 CMC, vol.55, no.2, pp.267-283, 2018

An Optimized Labeling Scheme for Reachability Queries

Xian Tang[1, *], Ziyang Chen[2], Haiyan Zhang[3], Xiang Liu[1], Yunyu Shi[1] and Asad Shahzadi[4]

Abstract: Answering reachability queries is one of the fundamental graph operations. Existing approaches either accelerate index construction by constructing an index that covers only partial reachability relationship, which may result in performing cost traversing operation when answering a query; or accelerate query answering by constructing an index covering the complete reachability relationship, which may be inefficient due to comparing the complete node labels. We propose a novel labeling scheme, which covers the complete reachability relationship, to accelerate reachability queries processing. The idea is to decompose the given directed acyclic graph (*DAG*) G into two subgraphs, G_1 and G_2. For G_1, we propose to use topological labels consisting of two integers to answer all reachability queries. For G_2, we construct 2-hop labels as existing methods do to answer queries that cannot be answered by topological labels. The benefits of our method lie in two aspects. On one hand, our method does not need to perform the cost traversing operation when answering queries. On the other hand, our method can quickly answer most queries in constant time without comparing the whole node labels. We confirm the efficiency of our approaches by extensive experimental studies using 20 real datasets.

Keywords: DAG, computing, detection, reachability queries processing.

1 Introduction

A reachability query $u?\rightarrow v$ asks, in a directed graph G, whether there exists a path from u to v. Answering reachability queries is one of the fundamental graph operations, which has been extensively studied in the past decades [Agrawal, Borgida and Jagadish (1989); Chen, Gupta and Kurul (2005); Chen and Chen (2008); Chen and Chen (2011); Cheng, Huang, Wu et al. (2013); Cheng, Yu, Lin et al. (2006); Cheng, Yu, Lin et al. (2008); Cohen, Halperin, Kaplan et al. (2002); Jagadish (1990); Jin, Ruan, Dey et al. (2012); Jin, Ruan, Xiang et al. (2011); Jin and Wang (2013); Jin, Xiang, Ruan et al. (2009); Jin, Xiang, Ruan et al. (2008); Seufert, Anand, Bedathur et al. (2013); Su, Zhu, Wei et al. (2017); Trißl and Leser (2007); van Schaik and de Moor (2011); Veloso, Cerf, Junior et al. (2014); Wang,

[1] School of Electronic and Electrical Engineering, Shanghai University of Engineering Science, Shanghai 201600, China.

[2] Lixin University of Accounting and Finance, Shanghai 201620, China.

[3] Yanshan University, Qinhuangdao 066004, China.

[4] NCBAE, Lahore 54660, Pakistani.

* Corresponding author: Xian Tang. Email: txianz@163.com.

He, Yang et al. (2006); Wei, Yu, Lu et al. (2014); Yano, Akiba, Iwata et al. (2013); Yildirim, Chaoji and Zaki (2010); Yildirim, Chaoji and Zaki (2012); Yu and Cheng (2010); Zhang, Yu, Qin et al. (2012); Zhu, Lin, Wang et al. (2014); Li and Li (2010)]. Its applications include, but not limited to, Semantic Web (RDF), online social networks, biological networks, ontology, transportation networks, etc. For reachability queries processing, there are three interrelated issues: Index size, index construction time, and query time. Given a reachability query $u?\rightarrow v$, the two extremes of answering it are: (1) traversing from u to v to find the final answer without building index in-advance, and (2) maintaining the whole transitive closure (*TC*) to check whether u can reach v. On one hand, the traversing approaches do not need to build index offline, however, it needs to visit all nodes that u can reach in the worst case by performing expensive depth-first-search (*DFS*) or breadth-first-search (*BFS*) operation to answer the given query. On the other hand, computing the *TC* makes it difficult to be scaled to large graphs, due to unacceptable index size and indexing time. Considering this problem, all existing approaches made trade-off among index construction, index size and query performance, where index construction is a one-time activity. Due to its importance and the emergence of large graphs in big data era [Wu, Zapevalova, Chen et al. (2018)], faster answering reachability queries *online* is still a challenging task, supposing that the index can be constructed *offline* with reasonable time and size.

We follow [Wei, Yu, Lu et al. (2014)] and classify existing methods into two categories according to the coverage of index on reachability information: *Label-Only* [Agrawal, Borgida and Jagadish (1989); Chen and Chen (2008); Cheng, Huang, Wu et al. (2013); Cohen, Halperin, Kaplan et al. (2002); Jagadish (1990); Jin and Wang (2013); Jin, Xiang, Ruan et al. (2009); Jin, Xiang, Ruan et al. (2008); van Schaik and de Moor (2011); Wang, He, Yang et al. (2006); Yano, Akiba, Iwata et al. (2013); Zhu, Lin, Wang et al. (2014)] and *Label+G* [Chen, Gupta and Kurul (2005); Jin, Ruan, Dey et al. (2012); Jin and Wang (2013); Seufert, Anand, Bedathur et al. (2013); Trißl and Leser (2007); Veloso, Cerf, Junior et al. (2014); Wei, Yu, Lu et al. (2014; Yano, Akiba, Iwata et al. (2013); Yildirim, Chaoji and Zaki (2010); Yildirim, Chaoji and Zaki (2012)]. Here, we say an algorithm belongs to *Label-Only*, it means that the index conveys the *complete* reachability information, and any given query $u?\rightarrow v$ can be answered by comparing labels of u and v. When we say that an algorithm belongs to *Label+G*, it means that the index covers *partial* reachability information, thus we may need to perform *DFS/BFS* from u to check whether u can reach v, if we cannot get the result by comparing labels of u and v. Most existing *Label+G* approaches possess the benefits of bounded index size and index construction time in theory. However, they all suffer from the same performance bottleneck, i.e. traversing from u to v to find the answer. As a comparison, among existing *Label-Only* approaches, recent variants of 2-hop labels, such as *PLL* [Yano, Akiba, Iwata et al. (2013)], *DL* [Jin and Wang (2013)], *TF* [Cheng, Huang, Wu et al. (2013)] and *TOL* [Zhu, Lin, Wang et al. (2014)], does not suffer from the expensive *DFS/BFS* operation on the underlying graph. Moreover, the index can be constructed efficiently with reasonable size, which makes these approaches can scale to large graphs. However, they could be *inefficient* when answering *unreachable* queries due to the higher cost of set intersection operation on node labels to get the result.

