v))
11 return E
12 Function isRedundant(u, v)
13 for each (w ∈ out_G(u), w ≠ v) do
14       if (RI(w, v) = TRUE ) then
15             return TRUE          /*w ⇝ v, and (u, v) is
   redundant*/
16 return FALSE
```

As shown by Algorithm 1, we first construct a certain reachability index *RI* in line 1, which is used to check the reachability relationship between two nodes in isRedundant() function. In line 2, we sort all edges according to a certain



(a) before topo-sorting (b) after topo-sorting (c) the inverse adjacency list w.r.t. (b)

FIGURE 2. The statuses of the adjacency list of G in Figure 1.

metric, such that edges are processed in a special order. After that, we process all edges one by one in lines 3-6. In each iteration, we pick up an edge in line 4, then check whether it is redundant or not in line 5. If the edge is redundant, we delete it in line 6. In Algorithm 1, all edges are processed in the order same as they are maintained in the adjacency lists, as shown by the sortEdge() function. For each edge, we check whether it is redundant or not by calling Function isRedundant(), which works based on Theorem 1. Assume that RI can correctly identify the reachability relationship between two nodes, the correctness of Algorithm 1 for TR computation can be guaranteed by Theorem 1 and the definition of TR.

Example 1: Assume that the adjacency list of G in Figure 1(a) is show as the one in Figure 2(a). Without loss of generality, assume that the edges are processed in a top-down manner from left to right. The first processed edge is (a, b). By Algorithm 1, we need to check whether there exist a node $x(\neq b)$ in a's out-neighbors $\{d, i, e, f\}$, such that x can reach b. After calling $RI(x, b)$ four times, we know that none of a's out-neighbors can reach b, thus (a, b) is a not a redundant edge. The second processed edge is (a, d). Similarly, we need to check whether there exist a node $x(\neq d)$ in a's out-neighbors $\{b, i, e, f\}$, such that x can reach d. After calling $RI(x, d)$ three times to check the reachability between b and d, i and d, and e and d, we find that e can reach d, thus (a, d) is a redundant edge. The following edges are processed similar. After processing all edges, we find all redundant edges shown as the dotted arrows in Figure 1(a). Then we delete them from G and get the TR G^{tr} of G shown in Figure 1(b). □

Analysis: In Algorithm 1, we construct BFL^+ [26] index as RI, the time cost of index construction is $O(s(|V| + |E|))$, where s is small user-given parameter. In line 2, we output all edges to a queue, the cost is $O(|V| + |E|)$. In lines 3-6, we check for each edge, whether it is redundant or not. For each edge (u, v), the time cost of checking its redundancy is $O(|out_G(u)|\theta)$, where θ is the cost of answering a reachability query by $RI()$. Therefore, the cost of processing all edges in the worst case is $O(\theta \sum_{u \in V} |out_G(u)|^2) \leq O(\theta d_{\max}|E|)$, where $d_{\max} = \max\{|out_G(v)||v \in V\}$. Note that in lines 3-6, when we find a redundant edge, we only need to mark it. The deletion of all edges can be done after processing all edges, and the time cost of deletion all redundant edges is $O(|V| + |E|)$. As usually in practice, s is a small integer and $d_{\max} < \theta d_{\max}$, the time complexity of Algorithm 1 is $O(\theta d_{\max}|E|)$.

For space complexity, the index size of BFL^+ is $O(s|V|)$, the size of the queue $\mathcal{E}$ is $O(|E|)$. In fact, we do not need to maintain $\mathcal{E}$ during the computation, we only need to process edges in the order as they are pushed into $\mathcal{E}$. Since s is a small integer, the space complexity of Algorithm 1 is $O(|V|)$.

IV. OPTIMIZATION

Consider the time complexity of Algorithm 1 again. Given a certain graph and a reachability approach, as the cost θ of answering a reachability query by $RI()$ and the number of edges $|E|$ cannot be changed, we know that for Algorithm 1, the dominating factor that affect the performance is the calling times of $RI()$ function for each edge. For Algorithm 1, the calling time of $RI()$ for each edge (u, v) is $|out_G(u)| - 1$. And for G in Figure 1(a), Algorithm 1 calls $RI()$ function 46 times to get the TR of G.

To reduce the calling times of $RI()$ function to improve the performance of TR computation, the basic idea of our approach is sorting all the edges, such that the edges can be processed in a certain order to avoid the unnecessary call of $RI()$ function. In this section, we discuss two kinds of sorting techniques to make optimization.

A. TOPO-SORTING BASED PROCESSING ORDER

Given a DAG G, we can assign each node u a topo-order t_u by performing a topo-sorting on G. With the topo-orders of all nodes, we have the following results.

Theorem 2: Given two nodes u and v, if $t_u > t_v$, then u cannot reach v. □

Proof 2: Assume that u can reach v, then there exists, from u to v, at least one path $p = v_0, v_1, v_2, \ldots, v_{i-1}, v_i (v_0 = u \wedge v_i = v)$, where each pair of adjacent nodes $v_{j-1} (j \in [1, i])$ and v_j are the two nodes of edge (v_{j-1}, v_j). As $t_{v_{j-1}} < t_{v_j}$, we know that $t_u = t_{v_0} < t_{v_i} = t_v$, which contradicts the assumption. Therefore, if $t_u > t_v$, then u cannot reach v. □

According to Theorem 2, for a given edge (u, v), to check whether it is redundant or not, we do not need to check whether v can be reached from all nodes of $out_G(u) \setminus \{v\}$. Instead, we only need to check whether v can be reached from nodes in $out_G(u) \setminus \{v\}$ satisfying topo-order $< t_v$. To do this, we need to first perform topo-sorting to assign each node a topo-order. Then, edges need to be sorted in the order that can utilize Theorem 2 to reduce the calling times of $RI()$.

Theorem 3: Given an edge $e = (u, v)$, if $\forall w \in out_G(u)(w \neq v)$, $t_v < t_w$, then e is not a redundant edge. □

Proof 3: Assume that e is a redundant edge, then there must exist at least one node $w' \in out_G(u)$, such that w' can reach v. According to the proof of Theorem 2, we know that $t_{w'} < t_v$, which contradicts the assumption that $\forall w \in out_G(u)(w \neq v)$, $t_v < t_w$. Therefore, if $\forall w \in out_G(u)(w \neq v)$, $t_v < t_w$, we know that e is not a redundant edge. □

According to Theorem 3, for a given edge $e = (u, v)$, if v has the smallest topo-order among u's out-neighbors, then e is not a redundant edge. Based on the above results, we have Algorithm 2 for *TR* computation, where changes over Algorithm 1 are marked with underlines. In line 8, we perform topo-sorting to get the topo-order. In lines 10-11, we push edges that have the same starting node into queue $\mathcal{E}$ in ascending topo-order w.r.t. the ending node. After that, we process all edges one by one in lines 14-17 by calling isRedundant() function, which works based on Theorem 2 and Theorem 3.

