ESTI: Efficient k-Hop Reachability Querying over Large General Directed Graphs

Yuzheng Cai and Weiguo Zheng(✉)

Fudan University, Shanghai, China
{yzcai17,zhengweiguo}@fudan.edu.cn

Abstract. As a fundamental task in graph data mining, answering k-hop reachability queries is useful in many applications such as analysis of social networks and biological networks. Most of the existing methods for processing such queries can only deal with directed acyclic graphs (DAGs). However, cycles are ubiquitous in lots of real-world graphs. Furthermore, they may require unacceptable indexing space or expensive online search time when the input graph becomes very large. In order to solve k-hop reachability queries for large general directed graphs, we propose a practical and efficient method named *ESTI* (Extended Spanning Tree Index). It constructs an extended spanning tree in the offline phase and speeds up online querying based on three carefully designed pruning rules over the built index. Extensive experiments show that *ESTI* significantly outperforms the state-of-art in online querying, while ensuring a linear index size and stable index construction time.

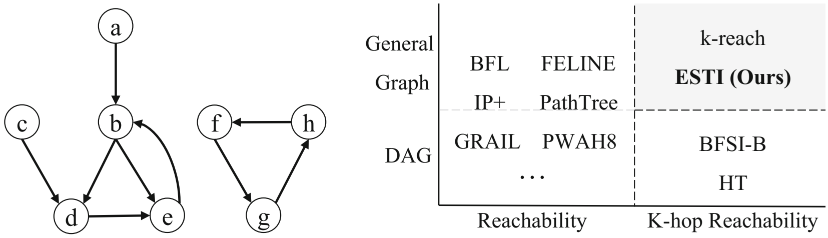
Keywords: k-hop reachability queries · General directed graphs · Extended spanning tree

1 Introduction

Graph is a flexible data structure representing connections and relations among entities and concepts, which has been widely used in real world, including XML documents, cyber-physical systems, social networks, biological networks and traffic networks [1–3,9,12]. Nowadays, the size of graphs such as knowledge graphs and social networks is growing rapidly, which may contain billions of vertices and edges. k-hop reachability query in a directed graph is first discussed by Cheng et al. [1]. It asks whether a vertex u can reach v within k hops, i.e., whether there exists a directed path from u to v in the given directed graph and the path is not longer than k. Note that the input general directed graph is not necessary to be connected. Take the graph G in Fig. 1(a) as an example, vertex a can reach vertex e within 2 hops, but a cannot reach vertex d within 1 hop.

Efficiently answering k-hop reachability queries is helpful in many analytical tasks such as wireless networks, social networks and cyber-physical systems

C. S. Jensen et al. (Eds.): DASFAA 2021 Workshops, LNCS 12680, pp. 71–89, 2021.
https://doi.org/10.1007/978-3-030-73216-5_6



(a) Input general directed graph G (b) Categories of existing works

Fig. 1. Illustration of input graph and existing works

[1,2,12]. Several methods for k-hop reachability has been proposed, providing different techniques to solve this kind of queries. However, existing methods suffer some shortcomings, which make them not practical or general enough to answer k-hop reachability queries efficiently. To the best of our knowledge, *k-reach* [1,2] is the only method aiming at dealing with k-hop reachability queries for general directed graph, which builds an index based on vertex cover of the graph. It is infeasible to build such an index for large graphs due to the huge space cost. Thus a partial coverage is employed in [2]. However, partial coverage technique is also not practical enough since most queries may fall into the worst case, which requires online BFS search.

A bunch of methods have been proposed to solve k-hop reachability queries in DAGs. *BFSI-B* [12] builds a compound index, containing both FELINE index [10] and breadth-first search index (BFSI). *HT* [3] works on 2-hop cover index, which selects some high-degree nodes in the DAG as hop nodes. Experiments have shown that both of them are practical and efficient to answer k-hop reachability queries. However, they are developed only for dealing with DAGs, which are not general enough since most graphs in real applications may have cycles, such as social networks and knowledge graphs.

A simple version of k-hop rechability query is reachability query. Given a graph G, reachability query can be taken as a specific case of k-hop reachability queries, since they are actually equivalent when $k \geq \lambda(G)$, where $\lambda(G)$ represents the length of the longest simple path in graph G. Note that for a general directed graph, we can obtain the corresponding DAG by condensing each strongly connected component (SCC) as a supernode, such that the reachability information in original graph can be completely preserved in the constructed DAG. Although lots of methods have been proposed to handle reachability queries [4,6,8,10,11,13], they cannot be directly used for k-hop reachability queries since more information such as distance is missing in the transformation above.

We categorize the methods related to k-hop reachability queries [1–4,6,8,10–13], as shown in Fig. 1(b). Clearly, right-top corner represents k-hop reachability in general directed graphs, which is the most general one. As discussed above, *k-reach*, the only existing method in this research area, is not practical enough to handle very large graphs. Hence, we develop a practical method named *ESTI* to answer k-hop reachability queries efficiently.

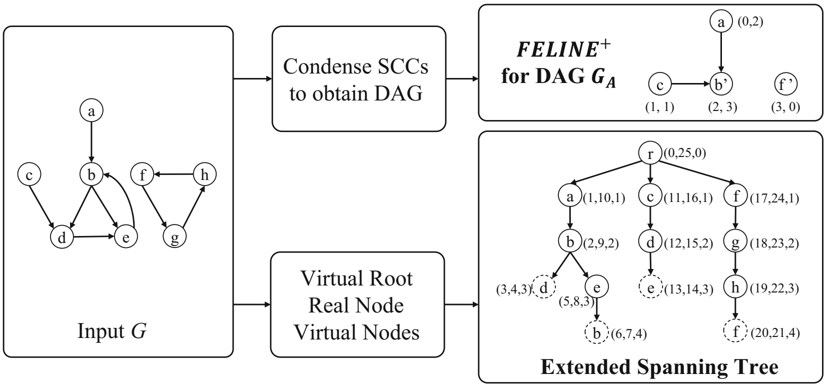


Fig. 2. Overview of Extended Spanning Tree Index (ESTI)

Our proposed approach, *ESTI*, follows the offline-and-online paradigm. It builds an index for a given graph in the offline phase, and answers arbitrary k-hop reachability queries in the online phase. In offline indexing process, both *FELINE*$^+$ index and Extended Spanning Tree Index (ESTI) are constructed. We introduce the concept of *Real Node* and *Virtual Node* to build the extended spanning tree with both BFS and DFS. As for online querying, the offline index helps to answer k-hop reachability queries efficiently, and three pruning strategies are devised to further speed up query process.