To overcome the shorting comings of existing approaches, we propose a novel *Label-Only*

labeling scheme, namely *T2H*, based on which reachability queries can be answered more efficiently than current ones. Given a *DAG G*, the basic idea is finding from *G* a set of "block nodes", such that after removing these block nodes from *G*, we get a reduced graph G_1, and the reachability relationship between any pair of nodes in G_1 can be answered in constant time with linear index size. Then, for reachability relationship conveyed by block nodes, we construct block nodes based 2-hop labels to reduce the 2-hop index size. The benefits of our labeling scheme lie in the following aspects: (1) compared with existing *Label+G* approaches, our method does not need to perform costly *DFS/BFS* operation, and can be much more efficient than existing ones when answering reachable queries, (2) compared with existing *Label-Only* approaches, our method significantly reduces the cost of set intersection operation on node labels to get final results. We conduct rich experiment on real datasets to show that our approach can answer both positive and negative queries very efficiently, and can scale to large graphs.

2 Background and related work

We follow the tradition of existing approaches and assume that the input graph is a *DAG* $G=(V, E)$, which can be constructed from the given directed graph $\mathcal{G}$ in linear time w.r.t. the size of $\mathcal{G}$ [Tarjan (1972)] by coalescing each *SCC* of $\mathcal{G}$ into a node in *G*, where each node $v \in V$ represents an *SCC* S_v of $\mathcal{G}$, and each edge $(u, v) \in E$ represents the edge from *SCC* S_u to S_v if there is an edge from a node in S_u to a node in S_v. Similar as Zhou et [Zhou, Zhou, Yu et al. (2017)], we use $in_G(u)=\{v|(u, v)\in E\}$ to denote the al. set of in-neighbors nodes of u in *G*, and $out_G(u)=\{v|(u, v)\in E\}$ the set of out-neighbors nodes 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 performing a topological sorting on *G* in linear time $O(|V|+|E|)$ [Simon (1988)]. A topological sorting of *G* is a mapping t^X: V→X, such that $\forall(u, v) \in E$, we have $t_u^X < t_v^X$, where $t_u^X (t_v^X)$ is the topo-order of $u(v)$ w.r.t. X.

We discuss existing algorithms from two categories: (1) *Label-Only* and (2) *Label+G*. By *Label-Only*, a given query $u?\rightarrow v$ can be answered by comparing labels of u and v. By *Label+G*, this query can be answered by *DFS/BFS* at run-time, when it cannot be answered by labels of u and v. We use $u\rightarrow v$ ($u \nrightarrow v$) to denote that u can (cannot) reach v.

The *Label-Only* methods [Agrawal, Borgida and Jagadish (1989); Cheng, Huang, Wu et al. (2013); Jin, Ruan, Xiang et al. (2011); Jin and Wang (2013); Jin, Xiang, Ruan et al. (2009); Jin, Xiang, Ruan et al. (2008); van Schaik and de Moor (2011); Yano, Akiba, Iwata et al. (2013); Zhu, Lin, Wang et al. (2014)] focus on compressing *TC* to get a smaller index size for fast query processing. The recent work includes variants of 2-hop labeling scheme, such as *TF* [Cheng, Huang, Wu et al. (2013)], *DL* [Jin and Wang (2013)], *PLL* [Yano, Akiba, Iwata et al. (2013)] and *TOL* [Zhu, Lin, Wang et al. (2014)]. 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 called the out (in) label of u consisting of a set of nodes that u can reach (be reached). Based on this labeling scheme, $u?\rightarrow v$ can be answered by testing whether the result of $L_{out}(u)\cap L_{in}(v)$ is empty or not. If $L_{out}(u)\cap L_{in}(v)\neq \emptyset$, then $u\rightarrow v$, otherwise $u\nrightarrow v$. To compute node labels, *TF* [Cheng, Huang, Wu et al. (2013)] folds the given *DAG* recursively based on topological level to reduce the cost of 2-hop computation. *DL* [Jin and Wang (2013)], *PLL* [Yano, Akiba, Iwata et al.

(2013)] and *TOL* [Zhu, Lin, Wang et al. (2014)] share the similar idea of computing 2-hop labels. Given all nodes in a certain order, the construction of *DL* and *PLL* labels is enumerating each node with a forward *BFS* and a backward *BFS* to add u to labels of nodes that *u* can reach and nodes that can reach *u*. During each *BFS*, an early stop condition is adopted to accelerate the computation and reduce the index size.

The *Label+G* methods [Seufert, Anand, Bedathur et al. (2013); Trißl and Leser (2007); Veloso, Cerf, Junior et al. (2014); Wei, Yu, Lu et al. (2014); Yildirim, Chaoji and Zaki (2010); Yildirim, Chaoji and Zaki (2012)] answer $u?\rightarrow v$ by performing *DFS* from *u* at run-time if needed. The recent work includes *GRAIL* [Yildirim, Chaoji and Zaki (2010); Yildirim, Chaoji and Zaki (2012)], *FERRARI* [Seufert, Anand, Bedathur et al. (2013)], *FELINE* [Veloso, Cerf, Junior et al. (2014)] and IP^+ [Wei, Yu, Lu et al. (2014)]. All these methods focus on efficient pruning techniques to answer most unreachable queries. They also use additional pruning strategies to make optimization on both positive and negative queries, such as comparing topological levels [Seufert, Anand, Bedathur et al. (2013); Veloso, Cerf, Junior et al. (2014); Wei, Yu, Lu et al. (2014); Yildirim, Chaoji and Zaki (2010); Yildirim, Chaoji and Zaki (2012)], comparing topological orders [Seufert, Anand, Bedathur et al. (2013)], and comparing intervals of *u* and *v* over a spanning tree [Veloso, Cerf, Junior et al. (2014)].

Apart from the above methods that work on a large *DAG G*, there are studies focusing on reducing *G* to a smaller *DAG* to accelerate reachability query processing, including *SCARAB* Framework [Jin, Ruan, Dey et al. (2012)] and equivalence reduction [Fan, Li, Wang et al. (2012); Zhou, Zhou, Yu et al. (2017)], which are orthogonal to the above approaches and can be combined with the above approaches to accelerate answering reachability queries.

