Information Fusion 38 (2017) 35–42

Contents lists available at ScienceDirect

Information Fusion

journal homepage: www.elsevier.com/locate/inffus

Full Length Article

BFSI-B: An improved K-hop graph reachability queries for cyber-physical systems☆

Xia Xie[a,*], Xiaodong Yang[a], Xiaokang Wang[b], Hai Jin[a], Duoqiang Wang[a], Xijiang Ke[a]

[a] Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[b] Cyber-Physical-Social Systems Lab, Huazhong University of Science and Technology, Wuhan, China

ARTICLE INFO

Article history:
Received 19 September 2016
Revised 21 January 2017
Accepted 20 February 2017
Available online 22 February 2017

Keywords:
K-hop reachability queries
Large graph
Online search
BFSI-B

ABSTRACT

Cyber-physical systems, encompass the real physical space and virtual cyber space for providing advanced service for humans, which together, are also namely Internet of Things. The complex and large scale relationships among nodes contain the potential information of user, which is required and essential for providing high quality personalized service. Graph is used to represent, make fusion and process the relationships data, which has been used in many domains with traditional small data sets. *K*-hop reachability query answering in graphs is seeing a growing number of applications, such as ranked keyword search in databases, social networking, ontology reasoning and bioinformatics. Currently, techniques for efficient evaluation of such queries are based on vertex-cover technology. These methods suffer from a lack of scalability; that is, they cannot be applied to very large graphs. To solve these problems, we propose the compound-index method, namely BFSI-B, which uses the special index to decrease the *k*-hop reachability query time and improve pruning efficiency. The theoretical analysis and experiment results demonstrate that our method has a smaller index size and a faster query time than the *k*-reach method, in both dense graphs and very large graphs.

1. Introduction

The cyber-physical systems (CPS), also referred as Internet of Things (IoT) [1], integrates the cyber space and physical space together and has the potention to provide predictive and personalized service for human beings [2]. The connections among the nodes including the objects such as the smart devices, smart sensors and humans contain the real intentions and requires of the users. Also, the data, collected from the two different spaces, are suggested as the research start point of the CPS. Therefore, to realize the significant potential, the data about relationships among the nodes in CPS must be efficiently represented and analyzed.

However, two important features of the relationships data in CPS should paid main attention on: (1) the data used to represented the relationships among the nodes are complex. The relationship among the nodes in CPS are complex. For example, the user, also referred as a node, will integrate with smart medical nodes such as the blood pressure sensor, height sensor, body temperature sensor, weight sensor, and heart rate sensor, when he goes to the smart hospital. So how can we represent the relationships data is one of the main challenges. (2) The relationship data collected from various devices are large scale. The connection among massive nodes in CPS brings the large scale relationship data, which are increasing all the time. How to efficiently process these data is a second challenging question about service providing as well.

Graph, as a relationship data represented and analysis tool, are studied and used in many domains such as social media, finance, communication networks, biological networks and traffic networks. Currently, reachability queries, as a basic graph data processing operation, are receiving more and more research attention to answers whether a path exists from vertex u to vertex v, when given a directed graph $G = (V, E)$, and a pair of vertices u and v, with $O(|V| + |E|)$ runtime by graph traversal. Furthermore, as the key questions of the reachability queries, the *k-hop reachability problem* and the *shortest path problem* have been discussed in many literatures to obtain better runtime of graph data processing.

☆ Fully documented templates are available in the elsarticle package on CTAN.

* Corresponding author.
E-mail address: shelicy@hust.edu.cn (X. Xie).

http://dx.doi.org/10.1016/j.inffus.2017.02.009
1566-2535/

The k-hop reachability problem, firstly discussed by Cheng [3], asks whether vertex u can reach vertex v within k hops, which can be considered a generalization of the basic reachability while $k = \infty$. A k-reach indexing approach is developed and proved to efficiently process the basic reachability query. As mentioned in [4], the more scalable k-reach indexing approach is based on a two-level vertex cover, which is a set of vertices that cover all the edges in the graph. The experimental results verified the efficiency of k-reach in answering k-hop reachability queries, for both small and large values of k, thus demonstrating its suitability for different real-life applications where the value of k may vary. But it is not suitable for large graph. Therefore, the k-hop reachability query with high efficiency has difficulty scaling to massive graphs.

Another research area of the reachability query problem is the *shortest path problem*, which has two main kinds of methods the landmark-based indexing including the PLL [5], LCA [6], and 2-hop cover indexing including 2-HOP [7], HCL [8], HLS [9]. Landmark nodes and single-source shortest paths among those is identified and pre-computed in the former one. To obtain the shortest path between a pair of arbitrary vertices, a connection of hops for each node is stored in the latter one. The shortest path methods can be widely used in large scale graph processing, with pre-computing time to obtain the shortest path bringing the unsatisfactory efficiency.

Therefore, a key question of how to exploit the high efficient lager scale graph analysis and processing strategy has triggered enormous attention and research activities. To tackle aforementioned challenge, we proposed an new method of answering the k-hop reachability query with a compound index, namely BFSI-B, to decrease the k-hop reachability query time by improving pruning efficiency. The contribution of this paper are summarized as follows. To decrease the search space, breadth-first search index (BFSI) is used to record the distance between nodes in the BFS-trees and improve the efficiency of pruning. To improve the search efficiency, FELINE(Fast rEfined onLINE search) index is used to discover whether the connect nodes in the BFS-trees are k-hop reachability.

The paper is organized as follows. Section 2 discusses related studies. Section 3 introduces our compound index, which includes the FELINE index, BFSI index (Breadth-first spanning-tree index), and their construction method. We provide theoretical justification by using this method. In Section 4, we introduce our query method with the compound index. Section 5 shows the experiment results. In Section 6, we give our conclusion, based on the results of the study.