Paper Organiztion. This paper is organized as follows. Section 3 explains the details of *ESTI* offline index, followed by the querying process as discussed in Sect. 4. Section 5 shows the results of experiments comparing *ESTI* with other k-hop reachability methods. In Sect. 6, some exciting works related to k-hop reachability queries are presented. Finally, Sect. 7 concludes the paper.

2 Problem Definition and Overview

2.1 Problem Definition

In this paper, the input general directed unweighted graph is represented as $G = (V, E)$, where V denotes the set of vertices and E denotes the set of edges. $|V|$ and $|E|$ denote the number of vertices and edges in G, respectively. For any two vertices $u, v \in V$ and $u \neq v$, we say that u can reach v within k hops if there exists a directed path from u to v in G which is not longer than k. Let $u \xrightarrow{?k} v$ represent a query asking whether u can reach v within k hops in G.

2.2 Overview

ESTI follows the offline-and-online paradigm, and Fig. 2 presents the overview of our offline index structure. For better understanding, we briefly introduce our basic ideas and techniques for answering arbitrary k-hop reachability queries.

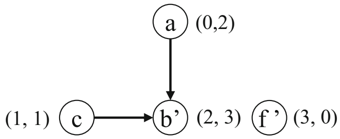


Fig. 3. FELINE index (X, Y) in DAG G_A

FELINE$^+$ Index. Since reachablity is the neccessary condition for k-hop reachability, *FELINE* index [10] including two topological orders can be utilized to efficiently filter unreachable queries. The time cost of generating index in offline phase is $O(|V|log|V| + |E|)$. In Sect. 3.1, we present an optimization named *FELINE*$^+$ to speed up index generation, which costs $O(|V|log(Deg_m^{(out)}) + |E|)$ time, where $Deg_m^{(out)}$ is the maximum outgoing degree of a vertex.

Extended Spanning Tree Index. In order to preserve as much information as possible for answering queries, we introduce *Virtual Root*, *Real Nodes* and *Virtual Nodes* to constuct an extended spanning tree from the input graph G in Sect. 3.2. Also, pre- and postorders and global level are assigned to nodes in the tree, which helps to efficiently answer k-hop queries online.

Online Querying. Given arbitrary query $u \xrightarrow{?k} v$, the constructed index is utilized to directly return the correct answer or prune search space. In Sect. 4.2, three pruning strategies are developed to further accelerate online querying.

3 Offline Indexing

3.1 FELINE$^+$ Index

If u cannot reach v in G, the answer of query $u \xrightarrow{?k} v$ is apparently $False$. To efficiently filter those unreachable queries in online querying phase, *FELINE* [10] condenses all strongly connected components (SCCs) in the given general directed graph G to obtain a DAG G_A, and two topological orders X and Y are generated for each vertex in G_A. Let X_v and Y_v denote the first and second topological order of a vertex v, respectively. If u can reach v, both $X_u < X_v$ and $Y_u < Y_v$ hold. Hence, for a query $u \xrightarrow{?k} v$, we can directly return the answer $False$ if $X_u > X_v$ or $Y_u > Y_v$ in FELINE index.

In *FELINE* [10], X is calculated by a topological ordering algorithm, and Y coordinate is assigned by applying a heuristic decision. When assigning Y coordinate, let R be a set storing all roots in current DAG. FELINE iteratively runs the following procedures until all vertices in G_A have Y coordinates.

Step 1. Choose the root r from R with largest X_r, assign r a coordinate Y_r;

Step 2. Remove all of r's outgoing edges. and some of its children may have no ancestors and become new roots. Thus, R should be updated.

Algorithm 1. FELINE$^+$ Index Construction

Input: DAG G_A;
Output: Two topological orders X and Y;

$X \leftarrow$ a Topological Order of G_A
$R \leftarrow$ all the roots in G_A sorted w.r.t descending X value
while R is not empty **do**
  pop the first element r from R and assign Y_r
  $R_{tmp} \leftarrow [\,]$
  for each outgoing neighbor t of r **do**
    remove edge (r, t)
    if t has no incoming neighbor **then**
      $R_{tmp} \leftarrow R_{tmp} \cup \{t\}$
  sort R_{tmp} according to descending X value
  insert all elements of R_{tmp} in the front of R, while preserving the order
return X, Y;

Example 1. By condensing all SCCs of graph G in Fig. 1(a), its corresponding DAG G_A is shown in Fig. 3. After assigning X, we start to assign Y and $R = \{a, c, f'\}$. Since $X_{f'} = 3$ is the largest one, $Y_{f'}$ is assigned to be 0, and next we assign $Y_c = 1$ and $Y_a = 2$. When all edges connecting with b' are removed, we update $R = R \cup \{b'\}$ to continue assigning Y coordinate to b'. As for online querying, for instance, vertex a cannot reach vertex c since $Y_a > Y_c$ in Fig. 3.

The time cost of condensing SCCs and generating X coordinate is $O(|V| + |E|)$. Note that FELINE utilizes a max-heap to store all the current roots R, in which those roots are sorted in the descending order according to X. It takes $O(1)$ to pop a root r from the max-heap in *Step 1*, and each vertex in G_A can only be inserted into R once which costs $O(log|V|)$ time. Hence, the overall time cost of building index construction for FELINE is $O(|V|log|V| + |E|)$.

In this paper, we propose an novel technique to accelerate Y coordinate generation, utilizing a simple array to store all the current roots R instead of a max-heap. Firstly, R is initialized by putting all the roots in original G_A, making sure they are sorted in descending order w.r.t. X value. Then the following two steps are processed iteratively until all the vertices have Y coordinate.

Step 1. Pop the first element r from the array R and assign its Y coordinate.

Step 2. Remove all of r's outgoing edges. Sort those new roots w.r.t descending X value, then insert them in the front of array R, while preserving the order.

Theorem 1. *The order of elements in array R is always the same as the descending order of their X value.*

Proof. At first, array R is initialized with all roots in original G_A, which are sorted in the descending order w.r.t. X value. Assume that elements in array R are in the descending order of X value. When we pop the first element r from array R to assign Y_r, $X_r \geq X_v$ holds for any vertex v in array R. After removing r's outgoing edges, some of its children w may become new roots and $X_w > X_r$

must hold. Thus, every w has larger X than any v in array R. After sorting those new roots w in descending X value and inserting them in the front of array R, all the vertices in array R are still in their descending X order. □

The enhanced algorithm, denoted by FELINE$^+$, for accelerating FELINE is shown in Algorithm 1. When generating Y coordinate, according to Theorem 1, the first element r of array R always has the largest X_r value in R, and it actually constructs the same index as FELINE. Note that to make sure the initial roots in arrary R are in descending order w.r.t. X value, we only need to reverse the initial root queue of X coordinate generation process, because their X values are generated following the order of it. Hence, the initialization time of array R is linear to the number of roots in original G_A. When processing each current root r, sorting the new roots takes $O(|w|log|w|)$, where $|w|$ is the number of new roots obtained by removing r's outgoing edges. Since each vertex in G_A can be a new root only once, the time cost of generating Y coordinate is $O(|V|log(Deg_m^{(out)}) + |E|)$, where $Deg_m^{(out)}$ is the max number of outgoing neighbors of a vertex and $|w| \leq Deg_m^{(out)}$ always holds.