3 Query processing

3.1 The T2H labeling scheme

Given a topo-order *X* of a *DAG G*, it is obvious that if $u\rightarrow v$, then we must have $t_u^X<t_v^X$, but the other side does not always hold, i.e. when $t_u^X<t_v^X$, we cannot conclude that $u\rightarrow v$. However, if for every unreachable query $u\nrightarrow v$, we have that $t_u^X>t_v^X$, then when $t_u^X<t_v^X$, we can safely say that $u\rightarrow v$. Therefore, the key problem is how to know that we can correctly identify all unreachable queries using topo-orders.

Given two nodes *u* and *v*, there are at most two queries, i.e. $u?\rightarrow v$ and $v?\rightarrow u$. Therefore, it is not possible to correctly check the unreachable relationship using one topo-orders, e.g. consider v_2 and v_3 in *G* of Fig. 1(a), since $t_{v_3}^X=3>2=t_{v_2}^X$, we know that $v_3\nrightarrow v_2$. But, for query $v_2?\rightarrow v_3$, we cannot get the results by comparing their topo-orders. The natural idea is using one more topo-order to check the unreachable relationship on the other direction. Here, we use reversed topo-order to check unreachable queries.

Definition 1. ***(Free Node)*** *During a topological sorting, a node u is called a free node, if u has not been visited, but all of u's in-neighbors have been visited.*

For example, for *G* in Fig. 1(a), the free node is v_1 before processing, and after processing v_1, the free nodes are v_2, v_3 and v_5.

Definition 2. ***(Reversed Topo-Order)*** *Given two topo-orders X and Y, we say X(Y) is the*

reversed topo-order of Y (X), if Y (X) is computed in the way that the topological sorting will always first visit and assign the topo-order in Y (X) to the free node with the maximal topo-order in X(Y).

Given the uniqueness of a topo-order, its reversed topo-order is also unique. For example, given the *circled* topo-order X of all nodes in G of Fig. 1(a), we can get the reversed topo-order Y according to Definition 2, as shown by the italic integer beside each node. Obviously, we know that $v_3 \nrightarrow v_2$ because $t^X_{v_3}=3>2=t^X_{v_2}$, and $v_2 \nrightarrow v_3$ because $t^X_{v_2}=7>4=t^X_{v_3}$.

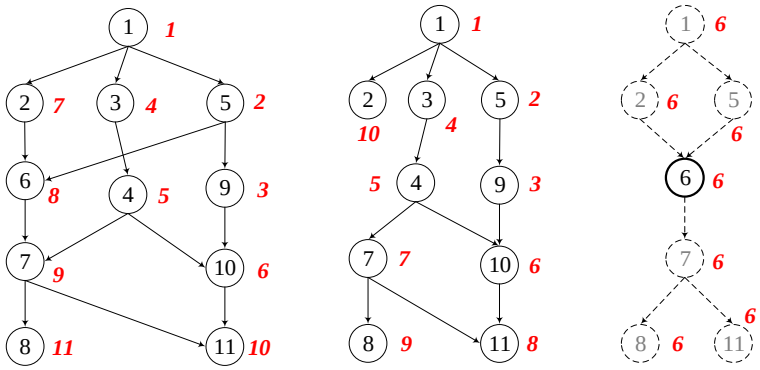

Figure 1: Illustration of the new labeling scheme, where (a) is a DAG with nodes denoted by their topo-orders, the italic integer beside each node is its reversed topo-order, (b) is the trivial subgraph $G1$ of G, the italic integer beside each node is its reversed topo-order, and (c) is the link graph $G2$ of G consisting of all nodes that can reach (be reached by) these block nodes

Even though we can identify more unreachable queries using multiple topo-orders, there exists some problematic *DAGs* for which a limited number of topo-orders will always produce false positive answers [Kornaropoulos and Tollis (2011); Veloso, Cerf, Junior et al. (2014)], i.e. even if $t^X_u<t^X_v$, we cannot tell that u can reach v. For example, for G in Fig. 1(a), even though $t^X_{v_3}=3<6=t^X_{v_6}$ and $t^X_{v_3}=4<8=t^X_{v_6}$, we know that $v_3 \nrightarrow v_6$ according to G. In fact, minimizing the number of false-positives is known to be NP-hard [Kornaropoulos and Tollis (2011); Veloso, Cerf, Junior et al. (2014)]. Moreover, using more topo-orders may result in unacceptable space and time cost to index construction and query processing.

Definition 3. ***(Trivial Graph)*** *Given a DAG G and one of its topo-order X. We say G is a trivial graph w.r.t. X, if every unreachable query on G can be identified by X and its reversed topo-order Y.*

For example, G_1 in Fig. 1(b) is a trivial graph. Interested readers can easily verify that for every unreachable query, we can always return FALSE based on the two reversed topo-orders shown in G_1.

Definition 4. ***(Block Node Set)*** *Given a DAG G, let S be a set of nodes in G. If G is a trivial graph after removing all nodes of S, then we call S a block node set of G and each node of S a block node.*

For example, for G in Fig. 1(a), v_6 is a block node, and $S=\{v_6\}$ is a block node set of G. After removing v_6 from G, we get the trivial graph G_1 in Fig. 1(b).

Based on two above definitions, we propose to find from the given *DAG G* a trivial graph G_1 by removing from *G* a set of "block nodes". The basic idea is that given a trivial graph G_1, we can identify all unreachable queries based on two reversed topo-orders. Then, for the reachability relationship between nodes in *G* that cannot be covered by the two topo-orders, we construct 2-hop labels from these block nodes. For example, for *G* in Fig. 1(a), we generate the trivial graph G_1 in Fig. 1(b) by removing the unique block node v_6. To answer reachability queries, we assign each node in G_1 two topo-orders shown in Fig. 1(b). Then, we performing forward and backward *DFS*/*BFS* from v_6 to construct 2-hop labels. Combining the two aspects together, we have the *T2H* (short for topo-order and 2-hop) label for each node, which we call as the *T2H* labeling scheme. Hereafter, we use t_v to denote both t_v^X and t_v^Y.We call it as *v*'s topo-label, and $t_u<t_v$ means that $t_u^X<t_v^X \wedge t_u^Y<t_v^Y$.We further use $L_v=\{L_{out}(v), L_{in}(v)\}$ to denote *v*'s block nodes based 2-hop labels.

3.2 Query algorithm

Before discussing the query algorithm, lets first introduce the following results.

Theorem 1. *Given a trivial graph G_1, we can answer all queries between nodes in G_1 using two reversed topo-orders.*

Proof. First, we can answer all unreachable queries using two reversed topo-orders according to Definition 3. Assume that there exists a reachable query $u \rightarrow v$ that we cannot correctly answer, it means that we have $t_u \not< t_v$. Then, we know that $u \not\rightarrow v$, which contradicts the assumption. Therefore, we can answer all queries using two reversed topo-orders.