It is worth noting that in Function isRedundant(), if (u, w) is a redundant edge, we do not need to call $RI(w, v)$ to check whether w can reach v. The correctness is guaranteed by the following result.

Theorem 4: Given a node u, assume that all nodes in $out_G(u) = \{v_1, v_2, \ldots, v_{|out_G(u)|}\}$ are sorted in ascending topo-order, and (u, v_k) is a redundant edge, where $v_k \in out_G(u)$. When checking the redundancy of edge (u, v_j), where $k < j \leq |out_G(u)|$, we do not need to check whether v_k can reach v_j. □

Proof 4: As (u, v_k) is a redundant edge, according to Theorem 1, we know that there exists a node $v_x \in out_G(u)$ satisfying that $x < k$ and v_x can reach v_k. Therefore, if v_k can reach v_j, we know that v_x can reach v_j. Since v_x is processed before v_k, we do not need to check whether v_k can reach v_j. □

Example 2: Assume that the adjacency list of G in Figure 1(a) is shown in Figure 2(a). To reduce the calling times of $RI()$, we first perform topo-sorting in line 8 in Algorithm 2. After that, the adjacency list is shown in Figure 2(b). Then, we push edges that have the same starting node into queue $\mathcal{E}$ in ascending topo-order w.r.t. the ending node.

Algorithm 2 *TR-O* $(G = (V, E))$

```
1  construct a certain reachability index RI for G
2  E ← sortEdge-O(G)
3  while (¬ isEmpty(E)) do
4        (u, v) ← deQueue(E)   /*remove an edge from E*/
5        if ( isRedundant-O(u, v)= TRUE) then
6              delete (u, v) from G
7  Function sortEdge-O(G)
8  perform a topo-sorting on G
9  for each (u ∈ V) do
10       for each (v ∈ out_G(u) in ascending topo-order) do
11             enQueue(E, (u, v))
12 return E
13 Function isRedundant-O(u, v)
14 for each (w ∈ out_G(u) satisfying t_w < t_v) do
15       if (RI(w, v) = TRUE ) then
16             return TRUE          /*w ⇝ v, and (u, v) is
   redundant*/
17 return FALSE
```

Consider a, the edges that start from a are pushed into $\mathcal{E}$ in the order $(a, b), (a, e), (a, d), (a, f), (a, i)$. To check their redundancy, we first process edge (a, b). According to Theorem 3, it is not a redundant edge. The second processed edge is (a, e). According to Theorem 2, we only need to check whether b can reach e. As b cannot reach e by calling $RI()$ function, we know that (a, e) is not a redundant edge. The third processed edge is (a, d). By Theorem 2, we know that we only need to check whether b or e can reach$d$. As e can reach d, we know that (a, d) is redundant. The fourth processed edge is (a, f). Since both b and d cannot reach f, we know that (a, f) is not a redundant edge. Note that when processing (a, f), we do not need to check whether d can reach f or not according to Theorem 4. The last processed edge is (a, i). To check whether it is redundant or not, we need to check whether b, e or f can reach i. As f can reach i, we know that (a, i) is a redundant edge. The following processing is similar. Compared with Algorithm 1 that needs to call $RI()$ 46 times to process all edges, $RI()$ has been called 21 times in Algorithm 2. □

Analysis: Given a *DAG* G, the cost of performing topo-sorting in line 8 of Algorithm 2 is $O(|V| + |E|)$ [3]. Thus the cost of line 2 is $O(|V| + |E|)$. Lines 3-6 enumerate every edge and check whether it is redundant or not. As we delete all redundant edges after processing all edges, for each processed edge (u, v), we need to scan all nodes $v_i \in out_G(u)$ satisfying $t_{v_i} < t_v$, and call $RI()$ function only if the edge (u, v_i) is not a redundant edge. Therefore, the time complexity of Algorithm 2 is $O(\theta d^{tr}_{\max}|E|)$, where $d^{tr}_{\max} = \max\{|out_{G^{tr}}(v)||v \in V\}$. Besides, both Algorithm 1 and Algorithm 2 have the same space complexity.

B. DEGREE BASED PROCESSING ORDER

By performing topo-sorting, Algorithm 2 reduces the calling times of $RI()$ function. In Algorithm 2, all edges with the

same starting node are clustered together, and are sorted in ascending order w.r.t. the topo-orders of their ending nodes. Therefore, edges starting with the same node are processed together, and the processing cost will increase when the starting node has many out-neighbors, i.e., *TR-O* works well only when all nodes have small number of out-neighbors.

Example 3: Consider G in Figure 1(a). The four edges starting from node j are processed together. The processing order is (j,k), (j,l), (j,m) and (j,n). To process (j,m), we need to call $RI()$ two times to check whether k and l can reach m. To process (j,n), we need to call $RI()$ three times to check whether k, l and m can reach n. Obviously, given a node u, assume that there is no redundant edge starting from u, Algorithm 2 will call $RI()$ function $|out_G(u)|-1$ times to process the last edge. And $RI()$ will be called $O(|out_G(u)|^2)$ times to process all edges starting from u. □

Theorem 5: Given an edge $e=(u,v)$, we say e is a redundant edge iff there exists a node $w \in in_G(v), w \neq u$, such that u can reach w. □

Proof 5: According to the definition of redundant edge, e is redundant iff there exists a node x, such that u can reach $x(x \in out^*_G(u))$ and x can reach v $(v \in out^*_G(x))$, which also means that $u \in in_G(x) \wedge x \in in_G(v)$. Similar to the proof of Theorem 1, we prove it from two aspects.

First, by $w \in in_G(v)$, we know that $w \in in^*_G(v)$. By u can reach w, we know that $u \in in^*_G(w)$. Therefore, if there exists a node $w \in in_G(v), w \neq u$, and u can reach w, then e is redundant.

Second, if e is redundant, it means that there exists a node x, such that u can reach $x(u \in in^*_G(x))$ and x can reach v $(x \in in^*_G(v))$. As x can reach v, we know that there exists at least one node $w \in in_G(v)$, such that x can reach w. As u can reach x, we know that u can reach w. That is, if e is redundant, we know that there exists a node $w \in in_G(v), w \neq u$, and u can reach w.