The total time cost of building index for FELINE$^+$ is $O(|V|log(Deg_m^{(out)}) + |E|)$. Theoretically, since $Deg_m^{(out)}$ is much smaller than $|V|$ in many graphs, our approach is faster than the original FELINE whose time cost is $O(|V|log|V| + |E|)$. Experiments confirm that the proposed optimization technique significantly accelerates the index construction for FELINE, as shown in Sect. 5.2.

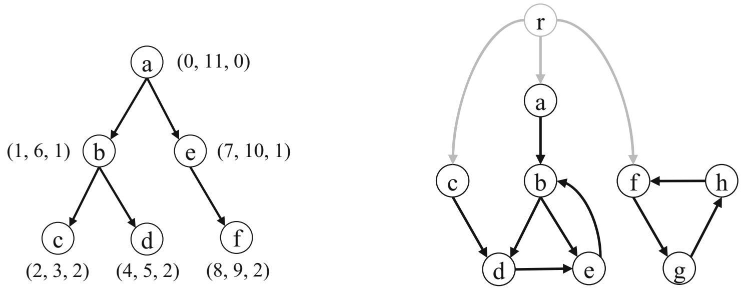
3.2 Extended Spanning Tree Index for General Directed Graph

Preliminary. We first briefly introduce pre- and postorder index and global level for a tree, which have been used in *GRIPP* [9] and *BFSI-B* [12]. Note that *BFSI-B* applies min-post strategy, which actually has the same effect as pre- and postorders. For any vertex v in the tree, pre_v and $post_v$ represent the pre- and postorder index of v, respectively. And $level_v$ is the global level of v, i.e., the distance from the tree root to v. pre_v and $post_v$ are generated during the DFS traversal, while $level_v$ is generated during the BFS traversal.

Example 2. Figure 4(a) illustrates the three labels. Following the visiting order in DFS, we start from root a and set pre_a to 0. Then we visit b and c and set pre_b and pre_c to 1 and 2, respectively. After returning from c, we set $post_c$ to 3. The process proceeds until all nodes have been visited. Each node is assigned both pre- and postorder index following the DFS. As for *level* index, $level_a$ is set to be 0 and we can assign *level* to other vertices following the BFS.

We say that $(pre_v, post_v) \subset (pre_u, post_u)$ iff $pre_v \geq pre_u \wedge post_v \leq post_u$. Based on the constructed index $(pre_v, post_v, level_v)$ discussed above, Theorem 2 holds in the tree, and query $u \xrightarrow{?k} v$ can be efficiently answered. For example, in Fig. 4(a) a can reach d in 2 hops, since $(4, 5) \subset (0, 11)$ and $level_d - level_a = 2$.

Theorem 2. *Given two vertices u and v in tree T, u can reach v within k hops if* $(pre_v, post_v) \subset (pre_u, post_u) \wedge level_v - level_u \in (0, k]$.



(a) Example of $(pre_v, post_v, level_v)$ index (b) New graph G' with *Virtual Root*

Fig. 4. Illustration of $(pre_v, post_v, level_v)$ index and *Virtual Root*

Proof. According to the process of pre- and postorder generation, $(pre_v, post_v) \subset (pre_u, post_u)$ indicates that v is in the subtree whose root is u. $level_v - level_u \in (0, k]$ implies that there is a path from u to v which is not longer than k. □

Clearly, if the input graph is a tree, both time and space cost for building the index are $O(|V| + |E|)$ and it only takes $O(1)$ for online query. However, when the input general directed graph G is not a tree, to make it practical and efficient enough for answering k-hop reachability queries, we introduce *Virtual Root*, *Real Node* and *Virtual Node* to transform G into an Extended Spanning Tree (EST). Note that our method is quite different from existing approaches like *GRIPP* [9] and *BFSI-B* [12]. *GRIPP* solves reachability queries while ignores distance information which is necessary for answering k-hop reachability queries, and *BFSI-B* is developed for only dealing with DAGs. However, most graphs in real life have cycles and *BFSI-B* cannot directly work on these graphs.

Virtual Root. Since the given graph G may not be connected, e.g., the graph in Fig. 1(a), we add a virtual root V_R to make sure that it can reach all vertices in G. We first add an edge from V_R to all the vertices which have no predecessors, then explore from V_R to mark all of its descendants visited. The second step is to randomly select an unvisited vertex v, and add an edge from V_R to v while all of v's descendants are marked visited. We repeat the second step until all vertices have been visited. Take graph G in Fig. 1(a) as an example. After adding a virtual root for it, we obtain a new graph G' in Fig. 4(b).

Real and Virtual Nodes. When starting BFS from virtual root V_R, we may encounter endless loop since there may exist cycles in G', or some visited vertices since they have multiple incoming edges. To solve this problem, we introduce *Real Nodes* and *Virtual Nodes*. In BFS process, if vertex v has never been visited, it will be added to the spanning tree as a *Real Node* and we will continue to visit its successors. If vertex v has been visited, it will be added to the tree as a *Virtual Node* while its successors will not be explored again. Following the

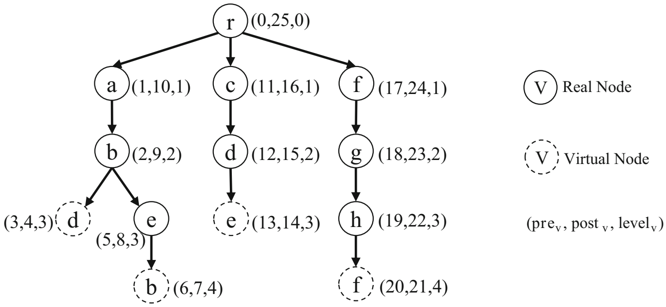


Fig. 5. Extended spanning tree of G and $(pre_v, post_v, level_v)$ index

above definition of *Real Node* and *Virtual Node*, we can construct an extended spanning tree from graph G', as shown in Example 3. Also, Theorem 3 holds.

Example 3. In Fig. 4(b), we start BFS from r and add real nodes for r, a, c, f, b, d and g. When exploring from b to visit d, we create a virtual node for d since it has been visited before. Figure 5 is the extended spanning tree of G'.