Definition 5. *Given a DAG G and the set of block nodes S, we call the graph consists of all nodes that can reach and be reached by every block node the link graph w.r.t. S.*

For example, for *G* in Fig. 1(a), v_6 is its unique block node, and the link graph is shown as G_2 in Fig. 1(c). We have the following result.

Corollary 1. *Given a DAG $G=(V, E)$, its trivial graph $G_1=(V_1, E_1)$ and link graph $G_2=(V_2, E_2)$, we have that $V=V_1 \cup V_2$ and $E=E_1 \cup E_2$.*

Theorem 2. *All queries can be correctly answered based on T2H labels.*

Proof. First, according to Theorem 1, we can answer all reachable queries between nodes in G_1. Second, the remaining queries over *G* can be either reachable through block nodes, or unreachable. Thus, by comparing block nodes based 2-hop labels, we can correctly answer the remaining reachable queries, and all other queries are definitely unreachable ones.

Based on Theorems 1 and 2, we have Algorithm 1, which first check whether u can reach *v* using topo-labels in lines 1-2, then check whether *u* can reach *v* using block nodes based 2-hop labels in lines 2-3. And return FALSE in line 5 to denote that *u* cannot reach *v*.

Algorithm 1: *T2H*-1(u, v) /*$u \neq v$*/

```
1 if (both u and v are not block nodes, and t_u < t_v) then
2     return TRUE
3 if (L_out(u) ⋂ L_in(v) ≠ ∅) then
4     return TRUE
5 return FALSE
```

Example 1. Consider *G* in Fig. 1(a). The *T2H* labels are shown in Figs. 1(b) and 1(c). For query $v_3?\rightarrow v_8$, Algorithm 1 can return TRUE in line 2 denoting $v_3\rightarrow v_8$ due to that $t_{v_3}<t_{v_8}$. For query $v_2?\rightarrow v_8$, Algorithm 1 will return TRUE in line 4 denoting $v_2\rightarrow v_8$. For query $v_3?\rightarrow v_6$, Algorithm 1 will return FALSE in line 5 denoting $v_3\nrightarrow v_6$, due to that v_6 is a block node and $L_{out}(v_3)\cap L_{in}(v_6)=\emptyset$.

Obviously, if the given query can be answered in line 2, then the time complexity is $O(1)$, otherwise, the time complexity is $O(L)$, where L is maximal length of block nodes based 2-hop labels.

Further, as block nodes based 2-hop labels capture all reachable relationships between nodes linked by them, we have the second algorithm to answer a given query, as shown by Algorithm 2. The correctness is based on Theorems 1 and 2. The difference between the two algorithms lies in whether we first compare topo-labels. And the time complexity of Algorithm 2 is $O(L)$.

Algorithm 2: *T2H*-$2(u, v)$ /*$u \neq v$*/

```
1 if (L_out(u) ⋂ L_in(v) ≠ ∅) then
2     return TRUE
3 if (t_u < t_v) then
4     return TRUE
5 return FALSE
```

In essence, either algorithm can work better than the other in different scenario. If most reachable queries cannot be answered by comparing topo-labels, then Algorithm 2 could be more efficient, otherwise Algorithm 1 could be better.

3.3 Optimization

Both Algorithms 1 and 2 involve two operations when answering reachability queries, i.e. comparing topo-labels and comparing block nodes based 2-hop labels. As the cost of comparing topo-labels is $O(1)$, which is better than that of comparing 2-hop labels with cost $O(L)$, if most reachable queries can be answered by comparing topo-labels, then Algorithm 1 is the better choice. Unfortunately, this is not true in practice, especially for randomly generated workload, where most queries are unreachable ones. The optimization in this section is to extend the topo-labels, such that it can be used in Algorithm 1 to test not only reachable queries, but also unreachable queries. The idea is to assign *G* one more topo-order *Z*, which is the reversed topo-order of *X*.

Algorithm 3: *T2H*-O(u, v) /*$u \neq v$*/

```
1 if (both u and v are not block nodes, and t_u < t_v) then
2     return TRUE
3 if (t_u^X > t_v^X ⋁ t_u^Z < t_v^Z) then
4     return FALSE
5 if (L_out(u) ⋂ L_in(v) ≠ ∅) then
6     return TRUE
7 return FALSE
```

In summary, in the optimized algorithm, there are three topo-orders *X*, *Y*, *Z*, where *X* and *Y* are mutually reversed to each other w.r.t. the trivial graph G_1, while X and Z are mutually reversed to each other w.r.t. the input graph *G*. Since the given reachability queries are answered according to *G*, and an unreachable query on G_1 could be reachable on *G*, thus *X* and *Y* cannot be used to find unreachable queries before comparing block nodes based 2-hop labels. Fortunately, we can use *X* and *Z* to find unreachable queries, which are defined on *G*. In this way, we can answer more queries in constant time. Based on the above discussion, we have the optimized algorithm with the same time complexity as Algorithm 1, as shown below.

4 Index construction

We first discuss how to generate the block nodes based 2-hop labels. Given a set of block nodes, we perform forward (backward) *BFS* from each block node to add it to the in (out) label of the set of nodes that it can reach (be reached by). This is similar to existing 2-hop approaches [Jin and Wang (2013); Yano, Akiba, Iwata et al. (2013); Zhu, Lin, Wang et al. (2014)]. The difference is that the number of nodes from which we perform *BFS* is reduced.

We then discuss how to generate topo-labels and block nodes. Given *G*, we perform the first topological sorting on *G* to generate topo-order *X*. This can be done by performing the following steps: (1) push all free nodes into a stack *S*, (2) pop a node *u* from *S*, assign *u* its topo-order t_u^X, and push all of *u*'s out-neighbors which are new free nodes into *S*, and (3) repeat the above two steps until *S* is empty. For example, every node u in Fig. 1(a) is denoted by its topo-order t_u^X.

Given *X*, we discuss how to generate *Z*. Similar as generating *X*, we generate *Z* by the following steps: (1) push all free nodes into a stack *S* in ascending order w.r.t. *X*, such that they are popped out from *S* in descending order, (2) pop a node *u* from *S*, assign *u* its topo-order t_u^Z, and push all of u's out-neighbors which are new free nodes into *S* in ascending order w.r.t. *X*, and (3) repeat the above two steps until *S* is empty, e.g. the reversed topo-order t_u^Z for each node *u* in Fig. 1(a) is denoted by the italic integers beside *u*.