2. Background

Two extreme methods are used to solve the reachability problem of social-network vertices in a graph. The first one is used to compute the storage and transmission and transitive closure in a given map. The one can be realized in constant time. Unfortunately, a quadratic space index is required. That means, it's consequently for large graphs. The second one is to verify reachability by DFS or BFS method. To achieve reachability, the breadth-first search (BFS) or depth-first search (DFS) algorithm was presented and even to determine the shortest path of two particular nodes. With the time complexity $O(|V| + |E|)$, the traditional BFS/DFS is unacceptable for large graphs as well. The main questions about the graph reachability indexing for large graph are as follow.

2.1. Reachability query

Many studies [10–12] use the transitive closure compression to answer the reachability query. To facilitate reachability checking, a directed acyclic graph (DAG) was decomposed in to a minimized set of disjoint chains in [13], which can be extended to directed cyclic graphs as well. Vertices are separated into k pair-wise disjoint paths, which is served as a vertex in a tree structure in path-tree cover method [14]. The methods are always used to compress the transitive closure equal to or even better than the optimal tree-cover and chain-cover approach, which also answers the reachability query in constant time. In [15], a 3-hop indexing is used to describe the reachability between source vertices and destination vertices.

With the index construction time complexity $O(m+n)$, a method, namely the GRIPP [16], was proposed to extract a tree cover from a DAG to speed up the DFS processing. Furthermore, GRAIL [17] is the latest scalable reachability-indexing technique, wherein each vertex u in the DAG is assigned multiple interval label Lu, which can help quickly to determine the non-reachability between two vertices. The fundamental idea behind FELINE [18] is to associate every vertex in V with a unique ordered pair of natural integers (X, Y). X and Y are two different kinds of topological order. Each pair of vertices has an upper-right relationship, which can negative the majority of unreachable queries on constant time, but cannot answer the queries directly. In [19], the authors introduce a unified SCARAB method, based on reachability backbone (similar to the motorway in a transportation network) to deal with their limitations: it can both help scale the transitive-closure approaches and speed up online searching. However, the query performance of transitive-closure approaches tends to be slowed down, and they may still not work if the size of the reachability backbone remains too large.

2.2. K-hop reachability query in graph

A k-reach index based on vertex cover was proposed in [3]. However, the vertex cover is also often large, this method is not feasible, especially for the large graphs. To address this challenge, a partial vertex cover has been proposed in [4] to exchange query-processing time with index-storage space. However, this study is still based on the vertex-cover method. Therefore, for very large graphs, even the new method affords better cover; we notice that 95% of the queries fall into the worst-case scenario of k-reach, which uses the traditional BFS method to get results. Our experiments indicate that the k-reach method performs much worse on the very large graph.

3. Compound index construction

3.1. Problem definition

In DAGs, a graph is defined as $G = (V, E)$ with a set E of edges and a set V of vertices. Supposing that u and v are two arbitrary vertices, the shortest distance is denotes as $d(u, v)$ among all the paths from u to v. And the $r(u, v)$ and $r(u, v, k)$ are used to denoted as the reachability query and k-hop reachability from u to v, respectively. Also, the $u \rightarrow v$ and $u \rightarrow_k v$ are used to defined the u can reach v and the u can reach v within k hops, respectively. We studied the following problem: given a DAG G, we need to determine that the distance between the two vertices is not greater than k. Table 1 lists the commonly used notations in this paper.

As a challenging problem, the k-hop reachability query has attracted the research attention in respect of large graphs. Firstly, the existing research into the k-hop reachability query cannot handle very large graphs, and the high cost of index makes the scalability of algorithms limited. Second, as a more popular problem, k-hop reachability has been widely used in many domains. In the k-hop reachability query, we need to estimate the distance between the query nodes. Using an online search, we found that k-hop reach-

Table 1
Commonly used notations.

Notation	Description
$G = (V, E)$	A graph with vertex set V and edge set E
$d(u, v)$	Shortest distance between u and v
Roots	Vertices with no predecessors
Leaves	Vertices with no successors
$r(u, v)$	A query whether u can reach v
$r(u, v, k)$	A query whether u can reach v within k hops
$u \to v$	u can reach v
$u \to v_k$	u can reach v within k hops
$\|V\|$	number of vertices
$\|E\|$	number of edges

ability query time will far exceed the reachable query, especially when $k = 4$, 5 or 6.

This paper presents a method to solve the k-hop reachability query using a compound index, and proposes a BFSI index to decrease the k-hop reachability query time by improving pruning efficiency. The following sections describe the structure of the index set.

3.2. FELINE index

Reachability is a necessary condition for k-hop reachability: if u cannot reach v, v must be unreachable from u within k hops. Therefore, we use the reachability index for denying the case that u cannot reach v. The FELINE index is the state-of-the-art method for the reachability query; it can deny the reachability between two vertices and has the best scalability. The FELINE index is used as one of the indices in our compound index. Given a DAG G, after the construction of the FELINE index, every vertex is associated with a pair x, y. For example, if there is a direct path from u to v, then the x-coordinate of u is less than or equal to that of v. The condition of y-coordinate is the same. Algorithm 1 generates the coordinates that will compose the FELINE index. The x-coordinates are first determined by a topological ordering algorithm, resulting in the set of coordinates X. Any topological ordering algorithm can be used (an $O(|V| + |E|)$ time is found in [18]).

Algorithm 1 FELINE index construction.