Theorem 3. *In extended spanning tree, each vertex v in graph G' must have exactly one real node. The total number of real and virtual nodes in this tree is equal to the number of edges in G' plus 1.*

Proof. Since virtual root V_R can reach all vertices in G' and we start BFS from V_R to construct the extended spanning tree, a real node is created for each vertex v in G' when it is visited for the first time. When v is visited again, we only create a virtual node for it. Hence, each v in G' must have exactly one real node.

At the beginning of BFS, we create a real node for virtual root V_R. As for the other vertices v in G', a real node or virtual node will be created for v only when we explore from its incoming neighbor. Hence, the number of real and virtual nodes in this tree is equal to the number of edges in G' plus one, where the additional one is the real node representing virtual root V_R. □

Index Generation. Recall that in a tree, the index of vertex v consists of pre_v, $post_v$ and $level_v$. When constructing the extended spanning tree from graph G', we have already run BFS in the tree, and *level* index will also be generated for all the nodes. Next, we explore the whole tree by DFS and assign each vertex with pre- and postorder index. Take the graph G' in Fig. 4(b) as an example. The index of its extended spanning tree is shown in Fig. 5. After assigning the above index, Theorem 4 holds for all the real and virtual nodes in the tree.

Theorem 4. *If vertex v of G' has virtual nodes in the extended spanning tree, denote its unique real node as v'_r. For any virtual node v'_i of v, $level_{v'_i} \geq level_{v'_r}$.*

Proof. When construting the extended spanning tree by BFS, all the virtual nodes of v are created after its real node is created. Hence, based on the exploration order of BFS, $level_{v'_i} \geq level_{v'_r}$. □

Let $|V'|$ and $|E'|$ denote the number of vertices and edges in G', respectively. When generating G' from original graph G, we add a virtual root V_R and at most $|V|$ edges to connect vertices in G. Thus, $O(|V'| + |E'|) = O(|V| + |E|)$.

The time and space comlexity of adding a virtual root is $O(|V| + |E|)$, since each vertex and edge is visited once. When constructing the extended spanning tree, each edge in G' is visited once since we explore from vertex v only when its unique real node is created. According to Theorem 3, it takes both time and space cost $O(|V| + |E|)$ to create all real and virtual nodes. And both BFS and DFS also take the time and space cost $O(|V|+|E|)$. Hence, the overall time and space cost for constructing the extended spanning tree and the three labels are $O(|V| + |E|)$, which indicates that it is feasible even for very large graphs.

3.3 Summary of Offline Indexing

The index of our proposed *ESTI* method consists of two parts: FELINE$^+$ (Sect. 3.1) and the extended spanning tree (Sect. 3.2). The whole generation process is shown in Algorithm 2. Recall that building FELINE$^+$ index takes $O(|V|log(Deg_m^{(out)}) + |E|)$ time and $O(|V|)$ space, where $Deg_m^{(out)}$ is the maximum outgoing degree in G_A. And the time and space cost of constructing the extended spanning tree and three labels are both $O(|V|+|E|)$. Hence, the overall index constrution time of *ESTI* is $O(|V|log(Deg_m^{(out)}) + |E|)$, and index size is $O(|V| + |E|)$. Next, we will show how the constructed index supports efficient online k-hop reachability queries.

Algorithm 2. *ESTI* Index Construction

Input: A general directed graph G;
Output: FELINE$^+$ index X, Y; EST mapping each v in G to its real or virtual node v' in extended spaning tree; $Pre, Post, Level$ index for each node v' in the tree.

```
G_A ← condense SCCs in G
X, Y ← generating FELINE+ index for G_A                 ▷ see Algorithm 1
G' ← add a virtual root V_R and virtual edges in G      ▷ see Section 3.2
F ← {(V_R, 0)}                                          ▷ a queue used as BFS frontier
i ← 0
while F is not empty do
    pop (u, l) from F
    Level[i] ← l
    if u has not been visited then
        EST[u].RealNode ← i
        for each out-neighbor v of u do
            F ← F ∪ {(v, l + 1)}
    else
        add node i to EST[u].VirtualNodes
    i ← i + 1
Pre, Post ← Assign pre- and postorder for all real and virtual nodes in the tree
return X, Y, EST, Pre, Post, Level;
```

Algorithm 3. Basic Query Fucntion $Query(u, v, k)$

Input: Start vertex u, target vertex v, k; Offline index X,Y,EST,Pre,$Post$,$Level$.
Output: $True$ or $False$.

1: **if** $X[u] > X[v] \vee Y[u] > Y[v]$ **then**
2: **return** $False$
3: $u'_r \leftarrow EST[u].RealNode$
4: **for** each node v' in $\{EST[v].RealNode\} \cup EST[v].VirtualNodes$ **do**
5: **if** $(Pre[v'], Post[v']) \subset (Pre[u'_r], Post[u'_r]) \wedge level[v'] - level[u'_r] \leq k$ **then**
6: **return** $True$
7: **if** $k > 1$ **then**
8: **if** number of outgoing edges of $u \leq$ number of incoming edges of v **then**
9: **for** each outgoing neighbor w of u **do**
10: **if** $Query(w, v, k-1)$ **then**
11: **return** $True$
12: **else**
13: **for** each incoming neighbor w of v **do**
14: **if** $Query(u, w, k-1)$ **then**
15: **return** $True$
16: **return** $False$;

4 Online Querying

4.1 Basic Query Process

After constructing *ESTI* index (Sect. 3) for the input graph G, we can utilize the index to answer k-hop reachability queries online. Given a query $u \xrightarrow{?k} v$, if $u = v$ or $k \leq 0$ we can directly return the answer. Assume that $u \neq v$ and $k > 0$, the basic query function is shown in Algorithm 3.

As discussed in Sect. 3.1, in Line 1–2, if the topological order X (or Y) of u's corresponding vertex in DAG G_A is larger than v's X (or Y), we can safely return $False$. In Line 3–6, the pre- and postorders of real and virtual nodes are compared. Note that in Line 7–15, we run DFS only when $k > 1$ (Line 7) because the exploration will never return $True$ when $k \leq 1$. If $k = 1$ the answer from Line 3–6 is the final answer, and $k = 0$ is impossible since the initial input assumes that $k > 0$ while funtion *Query* is invoked only when $k > 1$.