Finally, we discuss how to find block nodes and generate the reversed topo-order *Y* w.r.t. the trivial graph G_1. Here, we do not need to first find block nodes to actually generate the trivial graph G_1. Instead, we do both when computing the reversed topo-order *Y*.

Definition 6. ***(Candidate Free Node (CFN))*** *When performing topological sorting to compute Y according to X, we say a node v is a candidate free node (CFN), if v it is not a free node and at least one of its in-neighbors u has been assigned a topo-order* t_u^Y.

For example, when computing *Y* based on *X* for *G* in Fig. 1(a), after visiting v_1, we know that v_2, v_3 and v_5 are new free nodes. They are pushed into stack *S* in their *X* topo-order. Thus the next processed node is v_5. After processing v_5, v_6 is a CFN according to Definition 6. Now, we can formally give out the definition of block node, as show below.

Definition 7. ***(Block Node (BN))*** *When computing Y according to X, let w be the largest unprocessed in-neighbor of v w.r.t. X, and u the currently being processed free node, we say node v is a block node (BN), if v it is a CFN and* $t_w^X < t_u^X$.

Intuitively, for currently being processed node *u*, when we say that *v* is a block node, it means that *u* cannot reach *v*, and if we assign *u* its t_u^Y, then we have that $t_u^X < t_v^X \wedge < t_v^Y$,

and based on which we cannot say that $u \rightarrow v$ holds. That is, *v* prevents us from checking all reachability relationship by two topo-orders. Therefore, to avoid the costly traversing operation when answering reachability queries, while at the same time reduce the cost of set intersection operation of existing 2-hop approaches, we computes 2-hop labels on BN only, and for other nodes, we use topo-labels to check the reachability relationship.

Now, we are ready to give out the index construction algorithm, the idea is to compute *Y* according *X*, during which we perform forward and backward *BFS* to compute 2-hop labels from each block node, as shown by Algorithm 4.

Algorithm 4: *T2H* $(G = (u, v))$

```
1  Q ← ∅; S ← ∅
2  push all free node into S in ascending X order
3  while (S ≠ ∅) do
4      v ← head(S)
5      if (Q does not contain BN ) then
6          v ← pop(S)
7          set v with its topo-order t_v^Y in Y
8          push new free nodes that are out-neighbors of v into S
9          push new CFNs into Q
10     else
11         recursively remove every BN and compute its 2-hop labels
```

The time cost of getting topo-label can be done in $O(|V|+|E|)$. The cost of computing 2-hop labels is $O(\ n_b \times |V|+|E|)$., where n_b is the number of block nodes. Therefore, the time complexity of the index construction algorithm is $O(\ n_b \times |V|+|E|)$.

5 Experiment

In our experiment, we make comparison with both existing *Label-Only* and *Label+G* algorithms, the *Label-Only* algorithms include *TF* [Cheng, Huang, Wu et al. (2013)], *PLL* [Yano, Akiba, Iwata et al. (2013)] and our algorithms. The *Label+G* algorithms include *GRAIL* [Yildirim, Chaoji and Zaki (2010); Yildirim, Chaoji and Zaki (2012)] (abbreviated as *GRL*), *FELINE* [Veloso, Cerf, Junior et al. (2014)] (abbreviated as *FL*) and IP^+ [Wei, Yu, Lu et al. (2014)]. The source codes of these existing algorithm are kindly provided by the authors. We set k=5 for *GRL* and IP^+ on all datasets. For IP^+, we set h=5 and μ=100. We implemented our algorithms using C++ and all source codes are compiled by G++6.2.0.

All experiments were run on a PC with Intel(R) Xeon(R) CPU E7-4809 v3 2.00 GHz CPU, 32 GB memory, and Ubuntu 16.10 Linux OS. For algorithms that run more than 24 h or exceed the memory limit (32 GB), we will show their results as "-" in the tables.

Table 1: Statistics of real datasets, where d is the average degree, $out_G^*(\cdot)$ is the average number of reachable nodes for nodes of G, t is the number of topological levels, n_b is the ratio of the number of block nodes over $|V|$

Dataset	$\|V\|$	$\|E\|$	d	$out_G^*(\cdot)$	t	n_b(%)
amaze	3,710	3,600	0.97	639	16	2.91
vchocyc	9,491	10,143	1.07	14	21	2.62
kegg	3,617	3,908	1.08	729	26	3.73
xmark	6,080	7,025	1.16	57	38	2.66
nasa	5,605	6,537	1.17	19	35	9.26
go	6,793	13,361	1.97	11	16	60.19
citeseer	10,720	44,258	4.13	27	36	39.00
pubmed	9,000	40,028	4.45	31	19	41.78
yago	6,642	42,392	6.38	10	13	9.47
arxiv	6,000	66,707	11.12	821	167	66.38
unip150m	25,037,600	25,037,598	1.00	1	10	0.00
10citeseerx	770,539	1,501,126	1.95	24	36	23.46
05citeseerx	1,457,057	3,002,252	2.06	33	36	24.52
citeseerx	6,540,401	15,011,260	2.30	15,456	59	24.93
dbpedia	3,365,623	7,989,191	2.37	83,660	146	7.69
govwild	8,022,880	23,652,610	2.95	536	12	5.63
go-unip	6,967,956	34,769,339	4.99	17	21	0.25
10go-unip	469,526	3,476,397	7.40	24	21	3.50
twitter	18,121,168	18,359,487	1.01	1,346,819	22	5.76
webuk	22,753,644	38,184,039	1.68	3,417,929	2,793	4.07

Datasets: Tab. 1 shows the statistics of 20 real datasets, where the first ten are small datasets ($|V|\leq$100,000), and the following 10 datasets are large ones ($|V|>$100,000). These datasets are usually used in the recent works [Cheng, Huang, Wu et al. (2013); Jin, Ruan, Dey et al. (2012); Jin and Wang (2013); Seufert, Anand, Bedathur et al. (2013); Su, Zhu, Wei et al. (2017); Veloso, Cerf, Junior et al. (2014); Wei, Yu, Lu et al. (2014); Yano, Akiba, Iwata et al. (2013); Yildirim, Chaoji and Zaki (2010); Yildirim, Chaoji and Zaki (2012); Zhu, Lin, Wang et al. (2014)], where we can find the detailed description of these datasets. Among these datasets, there are both sparse graphs (with average degree $d\leq 2$) and dense graphs ($d>2$). For topological levels (the 6th column), Tab. 1 contains both graphs with smaller topological levels, such as unip150m, for which the topological level is 10, and graphs with very large topological levels, such as webuk with topological level as large as

2,793. For the average number of reachable nodes (the 5th column), Tab. 1 also contains both graphs with nodes having few reachable nodes, such as unip150m, for which each node can reach only one node on average, and graphs with nodes having very large number of reachable nodes, such as twitter and webuk, where each node can reach 1,346,819 and 3,417,929 nodes on average, respectively.