Input: G(V, E), a DAG, nodes number is from 1 to $|V|$
Output: (X, Y), two vectors of size $|V|$
1: $X \leftarrow TopologicalOrdering(V, E)$
2: $Y \leftarrow \oslash$
3: $heads \leftarrow (\oslash, \ldots\ldots, \oslash)$
4: $d \leftarrow (0, \ldots\ldots, 0)$
5: **for all** $(u, v) \in E$ **do**
6: $(heads)_u \leftarrow (heads)_u \cup \{v\}$
7: $d_v \leftarrow d_v + 1$
8: **end for**
9: $roots \leftarrow \{v \in V \mid d_v = 0\}$
10: **while** $roots \neq \oslash$ **do**
11: $u \leftarrow (argmax)_{v \in roots}(X_v)$
12: //Append u to Y
13: $Y \leftarrow (Y, u)$
14: //Delete u from the nodes set of roots
15: $roots \leftarrow roots \setminus \{u\}$
16: **for all** $v \in (heads)_u$ **do**
17: $d_v \leftarrow d_v - 1$
18: **if** $d_v = 0$ **then**
19: $roots \leftarrow roots \cup \{v\}$
20: **end if**
21: **end for**
22: **end while**

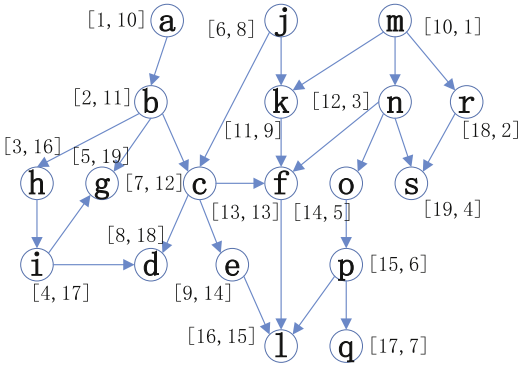

Fig. 1. Feline index of G.

For instance, consider the DAG of Fig. 1. A DFS-based topological ordering can generate the set X with the vertices $a, b, c, i, g \ldots, s$ associated with x-coordinates with rank values from 1 to 19. The algorithm first fills the roots set with a, b (line 9). Then, in line 11, it chooses the root vertex m (with the rank value 10, in X) and puts it into the Y set (in the first position). Next, in line 13, m is removed from the roots, which is updated with the new roots n and r, and now $roots = a, j, n, r$. Then, as r has the higher rank in X, it is also removed from the roots and inserted into the second position of Y. Note that G has no descendants, and, at that moment, $Y = m, r$ and $roots = a, j, n$. The vertex n is the next chosen and $Y = m, r, n$. In Fig. 1, the vertices m, r and n have the y-coordinates in 1, 2 and 3, r. The algorithm continues until the Y set is complete.

We can answer the k-hop reachability query using the FELINE index; we can use the FELINE index to deny the k-hop reachability query if u and v are not reachable by the FELINE index. For instance, in graph G, the FELINE index of a is [1, 10], m is [10, 1], r is [18, 2], s is [19, 4]; as the FELINE index of m is not all smaller than r and s, r and s are not reachable from m. However, the FELINE index of r is smaller than that of s, but it cannot ensure that s can be reachable from r within k hops.

The FELINE index can only determine the condition that u cannot reach v. If not, the FELINE index is not satisfactorily. In order to determine the second possibility, we use another index, BFSI to improve the efficiency of the k-hop reachability query.

3.3. BFSI index

As previously mentioned, two questions rely on the FELINE index pruning strategies. Firstly, in the case of k-hop reachability, if v is reachable from u within k hops, then it cannot judge directly, it needs to explore all vertices inside the region between u and v recursively. Secondly, u can reach v, but the shortest distance from u to v is bigger than k; in this case, the FELINE index also needs to explore recursively. If we can verify that v is reachable from u in k hops according to the index, the query regarding the k-hop reachable pairs of vertices can be verified without a recursive search (which takes a long time), reducing the access number of vertices and improving the efficiency.

For instance, to determine whether g can be reached from u in 4 hops, the FELINE index is inadequate; it will recursively explore the child node of u, and will stop the search until it traverses the node g. If we can immediately stop the search and give the correct answer according to the index, the time of query can be improved efficiently.

According to Yildirim [17], in a tree, indexing using a traditional min-post algorithm is enough to answer some queries in constant time. We constructed the breadth-first spanning tree from the original DAG. Each node in the DAG is in the unique breadth-first spanning tree, and has a unique index constructed by the BFSI. If we can immediately stop the search, and if u can reach v in a

tree and the distance from u to v in the tree is no more than k, the result is true; if the distance is more than k, the result is false.

The fundamental idea behind BFSI is to associate every vertex in v with a unique integer, so that the distance between the two vertices can be answered if one of the shortest path is in the BFSI tree. In other terms, given the index BFSI (min, post, TLE), built by BFSI, and two vertices (u, v), in this way, if $min(u) < min(v)$, $post(u) > post(v)$, then no traverse of the graph is needed to answer the query regarding the k-hop reachability from u to v. We can get a negative answer if $TLE(v) - TLE(u) > k$, or a positive answer.

Theorem 1 (Shortest path). *In a DAG, in the breadth-first spanning forest constructed by BFS, if u can reach v in the tree, the path from u to v in the forest is the shortest path in the DAG.*

Proof. We indicate the path from u to v in the breadth-first spanning forest is path p, the root of the tree is s, and us deep value in the tree is $level(u)$ and vs deep value is $level(v)$. We can establish from the BFS construction that the distance from s to v is $d(s, v) = level(v)$, $d(s, u) = level(u)$. If we suppose that there is a shortest path p with length less than k, the distance from s to v $d(s, v) = d(s, u) + d(u, v) < level(u)$, it is in contradiction with the truth that $d(s, v) = level(v)$, so the path from u to v in the tree is the shortest path in the DAG. □

Definition 1 (Global BFS level). Given a DAG G, the definition of the global BFS level $TLE(v)$ is as follows:

$$TLE(v) = \begin{cases} startlevel, & in\ N(v, G) = \emptyset \\ min\{TLE(u) + 1 \mid u \in N(v, G)\}, & otherwise \end{cases} \quad (1)$$

The *startlevel* is the start value of the breadth-first spanning tree; each of the root nodes has a different *startlevel*, to make sure that nodes in the different tree or level has different global BFS levels. The *startlevel* of the root node follows the highest level before it is visited. The global level of the first breadth-first spanning tree is 1. We can get the global BFS levels in a traverse of the graph G.

