buTCS: An Optimized Algorithm for Estimating the Size of Transitive Closure

XIAOZHE LI, XUAN WANG, JUNFENG ZHOU, AND MING DU

School of Computer Science and Technology, Donghua University, Shanghai 201620, China

Corresponding authors: Xuan Wang (dhwangxuan@dhu.edu.cn), Junfeng Zhou (zhoujf@dhu.edu.cn), and Ming Du (duming@dhu.edu.cn)

This work was supported in part by Shanghai Natural Science Foundation under Grant 20ZR1402700; and in part by the Natural Science Foundation of China under Grant 61472339, Grant 61873337, Grant 61572421, and Grant 61272124.

ABSTRACT Given a directed graph and a node v, the transitive closure (TC) of v is the set of nodes that v can reach in the graph. TC size is very important in many applications but the cost of TC size computation is high in both time and space, which makes computing accurate TC size not applicable to the scenario where we need to know the TC size quickly for large graphs. Considering that existing approaches are either inefficient or inaccurate, we propose an algorithm, namely buTCS, to efficiently make more accurate estimation. Our approach works in linear time and space. The basic idea is to compute a node’s TC size based on that of its out-neighbors and perform a top-down verification. We further propose two optimizations to improve the estimation accuracy. The experimental results on 13 real datasets show that buTCS achieves better estimation on TC size efficiently.

INDEX TERMS Directed graph, transitive closure size, spanning tree.

I. INTRODUCTION

Given a directed graph $G = (V, E)$, where $V$ is the set of nodes and $E$ is the set of edges, the transitive closure (TC) of a node $u$ is defined as the set of all nodes that $u$ can reach in $G$. Obtaining TC size for all nodes in directed graphs is a fundamental graph problem [1]–[18], [21], [23], [24] and has been extensively used in many applications. For example, to efficiently compute transitive reduction [23], [24], we need to know which node has a larger TC size, such that to determine a better processing order for all nodes to improve the efficiency. For another example, when constructing 2hop labels [16], [17], if we can quickly know the TC sizes of all nodes on forward and backward directions, we can get a better node processing order, such that to efficiently produce 2hop labels with smaller size.

However, the cost of TC size computation is high in both time and space. Calculating the accurate size of TC by using fast binary matrix multiplication (see e.g. [19], [20] for background) has time complexity $O(|V|^2 \cdot 38)$ and space complexity $O(|V|^2)$, therefore cannot scale to large graphs. Besides, in the scenario where we need to know the TC size for large graphs quickly, this method is not applicable due to its high time and space complexities.

The associate editor coordinating the review of this manuscript and approving it for publication was Fu Lee Wang.

Considering this problem, researchers proposed several approaches to estimate the size of TC [21]–[23]. These approaches usually have smaller time and space complexities, therefore can be used to efficiently process large graphs. However, these approaches still suffer from poor performance in accuracy or efficiency. In [21], Cohen proposed an algorithm that runs in $O(k(|V| + |E|))$ time and $O(k|V|)$ space to obtain a TC size estimation for every node by performing $k$ random permutations [21]. The problem of this approach is that its accuracy depends on the value of $k$. The larger the value of $k$, the more accurate for its TC size estimation. However, a large $k$ will result in inefficiency. In [22], Zhu et al. proposed heuristics to estimate the lower and upper bounds of TC size [22]. However, both lower and upper bounds are far from accurate. The most recent algorithm linearTC was proposed by Junfeng Zhou et al. to estimate every node’s TC size with time complexity [23], [24]. Specifically, to estimate the TC size of a node $u$, it firstly finds from $u$’s out-neighbors the node $v$ that has the maximum estimated TC size. Then, it takes the sum of $v$’s TC size and the number of $u$’s out-neighbors as $u$’s estimated TC size. This approach can work efficiently, but suffers from two problems. First, as the difference between the TC size of $u$ and $v$ may be huge, the estimated TC size of $u$ which is computed based on $v$’s estimated TC size may be far from accurate. Second, as $v$’s estimated TC size may be greater than the accurate TC size of
Considering these above problems, we propose an optimized approach for TC size estimation. The basic idea is similar to that of linear TC, i.e., u’s TC size is estimated based on that of its out-neighbors. Here, we make improvements from two aspects. First, we propose a novel spanning tree, based on which we can find the accurate TC size of many important nodes. In this way, we can reduce the difference between the TC sizes of every node u and u’s out-neighbor v, such that the estimated TC size of u can be more accurate. Second, we propose a verification operation to improve the estimated accuracy when the estimated TC size is greater than the accurate value. Our contributions are as follows.

1) We propose a two-step TC size estimation strategy.
   In the first step, we compute the initial estimated TC size for each node. In the second step, we perform a top-down verification to improve the accuracy of the estimated value.

2) We propose two heuristics for TC size estimation, which makes estimation based on a spanning tree. We further propose a novel spanning tree based on which we identify important nodes and get their accurate TC size, such that to make the overall estimation more accurate. We further propose to use two spanning trees to increase the number of nodes for which we can get the accurate TC size.

3) We conduct extensive experimental studies on real datasets. The experimental results show that our approach can efficiently get more accurate results and scale to large graphs.

The remainder of this paper is organized as follows. In Section II, we discuss the preliminaries and related work. In Section III, we discuss the general idea of our approach on TC size estimation and the algorithms. We report the experimental results in Section IV, and conclude our work in Section V.

II. BACKGROUND AND RELATED WORK

A. PRELIMINARIES

We model a graph as a directed graph, and focus on its direct acyclic graph (DAG) representation, since we can easily compute the size of each strongly connected component (SCC). Hereafter, we use $G = (V, E)$ to denote a DAG, where $V$ is the set of nodes and $E$ is the set of edges and focus on TC size estimation on $G$.

We use $i_G(v) = \{u | (u, v) \in E\}$ to denote the set of in-neighbor nodes of $v$, and $o_G(v) = \{u | (v, u) \in E\}$ the set of out-neighbor nodes of $v$. We define $i_G^+(v)$ as the set of nodes in $G$ that can reach $v (v \notin i_G^+(v))$, and $o_G^+(v)$ as the set of nodes in $G$ that $v$ can reach $v (v \notin o_G^+(v))$, which is also called the transitive closure of $v$. We call $o_G^+(v)$ as $u$’s descendants w.r.t. a tree $T$. Given a DAG $G = (V, E)$, we use $X = \{1, 2, \ldots, |V|\}$ 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 $t : V \rightarrow X$, such that

$$\forall (v, u) \in E, \text{ we have } t_v < t_u, \text{ where } t_v/t_u \text{ is the topo-order of } v/u \text{ w.r.t. } X \text{ and we use } I_v \text{ to denote node } v \text{’s topo-order interval. We use } \tilde{N}(v) \text{ to denote } v \text{ estimated TC size. A topo-order } X \text{ of } G \text{ can be got in linear time } O(|V| + |E|) \ [25].$$

We show important notations in Table 1 for ease of reference.