Workloads: We test these reachability algorithms using both random and equal workloads. Here, each workload contains 1,000,000 queries. The random workload is generated by sampling node pairs with the same probability on the node set of each graph. For equal workload, the "equal" means that it contains the same number of reachable and unreachable queries, i.e. it has 50% reachable queries and 50% unreachable queries. The query time is the running time of testing all queries in a workload.

Note that for Tabs. 2- 5, each italic number denotes the best result in the row.

5.1 Query time

From Tab. 2 we have the following observations for random workload: (1) our *T2H-O* algorithm works best on 19 out of 20 datasets, and on the remaining one dataset, *T2H-O* is also approaching the best result. The reason lies in that for reachable queries, it does not suffer from the costly traversing operation on the given *DAG* as *Label+G* algorithms do, for unreachable queries, it can quickly answer most unreachable ones using topo-labels, and does not suffer from performing costly set intersection operation on 2-hop labels for most queries compared with *Label-Only* algorithms; (2) even though *T2H-O* works better than *T2H*-1 and *T2H*-2, both the latter are very efficient compared with existing algorithms, and each one can beat the other on some datasets. This difference on query performance lies in that our method consists of comparing topo-labels and 2-hop labels, and *T2H*-1 and *T2H*-2 use different order on performing the two operations; (3) no one can beat all others on all datasets, and for existing algorithms, *TF* works better than others on most datasets, but it cannot work successfully on webuk dataset.

From Tab. 3 we know that for equal workload: (1) our *T2H*-O algorithm works best on 16 datasets, and *T2H*-1 works best on three datasets, together our approaches work best on 19 datasets, and are usually much better than existing *Label-Only* and *Label+G* algorithms; (2) for existing algorithms, *Label-Only* algorithms usually work better than *Label+G* algorithms, due to that for equal workload, the number of reachable queries is 50%, which can be answered more efficient using *Label-Only* algorithms by avoiding costly traversing operations on the given *DAG*.

From both the two tables we know that simply combining existing *Label-Only* and *Label+G* approaches together cannot improve the overall performance. For example, *FL+PLL* is the combination of *FL* and *PLL*, which is a *Label-Only* algorithm. We can see that in most cases, the performance of *FL+PLL* is in between the other two. Even though our approach is also a *Label-Only* algorithm, our labeling scheme is not a simple combination of *Label+G* label and *Label-Only* label, our topo-label significantly reduces the 2-hop label size to accelerate the query performance.

Table 2: Comparison of query time on random workload (ms)

Dataset	*GRL*	*FR*	*IP*$^+$	*FL*	*TF*	*PLL*	*FL+PLL*	*T2H-1*	*T2H-2*	*T2H-O*
amaze	95	37	28	48	16	37	70	20	22	***13***
vchocyc	42	***10***	12	19	20	32	63	12	19	11
kegg	104	39	33	49	20	38	66	21	23	***17***
xmark	109	34	56	291	39	53	83	24	25	***20***
nasa	74	36	47	79	35	50	79	22	26	***17***
go	152	57	61	249	42	64	98	61	65	***35***
citeseer	297	104	82	191	61	74	101	61	60	***35***
pubmed	279	89	69	139	54	71	100	65	62	***39***
yago	87	60	20	44	24	63	90	22	24	***14***
arxiv	3,264	361	1,387	1,607	396	103	131	108	111	***57***
unip150m	97	20	57	35	21	38	74	16	22	***15***
10citeseerx	361	62	58	146	57	95	133	82	102	***54***
05citeseerx	415	68	59	195	59	103	158	94	97	***53***
citeseerx	663	45	104	310	91	96	130	95	110	***44***
dbpedia	451	318	168	118	147	129	226	85	97	***70***
govwild	757	1,488	143	244	402	163	248	126	106	***82***
go-unip	213	147	57	75	53	94	147	58	57	***40***
10go-unip	224	132	45	80	49	95	130	83	59	***43***
twitter	275	651	111	120	151	119	217	142	101	***93***
webuk	721	871	451	380	-	184	233	117	114	***105***

Table 3: Comparison of query time on equal workload (ms)

Dataset	*GRL*	*FR*	*IP*$^+$	*FL*	*TF*	*PLL*	*FL+PLL*	*T2H-1*	*T2H-2*	*T2H-O*
amaze	167	29	53	49	15	24	48	14	16	***12***
vchocyc	148	21	67	49	30	26	45	14	16	***13***
kegg	179	34	84	52	18	41	49	16	16	***12***
xmark	160	45	181	73	71	44	58	24	23	***19***
nasa	231	50	206	149	50	56	60	27	27	***21***
go	290	89	244	308	57	75	83	68	65	***51***
citeseer	598	222	239	280	80	67	94	67	67	***53***
pubmed	620	218	188	252	105	98	119	84	88	***73***
yago	339	117	133	121	53	45	68	26	29	***21***
arxiv	2,328	393	1,145	1,306	434	78	102	89	96	***58***

unip150m	1,222	271	1,248	409	326	159	181	70	64	***57***
10citeseerx	1,002	554	320	252	296	184	233	***156***	157	164
05citeseerx	1,181	735	365	314	385	200	284	198	204	***188***
citeseerx	5,301	891	852	1,952	1,257	174	242	164	167	***173***
dbpedia	1,224	298	355	364	2,938	101	170	88	94	***80***
govwild	898	1,600	393	171	1,217	195	290	152	151	***147***
go-unip	2,523	1,490	478	568	8,738	195	209	***92***	97	95
10go-unip	1,112	606	392	471	375	171	191	***75***	78	79
twitter	600	824	394	267	197	***90***	179	136	95	134
webuk	2,585	974	4,956	2,328	-	179	218	95	136	***91***

Table 4: Comparison of index construction time (ms)