Therefore, e is a redundant edge iff there exists a node $w \in in_G(v), w \neq u$, such that u can reach w. □

According to Theorem 5, we know that for an edge $e=(u,v)$, if u cannot reach any parent of v, then e is not a redundant edge. By combining Theorem 1 and Theorem 5 together, we have two different ways to check the redundancy of each edge. That is, $e=(u,v)$ is redundant iff either one of the following conditions holds.

- $(C1)$ There exists a node $w \in out_G(u), w \neq v$, such that w can reach v, or,
- $(C2)$ There exists a node $w \in in_G(v), w \neq u$, such that u can reach w.

Given an edge $e=(u,v)$, if we use the first condition $C1$ to check the redundancy of e, we will need to call $RI()$ function $|out_G(u)|-1$ times in the worst case. On the other hand, if we use the second condition $C2$, we need to call $RI()$ function $|in_G(v)|-1$ times in the worst case. Therefore, to further reduce the calling times of $RI()$ function to improve the overall performance, for each edge $e=(u,v)$, we need to first make comparison between $|out_G(u)|$ and $|in_G(v)|$. If $|out_G(u)| > |in_G(v)|$, then we should use the second condition $C2$, otherwise use the first condition $C1$. For instance, consider processing edge (j,l) in G of Figure 1(a). If we use the first condition $C1$, we need to call $RI()$ once to check whether k can reach l. As a comparison, we do not need to call $RI()$ if we use the second condition $C2$.

When using the above idea to make optimization, it seems that we do not need to sort all edges $e=(u,v)$ in advance, due to that we know $|out_G(u)|$ and $|in_G(v)|$. However, if edges are not sorted, we cannot reduce the redundant call of $RI()$ as Algorithm 2 does. Furthermore, to facilitate processing edges based on the second condition $C2$, we need to use the inverse adjacency list, which also should be sorted already after performing topo-sorting.

Definition 1: Given a node u, we divide u into two nodes, $u^{\uparrow}$ and $u^{\downarrow}$, where $u^{\uparrow}(u^{\downarrow})$ is called the UP-node (DOWN-node) of u considering only u's in-neighbors (out-neighbors), and $d(u^{\uparrow})=|in_G(u)|(d(u^{\downarrow})=|out_G(u)|)$ is called the degree of $u^{\uparrow}(u^{\downarrow})$.

By Definition 1, for all nodes in the given *DAG* $G=(V,E)$, we have two sets of nodes, one is $V^{\uparrow}=\{v_1^{\uparrow}, v_2^{\uparrow}, \cdot, v_{|V|}^{\uparrow}\}$ containing only UP-nodes, the other is $V^{\downarrow}=\{v_1^{\downarrow}, v_2^{\downarrow}, \cdot, v_{|V|}^{\downarrow}\}$ containing only DOWN-nodes. Obviously, first processing nodes with smaller degree means less calling times to the $RI()$ function.

Based on the above discussion, we have Algorithm 3 for *TR* computation. Compared with Algorithm 2, the differences lie in functions sortEdge-O$^+$() and isRedundant-O$^+$(). In Function sortEdge-O$^+$(), we first perform a topo-sorting. Here, different with Algorithm 2 that only produces the sorted adjacency list, Algorithm 3 produces both adjacency list and inverse adjacency list, as shown in Figure 2(b) and (c). Then, in line 10, we sort all the $2|V|$ nodes in $\mathcal{V}=V^{\uparrow}\cup V^{\downarrow}$ in ascending order w.r.t. node degrees. After that, in lines 11-19, we push all edges into queue $\mathcal{E}$ by visiting nodes of $\mathcal{V}$ one by one. In this way, for any pair of edges $e_1=(u_1,v_1)$ and $e_2=(u_2,v_2)$, if e_1 is pushed into $\mathcal{E}$ before e_2, then we know that $\min\{|out_G(u_1)|, |in_G(v_1)|\} \leq \min\{|out_G(u_2)|, |in_G(v_2)|\}$. That is, the calling times of $RI()$ to process e_1 guarantees to be no more than that of e_2, if there is no redundant edge. It is worth noting that even though there are $2|V|$ nodes in $\mathcal{V}$, and $\sum_{v\in\mathcal{V}} d(v)=\sum_{v\in V^{\uparrow}}|in_G(v)|+\sum_{v\in V^{\downarrow}}|out_G(v)|=2|E|$, we only push $|E|$ edges into $\mathcal{E}$, which can be guaranteed by line 14 and 18.

After sorting all edges and push them into $\mathcal{E}$, Function isRedundant-O$^+$() is used to check whether a given edge $e=(u,v)$ is redundant or not. In line 22, we determine to use which condition to check e's redundancy. If $|out_G(u)| > |in_G(v)|$, we use condition $C2$ in lines 23-25 to check e's redundancy, otherwise, we use condition $C1$ in lines 27-29 to check e's redundancy.