**Problem Statement:** Given a DAG $G$, return the estimated TC size of all nodes.

### B. RELATED WORK

Existing works can be divided into two categories: (1) TC size computation [18], [20], which computes the accurate TC size. (2) TC size estimation [21]–[24], which computes an approximate TC size. Compared with the first category, the second category usually works much more efficiently and can scale to large graphs. In the following, we survey the techniques in each category.

1) **TC SIZE COMPUTATION**

Calculating the accurate TC size using fast binary matrix multiplication [2], [20] has time complexity $O(|V|^{2.38})$ and space complexity $O(|V|^2)$, which cannot scale to large graphs due to the expensive computation and space overheads. In [18], Tang et al. proposed an algorithm based on path-decomposition. The result is a set of $n$ paths. Then, it computes the TC size of nodes on the path in a bottom-up way. The time and space complexities are $O(n \times |E|)$ and $O(|V|)$, respectively. However, in practical applications, $n$ is usually very large, resulting in higher time complexity and poor scalability.

2) **TC SIZE ESTIMATION**

In [21], Cohen proposed to estimate TC size by performing $k$ random (kr) permutations. The basic idea is to generate a random rank value for each node. Then, it sorts all nodes according to this rank value. After that, it performs DFS/BFS from each node $v$ in the sorted order and then deletes all nodes that $v$ can reach. This estimated TC size of each node is calculated based on the rank value of the node that first reaches it. This algorithm guarantees that the estimated error

| TABLE 1. Table of notations. |
|-------------------------------|
| Notation | Description |
|---|---|
| $G = (V, E)$ | a DAG with a node set $V$ and an edge set $E$ |
| $\tilde{N}(v)$ | estimated TC size of $v$ |
| $X$ | a topo-order of a $G$ |
| $t_v$ | node $v$’s topo-order in $X$ |
| $T$ | the TPS tree w.r.t. a topo-order $X$ |
| $I_v$ | $v$’s interval covering $v$’s descendants in a spanning tree $T$ |
| $i_G(v)$ | the set of in-neighbors of $v$ in $G$ |
| $o_G(v)$ | the set of nodes that can reach $v$ in $G$ |
| $o_G^+(v)$ | the set of nodes that $v$ can reach in $G$ |
| $i_G^+(v)$ | the set of descendants of $v$ in a tree $T$ |
rate and iterative rounds \( k \) satisfy 
\[
\frac{|v_G^e(v)| - \bar{N}(v)}{|v_G^e(v)|} = O(1/\sqrt{k}).
\]
Obviously, its accuracy depends on the value of \( k \), and a large \( k \) usually means inefficiency. The cost of each random permutation is \( O(|V| + |E|) \), and the overall time complexity is \( O(k(|V| + |E|)) \).

In [22], Zhu et al. proposed an algorithm to estimate the lower and upper bounds (denoted as \( \ell_b \) and \( \ell_u \), respectively) of a node’s TC size. The lower bound of \( u \) is obtained by summing up the contributions of \( u \)'s out-neighbors in \( G \), where each out-neighbor \( v \) contributes \( \frac{1}{|v_G^e(v)|} \) of its lower bound to \( u \). If \( |v_G(u)| = 1 \) and \( |v_G(v)| > 1 \), 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 out-neighbors. Since many nodes share the same set of reachable nodes in \( G \), the upper bound of \( u \) may be much larger than the accurate result. Although both lower and upper bounds can be efficiently computed, they are usually far from accurate.

Zhou et al. [24] proposed an algorithm, namely linearTC, to estimate every node’s TC size more accurately with time complexity \( O(|V| + |E|) \). This approach can work efficiently, but suffers from inaccuracies in practice, as discussed in Introduction. We follow the similar idea and make improvements by proposing novel heuristics. The differences between linearTC and our approach lie in two aspects. First, the aim is different. Our approach is aiming at achieving more accurate estimation on TC size. As a comparison, linearTC is aiming at using the estimated TC size to get a better node processing order, that is, it concerns more on which node has larger TC size, but not on whether the estimated value is accurate. Second, the operation is different. According to our aim, i.e., achieving more accurate estimation, rather than which node has larger TC size. We made improvements on linearTC from three aspects: (1) we perform a verification operation to improve the accuracy, (2) we propose a novel spanning tree that can mark nodes with larger TC size, which will be used to generate more accurate estimation for more nodes, (3) we use two spanning trees to mark more important nodes.

### III. TC SIZE ESTIMATION

Given a DAG \( G = (V, E) \), Junfeng Zhou et al. proposed an algorithm linearTC to estimate every node’s TC size with time complexity \( O(|V| + |E|) \) [23], [24]. It is worth noting that in [23], [24], they want to know for a node, which of its out-neighbors has the largest TC size. They do not care the difference between the estimated value and the accurate value, as long as they can correctly identify the one that has the largest TC size. Different with [23], [24], the aim of our approach is achieving more accurate estimation on TC size.

#### A. PROCESSING STRATEGY

Our heuristics is based on careful observations, which help to verify the feasibility of our approach. Given a node \( u \), let \( v_{max} \) be the node that has the largest TC size among \( u \)'s out-neighbors, and \( n_u \) is the number of our-neighbors of \( u \), we define \( TC_{err} = \frac{\sum_{v \in \mathcal{N}(u)} (TC(v) - TC(v_{max}))}{|\mathcal{N}(u)|} \% \). The smaller the value of \( TC_{err} \), the smaller the difference between \( u \)'s TC size and the sum of the largest TC size among its out-neighbors and the number of its out-neighbors. By Table 2, we know that for most datasets, \( TC_{err} \) is very small, usually less than 0.1%, which means that if we can get the TC size of \( u \)'s out-neighbors as accurate as possible, then we can get more accurate estimation on \( u \)'s TC size. Therefore, we propose a bottom-up strategy to estimate the TC size of all nodes, during which we use a spanning tree to help get more accurate estimation.