Theorem 2 (*k*-reach by BFSI). *Given BFSI index (min, post, TLE) of the graph G, the min and post index is by applying a min-post strategy to the breadth-first spanning trees, the TLE is the height of the node in the same BFS-tree, if* $[min(v), post(v)] \subset [min(u), post(u)]$, $TLE(v) - TLE(u) \leq k$, *then u can reach v within k hops.*

Proof. If u and v are in the same BFS-tree, according to theorem 1 the distance from u to v is d, $d = TLE(v) - TLE(u) \leq k$ then u can reach v in k hops in the BFS-tree, also in Graph G. □

The construction process of the BFSI index has two main steps. The first one is to construct the breadth-first forest in Graph G to get the third value of the BFSI index. The second step is to perform a min-post strategy in the BFS-tree to get the first and the second value of the BFSI index. The BFSI index construction algorithm is shown in Algorithm 2. The max level of the visited node is denoted as *maxlevel*, and the *prepost* is the post order of the last node; the min-post function is the algorithm of the min-post.

For instance, consider the DAG of Fig. 2. It chooses the root vertex a and starts a BFS from a, generating the first BFS-tree; according to (1), the TLE value can be generate from the BFS that $TLE(a) = 1, TLE(b) = 2, TLE(g, c, h) = 3, TLE(d, e, f, i) = 4, TLE(l) = 5$. Next, we perform the min-post algorithm on the BFS-tree to get the first and the second value of the BFSI index: in Fig. 2, a is [1, 10, 1], b is [1, 9, 2] and the algorithm continues until the min-post algorithm is complete. It then chooses another root vertex j and repeats the steps until each node is visited. The BFSI index of Graph G is shown in Fig. 2.

Algorithm 2 BFSI index construction with G.

Input: (V, E), a DAG G
Output: $(min, post\ TLE)$, three vectors of size $|V|$

```
for all v ∈ Roots do
  TLE ← (v, maxlevel) //Initialization
  add v into S
  while S ≠ ∅ do
    pop up a node x from S
    for all (x, y) ∈ E do
      if y is not visited then
        //add(x, y) into BFS-tree
        add(BFS-Tree,x, y)
        visit y
        TLE ← (y, TLE(X) + 1)
        update maxlevel
        add y into S
      end if
    end for
  end while
  update prepost
  update maxlevel
  min-post(BFS-tree, v, prepost)
end for
```

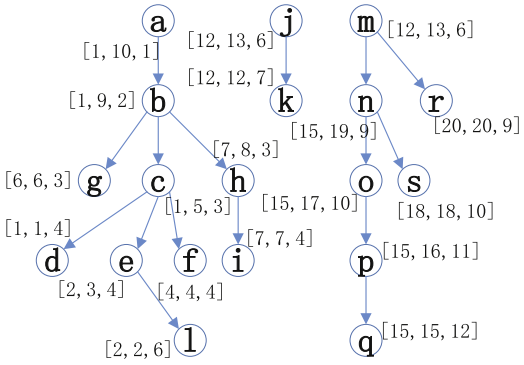

Fig. 2. BFSI index of G.

In Graph G, the BFSI index of a is [1, 10, 1], l is [2, 2, 6], so we know that in the graph a can reach l in 5 hops, according to Theorem 2. At the same time, we can establish $a \rightarrow_3 i, m \rightarrow_4 g, b \rightarrow_3 l$.

3.4. BFSI-B index

In many graphs, the out-degrees and in-degrees follow different distributions. To verify whether this difference can influence the BFSIs index, we constructed the indices for some graphs and their reversed versions; that is, for each DAG G, we generate a DAG $G^T = (V^T, E^T)$, where E^T is the set of edges with its reversed directions. We note that the vertices of the normal and the reversed graphs present different placements. This occurs because, in reversing the edges of a DAG, due to the in-degree and out-degree distributions of its vertices, the number of successors (and predecessors) of each vertex changes, as well as the number of roots and leaves of the DAG. Obviously, the query $r(u, v, k)$ in the DAG G is equivalent to the $r(v, u, k)$ in the DAG G^T. But, as the vertices have different values in each index, the shortest paths covered by the index are different. In some graphs, the number of roots is so large that the construction time is too long. This situation requires the construction of the BFSI-B index on the reversed graph to limit the construction time. To construct the index on the reversed graph, we get the set of leaves, then we start a BFS from one of the roots (different from the original BFS); we reversed the graph by using the in-degree list. Using the BFS-tree, we use the min-post strat-

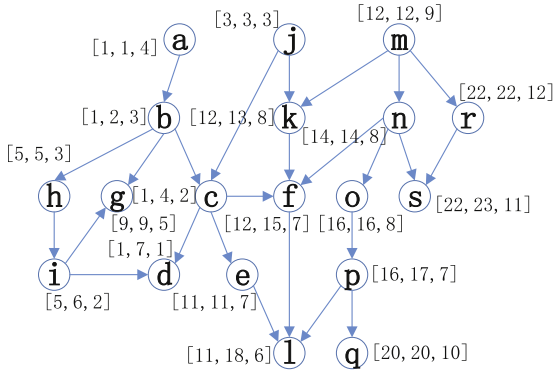

Fig. 3. BFSI-B index of G.

egy to create the BFSI-B index. The pseudo-code of the BFSI-B index construction algorithm is shown in Algorithm 3. The maximum level of the visited node is denoted as *maxlevel*, and the *prepost* is the post order of the last node.

For instance, consider the DAG of Fig. 3. It chooses the leaf vertex d and starts a BFS from d, generating the first BFS-tree; the *TLE* value can be generated from the BFS, $TLE(d) = 1, TLE(c, i) = 2, TLE(b, h) = 3, TLE(a) = 4$.