Example 4. Given the constructed index in Fig. 5, for query $c \xrightarrow{?3} b$, we invoke *Query(c, b, 3)*. The pre- and postorder of c's Real Node is $(11, 16)$, but the real node of b has index $(2, 9) \not\subset (11, 16)$ and its virtual node has index $(6, 7) \not\subset (11, 16)$. Then *Query(d, b, 2)* is invoked, which results in calling *Query(e, b, 1)*. Luckily, b's virtual node has index $(6, 7) \subset (5, 8)$ and the function returns $True$.

To further improve the performance of online querying, we develop three pruning strategies based on properties of the extended spanning tree.

4.2 Pruning Strategies

Prune I. For query $u \xrightarrow{?k} v$, denote u'_r, v'_r as the real node of u and v, respectively. Prune I strategy utilizes Theorem 5 to stop redundant exploration in advance, i.e., $Query(u, v, k)$ will directly return $False$ if $level_{v'_r} - level_{u'_r} > k$.

Theorem 5. *If* $level_{v'_r} - level_{u'_r} > k$, *$u$ cannot reach v within k hops.*

Proof. Note that as discussed above, we never invoke $Query(u, v, k)$ s.t. $k = 0$.

(Case 1). When $k = 1$, assume that $level_{v'_r} - level_{u'_r} > 1$. If u can reach v within 1 hop, v has a real or virtual node v' which is the child of u'_r and $level_{v'} = level_{u'_r} + 1$. According to Theorem 4, $level_{v'} \geq level_{v'_r}$ indicates that $level_{v'_r} - level_{u'_r} \leq level_{v'} - level_{u'_r} = 1$, which contradicts the assumption.

(Case 2). When $k > 1$, in function $Query(u, v, k)$, Line 3–6 will never return $True$ since $level_{v'} \geq level_{v'_r}$ and $level_{v'} - level_{u'_r} \geq level_{v'_r} - level_{u'_r} > k$. Hence we need to invoke $Query(w, v, k-1)$ or $Query(u, w, k-1)$. For $Query(w, v, k-1)$, since the real or virtual node w' is a child of u'_r in the tree, the real node of w satisfies $level_{w'_r} \leq level_{w'} = level_{u'_r} + 1$. Thus, we have $level_{v'_r} - level_{w'_r} \geq level_{v'_r} - level_{u'_r} - 1 > k - 1$, and $Query(w, v, k-1)$ falls into Case 1 or Case 2 again. For $Query(u, w, k-1)$, since w'_r is the parent of one of the real or virtual node v' in the tree, w'_r satisfies $level_{w'_r} = level_{v'} - 1 \geq level_{v'_r} - 1$. Thus, we have $level_{w'_r} - level_{u'_r} \geq level_{v'_r} - level_{u'_r} - 1 > k - 1$, and $Query(u, w, k-1)$ also falls into Case 1 or Case 2 again.

Hence, if $level_{v'_r} - level_{u'_r} > k$, u cannot reach v within k hops. □

Example 5. In Fig. 5, for query $f \xrightarrow{?1} e$, both real and virtual nodes of e have level 3, while the real node of f has level 1. Since $3-1 > k = 1$, we return $False$.

Prune II. In Line 3–6 of Algorithm 3, we iterate all real and virtual nodes v' to compare $(pre_{v'}, post_{v'})$ with $(pre_{u'_r}, post_{u'_r})$, where u'_r is the unique real node of u. From the generation process of pre- and postorder index, $(pre_i, post_i)$ and $(pre_j, post_j)$ can never overlap for any vertex i and j. Instead of utilizing $(pre_{v'}, post_{v'})$, we can only check whether $pre_{v'} \in (pre_{u'_r}, post_{u'_r})$. Hence, $post_{v'_i}$ index of all virtual nodes v'_i will never be used in online phase, which means that we do not need to store $post$ index for all virtual nodes in offline phase.

Moreover, when vertex v has lots of virtual nodes v'_i, checking whether $pre_{v'_i} \in (pre_{u'_r}, post_{u'_r})$ is not efficient enough. Instead of iterating them one by one for comparison, if all the virtual nodes v'_i have been sorted w.r.t. their $pre_{v'_i}$ in offline phase, we can spend only $log(|v'_i|)$ to find the first virtual node whose $pre_{v'_i} > pre_{u'_r}$ and start iterating from it until $pre_{v'_i} > post_{u'_r}$, where $|v'_i|$ is the number of virtual nodes representing v. Note that the number of virtual nodes representing vertex v is equal to its incoming degree in G' minus 1, since in the extended spanning tree construction (Sect. 3.2), we create a virtual node for v only when v is visited again from an incoming neighbor. Hence, sorting all virtual nodes v'_i w.r.t. $pre_{v'_i}$ for each vertex v costs $O(|E|log(Deg_m^{(in)}))$, where $Deg_m^{(in)}$ is the maximum incoming degree of a vertex. And the overall time cost of offline indexing is $O(|V|log(Deg_m^{(out)}) + |E|log(Deg_m^{(in)}))$ if Prune II strategy is used in online phase.

Algorithm 4. *ESTI* Online Query Function $Query(u, v, k)$

Input: Start vertex u, target vertex v, k; Offline index X,Y,EST,Pre,$Post$,$Level$,$dist$.
Output: $True$ or $False$.

if $X[u] > X[v] \vee Y[u] > Y[v] \vee level_{v_r'} - level_{u_r'} > k$ **then** ▷ Prune I
    return $False$
$u_r' \leftarrow EST[u].RealNode$
$v_i' \leftarrow$ the first virtual node of v s.t. $pre_{v_i'} > pre_{u_r'}$ ▷ Prune II
while $pre_{v_i'} < post_{u_r'}$ **do**
    if $level[v'] - level[u_r'] \leq k$ **then**
        return $True$
    $v_i' \leftarrow$ next virtual node of v
if $k > 1 \wedge Dist[u] < k$ **then** ▷ Prune III
    if number of outgoing edges of $u \leq$ number of incoming edges of v **then**
        for each outgoing neighbor w of u **do**
            if $Query(w, v, k-1)$ **then**
                return $True$
    else
        for each incoming neighbor w of v **do**
            if $Query(u, w, k-1)$ **then**
                return $True$
return $False$;

Prune III. For each real node u_r' of u, while performing DFS traversal in offline index construction, we can find out $dist_u$ which represents the distance from u_r' to the nearest virtual node w_i' among all its successors in extended spanning tree. Given $dist$ index for every real node in the tree, for query $u \xrightarrow{?k} v$, if $dist_u \geq k$, we do not have to explore u's successors. That is because when exploring from u_r' in the tree, virtual nodes can only exists in the k^{th} hop. Assume that u can reach v within k hops. When one of v's real or virtual node is in the subtree rooted at u_r', the query will return $True$ in Line 3–6 in Algorithm 3. When all of v's real and virtual nodes are not in the subtree rooted at u_r', there must exist a virtual node w_i' which can jump out of the subtree to reach v. Note that $level_{w_i'} - level_{u_r'} < k$ holds, or it needs more than k hops from u to v. However, it contradicts $dist_u \geq k$ since the distance from u_r' to w_i' is smaller than k.