### TABLE 2. The TC statistics of each dataset, where |TC_{avg}| is the accurate TC size of all nodes on average.

| DataSet   | TC_{avg} | TC_{err} |
|-----------|----------|----------|
| email     | 11,698   | <0.1%    |
| unip150m  | 1        | 0        |
| wiki      | 18,522   | <0.1%    |
| LJ        | 206,903  | <0.1%    |
| web       | 55,055   | <0.1%    |
| citeseerx | 15,510   | <0.1%    |
| dbpedia   | 83,658   | <0.1%    |
| govwild   | 561      | 0.7%     |
| Patent    | 1,544    | 2.1%     |
| go-unip   | 26       | 31.3%    |
| 10go-unip | 39       | 32.1%    |
| twitter   | 1,346,820| <0.1%    |
| web-uk    | 3,417,930| <0.1%    |

#### 1) THE SPANNING TREE

Given a toporder-X of DAG \( G \), the toporder spanning (TPS) tree \( T \) is a spanning tree of \( G \). The incoming edge to a node \( v \) in \( T \) is from its last in-neighbor \( u \), which has the maximum toporder among \( v \)'s in-neighbors w.r.t. \( X \). For example, given \( G \) in FIGURE 1(a), the integer on the right of each node is its top-order, the tree \( T \) is shown in FIGURE 1(b).

![FIGURE 1. A DAG G with top-order X and its spanning tree T. The integer on the right of each node in G is its top-order, and the pair on the right of each node in T is its interval.](image)
We assign each node \( v \) an interval \( I_v = [s, e] \) to facilitate checking the ancestor-descendant relationship for nodes in \( T \), where \( I_v.s = t_v \), and \( I_v.e \) is the maximum Topo-order of \( v \)'s descendants in the spanning tree \( T \). \( I_v \subset I_u \) means that \( u \) is an ancestor of \( v \).

Given a DAG \( G \), we use Algorithm 1 to obtain the Topo-order \( X \) and generate a TPS tree \( T \). First, to get the Topo-order \( X \), the topological sorting can be done by (1) finding a set of “start nodes” without incoming edges and inserting them into a stack \( S \) (lines 2-5); (2) popping out a node \( v \) from \( S \), assigning \( v \) its visiting order \( t_v \), and pushing \( v \)'s out-neighbors which have no incoming edges into \( S \) after deleting edges starting from \( v \) (lines 6-14); and (3) repeating (2) until \( S \) becomes empty. The time complexity of topological sorting is \( O(|V| + |E|) \), and the space complexity is \( O(|V|) \) (line 6).

Second, we construct the TPS tree \( T \) during performing topological sorting. Specifically, after processing a node \( v \), each of \( v \)'s out-neighbors in \( G \) with in-degree = 0 is inserted into \( T \) as \( v \)'s child node (line 14). Therefore, \( T \) is constructed by inserting nodes into it in the ascending Topo-order \( X \). The interval of each node \( v \in V \) is constructed in descending topo-order \( X \) (lines 15-17). In Algorithm 1, \( p(v) \) is \( v \)'s parent node in \( T \). The time complexity of Algorithm 1 is \( O(|V| + |E|) \), and the space complexity is \( O(|V|) \).

Algorithm 1 ConstructTPS Tree(\( G = (V, E) \))

1 InitStack \( S \)
2 for each \( (v \in V) \) do
3   if (\(|I_G(v)| = 0\)) then
4     \( S.push(v) \)
5     insert \( v \) as root’s child in \( T \)
6 while (\( S \) is not empty) do
7     \( v = S.pop() \)
8     assigning \( v \) its Topo-order \( t_v \)
9     \( I_v.s = I_v.e = t_v \)
10    for each (\( u \in o_G(v) \)) do
11      \( |I_G(u)| = |I_G(u)| - 1 \)
12      if (\(|I_G(u)| = 0\)) then
13        \( S.push(u) \)
14      insert \( u \) as \( v \)'s child in \( T \)
15    for each \( (v \in V \) in descending Topo-order \( X \)) do
16      if (\(|I_G(v)|.e < I_v.e\)) then
17        \( I(v).e = I_v.e \)
18 return \( T \)

Definition 1 (Complete Node, CN): Given a node \( v \) of the input graph \( G \), we say \( v \) is a CN, if every node \( u \) that \( v \) can reach in \( G \) is a descendant of \( v \) in the TPS tree \( T \) w.r.t. \( G \).

Here, intuitively, if \( v \) is a CN, then every node in \( o_G(v) \) is a descendant of \( v \) in \( T \). E.g., the CNs of \( T \) in FIGURE 1 (b) are \( a, f, g, h, i, j, k \) according to Definition 1.

Theorem 1: Given a CN, then its TC size is equal to its interval length.

Proof: For node \( v \), the \( v \)'s TC size is the number of nodes that \( v \) can reach. If \( v \) is CN, the interval of \( v \) is composed of all its reachable nodes according to the definition 1, so the length of the interval is the accurate TC size.

Example 1: Consider node \( a \) in FIGURE 1 (a), \( o_G^*(a) = \alpha^*(a) \), i.e., \( a \) is a CN in FIGURE 1 (b). According to Theorem 1, \( N(a) = I_v.e - I_v.s = 11 - 1 = 10 \), which is its accurate TC size.

According to Theorem 1, if we know which node is a CN, we know its accurate TC size, based on which we can make a more accurate TC size estimation. To mark all CNs, we first need to know for each node \( v \), the farthest node \( u \) that \( v \) can reach. Let \( far(v) \) be \( u \)'s Topo-order satisfying that \( v \) can reach \( u \) and \( u \) has the largest Topo-order among all nodes of \( o_G(v) \). To mark all CNs, we first compute \( far(v) \) for every node \( v \). After that, if \( I_v.e = far(v) \), then we can safely say that \( v \) is a CN. The correctness is based on the following result.

Theorem 2: Given a node \( v \), \( v \) is a CN, if \( I_v.e = far(v) \).

Proof: If we first prove that if \( I_v.e = far(v) \), then \( v \) is a CN. For every node \( u \) in \( o_G(v) \), we know that \( t_v < t_u \). For every node \( w \), if \( t_v < t_w < t_u \), we know that \( w \) is a descendant of \( v \) in the TPS tree \( T \), i.e., all nodes in the subtree rooted at \( v \) in \( T \) are \( v \)'s descendants. Since every node \( u \) in \( o_G(v) \) satisfies that \( (1) \ t_v < t_u \) and \( (2) \ t_u < I_v.e \), we know that \( I_v.e = far(v) \) means that all nodes of \( o_G(v) \) are \( v \)'s descendants in \( T \). Therefore, \( I_v.e = far(v) \) means that \( v \) is a CN.

We then prove that if \( v \) is a CN, then \( I_v.e = far(v) \). According to Definition 1, if \( v \) is a CN, then every node \( u \) in \( o_G(v) \) is a descendant of \( v \) in the TPS tree \( T \), thus all nodes of \( o_G(v) \) are \( v \)'s descendants in \( T \), that is, \( I_v.e \geq far(v) \). On the other hand, all nodes in the subtree rooted at \( v \) are \( v \)'s descendants and \( v \) can reach them, thus \( I_v.e \leq far(v) \). Then we have that \( I_v.e = far(v) \).