Next, we perform the min-post algorithm on the BFS-tree to get the first and the second value of the BFSI index, in Fig. 3, d is [1,7,1], c is [1,4,2], i is [5,6,2]; the algorithm continues until the min-post algorithm is complete. Then it chooses another root vertex g and repeats the steps until each node is visited. As shown in Fig. 3, the BFSI index of Graph G is presented. For example, the BFSI-B index of d is [1, 7, 1] and a is [1, 1, 4]; we can establish that, in the reverse graph, d can reach a in 3 hops, according to theorem 2. The shortest paths covered by the BFSI and BFSI-B, respectively, are different. Sometimes the BFSI-B index is more scalable than that of the BFSI when the number of roots is larger than the set of leaves.

4. K-hOP reachability query

Algorithm 4 is a pseudo-code that summarizes our query strategy. As mentioned earlier, for two vertices, u and v, the algorithm first checks whether u is equal to v, in which case the search u has reached v. In line 4, if $k = 0$ is true, we stop the search, because the search length exceeds k. In line 7, it checks the FELINE index; if $i(u) < i(v)$ is not true, we can immediately stop the search, in accordance with the FELINE-index rule. If $i(u) < i(v)$, no conclusion can be reached. In line 8, if $L_v \subset L_u$ is true, we can immediately stop the search; if $TLE(v) - TLE(u) < k$, the result is true, if $TLE(v) - TLE(u) > k$, the result is false. If we cannot stop the search in accordance with the FELINE index and BFSI index, we need to explore all vertices inside the region between u and v, recursively. In line 14, we always choose the node that has a small degree to enter in. If $outDeg(u, G) < inDeg(u, G)$, we search all the successors of u to check whether v can be reached from u within k hops; otherwise ,we check all the predecessors of v. This proved to accelerate the search efficiently.

Algorithm 3 BFSI-B index construction with G.

Input: (V, E), a DAG G
Output: (min, $post$ TLE), three vectors of size $|V|$
1: **for all** $v \in Leaves$ **do**
2: $TLE \leftarrow (v, maxlevel)$
3: add v into S
4: **while** $S \neq \emptyset$ **do**
5: pop up a node x from S
6: **for all** $(y, x) \in E$ **do**
7: **if** y is not visited **then**
8: //add(x, y) into BFS-tree
9: add(BFS-Tree, y, x)
10: visit y
11: $TLE \leftarrow (y, TLE(X) + 1)$
12: update *maxlevel*
13: add y into S
14: **end if**
15: **end for**
16: **end while**
17: update *prepost*
18: update *maxlevel*
19: min-post(BFS-tree, v, *prepost*)
20: **end for**

Algorithm 4 K-hop reachability query.

Input: k, $(u, v) \in V^2$, two vertices of the DAG
Output: $r(u, v, k)$, whether v is reachable from u within k hops
1: **if** $u = v$ **then**
2: return true
3: **end if**
4: **if** $k = 0$ **then**
5: return false
6: **end if**
7: **if** $i(u) < i(v)$ **then**
8: **if** $L_v \subset L_u$ **then**
9: **if** $TLE(v) - TLE(u) \leq k$ **then**
10: return true
11: **end if**
12: return false
13: **end if**
14: **if** $outDeg(u, G) \leq inDeg(u, G)$ **then**
15: **for all** $(u, w) \in E$ **do**
16: **if** kreachable $(w, v, k - 1)$ **then**
17: return true
18: **end if**
19: **end for**
20: **end if**
21: **if** $outDeg(u, G) > inDeg(u, G)$ **then**
22: **for all** $(p, v) \in E$ **do**
23: **if** kreachable $(u, p, k - 1)$ **then**
24: return true
25: **end if**
26: **end for**
27: **end if**
28: **end if**
29: return false

5. Experiments

We ran the experiments on different kinds of graphs, including small and dense graph and large graphs; then, we made comparisons between our method and the state-of-the-art k-reach method.

5.1. Datasets

We used five small (that is, fewer than 100,000 vertices) and dense real-world DAGs: Arxiv, Citeseer, Go, Pubmed and Yago. We also used five large, real-world DAGs (sparse and dense): Cit-Patents, Citeseerx, Uniprot100m, Go-uniprot and Uniprot150m. Table 2, below, shows some characteristics of the graphs, such as the number of vertices and edges, e-dia (effective diameter, rather than full diameter), the number of root nodes and leaf nodes.

Table 2
Datasets.

Graph	$\|V\|$	$\|E\|$	*e-dia*	Roots	Leaves
Arxiv	6000	66,707	5.48	961	624
Yago	6642	42,392	6.57	5176	263
Go	6793	13,361	10.92	64	3087
Pubmed	9000	40,028	6.32	2609	4702
citeseer	10,720	44,258	8.36	4572	1868
Uniprot22m	1,595,444	1,595,442	3.3	1,556,157	1
Cit-patents	3,774,768	16,518,947	10.5	515,785	1,685,423
citeseerx	6,540,401	15,011,260	8.4	567,149	5,740,710
Go-uniprot	6,967,956	34,770,235	4.8	6,945,721	4
Uniprot100m	16,087,295	16,087,293	4.1	14,598,959	1
Uniprot150m	25,037,600	25,037,598	4.4	21,650,056	1

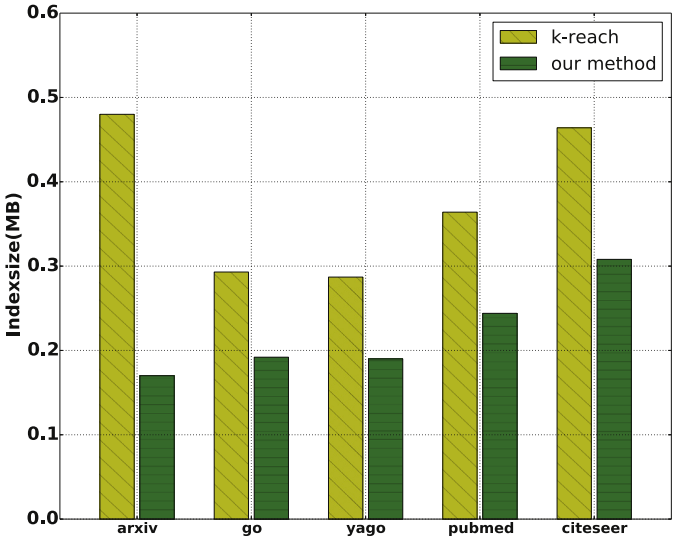

Fig. 4. Index size on small graphs.