Example 4: Assume that the adjacency list of G in Figure 1(a) is shown in Figure 2(a). Algorithm 3 will perform topo-sorting in line 8 and produce the adjacency list and the inverse adjacency list in Figure 2(b) and (c),

Algorithm 3 *TR-O*$^+$ $(G = (V, E))$

```
1  construct a certain reachability index RI for G
2  E ← sortEdge-O+(G)
3  while (¬ isEmpty(E)) do
4        (u, v) ← deQueue(E)  /*remove an edge from E*/
5        if ( isRedundant-O+(u, v)= TRUE) then
6              delete (u, v) from G
7  Function sortEdge-O+(G)
8  perform a topo-sorting on G
9  let V↑ = {v1↑, v2↑, ·, v|V|↑}, V↓ = {v1↓, v2↓, ·, v|V|↓}
10 sort all nodes in V = V↑ ∪ V↓ in ascending degrees
11 for each (u ∈ V in ascending degrees) do
12       if (u ∈ V↑) then
13             for each (v ∈ inG(u) in descending topo-order)
   do
14                   if ((v, u) ∉ E) then
15                         enQueue(E, (v, u))
16       if (u ∈ V↓) then
17             for each (v ∈ outG(u) in ascending topo-order)
   do
18                   if ((u, v) ∉ E) then
19                         enQueue(E, (u, v))
20 return E
21 Function isRedundant-O+(u, v)
22 if (|outG(u)| > |inG(v)|) then
23       for each (w ∈ inG(v) satisfying tw > tv) do
24             if (RI(u, w) = TRUE ) then
25                   return TRUE       /*u ⇝ w, and (u, v) is
   redundant*/
26 else
27       for each (w ∈ outG(u) satisfying tw < tv) do
28             if (RI(w, v) = TRUE ) then
29                   return TRUE       /*w ⇝ v, and (u, v) is
   redundant*/
30 return FALSE
```

respectively. Then in line 10, it sorts all the $2|V|$ nodes in $\mathcal{V}$ in ascending order w.r.t. their degrees. The resulting ordered list is "$\mathcal{V} = \{o^{\downarrow}, a^{\uparrow}, d^{\downarrow}, f^{\downarrow}, k^{\downarrow}, l^{\downarrow}, m^{\downarrow}, n^{\downarrow}, b^{\uparrow}, c^{\uparrow}, e^{\uparrow}, f^{\uparrow}, g^{\uparrow}, h^{\uparrow}, l^{\uparrow}, c^{\downarrow}, e^{\downarrow}, g^{\downarrow}, h^{\downarrow}, i^{\downarrow}, d^{\uparrow}, i^{\uparrow}, k^{\uparrow}, m^{\uparrow}, b^{\downarrow}, n^{\uparrow}, j^{\downarrow}, a^{\downarrow}, j^{\uparrow}, o^{\uparrow}\}$". Then, we have the ordered edges in $\mathcal{E} = \{(d, j), (f, h), (k, o), (l, o), (m, o), (n, o), (a, b), (b, c), (a, e), (a, f), (b, g), (j, l), (c, j), (c, k), (e, d), (e, n), (g, j), (g, o), (h, i), (h, n), (i, j), (i, m), (a, d), (a, i), (j, k), (j, m), (b, j), (j, n)\}$. After that, we are ready to process all edges. Consider processing edge (a, i). Algorithm 2 will call $RI()$ three times to check whether b, e and f can reach i. As a comparison, due to $|out_G(a)| = 5 > |in_G(i)| = 2$ and $h \in in_G(i)$, Algorithm 3 will call $RI()$ once in lines 23-25 to check whether a can reach h according to Theorem 5. As a result, to process all edges in $\mathcal{E}$, Algorithm 3 only needs to call $RI()$ 8 times. Specifically, no one of the non-redundant edges needs to call $RI()$ function, and each redundant edge needs to call $RI()$ only once. As a comparison, to process all these edges, Algorithm 2 needs to call $RI()$ 21 times, and the number for Algorithm 1 is 46. □

Analysis: We discuss the time and space complexities of Algorithm 3.

Definition 2: (Minimum Degree Cover Set (MDCS)) Given the set of all UP-nodes and DOWN-nodes $\mathcal{V} = V^{\uparrow} \cup V^{\downarrow}$ of $G = (V, E)$ sorted in ascending degrees, we say $S \subseteq \mathcal{V}$ is a minimum degree cover set (MDCS) of $\mathcal{V}$ if the following conditions hold.

1) $S = \{v_1, v_2, \ldots, v_{|S|}\}$ consists of the first $|S|$ nodes of $\mathcal{V}$,
2) $\bigcup_{v \in S} adjEdges(v) = E$,
3) $\bigcup_{v \in S \setminus \{v_{|S|}\}} adjEdges(v) \subset E$, where $adjEdges(v)$ is defined as Equation 1.

$$adjEdges(v) = \begin{cases} in_G(v), & v \in V^{\uparrow} \\ out_G(v), & v \in V^{\downarrow} \end{cases} \tag{1}$$

According to Definition 2, we only need to process all edges adjacent to nodes in S for TR computation, due to that $\bigcup_{v \in S} adjEdges(v) = E$. Let $d^S_{\max} = \max\{|adjEdges(v)||v \in S\}$ be the maximum degree for all nodes in S, then we know that the calling times of $RI()$ function to process any edge in Algorithm 3 is $d^S_{\max} - 1$ in the worst case. For instance, for the sorted $\mathcal{V}$ in Example 4, we know that $S = \{o^{\downarrow}, a^{\uparrow}, d^{\downarrow}, f^{\downarrow}, k^{\downarrow}, l^{\downarrow}, m^{\downarrow}, n^{\downarrow}, b^{\uparrow}, c^{\uparrow}, e^{\uparrow}, f^{\uparrow}, g^{\uparrow}, h^{\uparrow}, l^{\uparrow}, c^{\downarrow}, e^{\downarrow}, g^{\downarrow}, h^{\downarrow}, i^{\downarrow}, d^{\uparrow}, i^{\uparrow}, k^{\uparrow}, m^{\uparrow}, b^{\downarrow}, n^{\uparrow}\}$, for which $d^S_{\max} = 3$. Thus for any edge, the calling times of $RI()$ function is at most twice in the worst case.

In Algorithm 3, the sorting operation in line 10 can be done in linear time $O(|V|)$ using radix sorting. In lines 14 and 18, we use a hash map to check whether an edge is contained in $\mathcal{E}$ or not, thus the cost is $O(1)$. Since the cost of processing each edge is $O(\theta d^S_{\max})$, the time complexity of Algorithm 3 is $O(\theta d^S_{\max}|E|)$. And Algorithm 3, Algorithm 2 and Algorithm 1 have the same space complexity.

TABLE 3. The comparison of time and space complexities.

algorithm	time complexity	space complexity	d for G
TR-B	$O(\theta d_{\max}\|E\|)$	$O(\|V\|)$	$d_{\max} = 5$
TR-O	$O(\theta d^{tr}_{\max}\|E\|)$	$O(\|V\|)$	$d^{tr}_{\max} = 4$
TR-O$^+$	$O(\theta d^S_{\max}\|E\|)$	$O(\|V\|)$	$d^S_{\max} = 3$

We show the comparison of the time and space complexities of our approaches in Table 3, from which we know that the three algorithms have the same space complexity. For time complexity, as $d^S_{\max} \leq d^{tr}_{\max} \leq d_{\max}$, we know that *TR-O*$^+$ should work best in practice. For G in Figure 1(a), the values of d for the three algorithms in Table 3 are shown in the last column.

V. EXPERIMENT

In this section, we make comparison with existing approaches for TR computation, including *CWO* [7], *PTR* [3], *DFS*, and *buTR* [12]. We implemented all algorithms using C++ and compiled by G++ 6.2.0. All experiments were run on a PC with Intel(R) Core(TM) i5-3230M CPU @ 3.0 GHz CPU, 16 GB memory, and Ubuntu 18.04.1 Linux OS. For

TABLE 4. Statistics of datasets, where $d = |E|/|V|$ is the average degree of G, $d_{\max} = \max\{|out_G(v)||v \in V\}$, $d^{tr}_{\max} = \max\{|out_{G^{tr}}(v)||v \in V\}$, $d^{S}_{\max} = \max\{|adjEdges(v)||v \in S\}$ is the maximum degree for all nodes in the minimum degree cover set S.