Dataset	*FR*	*IP*$^+$	*FL*	*TF*	*PLL*	*FL+PLL*	*T2H-1*	*T2H-2*	*T2H-O*
amaze	3.88	2.54	***1.31***	4.23	2.00	2.54	1.33	1.33	1.33
vchocyc	9.05	7.03	2.91	36.85	4.80	5.82	***2.21***	2.21	2.21
kegg	2.64	1.55	1.50	3.37	2.20	2.83	***1.30***	1.30	1.44
xmark	4.38	3.21	2.20	18.92	5.35	5.72	***1.75***	1.75	2.85
nasa	4.59	2.97	***2.15***	17.76	5.64	5.82	3.47	3.47	3.86
go	10.45	5.43	***3.55***	30.54	11.13	11.53	10.61	10.61	11.34
citeseer	32.16	18.72	***7.84***	120.29	24.12	25.08	15.75	15.75	16.91
pubmed	40.59	13.21	***5.82***	93.27	24.52	22.60	23.07	23.07	21.96
yago	22.99	8.82	***4.50***	52.58	7.00	11.86	10.92	10.92	8.95
arxiv	28.30	33.70	***6.26***	9,225.31	41.84	39.04	44.60	44.60	44.72
unip150m	40,089	35,932	23,452	58,441	14,138	36,604	***6,146***	6,146	6,665
10citeseerx	1,641	805	***595***	4,144	1,313	2,070	1,215	1,215	1,199
05citeseerx	3,360	1,708	***1,194***	9,551	3,369	4,945	3,333	3,333	3,253
citeseerx	18,467	9,759	***6,161***	100,805	10,677	18,022	11,018	11,018	11,135
dbpedia	8,229	4,778	3,471	12,629	3,681	7,674	***3,312***	3,312	3,437
govwild	33,710	10,762	***7,584***	145,695	11,759	22,304	11,137	11,137	11,162
go-unip	39,820	12,851	***6,849***	64,475	15,477	24,320	17,179	17,179	17,345
10go-unip	2,360	874	***517***	5,813	1,268	1,998	1,133	1,133	1,093
twitter	26,714	22,460	11,050	12,875	7,707	18,170	***5,492***	5,492	5,793
webuk	34,304	33,114	14,802	-	17,493	32,972	***14,788***	14,788	15,619

Table 5: Comparison of index sizes (MB)

Dataset	*GRL*	*FR*	*IP*$^+$	*FL*	*TF*	*PLL*	*FL+PLL*	*T2H-1*	*T2H-2*	*T2H-O*
amaze	0.21	0.10	0.10	0.07	***0.02***	0.04	0.09	0.04	0.04	0.06
vchocyc	0.54	0.20	0.29	0.18	0.31	0.12	0.23	***0.08***	0.08	0.12
kegg	0.21	0.10	0.11	0.07	***0.02***	0.05	0.09	0.04	0.04	0.06
xmark	0.35	0.16	0.21	0.12	0.12	0.12	0.19	***0.09***	0.09	0.11
nasa	0.32	0.15	0.17	0.11	0.12	0.11	0.17	***0.08***	0.08	0.10
go	0.39	0.23	0.22	***0.13***	0.18	0.19	0.27	0.18	0.18	0.22
citeseer	0.61	0.39	0.32	***0.20***	0.83	0.28	0.41	0.29	0.29	0.35
pubmed	0.51	0.33	0.23	***0.17***	0.80	0.27	0.37	0.28	0.28	0.33
yago	0.38	0.24	0.18	***0.13***	0.23	0.19	0.27	0.20	0.20	0.23
arxiv	0.34	0.22	0.28	***0.11***	14.66	0.35	0.42	0.36	0.36	0.40
unip150m	1,433	716	440	478	***132***	318	604	191	191	287
10citeseerx	44	21	21	***15***	45	18	26	16	16	20
05citeseerx	83	40	38	***28***	115	36	52	33	33	40
citeseerx	374	182	151	125	1,523	116	191	***108***	108	139
dbpedia	193	122	94	64	52	53	91	***48***	48	62
govwild	459	291	194	***153***	3,123	188	280	166	166	198
go-unip	399	253	185	***133***	431	251	331	251	251	278
10go-unip	27	17	13	***9***	44	21	27	21	21	23
twitter	1,037	375	745	346	***70***	202	410	206	206	279
webuk	1,302	632	816	434	-	357	618	***329***	329	419

5.2 Index construction time and index size

For index construction time, we have the following observations according to Tab. 4: *FL*, *T2H*-1 and *T2H*-2 usually work better than others, the reasons lie in that on one hand, *FL* is a *Label+G* algorithm, it has the lowest time complexity, on the other hand, both *T2H*-1 and *T2H*-2 has linear time complexity to get the topo-labels, which is used to help find block nodes. We can see from the last column of Tab. 1 that the ratio of block nodes is usually much less than the number of nodes in the given graph. Based on these block nodes, we can quickly get the 2-hop labels. Even though other existing *Label+G* algorithms have smaller time complexity on index construction, they may need more time due to traversing several times on the given *DAG*. For existing algorithm, *TF* usually needs more time to get smallest index size and query time on several datasets, and it cannot work successfully on the webuk dataset. We can see from Tab. 5 that even though *Label+G* algorithms have linear index size, our approaches can achieve the best result due to reduced number of block nodes to generate 2-hop labels. And both *T2H*-1 and *T2H*-2 achieves best results on 6 datasets.

By combining the two tables together, we can see that for index construction time and index size, no one can beat all others for all datasets. And all our approaches can complete the index construction in reasonable time with acceptable index size. Based on this result, from Tab. 2 and Tab. 3 we can see that our approaches make significant improvement on query performance.

6 Conclusions

In this paper, we propose a novel *Label-Only* labeling scheme to accelerate reachability query processing. The idea comes from the fact that index construction is a *one-time* activity, while query testing is a fundamental *online* graph operation. Considering that existing *Label+G* algorithms are inefficient for reachable queries and existing *Label-Only* algorithms are inefficient for unreachable queries, we propose to use topo-labels to reduce the size of 2-hop labels to construct a new *Label-Only* labels for fast query answering. Our experimental results show that our approaches can answer both reachable and unreachable queries more efficiently on 19 out of 20 real datasets.

Acknowledgement: This work was partly supported by National Key R&D Program of China, Grant No. 2017YFB0309800, the grants from the Natural Science Foundation of China (No. 61472339, No. 61303040, No. 61572421, No. 61272124), and Shanghai Alliance Program(LM201552), and Shanghai University of Engineering and Technology School-Enterprise cooperation projects(15)(DZ-025).