Example 6. In Fig. 5, for query $f \xrightarrow{?3} c$, the pre- and postorder index of c is not in the interval of f's index, i.e., $(11, 16) \not\subset (17, 24)$. Next, instead of exploring g and h, we can safely return $False$ directly since $dist_f = k = 3$.

4.3 Summary of Online Querying

After utilizing the three pruning strategies as discussed in Sect. 4.2, the *ESTI* query function $Query(u, v, k)$ is shown in Algorithm 4. Though in the worst case we still need to explore the whole graph, *ESTI* index still helps a lot for pruning online search space. Section 5 will demonstrate its practical efficiency.

Table 1. Statistics of datasets

Graph	$\|V\|$	$\|E\|$	Graph	$\|V\|$	$\|E\|$
kegg	3,617	4,395	p2p-Gnutella31	62,586	147,892
amaze	3,710	3,947	soc-Epinions1	75,879	508,837
nasa	5,605	6,538	10go-uniprot	469,526	3,476,397
go	6,793	13,361	10cit-Patent	1,097,775	1,651,894
mtbrv	9,602	10,438	uniprotenc22m	1,595,444	1,595,444
anthra	12,499	13,327	05cit-Patent	1,671,488	3,303,789
ecoo	12,620	13,575	WikiTalk	2,394,385	5,021,410
agrocyc	12,684	13,657	cit-Patents	3,774,768	16,518,948
human	38,811	39,816	citeseerx	6,540,401	15,011,260
p2p-Gnutella05	8,846	31,839	go-uniprot	6,967,956	34,770,235
p2p-Gnutella06	8,717	31,525	govwild	8,022,880	23,652,610
p2p-Gnutella08	6,301	20,777	soc-Pokec	1,632,803	30,622,564
p2p-Gnutella09	8,114	26,013	uniprotenc100m	16,087,295	16,087,295
p2p-Gnutella24	26,518	65,369	yago	16,375,503	25,908,132
p2p-Gnutella25	22,687	54,705	twitter	18,121,168	18,359,487
p2p-Gnutella30	36,682	88,328	uniprotenc150m	25,037,600	25,037,600

5 Experiments

We evaluate the effectiveness and efficiency of the proposed *ESTI* method by carrying extensive experiments on both small and large graphs. All the experiments are conducted on a Linux machine with an Intel(R) Xeon(R) E5-2678 v3 CPU @2.5GHz and 220G RAM, and all algorithms are implemented using C++ and complied by G++ 5.4.0 with -O3 Optimization. Each experiment has been run for 10 times and the results are consistent among 10 executions. In this section, we report the average value from 10 executions of each experiment.

5.1 Datasets

A variety of real graphs are used in our experiments, as shown in Table 1. *kegg*, *amaze*, *nasa*, *go*, *mtbrv*, *anthra*, *ecoo*, *agrocyc* and *human* are small graphs from different sources [13]. *p2p-Gnutella* graphs are 8 snapshots of Gnutella peer to peer file network, while *soc-Epinions1* is a who-trust-whom online social network [5]. As for large graphs, *10go-uniprot*, *go-uniprots*, *uniprotenc22m*, *uniprotenc100m* and *uniprotenc150m* come from Uniprot database. *10cit-Patent*, *05cit-Patent*, *cit-Patents* and *citeseer* are citation networks [3]. *WikiTalk* is a Wikipedia communication network, while *soc-Pokec* and *twitter* are large-scale social networks [5,7]. *govwild* and *yago* are RDF datasets [7].

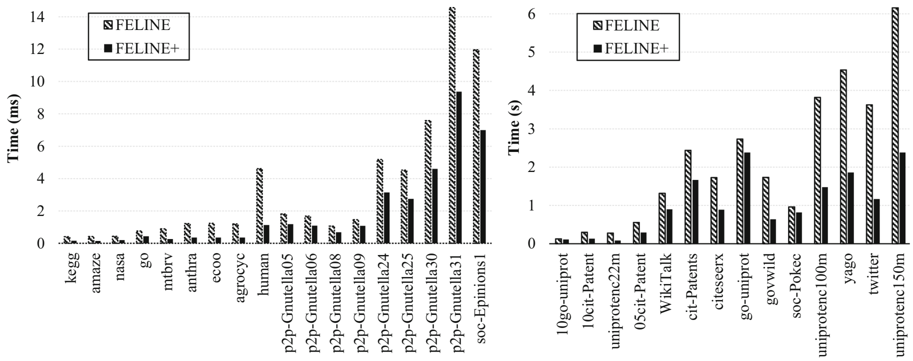


Fig. 6. Index construction time of FELINE and FELINE$^+$

5.2 Performance of FELINE$^+$

As discussed in Sect. 3.1, we propose an optimized approach named FELINE$^+$ to accelerate FELINE index generation, while obtaining exactly the same index as FELINE. Figure 6 shows the index construction time, in which FELINE$^+$ significantly speeds up the construction process in all graphs.

5.3 Queries with Different k

The efficiency of online querying is crucial for k-hop reachability query answering, and different values of k can significantly affect the performance. We report the query time of the proposed *ESTI* method with different values of k ($k = 2, 4, 8$) in Table 2, comparing it with the state-of-art *k-reach* approach [2]. For each k, we generate a million queries with randomly selected start and target vertices. Note that *k-reach* requires a fixed budget b to construct the partial vertex cover and we set $b = 1000$, which is the same as the budget used in [2].

When the value of k increases, the time cost of both *k-reach* and *ESTI* also tend to increase, because a larger k indicates a larger search space when the built index cannot directly answer a query. We notice that most of queries fall into the worst case in *k-reach*, which needs traditional BFS search over the whole graph. Note that when $k = 4$ and $k = 8$, *k-reach* exceeds our time limit (4 h) in graph *soc-Pokec*. Clearly, *ESTI* is faster than *k-reach* over all graphs when $k = 2$ and $k = 4$, and it also beats *k-reach* in most graphs except for graph *WikiTalk*. Note that the diameter of *WikiTalk* is 9, which is relatively small and is quite closed to $k = 8$. In practice, k will not be too large for social networks.