We use Algorithm 2 to mark all CN nodes which computes \( far(v) \) for every node \( v \) in descending Topo-order \( X \) (lines 2-5). If \( far(v) \) is equal to the end value of \( v \)'s interval, it means that \( v \) is a CN, i.e., \( o_G(v) = \alpha(v) \). We mark \( v \) as CN node according to Definition 1 (line 3). After that, we update \( far(u) \) for each \( u \) of \( v \)'s in-neighbors (lines 4-5). The time complexity of this algorithm is \( O(|V| + |E|) \), and the space complexity is \( O(|V|) \) (line 1).

Algorithm 2 markCN(\( G = (V, E) \), Topo-Order \( X \))

1 \( far(v) = t_v \) for all nodes
2 for each \( (v \in V \) in descending order w.r.t. \( X \)) do
3   if (\( far(v) = I_v.e \)) then mark \( v \) as a CN
4   for each (\( u \in i_G(v) \)) do
5     if (\( far(u) < t_u \)) then \( far(u) = t_v \)

2) ESTIMATION HEURISTICS

According to Theorem 1, we know that if \( u \) is a CN, then we do not need to estimate \( u \)'s TC size, due to that we know the accurate TC size. For other nodes, we use the following heuristics to make estimation, where \( v_{max} \) is the node with the largest estimated TC size in \( o_G(u) \).
(H1) Using $v_{\text{max}}$ to make estimation, $\bar{N}(u) = |o_G(u)| + \bar{N}(v_{\text{max}})$.

(H2) Using the sum of sub-tree sizes as the estimation, $|o_G(u)| + \sum_{v \in o_G(u)} |o_T^v(v)|$.

Example 2: For H1, consider d in FIGURE 1(a). $o_G(d) = \{e, h\}$, e is the node with the largest estimated TC size in $o_G(d)$ and $\bar{N}(e) = 4$. With H1, we can get the estimated size of d, i.e., $\bar{N}(d) = |o_G(d)| + \bar{N}(e) = 2 + 4 = 6$. For H2, consider node d in FIGURE 1(a), $o_G(d) = \{e, h\}$, we know that $\bar{N}(o_G(d)) + |o_T^e(d)| = 2 + 0 = 2$.

As can be seen above, H1 and H2 are complementary to each other. When one gets a smaller value, the other usually gets a larger value. For example, for d in FIGURE 1(a), H2's result is 2, which is smaller than that of H1, and both are not greater than the accurate result. Further, when the set of subtrees of a node u have similar sizes, H1 may get results smaller than that of H2. Considering the above problems, we take the larger value of H1 and H2 as the estimated result, as shown by Equation (1).

$$\bar{N}(u) = \begin{cases} |o_T^u(u)|, & u \text{ is a CN} \\ |o_G(u)| + \max\{\bar{N}(v_{\text{max}}), \sum_{v \in o_G(u)} |o_T^v(v)|\}, & \text{otherwise} \end{cases}$$

(1)

There are two cases in Equation (1). (Case-1) u is a CN, we take $|o_T^u(u)|$ as the accurate result, which can be got in O(1) time, i.e., $|o_T^u(u)| = I_u.e - I_u.s$. (Case-2) u is not a CN, we take the larger value of $|o_G(u)| + \bar{N}(v_{\text{max}})$, $|o_G(u)| + \sum_{v \in o_G(u)} |o_T^v(v)|$ as the estimated result.

We use Algorithm 3 to perform the estimation, where $\bar{N}$ is used to store the estimated values of all nodes. We use Equation (1) to estimate the TC size of non-CNs (lines 6-8). The time complexity of this algorithm is $O(|V| + |E|)$, and the space complexity is $O(|V|)$.

**Algorithm 3 EstimateNodeValue** ($G = (V, E)$)

1. generate TPS tree $T$ with Topo-order $X$
2. initialize $\bar{N}$
3. for each $(u \in V$ in descending Topo-order $X$) do
4.  if $(u \in \text{CN})$ then $\bar{N}(u) = |o_T^u(u)|$
5.  else
6.    if $(o_G(u) = \emptyset)$ then $v_{\text{max}} \leftarrow 0$
7.    else $v_{\text{max}} \leftarrow \max_{v \in o_G(u)} \bar{N}(v)$
8.    $\bar{N}(u) = |o_G(u)| + \max\{\bar{N}(v_{\text{max}}), \sum_{v \in o_G(u)} |o_T^v(v)|\}$
9.  return $\bar{N}$

3) TOP-DOWN VERIFICATION

Although both H1 and H2 usually get values smaller than the accurate TC sizes, they may also produce estimated values larger than the accurate results. For example, for b in FIGURE 1(a), $|o_G^b(b)| = 8$, with Case-1 in Equation (1), $\bar{N}(a) = 10$. With Case-2 in Equation (1), $\bar{N}(b) = |o_G(b)| + \bar{N}(c) = 5 + 8 = 13$. According to Equation (1), the estimated TC size of b is 13. However, b’s in-neighbor is a, which is a CN, and we know its accurate TC size is 10 according to Equation (1). Given the accurate TC size of a, we know that the upper bound of the TC size of b, i.e., 13, is far from accurate and should be avoided.

Motivated by this, we propose to revise the estimated results by a top-down verification based on CNs.

The basic idea of our approach is to visit all nodes in top-down way. For each visited node $v \in V$, if the estimated value of each node $u$ of $v$’s out-neighbors is larger than that of $v$, we set its value as $\bar{N}(u) = \bar{N}(v) - 1$, such that for each node $v$, its estimated value is smaller than that of its in-neighbors.

**Algorithm 4 VerifyValue** ($G = (V, E)$, Topo-Order $X$)

1. for each $(v \in V$ in ascending Topo-order $X$) do
2.  for each $(u \in o_v(v))$ do
3.    if $\bar{N}(u) \geq \bar{N}(v)$ then
4.      $\bar{N}(u) = \bar{N}(v) - 1$
5.  return verified $\bar{N}$

**B. OPTIMIZATION**

1) LARGE DEGREE NODES ON THE RIGHT

Assuming that node $v$ is one of $u$’s out-neighbors, and $a$ has a large out degree. When the difference between the TC size of $u$ and $v$ is large, the estimated TC size of $u$ which is computed...
based on v’s estimated TC size may be far from accurate. According to Definition 1, every node on the rightmost path of the TPS tree will be marked as CN. Since node u has a large out degree, it will be more probable that u has a large TC size. If we can get its TC size more accurate and use it to help estimate the TC size of u’s in-neighbors, the estimation accuracy of many nodes can then be improved. We propose an optimization strategy that first processes nodes with a small out degree when constructing the TPS tree, such that to make sure nodes with the large out degree are on the rightmost path of the spanning tree T.