Dataset	$\|V\|$	$\|E\|$	d	$\|E \setminus E^{tr}\|$	$d_{\max}$	$d^{tr}_{\max}$	$d^{S}_{\max}$	source
amaze	3,710	3,600	0.97	216	1,539	1,452	5	small real
agrocyc	12,684	13,408	1.06	314	5,485	5,477	382	
xmark	6,080	7,025	1.16	71	803	803	58	
mtbrv	9,602	10,245	1.07	294	4,003	3,999	328	
go	6,793	13,361	1.97	1,101	70	70	6	
email	231,000	223,004	0.97	6,941	17,847	17,762	7,631	large real
uniprot150m	25,037,600	25,037,598	1.00	0	1	1	1	
LJ	971,232	1,024,140	1.05	49,701	546,729	538,531	586	
web	371,764	517,805	1.39	104,810	97,957	88,784	1,791	
05cit-Patent	1,671,488	3,303,789	1.98	328,585	128	96	76	
dbpedia	3,365,623	7,989,191	2.37	3,262,298	125,371	122,703	5,371	
cit-Patents	3,774,768	16,518,947	4.38	4,693,568	770	338	219	
go_uniprot	6,967,956	34,769,339	4.99	11,405,409	170	107	89	
10go-unip	469,526	3,476,397	7.40	1,435,296	170	170	170	
twitter	18,121,168	18,359,487	1.01	1,667,042	14,892,425	14,782,690	22,073	
1M-1M	1,000,000	1,000,000	1.00	0	11	11	8	large synthetic
1M-5M	1,000,000	5,000,000	5.00	20,171	28	27	20	
1M-10M	1,000,000	10,000,000	10.00	1,981,296	41	29	23	
1M-15M	1,000,000	15,000,000	15.00	6,109,021	54	28	24	
1M-20M	1,000,000	20,000,000	20.00	10,788,966	66	29	24	
10M-50M	10,000,000	50,000,000	5.00	21,683	29	29	20	
20M-100M	20,000,000	100,000,000	5.00	22,002	28	28	22	
30M-150M	30,000,000	150,000,000	5.00	21,643	28	28	23	
40M-200M	40,000,000	200,000,000	5.00	22,061	32	32	22	

algorithms that run $\geq$ 24 hours or exceed the memory limit (16GB), we will show their results as "–" in the tables.

Datasets: Table 4 shows the statistics of 15 real datasets and 9 synthetic datasets. For real datasets, the first five are small datasets ($|V| \leq 100,000$) and are downloaded from the same web page[2]. The following 10 datasets are large ones ($|V| > 100,000$). These datasets are usually used in the recent works [12], [25]–[30], [32], [34], [35]. Among these datasets, **amaze** is a metabolic network, **agrocyc** and **mtbrv** are both graphs describing the genome and biochemical machinery of E. coli K-12 MG1655. **xmark** is an XMark document, **email**[3] is an email network. As indicated by [32], **go**[2] and **10go-unip**[4] (10go-uniprot) are the joint graphs of Gene Ontology terms and the annotations file from the UniProt[5] database. **uniprot150m**[2] (uniprotenc_150m) is a *DAG* that is a subgraph of the RDF graph of UniProt, which contain many nodes without incoming edges and few nodes without outgoing edges. **05cit-Patent**[4] (05cit-Patent) and **cit-Patents**[2] (cit-Patents) are both citation networks with out-degree of non-leaf nodes ranging from 10 to 30. **LJ** is an online social network soc-LiveJournal1[3]. **go_uniprot**[2] (go_uniprot) is a *DAG* transformed from the joint graph of Gene Ontology terms with the annotations file from the UniProt. **dbpedia**[6] is a knowledge graph Dbpedia. **web** is a web graph web-Google.[7] **twitter**[7] is a large-scale social network [36]. For these real datasets, the first 4 small datasets and three large datasets, including **email**, **LJ** and **web**, are direct graphs initially. We transform each of them into a *DAG* by coalescing each strongly connected component into a node. Note that this can be done in linear time [37]. All other datasets are *DAG*s initially. The statistics in Table 4 are that of *DAG*s.

Besides real ones, we also generate 9 synthetic datasets shown in Table 4. The synthetic datasets are used to test the scalability of the algorithms on *TR* computation with the change of the average degree and the number of nodes in the graph. These datasets are generated as follows. We first fix the number of nodes $|V|$. Then, we randomly generate two integers between 1 and 1,000,000 representing two nodes u and v. If $u > v$ and edge (v, u) does not exist, then we insert an edge from v to u; otherwise, we insert an edge (u, v), if (u, v) does not exist. We perform this operation repeatedly until the number of edges satisfies our requirement.

A. COMPARISON ON REAL DATASETS

Table 5 shows the running time of different approaches on *TR* computation, from which we know that *CWO* cannot scale to large graphs due to large space complexity.

Similar to *CWO*, *PTR* cannot compute *TR* successfully on several datasets. The reason lies in that the time and space complexities of *PTR* is determined by the number of decomposed paths k and in practice, k is usually large that approaches the number of nodes in the given graph. For example, for **amaze**, **10go-unip** and **email**, $k > 0.8|V|$, which makes *PTR* cannot scale to large graphs, and is inefficient on datasets it can process. When the given graph becomes large, *PTR* fails to get the result in limited time due to large space consumption.

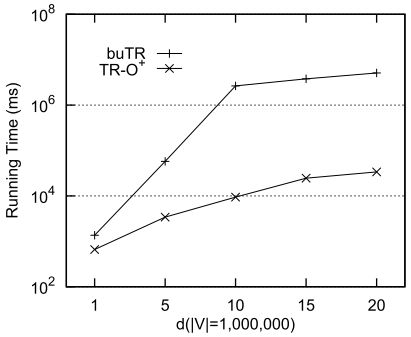
[2]https://code.google.com/archive/p/grail/downloads
[3]http://snap.stanford.edu/data/index.html
[4]http://pan.baidu.com/s/1bpHkFJx
[5]http://www.uniprot.org/
[6]http://pan.baidu.com/s/1c00Jq5E
[7]https://code.google.com/p/ferrari-index/downloads/list

TABLE 5. Comparison of running time on real datasets (ms), where the number in the parentheses of each row of the last column is the index construction time for reachability index of *BFL*$^+$.