The effective diameter is more robust than the full diameter, and has been successfully used to analyse the topological properties of Internet graphs.

5.2. Experimental set-up

We empirically evaluate our method in respect of different kinds of data. The experiments are performed on an Intel(R) Xeon(R) CPU (8-core, 2.60GHz) machine with 16GB RAM. Both our method and *k*-reach algorithm are implemented in C++; the compiler is g++ 4.8.5. We generated a specific set of queries for each graph and randomly selected 500 K pairs of vertices for each set. We submit the respective set of queries to each dataset, and all results shown are the average values of 10 executions. With these sets of queries, we are able to estimate the performance in terms of query times, construction times of the indices, and their size.

5.3. Index size

The index size of the *k*-reach and the proposed method (BFSI-B) is shown in Fig. 4 in the small graph. Interestingly, there is an increasing linear relationship between the index size of suggested method and the number of vertices. This is because the index of our method is composed of all the labels of the vertices, and each of the vertices has the same space size of label. However, it is not difficult to see that the *k*-reach index size of the Arxiv is surprisingly high. The Arxiv is so dense that the vertex cover of the *k*-reach method is very large. In all the graphs in Fig. 4, the index size of the (BFSI-B) is smaller than that of the *k*-reach method.

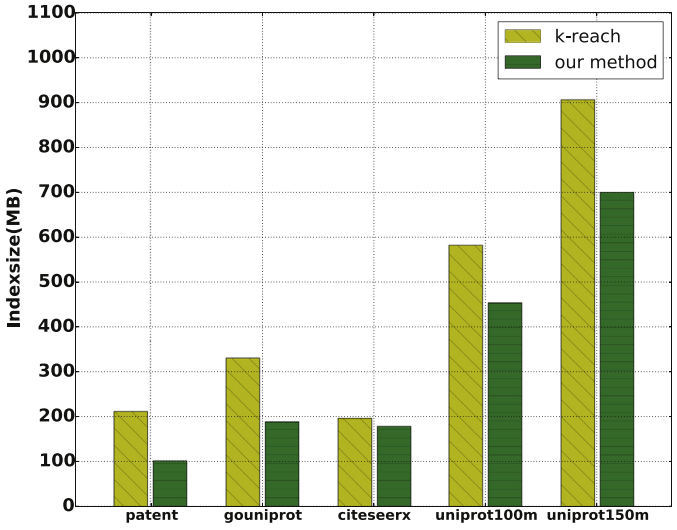

Fig. 5. Index size on large graphs.

Therefore, the scalability of our method is superior to that of the *k*-reach method, especially in the dense graph.

As shown in Fig. 5, the index size of *k*-reach and our method in respect of very large graphs is presented. For all the very large graphs in Fig. 5, the index size comparison between our method and the *k*-reach method is the same with that of the Fig. 4. In graph Uniprot150m, the index size of our method is 22% less than that of the *k*-reach method. It is not hard to see that, in graph Citeseerx, the difference between the two methods is very small, because the graphs are very sparse. In the large dense graphs, such as Uniprot100m, Uniprot150m and Go-uniprot, our method has obvious advantages over the k-reach method.

5.4. Query time

As shown in the Fig. 6, the query time is measured between the *k*-reach and our method in the small and dense graphs. For all the small and dense graphs in Fig. 6, the query time of our method is more stable and even less than that of the *k*-reach method. However, the query time of the *k*-reach method changed considerably in different datasets. In dense graphs Pubmed and Citeseer, the performance of the *k*-reach method is very slow. We conclude that the query time of our method is superior to that of the *k*-reach method, especially in the dense graph.

As shown in Fig. 7, we measure the query time of the *k*-reach and our method in respect of very large graphs. Overall, the query time of our method is less than that of the *k*-reach method. The *k*-reach method has a longer query time than that of our method. In the graph patents, the *k*-reach method has a very slow query time, and many tested vertices require a BFS search to attain the result of the queries. Our method has a stable performance in the different large graphs, and can quickly negate the impossible queries in constant time. In the graph for Uniprot100m and Uniprot150m, the query times showed a smaller difference, but the index sizes were much larger than for our method.

Table 3 gives the complete results of the query time of our method, including the cases *k* from 2 to 6. The results show that the query time of our method is always less than the *k*-reach method. The query time of k=6 is the slowest. In the dense graph arxiv, the difference between different *k* is very significant. The increase of the value of *k* makes the search space increase greatly, which increases the query time.