![DAG G with topo-order X2](image)

(a) DAG G with topo-order X2

![The TPS tree T2 of G generated from X2](image)

(b) The TPS tree T2 of G generated from X2

**FIGURE 3.** A DAG G and its spanning tree T2.

**Example 4:** We process G using Topo-order X2 in FIGURE 3(a), which is generated by first processing nodes with small out-degree during topological sorting. As shown by the integer on the right of each node. During the topological sorting, we generate the TPS tree T2, as shown by FIGURE 3(b). For example, for node a and its out-neighbors b and f, we first process f due to that o_G(b) > o_G(f).

For each node v of the given graph, let big be the one that has the largest out-degree among v’s out-neighbors. We use Algorithm 5 to move big to the last position of v’s adjacency list, such that node big will be processed last. Then, the TPS tree constructed by Algorithm 1 will produce CNs with large TC size. Based on these CNs, we can make more accurate TC size estimation. Algorithm 5 processes each node in the graph G one by one (line 1). During processing each node v, it identifies the out-neighbor with the largest out-degree, and moves it to the last position in v’s adjacency list. The time complexity of this algorithm is O(|V| + |E|), and the space complexity is O(|V|).

2) ESTIMATING WITH TWO TREES

According to Definition 1, we can get the accurate TC size of a CN. The more CNs we identified, the more accurate for TC size estimation. Considering this, we propose to use two TPS trees to find more CNs.

**Algorithm 5 SortGraph (G = (V, E))**

1. for each (v ∈ V) do
   2. initialize big = the first out-neighbor of v
   3. for each (i ∈ o_G(v)) do
   4. if (|o_G(i)| > |o_G(big)|) then
   5. big = i
   6. moving big to last position in o_G(v)
7. return sorted graph G

**Example 5:** By the TPS tree T1 in FIGURE 1(a), we have 7 CNs. Further, by the TPS tree T2 in FIGURE 3(a), we get three more CNs b, c, e.

The heuristics for TC size estimation is shown as Equation 2, as shown at the bottom of the page. To use Equation 2, we need to first construct two TPS trees and identify CNs on both trees. After that, we can make more accurate estimation using Equation 2.

There are three cases in Equation (2). (Case-1) and (Case-2) u is a CN, we take \( |o^*_T(u)| \) as the accurate TC size, i.e., \( |o^*_G(u)| = I_u,e - I_u,s \) (Case-3) u is not a CN, we take the larger value of \( |o_G(u)| + \bar{N}(v_{\text{max}}), |o_G(u)| + \sum_{v \in o_G(u)} |o^*_T(v)| \) in two trees as the estimated result, which guarantees that \( \bar{N}(u) \) is closer to the accurate value.

Given a DAG G, our algorithm, namely buTCS in Algorithm 6, obtains TC size estimation for every node, which is done by first calls Algorithm 5 to sort the graph G to ensure the maximum out degree node is at the last position of the current node’s adjacency list, then calls Algorithm 1 to get two different TPS trees. After that, it marks CNs by calling Algorithm 2, then calls Algorithm 3 which uses Equation 2 to obtain the estimated result, finally calling Algorithm 4 to verify the estimated result.

**Algorithm 6 buTCS (G = (V, E))**

1. sort graph G (Algorithm 5)
2. construct two TPS trees (Algorithm 1)
3. mark CNs (Algorithm 2)
4. estimate TC size (Algorithm 3)
5. verify value (Algorithm 4)

Since the time complexity of Algorithm 1, Algorithm 2, Algorithm 3, Algorithm 4 and Algorithm 5 is \( O(|V| + |E|) \). The overall time complexity of buTCS is \( O(|V| + |E|) \). During the processing, we need to maintain two TPS tree and CNs, the space complexity of buTCS is \( O(|V|) \).

**Example 6:** Given G with topo-order X in FIGURE 1(a). The estimated results of \( \text{linearTC} \) [23], [24] and buTCS are shown in FIGURE 4 (a) and (b) respectively.
TABLE 3. The statistics of each dataset, where $n_t$ is the number of topo-levels (the length of the longest path) of G, $p_{\text{leaf}}$ is the percent of the number of nodes without out-neighbors (in-neighbors), $p_{\text{CN}}$ is the percent of the number of CNs of G.

| Dataset  | $|V|$  | $|E|$   | $n_t$ | $p_{\text{leaf}}$ | $p_{\text{leaf}}$ | $p_{\text{CN}}$ |
|---------|-------|--------|-------|------------------|------------------|-----------------|
| email   | 231,000 | 223,004 | 7     | 17.3%            | 41.9%            | 24.1%           |
| unip150m| 25,037,600 | 25,037,598 | 10    | 0                | 86.4%            | 0%              |
| wiki    | 2,281,879 | 2,311,570 | 8     | 98.5%            | 1.1%             | 98.9%           |
| LJ      | 971,232  | 1,024,140 | 24    | 57.7%            | 40.0%            | 58.8%           |
| web     | 371,764  | 517,805  | 34    | 37.9%            | 43.7%            | 44.4%           |
| citeeseer| 6,540,401 | 15,011,260 | 59    | 87.7%            | 8.6%             | 88.7%           |
| dbpedia | 3,365,623 | 7,989,191 | 146   | 23.2%            | 50.3%            | 24.1%           |
| gowild  | 8,022,880 | 23,652,610 | 12    | 64.6%            | 16.2%            | 64.6%           |
| Patent  | 3,774,768 | 16,518,947 | 32    | 44.6%            | 13.6%            | 46.1%           |
| go-unip | 6,967,956 | 34,769,339 | 21    | 0.04%            | 99.6%            | <0.1%           |
| 10go-unip | 469,526  | 3,476,397 | 21    | <0.1%            | 95.5%            | <0.1%           |
| twitter | 18,121,168 | 18,359,487 | 22    | 90.4%            | 17.3%            | 90.6%           |
| web-uk  | 22,753,644 | 38,184,039 | 2,793 | 55.4%            | 38.4%            | 58.1%           |

FIGURE 4. The graph $G$ with estimated TC size using linearTC algorithm (a) and buTCS algorithm (b). The integer on the right of each node is its estimated TC size.

linearTC algorithm, we obtain $\tilde{N} (a) = 13$, $\tilde{N} (b) = 13$ and $\tilde{N} (c) = 8$, which is far from their accurate TC size. However, with buTCS algorithm, we mark $a$, $b$, $c$ as CNs and can obtain their accurate TC size.