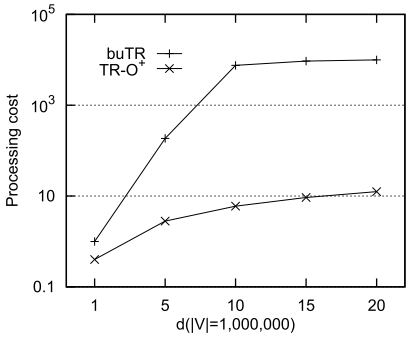
Dataset	*CWO*	*PTR*	*DFS*	*buTR*	*TR-B*	*TR-O*	*TR-O*$^+$
amaze	54.45	37.21	22.00	2.69	107.43	69.45	**0.54** (0.21)
agrocyc	679.19	250.55	2.31	4.88	4,811.57	1,484.72	**1.33** (0.43)
xmark	148.19	56.29	9.15	4.94	62.16	20.32	**1.43** (0.39)
mtbrv	378.87	149.62	1.99	2.33	2,273.17	720.83	**0.93** (0.31)
go	244.46	51.67	3.41	7.79	18.49	10.85	**1.67** (0.42)
email	—	117,817.00	14,100.40	51.76	79,115.90	21,342.20	**24.49** (12.33)
uniprot150m	—	—	**1,705.80**	12,081.94	53,671.50	42,962.80	4,134.41 (2,325.21)
LJ	—	—	1,661,850.00	360.39	—	—	**116.96** (43.02)
web	—	—	298,438.00	427.96	2,216,000.00	435,508.00	**93.75** (33.24)
05cit-Patent	—	1,244,350.00	**1,078.93**	5,249.38	16,071.80	4,035.78	1,416.03 (323.29)
dbpedia	—	—	2,428,480.00	5,443.52	7,788,970.00	785,305.00	**2,098.37** (732.19)
cit-Patents	—	482,117.00	579,911.00	299,910.00	229,156.00	32,451.40	**13,118.70** (1,783.43)
go_uniprot	—	—	**3,661.49**	15,371.08	283,899.00	53,305.00	6,229.19 (1,341.71)
10go-unip	—	395,541.00	**382.97**	1,378.78	20,741.40	3,124.76	647.94 (130.10)
twitter	—	—	—	6,844.88	—	—	**3,690.52** (1,721.35)

TABLE 6. Comparison of average processing cost for each edge on real datasets, where for *buTR*, it means the average number of visited nodes for each processed edge, and for other approaches, it means the average number of calling times of the *BFL*$^+$ algorithm for each processed edge.

Dataset	*buTR*	*TR-B*	*TR-O*	*TR-O*$^+$
amaze	0.27	637.17	318.62	0.09
agrocyc	2.88	5,660.29	1,713.53	0.07
xmark	3.19	95.75	47.88	0.05
mtbrv	3.23	2,319.81	1,161.07	0.08
go	4.38	7.70	3.88	0.58
email	0.24	1,424.05	710.71	0.05
uniprot150m	0.00	0.00	0.00	0.00
LJ	0.09	—	—	0.07
web	0.73	16,962.58	8,392.08	0.48
05cit-Patent	2.42	6.97	3.49	1.44
dbpedia	2.43	1,938.27	963.67	1.54
cit-Patents	211.53	15.60	7.82	4.03
go_uniprot	4.39	6.37	3.16	3.16
10go-unip	5.27	8.66	4.34	4.35
twitter	0.30	—	—	0.11



(a) Comparison of running time



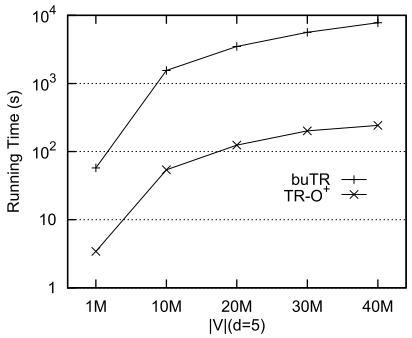
(b) Comparison of the average processing cost

FIGURE 3. Impacts of degrees on the performance of *buTR* and *TR-O*$^+$, where the number of nodes $|V| = 1,000,000$, for *buTR*, the processing cost means the average number of visited nodes for each processed edge, and for *TR-O*$^+$, it means the average calling times of the *BFL*$^+$ algorithm for each processed edge.

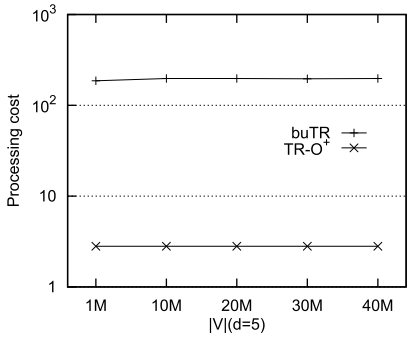
An interesting thing is that even though *DFS* is proposed earlier than *PTR*, it can work successfully on more datasets than *CWO* and *PTR*. The reason lies in that *DFS* has linear space complexity w.r.t. the number of nodes in the graph. However, as *DFS* computes *TR* based on each node with traversal, its performance is greatly affected by the size of the average transitive closure $|out^*_G(\cdot)|$. When $|out^*_G(\cdot)|$ increases, such as for twitter where $|out^*_G(\cdot)|/|V| = 0.15$, *DFS* fails to get the result in limited time.

The state-of-the-art approach *buTR* outperforms *DFS* on most datasets, because it has linear space complexity and its performance is not affected by the size of transitive closure.

As a comparison, our *TR-B* does not show better performance even compared with *DFS*. *TR-O* works better than *TR-B*, but it is still beaten by *buTR* on most datasets. Even though, *TR-O* can work more efficient than both *DFS* and *buTR* on cit-Patents dataset. This is because for the cit-Patents dataset, there does not exist nodes with much larger number of in- or out-neighbors than other nodes. Compared with the state-of-the-art approach *buTR*, our *TR-O*$^+$ is more efficient. The reason is that by using a new edge processing strategy, we can greatly reduce the average processing cost of each edge, which can be further explained by the average processing cost of each edge shown in Table 6. From Table 6 we know that for each edge, the average calling times of the *BFL*$^+$ algorithm by *TR-O*$^+$ is much less than *TR-B* and *TR-O*. The number in Table 6 for *buTR* is the



(a) Comparison of running time



(b) Comparison of the average processing cost

FIGURE 4. Impacts of the number of nodes on the performance of *buTR* and *TR-O*$^+$, where the average degree $k = 5$, for *buTR*, the processing cost means the average number of visited nodes for each processed edge, and for *TR-O*$^+$, it means the average calling times of the *BFL*$^+$ algorithm for each processed edge.