Fig. 8 gives the query time of the *k*-reach method on all the datasets, including the cases *k* from 2 to 6It is not hard to see that

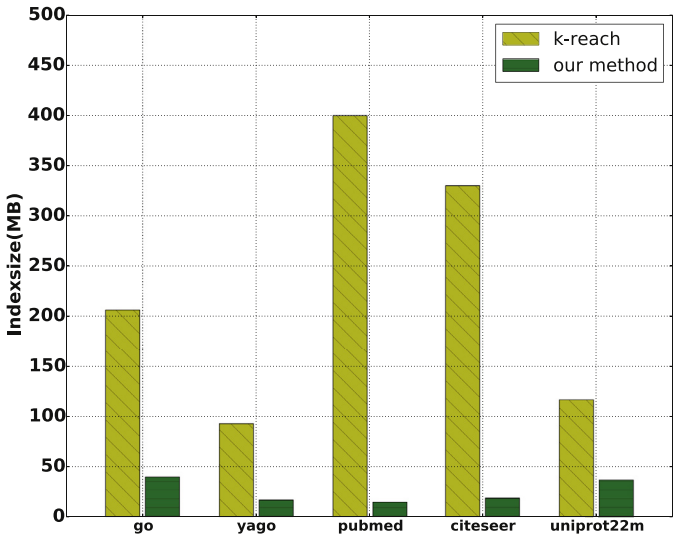

Fig. 6. Query time on small graphs.

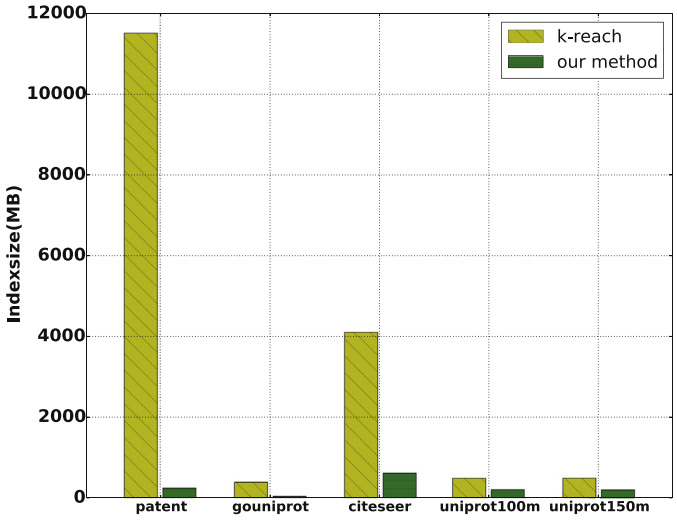

Fig. 7. Query time on large graphs.

Table 3
Query time (MS) by our method.

Graph	k=2	k=3	k=4	k=5	k=6
Arxiv	216	851	2537	5968	9797
Go	23	28.5	33.1	36.7	38.6
Yago	14.7	14.7	15.1	15	14.7
Pubmed	14.2	14.1	14.2	14.1	14.3
Citeseer	16.5	16.8	17	17.2	17.3
Unipro22m	31.7	31.5	31.6	31.6	31.6
Go-uniprot	38.1	39.7	39.3	38.4	41.8
Citeseerx	256.1	446.4	583.2	672.3	722.4
Cit-patents	263	263	264.1	264.5	265.1
Unprot100m	219.6	218.3	218.3	218.2	218.2
Unprot150m	204.2	203.8	203.8	203.8	203.8

the query time of our method is superior to the k-reach method in all cases. After the experiments in both the dense graphs and the very large graphs, we found that our method has a smaller index size, especially in the large dense graphs, and the query time of our method is much faster than for the k-reach method, and our method is more scalable than is the k-reach method.

6. Conclusion

In this paper, we propose a new method to answer the k-hop reachability query. We have shown that our approach achieves both small index sizes and short query times. Our method is a refined online-search method, consisting of two stages: index and

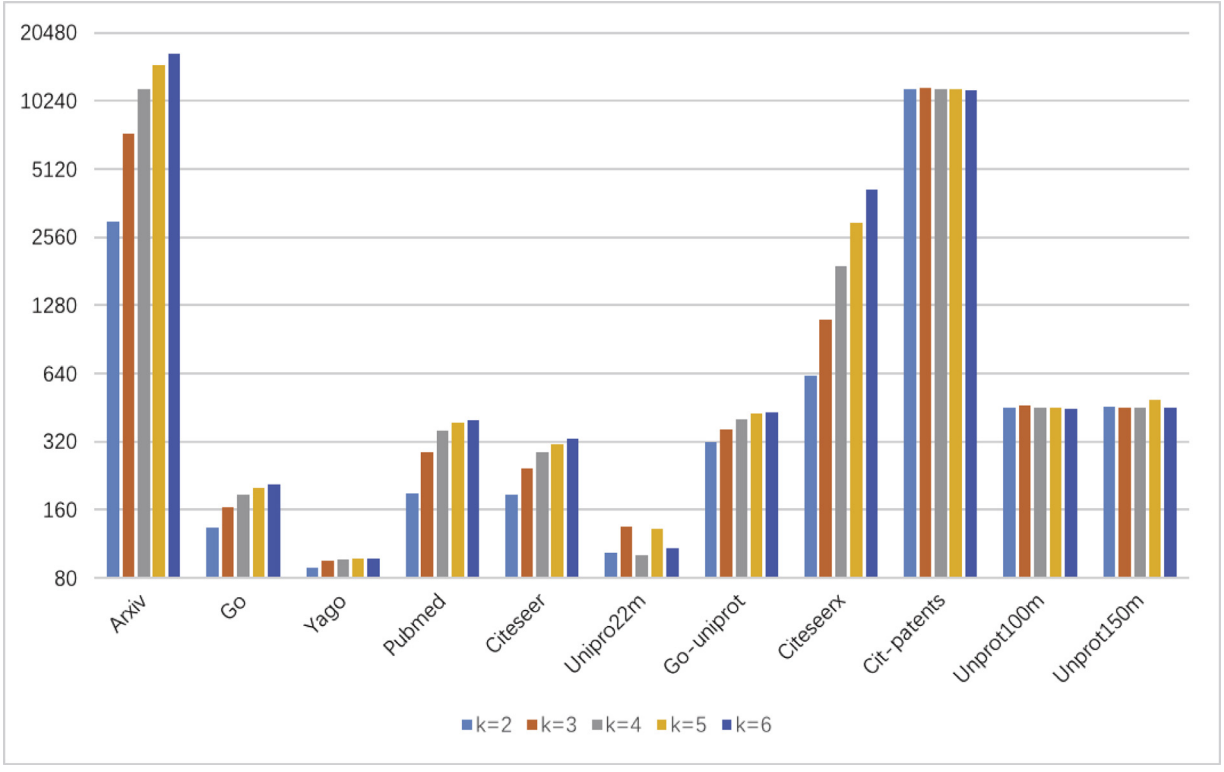

Fig. 8. Query time (MS) by K-reach.

query. Our method constructs the FELINE index and the BFSI index, which can effectively decrease the query space of the online search. The query time of our method is 10 times shorter than the state-of-the-art *k*-reach method, and our index size is smaller than the *k*-reach method as well, especially in respect of the large dense graphs. Experimental results from different kinds of graphs show the effectiveness of the proposed method.