IV. EXPERIMENT

We make comparison with several existing TC estimation approaches, including lb, ub [22], kr ($k = 50/100/150$) [21], linearTC [23], [24], and our buTCS. All experiments were run on a PC with Intel(R) Core (TM) i5-7300HQ CPU @ 2.50GHz, 8 GB memory, and Ubuntu 18.04 Linux OS.

Table 3 shows the statistics of 13 real datasets used in our experiments. Among these datasets used in our experiments, email, wiki, LJ, and web are directed graphs initially, we transformed them into DAGs by coalescing each strongly connected component into a node of DAGs, and these are downloaded from the same web page.

All other datasets are DAGs initially, and all of these are large datasets ($|V| > 100,000$). email is a DAG transformed from directed graph. email-EuAll, which is an email network from a EU research institution. unip150m is a DAG obtained from the RDF graph of UniProt, 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 a web graph web-Google. Patent (cit-Patents) and citeseerx are all citation networks with out-degree of non-leaf nodes ranging from 10 to 30. dbpedia is the DAG of a knowledge graph. gowild is a DAG transformed from a large RDF graph. go-unip (go-uniprot) and 10go-unip (10go-uniprot) are DAGs transformed from the joint graph of Gene Ontology terms with the annotations file from the UniProt. twitter is a DAG transformed from a large-scale social network obtained from twitter.com. web-uk is a DAG of a web graph dataset. It is worth noting that for twitter dataset in Table 3, $p_{\text{leaf}} + p_{\text{leaf}} > 1$, due to that there are many nodes without in- and out-neighbors.

A. TC SIZE ESTIMATION ACCURACY

Let $N(u) = |out^*_G (u)|$, $X$ is an algorithm for TC size estimation. We use two metrics error(u) and dev $(X, G)$ to show the accuracy of TC size estimation for different approaches. error(u) is computed based on equation (3), which is used as a metric to evaluate the goodness of TC size estimation on a single node. For node $u$, if error(u) is smaller, it means that $u$’s estimated TC size is more accurate. As a supplement, dev $(X, G)$ is the standard deviation computed based on equation (4), which measures the degree of the standard deviation.

1 http://snap.stanford.edu/data/index.html

2 http://www.uniprot.org/

3 http://pan.baidu.com/s/1c00Iq5E

4 https://code.google.com/p/ferrari-index/downloads/list
of the estimated $TC$ sizes of nodes from their accurate value w.r.t $X$ on $G$. $dev(X, G)$ is used to evaluate the goodness of $TC$ size estimation over all nodes of the input graph $G$. The smaller the standard deviation, the less the estimated values deviating from the accurate values. Besides, this metric can measure the goodness of the estimation method, that is, whether the overall estimation result is fluctuating greatly.

$$error(u) = \frac{|N(u) - \tilde{N}(u)|}{N(u)}$$  \hspace{1cm} (3)

$$dev(X, G) = \sqrt{\frac{\sum_{i=1}^{\mid V \mid} \left( \frac{N(i) - \tilde{N}(i)}{N(i)} \right)^2}{\mid V \mid - 1}}$$  \hspace{1cm} (4)

Further, we define the deviation promotion rate as Equation 5, by which we can evaluate the improvement of $buTCS$ on each $X$ of existing approaches. For one graph $G$ and method $X$, a large $promotion(buTCS, X, G)$ means that the $buTCS$ has a significant improvement effect on $G$ compared to $X$.

$$promotion(buTCS, X, G) = \frac{dev(X, G) - dev(buTCS, G)}{dev(X, G)}$$  \hspace{1cm} (5)

FIGURE 5 shows the comparison of error rates on 13 datasets of different $TC$ size estimation algorithms, from which we know that for all graphs, our $buTCS$’s is more accurate than existing methods because for our method many nodes are with error($u$) $\in$ [0, 0.2). For example, for email, unip150m, wiki, LJ, web, citeseerx and twitter, every node’s error is in the [0, 0.2) interval using $buTCS$.

Table 4 shows the standard deviation for different $TC$ size estimation algorithms, from which we can see that for all graphs, $buTCS$ has the smallest standard deviation, which means that $buTCS$ can make more accurate estimation compared with existing algorithms. For example, $buTCS$ gets
an estimation with the standard deviation less than 0.01 on several datasets, such as email, unip150m, wiki, LJ, web, citeseerx, twitter and web-uk. Even though linearTC also gets better estimation than other existing approaches, buTCS still works better than linearTC on all datasets. The reason lies in that (1) our spanning tree can help to identify more CNs, (2) our approach performs a top-down verification after estimation. Both two steps make the estimation more accurate than that of linearTC. For kr \((k = 50, k = 100, k = 150)\) method, as \(k\) increases from 50 to 150, the
TABLE 4. The standard deviation for different estimating methods.

| Dataset  | ub  | lb  | kr(k = 50) | kr(k = 100) | kr(k = 150) | linearTC | buTCS    |
|----------|-----|-----|------------|-------------|-------------|----------|----------|
| email    | 0.11| 0.61| 0.11       | 0.10        | 0.10        | 0.0067   | 0.0056   |
| unip150m | 0.10| 0.33| 0.19       | 0.18        | 0.17        | 0.015    | 0        |
| wiki     | 0.10| 0.11| 0.13       | 0.12        | 0.12        | 0.00042  | 0.00032  |
| LJ       | 0.10| 0.38| 0.57       | 0.56        | 0.55        | 0.0067   | 0.0057   |
| web      | 0.33| 0.49| 0.41       | 0.40        | 0.44        | 0.049    | 0.030    |
| citeseerx| 0.19| 0.18| 0.12       | 0.12        | 0.11        | 0.037    | 0.033    |
| dbpedia  | 0.40| 0.56| 0.34       | 0.34        | 0.33        | 0.33     | 0.21     |
| govwild  | 0.27| 0.34| 0.15       | 0.14        | 0.13        | 0.098    | 0.069    |
| Patent   | 0.62| 0.48| 0.56       | 0.54        | 0.54        | 0.50     | 0.40     |
| go-unip  | 0.84| 0.75| 0.52       | 0.50        | 0.50        | 0.42     | 0.29     |
| 10go-unip| 0.83| 0.75| 0.57       | 0.53        | 0.52        | 0.42     | 0.31     |
| twitter  | 0.14| 0.13| 0.21       | 0.21        | 0.20        | 0.0029   | 0.0014   |
| web-uk   | 0.44| 0.28| 0.30       | 0.30        | 0.28        | 0.071    | 0.058    |