References

Agrawal, R.; Borgida, A.; Jagadish, H. V. (1989): Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Conference*, pp. 253-262.

Chen, L.; Gupta, A.; Kurul, M. E. (2005): Stack-based algorithms for pattern matching on dags. *International Conference on Very Large Data Bases*, pp. 493-504.

Chen, Y.; Chen, Y. (2008): An efficient algorithm for answering graph reachability queries. *IEEE International Conference on Data Engineering*, pp. 893-902.

Chen, Y.; Chen, Y. (2011): Decomposing dags into spanning trees: A new way to compress transitive closures. *IEEE International Conference on Data Engineering*, pp. 1007-1018.

Cheng, J.; Huang, S.; Wu, H.; Fu, A. (2013): Tf-label: A topological-folding labeling scheme for reachability querying in a large graph. *ACM Sigmod International Conference on Management of Data*, pp. 193-204.

Cheng, J.; Yu, J.; Lin, X.; Wang, H.; Yu, P. (2006): Fast computation of reachability labeling for large graphs. *International Conference on Advances in Database Technology*, pp. 961-979.

Cheng, J.; Yu, J.; Lin, X.; Wang, H.; Yu, P. (2008): Fast computing reachability labelings for large graphs with high compression rate. *International Conference on Extending Database Technology*, pp. 193-204.

Cohen, E.; Halperin, E.; Kaplan, H.; Zwick, U. (2002): Reachability and distance queries via 2-hop labels. *Symposium on Discrete Algorithms*, pp. 937-946.

Fan, W.; Li, J.; Wang, X.; Wu, Y. (2012): Query preserving graph compression. *ACM Sigmod International Conference on Management of Data*, pp. 157-168.

Jagadish, H. V. (1990): A compression technique to materialize transitive closure. *ACM Transactions on Database Systems*, vol. 15, no. 4, pp. 558-598.

Jin, R.; Ruan, N.; Dey, S.; Yu, J. (2012): SCARAB: Scaling reachability computation on large graphs. *ACM Sigmod International Conference on Management of Data*, pp. 169-180.

Jin, R.; Ruan, N.; Xiang, Y.; Wang, H. (2011): Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Transactions on Database Systems*, vol. 36, no. 1, pp. 1-44.

Jin, R.; Wang, G. (2013): Simple, fast, and scalable reachability oracle. *Proceedings of the VVLDB Endowment*, vol. 6, no. 14, pp. 1978-1989.

Jin, R.; Xiang, Y.; Ruan, N.; Fuhry, D. (2009): 3-hop: A high-compression indexing scheme for reachability query. *Acm Sigmod International Conference on Management of Data*, pp. 813-826.

Jin, R.; Xiang, Y.; Ruan, N.; Wang, H. (2008): Efficiently answering reachability queries on very large directed graphs. *ACM Sigmod International Conference on Management of Data*, pp. 595-608.

Kornaropoulos, E. M.; Tollis, I. G. (2011): Weak dominance drawings and linear extension diameter. *Data Structures and Algorithms*.

Li, F.; Li, G. (2010): Interval-based uncertain multi-objective optimization design of vehicle crashworthiness. *Computers, Materials & Continua*, vol. 15, no. 3, pp. 199-219.

Seufert, S.; Anand, A.; Bedathur, S. J.; Weikum, G. (2013): FERRARI: Flexible and efficient reachability range assignment for graph indexing. *IEEE International Conference on Data Engineering*, pp. 1009-1020.

Simon, K. (1988): An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, vol. 58, no. 1, pp. 325-346.

Su, J.; Zhu, Q.; Wei, H.; Yu, J. (2017): Reachability querying: Can it be even faster? *IEEE Transactions on Knowledge & Data Engineer*, vol. 29, no. 3, pp. 683-697.

Tarjan, R. E. (1972): Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146-160.

Trißl, S.; Leser, U. (2007): Fast and practical indexing and querying of very large graphs. *ACM Sigmod International Conference on Management of Data*, pp. 845-856.

van Schaik, S. J.; de Moor, O. (2011): A memory efficient reachability data structure through bit vector compression. *ACM Sigmod International Conference on Management of Data*, pp. 913-924.

Veloso, R. R.; Cerf, L.; Junior, W. M.; Zaki, M. J. (2014): Reachability queries in very large graphs: A fast refined online search approach. *International Conference on Extending Database Technology*, pp. 511-522.

Wang, H.; He, H.; Yang, J.; Yu, P.; Yu, J. (2006): Dual labeling: Answering graph reachability queries in constant time. *International Conference on Data Engineering*, pp. 75-75.

Wei, H.; Yu, J.; Lu, C.; Jin, R. (2014): Reachability querying: An independent

permutation labeling approach. *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1191-1202.

Wu, C.; Zapevalova, E.; Chen, Y.; Li, F. (2018): Time optimization of multiple knowledge transfers in the big data environment. *Computers, Materials & Continua*, vol. 54, no. 3, pp. 269-285.

Yano, Y.; Akiba, T.; Iwata, Y.; Yoshida, Y. (2013): Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. *ACM International Conference on Conference on Information & knowledge Management*, pp. 1601-1606.

Yildirim, H.; Chaoji, V.; Zaki, M. J. (2010): GRAIL: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, vol. 3, no. 1, pp. 276-284.

Yildirim, H.; Chaoji, V.; Zaki, M. J. (2012): GRAIL: A scalable index for reachability queries in very large graphs. *VLDB Journal*, vol. 21, no. 4, pp. 509-534.

Yu, J. X.; Cheng, J. (2010): Graph reachability queries: A survey. *Managing and Mining Graph Data*, pp. 181-215.

Zhang, Z.; Yu, J.; Qin, L.; Zhu, Q.; Zhou, X. (2012): I/O cost minimization: Reachability queries processing over massive graphs. *International Conference on Extending Database Technology*, pp. 468-479.

Zhou, J.; Zhou, S.; Yu, J.; Wei, H.; Chen, Z. et al. (2017): *DAG* reduction: Fast answering reachability queries. *ACM Sigmod International Conference on Management of Data*, pp. 375-390.

Zhu, A.; Lin, W.; Wang, S.; Xiao, X. (2014): Reachability queries on large dynamic graphs: A total order approach. *ACM Sigmod International Conference on Management of Data*, pp. 1323-1334.