In future work, we plan to improve the BFSI index to cover the larger shortest path to accelerate the query resolution further. Currently, our method cannot cover all the shortest paths in the graph. Because the online search approaches rely on the cover rate of the shortest path, if we can get the higher cover rate of the shortest path, the query time will be improved.

References

[1] A. Kremyzas, N. Jaklin, R. Geraerts, Towards social behavior in virtual-agent navigation, Sci. China Inf. Sci. 59 (11) (2016) 112102, doi:10.1007/s11432-016-0074-9.

[2] X. Wang, L.T. Yang, J. Feng, X. Chen, M.J. Deen, A tensor-based big service framework for enhanced living environments, IEEE Cloud Comput. 3 (6) (2016) 36–43, doi:10.1109/MCC.2016.130.

[3] J. Cheng, Z. Shang, H. Cheng, H. Wang, J.X. Yu, K-reach: who is in your small world, Proc. VLDB Endow. 5 (11) (2012) 1292–1303, doi:10.14778/2350229.2350247.

[4] J. Cheng, Z. Shang, H. Cheng, H. Wang, J.X. Yu, Efficient processing of K-hop reachability queries, VLDB J. 23 (2) (2014) 227–252, doi:10.1007/s00778-013-0346-6.

[5] T. Akiba, Y. Iwata, Y. Yoshida, Fast exact shortest-path distance queries on large networks by pruned landmark labeling, in: Proceedings of the 2013 ACM SIGMOD Interanational Conference on Management of Data, New York, USA, 2013, pp. 349–360, doi:10.1145/2463676.2465315.

[6] M. Qiao, H. Cheng, L. Chang, J.X. Yu, Approximate shortest distance computing: a query-dependent local landmark scheme, IEEE Trans. Knowl. Data Eng. 26 (1) (2014) 55–68, doi:10.1109/TKDE.2012.253.

[7] E. Cohen, E. Halperin, H. Kaplan, U. Zwick, Reachability and distance queries via 2-Hop labels, SIAM J. Comput. 32 (5) (2003) 1338–1355, doi:10.1137/S0097539702403098.

[8] R. Jin, N. Ruan, Y. Xiang, V. Lee, A highway-centric labeling approach for answering distance queries on large sparse graphs, in: Proceedings of the 2012 ACM SIGMOD Interanational Conference on Management of Data, Scottsdale, USA, 2012, pp. 445–456, doi:10.1145/2213836.2213887.

[9] T.L. Wong, Answering reachability queries on incrementally updated graphs by hierarchical labeling schema, J. Comput. Sci. Technol. 31 (2) (2016) 381–399, doi:10.1007/s11390-016-1633-7.

[10] Y. Chen, Y. Chen, Decomposing DAGs into spanning trees: a new way to compress transitive closures, in: Proceedings of the 27th International Conference on Data Engineering, Hannover, Germany, 2011, pp. 1007–1018, doi:10.1109/ICDE.2011.5767832.

[11] S.J. Van Schaik, O. de Moor, A memory efficient reachability data structure through bit vector compression, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, Athens, Greece, 2011, pp. 913–924, doi:10.1145/1989323.1989419.

[12] F. Wei-Kleiner, Tree decomposition-based indexing for efficient shortest path and nearest neighbors query answering on graphs, J. Comput. Syst. Sci. 82 (1) (2016) 23–44, doi:10.1016/j.jcss.2015.06.008.

[13] Y. Chen, Y. Chen, An efficient algorithm for answering graph reachability queries, in: Proceedings of the 2008 IEEE International Conference on Data Engineering, Cancun, Mexico, 2008, pp. 893–902, doi:10.1109/ICDE.2008.4497498.

[14] R. Jin, Y. Xiang, N. Ruan, H. Wang, Efficiently answering reachability queries on very large directed graphs, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, Canada, 2008, pp. 595–608, doi:10.1145/1376616.1376677.

[15] R. Jin, Y. Xiang, N. Ruan, D. Fuhry, 3-hop: a high-compression indexing scheme for reachability query, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, Providence, USA, 2009, pp. 813–826, doi:10.1145/1559845.1559930.

[16] S. Trißl, U. Leser, Fast and practical indexing and querying of very large graphs, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, Beijing, China, 2007, pp. 845–856, doi:10.1145/1247480.1247573.

[17] H. Yıldırım, V. Chaoji, M.J. Zaki, GRAIL: a scalable index for reachability queries in very large graphs, VLDB J. Int. J. Very Large Data Bases 21 (4) (2012) 509–534, doi:10.1007/s00778-011-0256-4.

[18] R.R. Veloso, L. Cerf, W. Meira Jr, M.J. Zaki, Reachability queries in very large graphs: A fast refined online search approach, in: Proceedings of the 2014 International Conference on Extending Database Technology, Athens, Greece, 2014, pp. 511–522. doi: 10.1.1.676.9982

[19] R. Jin, N. Ruan, S. Dey, J.X. Yu, SCARAB: scaling reachability computation on large graphs, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, USA, 2012, pp. 169–180, doi:10.1145/2213836.2213856.