TABLE 5. Ratio of the deviation improvement compared with previous methods.

| Dataset  | ub  | lb  | kr(k = 50) | kr(k = 100) | kr(k = 150) | linearTC |
|----------|-----|-----|------------|-------------|-------------|----------|
| email    | 0.95| 0.99| 0.95       | 0.94        | 0.94        | 0.15     |
| unip150m | 1   | 1   | 1          | 1           | 1           | 1        |
| wiki     | 0.99| 0.99| 0.99       | 0.99        | 0.99        | 0.99     | 0.25     |
| LJ       | 0.94| 0.99| 0.99       | 0.99        | 0.99        | 0.99     | 0.16     |
| web      | 0.89| 0.92| 0.91       | 0.91        | 0.92        | 0.92     | 0.23     |
| citeseerx| 0.82| 0.82| 0.73       | 0.72        | 0.69        | 0.69     | 0.12     |
| dbpedia  | 0.55| 0.53| 0.24       | 0.23        | 0.22        | 0.22     | 0.20     |
| govwild  | 0.75| 0.80| 0.55       | 0.52        | 0.47        | 0.47     | 0.31     |
| Patent   | 0.28| 0.18| 0.20       | 0.18        | 0.17        | 0.17     | 0.10     |
| go-unip  | 0.65| 0.58| 0.39       | 0.37        | 0.37        | 0.37     | 0.25     |
| 10go-unip| 0.63| 0.53| 0.38       | 0.34        | 0.33        | 0.33     | 0.15     |
| twitter  | 0.99| 0.99| 0.99       | 0.99        | 0.99        | 0.99     | 0.50     |
| web-uk   | 0.86| 0.78| 0.80       | 0.79        | 0.78        | 0.78     | 0.14     |

standard deviation becomes smaller, but still larger than our approach.

Table 5 shows the deviation improvement of our approach over existing approaches, from which we have the following observations.

1. Compared with lb and ub, buTCS’s improvement ratio is more than 0.9 for email, unip150m, wiki, LJ and twitter for email, unip150m, wiki, LJ, twitter.

2. Compared with kr \(k = 50, 100, 150\), buTCS’s improvement ratio is more than 0.90 on amaze, kegg, email, unip150m, wiki, LJ, web and twitter. For kr, it estimates the results based on \(k\) random permutations, its accuracy depends on the value of \(k\), the larger the value of \(k\), the more accurate of its estimation. Even though, the improvement ratio is always a positive number on each dataset, which means that buTCS works better than kr on all datasets.

3. Compared with linearTC, buTCS’s improvement ratio is more than 0.20 on email, unip150m, wiki, web, dbpedia, govwild and twitter. Especially on unip150m, the improvement ratio is 1, which also means that linearTC does not work well on this dataset. Further, for linearTC, the estimated TC size of a node may be greater than the accurate value of its in-neighbor, which may lead to the estimated value of its in-neighbor far from accurate.

However, it is worth noting that buTCS does not work well on all datasets, especially on graphs where most nodes have no in-neighbors, such as go-unip and 10go-unip, where more than 95% nodes have no in-neighbors. The reason lies in that the two graphs, the number of CNs is less than 0.1%, which means that for most nodes, the estimated TC size may not be equal to the accurate value by our approach. As a comparison, for wiki dataset, where 98.9% nodes are CNs, which means that for 98.9% nodes, the estimation will be equal to the accurate TC size. Further, our approach is inspired by \(TC_{err}\), from the fifth column of Table 2, we know that \(TC_{err}\) is more than 30% for go-unip and 10go-unip, which also means that our approach may not work well on the two...
TABLE 6. The standard deviation for different optimization strategies.

| Dataset  | buTCS-B | buTCS-1 | buTCS-2 |
|----------|---------|---------|---------|
| email    | 0.0064  | 0.0063  | 0.0056  |
| unip150m | 0.0013  | 0.0011  | 0.0084  |
| wiki     | 0.00039 | 0.00036 | 0.00032 |
| LJ       | 0.0065  | 0.0062  | 0.0057  |
| web      | 0.038   | 0.035   | 0.030   |
| citeseerx| 0.035   | 0.034   | 0.033   |
| dbpedia  | 0.31    | 0.29    | 0.21    |
| govwild  | 0.091   | 0.087   | 0.069   |
| Patent   | 0.45    | 0.44    | 0.40    |
| go-unip  | 0.36    | 0.34    | 0.29    |
| 10go-unip| 0.35    | 0.34    | 0.31    |
| twitter  | 0.0024  | 0.0023  | 0.0014  |
| web-uk   | 0.064   | 0.061   | 0.058   |

From FIGURE 5, Table 4 and Table 5, we know that even though our approach does not work well on go-unip and 10go-unip compared with other datasets, compared with all existing approaches on the two datasets, it still achieves the best estimation accuracy.

B. TC SIZE ESTIMATION EFFICIENCY

FIGURE 6 shows the comparison of actual memory usages, from which we know that buTCS consumes more memory than that of linearTC (usually 7%-10%), due to that buTCS uses two spanning trees. However, as both approaches have linear space complexity, all these graphs can be loaded into memory and processed with limited space. For example, web-uk has more than 20,000,000 nodes, the actual memory usage is 985MB for linearTC and 1,072MB for buTCS.

FIGURE 7 shows the comparison of running times, from which we know that buTCS consumes similar time as ub, lb and linearTC, due to that the time complexity of these algorithms is O(|V| + |E|). As a comparison, kr needs to consume much more time for TC size estimation, and with the increase of k, it consumes more time.

From FIGURE 6 and FIGURE 7, we know that for buTCS, all these datasets can be loaded into memory and efficiently processed with limited space, due to that it has linear space and time complexities.

C. IMPACTS OF THE OPTIMIZATION

To show the impacts of the optimization techniques used in our algorithm, we test three versions of our approach, including: (1) buTCS-B, which is the basic version without using the two optimizations; (2) buTCS-1, which uses only the first optimization, and (3) buTCS-2, which uses both the two optimizations and is the buTCS algorithm.

From II we know that buTCS-2 works more accurately than buTCS-B and buTCS-1. The reason lies in that buTCS-2 not only constructs two TPS tree, but also moves large degree nodes to the right of the spanning tree. Both optimizations can help find more CNs, which results in more accurate estimation.

From Table 6 we know that buTCS-2 has a smaller standard deviation than buTCS-B and buTCS-1.

Table 7 shows the comparison of running times, from which we know that buTCS-2 needs to consume a little more time than buTCS-B and buTCS-1, due to buTCS-2 using both optimizations.