Table 2. Query time (ms) of different k

Graph	k = 2		k = 4		k = 8	
	k-reach	ESTI	k-reach	ESTI	k-reach	ESTI
kegg	63	**27**	97	**42**	103	**40**
amaze	58	**25**	83	**34**	90	**37**
nasa	96	**20**	193	**33**	212	**45**
go	177	**36**	380	**70**	391	**111**
mtbrv	66	**16**	124	**24**	116	**23**
anthra	68	**17**	108	**25**	129	**26**
ecoo	67	**17**	123	**25**	114	**28**
agrocyc	72	**17**	105	**26**	124	**27**
human	69	**20**	138	**26**	255	**29**
p2p-Gnutella05	630	**95**	8,069	**723**	47,595	**9,507**
p2p-Gnutella06	624	**93**	9,260	**755**	56,856	**9,890**
p2p-Gnutella08	656	**73**	5,654	**462**	26,063	**5,061**
p2p-Gnutella09	529	**74**	5,078	**482**	34,501	**7,006**
p2p-Gnutella24	467	**74**	4,862	**514**	100,276	**15,639**
p2p-Gnutella25	509	**65**	4,534	**430**	79,991	**14,438**
p2p-Gnutella30	668	**74**	6,166	**478**	129,051	**24,226**
p2p-Gnutella31	806	**78**	5,784	**503**	195,265	**34,490**
soc-Epinions1	132,712	**596**	999,966	**3,747**	765,753	**11,932**
10go-uniprot	788	**109**	1,622	**204**	2,505	**469**
10cit-Patent	245	**90**	400	**159**	815	**322**
uniprotenc22m	491	**77**	644	**102**	776	**129**
05cit-Patent	458	**130**	722	**222**	1,649	**480**
WikiTalk	1,112,536	**240**	8,091,777	**842**	**769,542**	11,162,590
cit-Patents	5,259	**382**	36,879	**1,605**	306,144	**21,195**
citeseerx	927	**205**	2,935	**264**	23,154	**763**
go-uniprot	1,196	**214**	1,386	**201**	2,744	**351**
govwild	3,419	**147**	9,993	**211**	19,229	**483**
soc-Pokec	1,510,794	**4,057**	-	**653,194**	-	**6,430,662**
uniprotenc100m	744	**95**	987	**113**	1,748	**160**
yago	501	**113**	861	**168**	1,255	**257**
twitter	592	**211**	647	**215**	1,432	**435**
uniprotenc150m	938	**103**	1,367	**146**	2,019	**205**

Table 3. Index size, index construction time and query time on small graphs

Graph	Index size (KB)		Index time (ms)		Query time (ms)	
	k-reach	ESTI	k-reach	ESTI	k-reach	ESTI
kegg	129	**101**	80	**1**	107	**44**
amaze	127	**101**	77	**1**	97	**42**
nasa	229	**139**	78	**1**	200	**43**
go	284	**212**	81	**3**	298	**68**
mtbrv	345	**233**	76	**2**	120	**31**
anthra	448	**301**	79	**3**	118	**30**
ecoo	452	**305**	79	**3**	120	**30**
agrocyc	454	**306**	81	**3**	119	**30**
human	1,380	**922**	87	**10**	121	**32**
p2p-Gnutella05	450	**389**	80	**7**	34,067	**5,659**
p2p-Gnutella06	445	**384**	79	**7**	39,910	**5,747**
p2p-Gnutella08	383	**262**	80	**4**	19,977	**2,989**
p2p-Gnutella09	407	**332**	80	**6**	24,226	**4,514**
p2p-Gnutella24	1,680	**931**	89	**19**	77,227	**12,457**
p2p-Gnutella25	2,102	**787**	91	**14**	61,227	**9,966**
p2p-Gnutella30	3,056	**1,269**	99	**28**	103,566	**11,235**
p2p-Gnutella31	5,189	**2,143**	123	**55**	160,885	**23,031**
soc-Epinions1	50,211	**5,361**	957	**134**	1,214,127	**6,579**

5.4 Comparison with the State-of-art

As discussed in Sect. 1, *k-reach* [1,2] is the only method solving k-hop reachablity queries on general directed graphs. We conduct experiments on both small and large graphs to compare the proposed *ESTI* method with *k-reach*. For each graph, we randomly generate a million queries while values of k are generated following the distance distribution of all reachable pairs. Their index size, index construction time and query time are reported in Table 3 and 4.

The results in Table 3 shows that *ESTI* completely beats *k-reach* in all small graphs. Note that the budget of *k-reach* is also set to be 1000. *ESTI* constructs smaller index and is approximately an order of magnitude faster when building index for most small graphs. As for online querying, *ESTI* costs significantly less time. It is even more than a hundred times faster in graph *soc-Epinions1*.

For large graphs, we compare our *ESTI* method with *k-reach* in Table 4, where the budget of *k-reach* are set to be 1,000 and 50,000, respectively. Note that *k-reach* exceeds our time limit (4 h) on graph *soc-Pokec*. When answering queries online, *ESTI* method costs much less time over all large graphs. Though *ESTI* needs longer index construction time on most graphs, we believe that the efficiency of online query processing is more important than

Table 4. Index size, index construction time and query time on large graphs

Graph	Index size (MB)			Index time (s)			Query time (s)		
	k-reach (b=1k)	k-reach (b=50k)	ESTI	k-reach (b=1k)	k-reach (b=50k)	ESTI	k-reach (b=1k)	k-reach (b=50k)	ESTI
10go-uniprot	**24**	**24**	34	0.7	**0.4**	1.1	1.2	1.0	**0.2**
10cit-Patent	**26**	39	31	**0.2**	0.6	0.8	0.4	0.4	**0.1**
uniprotenc22m	55	55	**37**	0.6	**0.5**	0.8	0.4	0.4	**0.1**
05cit-Patent	**41**	60	53	**0.3**	1.2	1.7	0.6	0.7	**0.2**
WikiTalk	217	217	**75**	4.7	4.5	**4.0**	6392	6049	**0.8**
cit-Patents	**181**	558	188	**5.2**	22.4	9.2	94.0	92.5	**9.7**
citeseerx	**171**	237	219	**3.5**	11.1	6.4	2.1	2.2	**0.3**
go-uniprot	331	**321**	372	9.8	**6.9**	13.7	1.0	1.0	**0.3**
govwild	**256**	262	314	5.1	**4.4**	8.0	9.9	6.6	**0.2**
soc-Pokec	**183**	**183**	260	11.3	**10.5**	13.8	-	-	**2281**
uniprotenc100m	556	556	**370**	8.2	**6.0**	10.0	0.7	0.7	**0.1**
yago	**411**	446	472	**2.9**	4.2	12.3	0.5	0.6	**0.1**
twitter	609	609	**443**	**5.5**	6.9	11.3	0.6	0.6	**0.3**
uniprotenc150m	866	866	**576**	14.3	**10.1**	16.8	0.9	0.9	**0.1**

offline indexing. Theorectically, the overall time cost of *ESTI* offline indexing is $O(|V|log(Deg_m^{(out)}) + |E|log(Deg_m^{(in)}))$, which is a stable bound.