average number of visited nodes for each edge. Even though the units of *buTR* and *TR-O*$^+$ are different, they are the basic processing unit and can be used to explain the performance difference to a large extent. For example, the average processing cost of *buTR* is 50 times more than that of *TR-O*$^+$, and *TR-O*$^+$ is 23 times faster than *buTR* for *TR* computation.

From Table 5 we know that for existing approaches, *buTR* works much better than all the other existing ones, which was also confirmed by [12], [13]. For our approaches, *TR-O*$^+$ works much better than the other two. Therefore, in the following discussion, we only make comparison between *buTR* and *TR-O*$^+$ to show their scalability with the change of graph size and density.

B. COMPARISON ON SYNTHETIC DATASETS

In this subsection, we make the comparison between the state-of-the-art approach *buTR* and our *TR-O*$^+$ on synthetic datasets to show their scalability with the change of degree and the number of nodes in the graph. The results are shown in Figure 3 and Figure 4, from which we have the following observations.

First, with the increase of the graph density by fixing the number of nodes $|V| = 1,000,000$, *TR-O*$^+$ achieves much better scalability than *buTR*. As shown in Figure 3(a), when the average degree $k = 1$, both *buTR* and *TR-O*$^+$ can compute *TR* efficiently, but with the increase of the average degree, *buTR* needs much more time than *TR-O*$^+$ for *TR* computation. When the average degree $d \geq 10$, *TR-O*$^+$ is more than two orders of magnitude faster than *buTR*. The reason lies in that for every single edge, the average processing cost of *TR-O*$^+$ is much less than that of *buTR*, as shown in Figure 3(b). For example, when $k = 10$, *TR-O*$^+$ is 280 times faster than *buTR* for *TR* computation, due to that the average processing cost of *buTR* is 1,259 times more than that of *TR-O*$^+$.

Second, with the increase of the number of nodes in the graph by fixing the average degree $k = 5$, *TR-O*$^+$ also works much better than *buTR* on all datasets, as shown in Figure 4(a). This can also be explained by Figure 4(b). For example, for all datasets with the same degree $k = 5$, when $|V| = 1,000,000$, the average processing cost of *buTR* is 67 times more than that of *TR-O*$^+$, and *TR-O*$^+$ is 16.8 times faster than *buTR*. When $|V| = 40,000,000$, the average processing cost of *buTR* is 71 times more than that of *TR-O*$^+$, and *TR-O*$^+$ is 30 times faster than *buTR*.

VI. CONCLUSION

Considering that existing *TR* computation approaches are inefficient with the increase of the graph density and size, we propose a novel strategy that compute *TR* without graph traversal. Our approach utilizes existing reachability algorithms to check whether each edge in the given graph is redundant or not, and propose several heuristics to significantly reduce the calling times of the underlying reachability algorithm. In this way, our approach, namely *TR-O*$^+$, can efficiently compute *TR* with the increase of both the graph density and size. Our experimental results show that our approach *TR-O*$^+$ is more efficient than the state-of-the-art algorithm *buTR* on both real and synthetic datasets. As an indication, for real datasets, *TR-O*$^+$ is 23 times faster than *buTR* on cit-Patents dataset. For synthetic dataset, our *TR-O*$^+$ is 280 times faster than *buTR* on 1M-10M dataset.

REFERENCES

[1] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," *SIAM J. Comput.*, vol. 1, no. 2, pp. 131–137, Jun. 1972.
[2] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," *SIAM J. Comput.*, vol. 11, no. 2, pp. 298–313, May 1982.
[3] K. Simon, "An improved algorithm for transitive closure on acyclic digraphs," *Theor. Comput. Sci.*, vol. 58, nos. 1–3, pp. 325–346, Jun. 1988.
[4] K. Simon, "Finding a minimal transitive reduction in a strongly connected digraph within linear time," in *Proc. 15th Int. Workshop Graph-Theoretic Concepts Comput. Sci. (WG)*, Castle Rolduc, The Netherlands, Jun. 1989, pp. 245–259.
[5] M. Habib, M. Morvan, and J.-X. Rampon, "On the calculation of transitive reduction—Closure of orders," *Discrete Math.*, vol. 111, nos. 1–3, pp. 289–303, Feb. 1993.
[6] P. Berman, B. DasGupta, and M. Karpinski, "Approximating transitive reductions for directed networks," in *Proc. 11th Int. Symp. Workshop Algorithms Data Struct. (WADS)*, Banff, AB, Canada, Aug. 2009, pp. 74–85.
[7] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd," in *Proc. 44th Symp. Theory Comput. (STOC)*, New York, NY, USA, May 2012, pp. 887–898.
[8] D. Gries, A. J. Martin, J. L. A. van de Snepscheut, and J. T. Udding, "An algorithm for transitive reduction of an acyclic graph," *Sci. Comput. Program.*, vol. 12, no. 2, pp. 151–155, Jul. 1989.

[9] J. A. L. Poutré and J. van Leeuwen, "Maintenance of transitive closures and transitive reductions of graphs," in *Proc. Int. Workshop Graph-Theoretic Concepts Comput. Sci. (WG)*, Staffelstein, Germany, Jun./Jul. 1987, pp. 106–120.

[10] P. Chang and L. J. Henschen, "Parallel transitive closure and transitive reduction algorithms," in *Proc. 1st Int. Conf. Databases, Parallel Archit., Appl. (PARBASE)*, Miami Beach, FL, USA, Mar. 1990, pp. 152–154.

[11] A. Goralciková and V. Koubek, "A reduct-and-closure algorithm for graphs," in *Proc. 8th Int. Symp. Math. Found. Comput. Sci.*, Olomouc, Czech Republic, Sep. 1979, pp. 301–307, 1979.

[12] J. Zhou, S. Zhou, J. X. Yu, H. Wei, Z. Chen, and X. Tang, "DAG reduction: Fast answering reachability queries," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, Chicago, IL, USA, May 2017, pp. 375–390.

[13] J. Zhou, J. X. Yu, N. Li, H. Wei, Z. Chen, and X. Tang, "Accelerating reachability query processing based on DAG reduction," *VLDB J.*, vol. 27, no. 2, pp. 271–296, 2018.

[14] J. R. Clough, J. Gollings, T. V. Loach, and T. S. Evans, "Transitive reduction of citation networks," *J. Complex Netw.*, vol. 3, no. 2, pp. 189–203, Sep. 2014.