V. CONCLUSION

In this paper, we focus on TC size estimation. We propose an algorithm, namely buTCS, to estimate all nodes’ TC size in linear time \( O(|V| + |E|) \), where \( V \) is the set of nodes and \( E \) is the set of edges of the input graph. The basic idea is to estimate a node’s TC size in two steps. First, it gets the initial estimation result of a node based on the estimated TC size of its out-neighbors. Second, it revises the result with a top-down verification. We further propose two optimization strategies to make improvements. We conduct rich experiments on 13 real datasets. The experimental results show that we can get a better estimation efficiently compared with the existing algorithms. As an indication, among 13 real datasets being tested, the ratio of the standard deviation improvement for buTCS is around 97% compared with lb, ub and kr. Compared with linearTC, buTCS get a better estimation on all datasets and the ratio of the standard deviation improvement is more than 30% on several datasets.

REFERENCES

[1] M. Yannakakis, “Graph-theoretic methods in database theory,” in Proc. 9th ACM SIGACT-SIGMOD-SIGART Symp. Princ. Database Syst. (PODS), 1990, pp. 230–242.

[2] 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, 1972.
REFERENCES

[3] Y. Chen and Y. Chen, “Decomposing DAGs into spanning trees: A new way to compress transitive closures,” in Proc. IEEE 27th Int. Conf. Data Eng., Hannover, Germany, Apr. 2011, pp. 1007–1018.

[4] C. Ding, X. He, and H. Peng, “Finding cliques in protein interaction networks via transitive closure of a weighted graph,” in Proc. 5th Int. Conf. Bioinformatics (BIOKDD), Chicago, IL, USA, Aug. 2005, pp. 69–75.

[5] G. H. L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu, “The impact of transitive closure on the Boolean expressiveness of navigational query languages on graphs,” in Foundations of Information and Knowledge Systems. Kiel, Germany: Springer, Mar. 2012, pp. 124–143.

[6] G. H. L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu, “The impact of transitive closure on the expressiveness of navigational query languages on unlabeled graphs,” Ann. Math. Artif. Intell., vol. 73, nos. 1–2, pp. 167–203, Jan. 2015.

[7] J. J. Pfeiffer, T. La Fond, S. Moreno, and J. Neville, “Fast generation of large scale social networks while incorporating transitive closures,” in Proc. Int. Conf. Privacy, Secur., Risk Trust, Int. Conf. Social Comput. (SocialCom), 2012, Amsterdam, The Netherlands, Sep. 2012, pp. 154–165.

[8] V. King, “Fully dynamic transitive closure,” in Encyclopedia of Algorithms. New York, NY, USA: Springer, 2016, pp. 808–809.

[9] I. Krommidas and C. Zaroliagis, “An experimental study of algorithms for fully dynamic transitive closure,” ACM J. Exp. Algorithms, vol. 12, pp. 1–22, Jun. 2008.

[10] P. Niesink, K. Poulin, and M. Sajna, “Computing transitive closure of bipolar weighted digraphs,” Discrete Appl. Math., vol. 161, nos. 1–2, pp. 217–247, Jan. 2013.

[11] M. Palkowski and W. Bielecki, “Optimizing numerical code by means of the transitive closure of dependence graphs,” in Proc. Federated Conf. Comput. Sci. Inf. Syst., Prague, Czech Republic, Sep. 2017, pp. 523–526.

[12] L. Roditty, “A faster and simpler fully dynamic transitive closure,” ACM Trans. Algorithms, vol. 4, no. 1, pp. 1–16, Mar. 2008.

[13] P. Sankowski and M. Mucha, “Fast dynamic transitive closure with lookahead,” Algorithmica, vol. 56, no. 2, pp. 180–197, Feb. 2010.

[14] S. Anirban, J. Wang, and M. S. Islam, “Modular decomposition-based graph compression for fast reachability detection,” Data Sci. Eng., vol. 4, no. 3, pp. 193–207, Sep. 2019.

[15] S. Dar, “Augmenting databases with generalized transitive closure,” M.S. thesis, Dept. Eng. Exp. Station Madison, Univ. Wisconsin-Madison, Madison, WI, USA, 1993.

[16] R. Jin and G. Wang, “Simple, fast, and scalable reachability Oracle,” Proc. VLDB Endowment, vol. 6, no. 14, pp. 1978–1989, Sep. 2013.

[17] 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. Inf. Knowl. Manage. (CIKM), 2013, pp. 1601–1606.

[18] X. Tang, Z. Chen, K. Li, and X. Liu, “Efficient computation of the transitive closure size,” Cluster Comput., vol. 22, no. S3, pp. 6517–6527, May 2019.

[19] C. E. Leiserson, R. L. Rivest, T. H. Cormen, and C. Stein, Introduction to Algorithms. New York, NY, USA: McGraw-Hill, 1990.

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

[21] E. Cohen, “Estimating the size of the transitive closure in linear time,” in Proc. 35th Ann. Symp. Found. Comput. Sci., Santa Fe, NM, USA, Nov. 1994, pp. 190–200.

[22] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, “Reachability queries on large dynamic graphs: A total order approach,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, Jun. 2014, pp. 1323–1334.

[23] 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, May 2017, pp. 1–16.

[24] 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.

[25] K. Simon, “An improved algorithm for transitive closure on acyclic digraphs,” Theor. Comput. Sci., vol. 58, nos. 1–3, pp. 325–346, Jun. 1988.

---

XIAOZHE LI was born in Xi’an, Shanxi, China, in 2000. He is currently pursuing the bachelor’s degree with Donghua University, Shanghai, China. His current research interests include graph data management and data mining.

XUAN WANG was born in Qiqihar, Heilongjiang, China, in 1977. She received the B.S. and M.S. degrees in computer science and the Ph.D. degree in circuits and systems from Yanshan University, Qinhuangdao, China, in 1999, 2002, and 2008, respectively.

From 2008 to 2019, she was an Associate Professor with Yanshan University. Since 2019, she has been an Associate Professor with the School of Computer Science and Technology, Donghua University, Shanghai. Her research interests include information query processing on semi-structured data and graph data.

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.

MING DU was born in Hulin, Heilongjiang, China, in 1975. He received the B.S. degree in electronic technology from Chinese Textile University, in 1998, and the M.S. degree in control theory and control engineering and the Ph.D. degree in management science and engineering from Donghua University, Shanghai, China, in 2005 and 2013, respectively.

Since 2010, he has been an Associate Professor with Donghua University. His research interests include information retrieval techniques, data analysis and mining, and natural language processing.

---

**VOLUME 9, 2021**

116539