The index size of *ESTI* is $O(|V| + |E|)$, which is strictly linear to the size of input graph. However, *k-reach* with budget 1,000 has the smallest index size on some large graphs, and it also costs a lot of time to answer queries online. It seems that 1,000 is a relatively small budget, which may limit the querying performance of *k-reach*. But when the budget is set to be 50,000, *k-reach* has larger index size than *ESTI* in many graphs, while it still cost more time in online querying process. Hence, the overall query answering ability of *ESTI* method is also better over large graphs.

6 Related Works

6.1 Reachabilty Query

Before Cheng et al. [1] first proposed k-hop reachability problem, lots of studies about reachability query over large graphs have been carried. Reachability query is a special case of k-hop reachability query when $k = \infty$. Since the lack of distance information, existing reachability query methods including *BFL* [8], *IP+* [11], *GRIPP* [9], *PWAH8* [6], *GRAIL* [13] and *Path-Tree Cover* [4], etc. are not sufficient to answer k-hop reachability queries.

6.2 k-hop Reachabilty Query

To answer k-hop reachability problems, a naive idea is to process BFS or DFS in given directed graph. Both BFS and DFS don't need any pre-computed index, but they are not efficient when the graph becomes very large, since lots of search branches will be expanded while exploring in the original large graph. In contrast, storing the shortest distance between each pair of vertices helps to answer any queries within *O(1)* time. However, in order to compute and store such distance, performing BFS from every vertex in G costs $O(|V|(|V|+|E|))$ time and $O(|V|^2)$ space, which is also inefficient and even infeasible for large graphs.

Vertex Cover Based Method. Vertex cover is a subset of all the vertices in a given graph G, making sure that for each edge in G, at least one of the two vertices connected by this edge is contained in the vertex cover. *k-reach* [1,2] makes good use of vertex cover, and runs BFS in the subgraph constructed from vertex cover to build index. Though it is proved efficient in small graphs, when dealing with larger graphs, *k-reach* still costs infeasible index time and space.

To overcome this drawback, Cheng et al. also proposes a partial vertex cover [2] to make a trade-off between offline index and online query performance. Though it can work on very large graphs, the partial vertex cover index cannot answer a large proportion of online queries directly. In fact, traditional online BFS would be invoked for more than 95% of the queries. Hence, it is still not practical enough for answering k-hop reachability queries efficiently.

Methods Work on DAGs. To improve index efficiency, Xie et al. [12] proposed *BFSI-B* Algorithm, which uses the breadth-first spanning tree to build *BFSI* index, including *min-post* index and global BFS level TLE. Also, *FELINE* index [10] is adopted to filter those unreachable queries. Another method developed for DAGs is *HT* [3], which adopts the idea of partial 2-hop cover. In its indexing process, vertices with high degree are selected as hop nodes. Both backward and forward BFS are started from each hop node u. When visiting a new vertex v, current hop node's id u and the distance from u to v will be stored as the index of v. Topological order is also used for filtering unreachable queries.

Though both *BFSI-B* and *HT* are more efficient than *k-reach*, they can only work for DAGs and cannot directly deal with directed graphs with cycles. Also, more efficient pruning strategies need to be utilized to further improve online querying performance.

Algorithms for Distributed Systems. To deal with multiple k-hop reachability queries concurrently on distributed infrastructures, *C-Graph* [14] focuses on improving both disk and network I/O performance when performing BFS. Compared with developing methods for a single machine, designing optimizations for distributed systems is a significantly different task.

7 Conclusion

We propose *ESTI* method to efficiently solve k-hop reachability queries for general directed graphs, which builds an extended spanning tree in offline phase and utilizes three pruning strategies to accelarte query processing. Also, an optimization named FELINE$^+$ is developed to speeds up FELINE index generation, which helps to effectively filter unreachable queries in online searching.

We also conduct extensive experiments to compare *ESTI* with the state-of-art method *k-reach*. Our experiment results confirm that on most graphs the overall performance of *ESTI* is the best, and in online querying it is significantly faster.

Acknowledgments. This work was supported by National Natural Science Foundation of China (Grant No. 61902074) and Science and Technology Committee Shanghai Municipality (Grant No. 19ZR1404900).

References

1. Cheng, J., Shang, Z., Cheng, H., Wang, H., Yu, J.X.: K-reach: who is in your small world. CoRR abs/1208.0090 (2012)
2. Cheng, J., Shang, Z., Cheng, H., Wang, H., Yu, J.X.: Efficient processing of k-hop reachability queries. VLDB J. **23**(2), 227–252 (2014)
3. Du, M., Yang, A., Zhou, J., Tang, X., Chen, Z., Zuo, Y.: HT: a novel labeling scheme for k-hop reachability queries on DAGs. IEEE Access **7**, 172110–172122 (2019)
4. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: SIGMOD 2008, pp. 595–608 (2008)
5. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection
6. van Schaik, S.J., de Moor, O.: A memory efficient reachability data structure through bit vector compression. In: SIGMOD 2011, pp. 913–924 (2011)
7. Seufert, S., Anand, A., Bedathur, S., Weikum, G.: Ferrari: flexible and efficient reachability range assignment for graph indexing (2013)
8. Su, J., Zhu, Q., Wei, H., Yu, J.X.: Reachability querying: can it be even faster? IEEE Trans. Knowl. Data Eng. **29**(3), 683–697 (2017)
9. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD 2007, pp. 845–856 (2007)
10. Veloso, R.R., Cerf, L., Meira Jr., W., Zaki, M.J.: Reachability queries in very large graphs: a fast refined online search approach. In: EDBT, pp. 511–522 (2014)
11. Wei, H., Yu, J.X., Lu, C., Jin, R.: Reachability querying: an independent permutation labeling approach. Proc. VLDB Endow. **7**(12), 1191–1202 (2014)
12. Xie, X., Yang, X., Wang, X., Jin, H., Wang, D., Ke, X.: BFSI-B: an improved k-hop graph reachability queries for cyber-physical systems. Inf. Fusion **38**, 35–42 (2017)
13. Yildirim, H., Chaoji, V., Zaki, M.: Grail: a scalable index for reachability queries in very large graphs. VLDB J. **21**, 1–26 (2012)
14. Zhou, L., Chen, R., Xia, Y., Teodorescu, R.: C-graph: a highly efficient concurrent graph reachability query framework. In: ICPP, pp. 79:1–79:10. ACM (2018)