[15] V. Dubois and C. Bothorel, "Transitive reduction for social network analysis and visualization," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. (WI)*, Compiegne, France, Sep. 2005, pp. 128–131.

[16] S. Klamt, R. J. Flassig, and K. Sundmacher, "TRANSWESD: Inferring cellular networks with transitive reduction," *Bioinformatics*, vol. 26, no. 17, pp. 2160–2168, Jul. 2010.

[17] X. Tannier and P. Muller, "Evaluating temporal graphs built from texts via transitive reduction," *J. Artif. Intell. Res.*, vol. 40, pp. 375–413, Feb. 2011.

[18] A. Pinna, S. Heise, R. J. Flassig, A. Fuente, and S. Klamt, "Reconstruction of large-scale regulatory networks based on perturbation graphs and transitive reduction: Improved methods and their evaluation," *BMC Syst. Biol.*, vol. 7, no. 1, p. 73, 2013.

[19] D. Bošnački, M. R. Odenbrett, A. Wijs, W. Ligtenberg, and P. Hilbers, "Efficient reconstruction of biological networks via transitive reduction on general purpose graphics processors," *BMC Bioinf.*, vol. 13, no. 1, p. 281, Oct. 2012.

[20] N. J. van Eck and L. Waltman, "Citnetexplorer: A new software tool for analyzing and visualizing citation networks," *J. Inf.*, vol. 8, no. 4, pp. 802–823, 2014.

[21] S. Hou, X. Huang, J. K. Liu, J. Li, and L. Xu, "Universal designated verifier transitive signatures for graph-based big data," *Inf. Sci.*, vol. 318, pp. 144–156, Oct. 2015.

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

[23] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, Scottsdale, AZ, USA, May 2012, pp. 157–168.

[24] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *J. Symbolic Comput.*, vol. 9, no. 3, pp. 251–280, Mar. 1990.

[25] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *VLDB J.*, vol. 27, no. 1, pp. 1–26, May 2017.

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

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

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

[29] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida, "Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths," in *Proc. 22nd ACM Int. Conf. Conf. Inf. Knowl. Manage. (CIKM)*, 2013, pp. 1601–1606.

[30] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: a total order approach," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Snowbird, UT, USA, Jun. 2014, pp. 1323–1334.

[31] H. Yildirim, V. Chaoji, and M. J. Zaki, "Grail: Scalable reachability index for large graphs," *Proc. VLDB Endowment J.*, vol. 3, nos. 1–2, pp. 276–284, 2010.

[32] H. Yıldırım, V. Chaoji, and M. J. Zaki, "GRAIL: A scalable index for reachability queries in very large graphs," *VLDB J.*, vol. 21, no. 4, pp. 509–534, Sep. 2011.

[33] Z. Zhang, J. X. Yu, L. Qin, Q. Zhu, and X. Zhou, "I/O cost minimization: Reachability queries processing over massive graphs," in *Proc. 15th Int. Conf. Extending Database Technol. (EDBT)*, Berlin, Germany, Mar. 2012, pp. 468–479.

[34] S. Seufert, A. Anand, S. Bedathur, and G. Weikum, "FERRARI: Flexible and efficient reachability range assignment for graph indexing," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Brisbane, QLD, Australia, Apr. 2013, pp. 1009–1020.

[35] R. R. Veloso, L. Cerf, W. M. Jr, and M. J. Zaki, "Reachability queries in very large graphs: A fast refined online search approach," in *Proc. 17th Int. Conf. Extending Database Technol. (EDBT)*, Athens, Greece, Mar. 2014, pp. 511–522.

[36] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proc. 4th Int. Conf. Weblogs Social Media (ICWSM)*, Washington, DC, USA, May 2010, pp. 1–8.

[37] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, Jun. 1972.

XIAN TANG was born in Weifang, Shandong, China, in 1978. She received the B.S. and M.S. degrees in computer science from Yanshan University, in 2002 and 2006, respectively, and the Ph.D. degree in computer applications from the Renmin University of China, Beijing, China, in 2011.

From 2011 to 2018, she was a Lecturer with Yanshan University. Since 2019, she has been an Associate Professor with the School of Electronic and Electrical Engineering, Shanghai University of Engineering Science, Shanghai. Her research interests include query processing and optimization on graph data and semi-structured data, and flash based databases.

JUNFENG ZHOU was born in Xi'an, Shanxi, China, in 1977. He received the B.S. and M.S. degrees in computer science from Yanshan University, in 1999 and 2002, respectively, and the Ph.D. degree in computer applications from Renmin University of China, Beijing, China, in 2009.

From 2009 to 2017, he was a Professor with Yanshan University. Since 2017, he has been a Professor with the School of Computer Science and Technology, Donghua University, Shanghai. His research interests include information retrieval techniques, query processing on semi-structured data and graph data, and optimization.

YAXIAN QIU was born in Anyang, Henan, China, in 1997. She received the B.S. degree from Yanshan University, in 2019.

Since 2019, she has been a Graduate Student of Donghua University, Shanghai, China. Her current research interest includes graph data management.

XIANG LIU was born in Zhenjiang, Jiangsu, China, in 1972. He received the B.Sc. degree from Nanjing Normal University, the M.Eng. degree from Jiangsu University, and the Ph.D. degree from Fudan University.

He is currently an Associate Professor and the Director of the Computer Department, School of Electronic and Electric Engineering, Shanghai University of Science Engineering, Shanghai, China. His current research interests include medical image analysis, biological image processing, and pattern recognition.

YUNYU SHI was born in Yanzhou, Shandong, China, in 1982. She received the B.S. and M.S. degrees in computer science from the East China University of Technology, in 2004 and 2007, respectively, and the Ph.D. degree in computer applications from Shanghai University, Shanghai, China, in 2012.

From 2012 to 2014, she was a Postdoctoral Researcher with Shanghai Jiao Tong University. Since 2014, she has been a Lecturer with the School of Electronic and Electrical Engineering, Shanghai University of Engineering Science, Shanghai. Her research interests include image and video processing and analysis, video compression, and quality assessment.

JINGWEN ZHAO received the B.Eng. degree from the East China University of Science and Technology, in 2013. She received the Ph.D. degree from the School of Computer Science, Fudan University.

She is currently a Faculty Member with the Computer Department, School of Electronic and Electrical Engineering, Shanghai University of Science Engineering, Shanghai, China. Her current research interests include biological image processing, medical image analysis, and pattern recognition.