Min-Forest: Fast Reachability Indexing Approach for Large-Scale Graphs on Spark Platform

Liu Yang[1], Tongyong Liu[1(✉)], Zhigang Hu[1], Zhifang Liao[1], and Jun Long[2]

[1] School of Software, Central South University, Changsha 410075, China
{yangliu, tongyongliu, zghu, zfliao}@csu.edu.cn
[2] School of Information Science and Engineering, Central South University, Changsha 410075, China
jlong@csu.edu.cn

Abstract. Reachability query is an important graph operation in graph database which answers whether a vertex can reach another vertex through a path over the graph, and it is also fundamental to real applications involved with graph-shaped data. However, the increasingly large amount of data in real graph database makes it more challenging for query efficiency and scalability. In this paper, we propose *Min-Forest* approach to handle with reachability problem in large graphs. We present *Min-Forest* structure to transfer and label the original DAG, and introduce a 4-tuple labeling scheme to construct index for each vertices, which integrate interval labels for trees and non-tree labels. We design efficient reachability query algorithms for *Min-Forest* approach on the Cloud Platform of Spark. The experiment results show that query time of *Min-Forest* approach is also on average about 10^{-4} ms for large dense graphs, and query time and index construction time of our approach are linear for both sparse graphs and dense graphs. It can answer reachability queries much faster than the state-of-art approaches on real graphs database, especially on large and dense ones.

Keywords: *Min-Forest* · Reachability query · Large-scale graphs GraphX

1 Introduction

Reachability query is one of most important research in graph processing, especially in large-scale graph processing, which is widely used in Semantic Web, such as Light-weight Service [1], semantic query [19] and Semantic Mining [2], knowledge ontology, biological network and social network. We know that a directed graph can always be transformed into a directed acyclic graph (DAG) by coalescing strongly connected components into vertices, and the reachability query of the original graph can be answered on DAG [3]. Let $G = (V, E)$ be the DAG with n vertices ($n = |V|$) and m edges ($m = |E|$), and a reachability query ($u \rightarrow v?\ u, v \in V$) is to answer if there exists a path $(u, v) = (v_1, v_2, \ldots, v_p)$ in G where (v_i, v_{i+1}) is an edge in E, for $1 \leq i < p$, $u = v_1$, and $v = v_p$. However, the recent dramatically increasing graph data poses new

© Springer International Publishing AG, part of Springer Nature 2018
H. Jin et al. (Eds.): ICWS 2018, LNCS 10966, pp. 437–454, 2018.
https://doi.org/10.1007/978-3-319-94289-6_28

challenges for reachability computing. For example, the Linked Open Data (LOD) project [4] has contained 2973 open datasets and more than 149.4 billion triplets up to August 2017. Therefore, some graph indexing approaches were proposed to improve the efficiency of reachability query.

2 Related Work

There are two kinds of graph indexing approaches according to the size of graph data:

(1) the approaches for small-scaled and medium-scaled graph with below 1 million vertices, including Chain-Cover [5, 6], Tree-Cover [7–10], 2-Hop and 3-Hop [11–13].
(2) the approaches for large-scaled graph with above 1 million vertices, including Refined Online Search [14–16], and Bloom Filter Labeling [17]. All these approaches are trade-off between online processing cost and the offline processing cost, while online processing cost is reachability query time, and offline processing cost contains index construction time and index size.

The above approaches present different reachability query algorithms based on different indexing approaches. However, there are several problems:

(1) The balance between reachability query time, index construction time and index size of graph: those algorithms are trying to speed up the query answering time while reducing the index construction time with a reasonable index size.
(2) The scalability bottleneck for handling massive graphs: some reachability algorithms cannot scale to very large real-world graphs.
(3) The limitation of platform: most of the algorithms are implemented in C++ based on the Standard Template Library (STL), and they have not been extended to Cloud Platform [18], which have significant advantages for large-scale data processing.

In this paper, we propose *Min-Forest* by using forest structure index to prune the search space of original graph, so as to speed up query time. In addition, *Min-Forest* algorithms are implemented on the Cloud platform of Spark to increase scalability for large-scale graph. The main idea of *Min-Forest* is as follows:

(1) (*Min-Forest*) The original DAG is divided into a Forest structure with minimal number of trees (*Min-Forest*) by cutting some edges from the original graph, and then each tree in *Min-Forest* carries the major reachability information of the whole tree vertices, and the deleted edges called *Non-Forest Edge* carry the relation information between trees. Each *Non-Forest* Edge in the original DAG is the deleted incoming edge of ending vertex in the *Min-Forest*, so as to ensure the in-degree of ending vertex is at most 1 in the *Min-Forest*.
(2) (Interval Labeling) Each vertex in *Min-Forest* is assigned an interval labels (X, Y), where X is the tree id in the forest, and Y is the vertex position on the tree. Therefore, the positive reachability query between the two vertices in the *Min-Forest* can be immediately answered by the interval labels.

(3) (Start Vertex Set of *Non-Forest* Edge) As for each *Non-Forest* Edge carrying the relation information between trees, we record all the starting vertices of it in order to achieve the reachability query, and name them Start Vertex Set of *Non-Forest* Edge.

Therefore, the positive reachability query between any two vertices in the original DAG can further be answered by Start Vertex Set of *Non-Forest* Edge and Nearest Ancestor Vertex of *Non-Forest* Edge, because they connect trees in *Min-Forest*.

The rest of the paper is organized as follows. In Sect. 3, we introduce how to construct a *Min-Forest* from original DAG. In Sect. 4, we assign interval labels of each vertex of *Min-Forest*, which label the tree and the branch each vertex belongs to. In Sect. 5, we assign connectivity label for each vertex of *Min-Forest* to ensure the connectivity of original DAG. In Sect. 6, we present reachability query approach of *Min-Forest*, and describe the corresponding query algorithm and its optimized query algorithm for special graph with redundant data. We analyze experiment results of four kinds of graphs from query time, index size and construction time, including small sparse graphs, large sparse graphs, small dense graphs and large dense graphs. We also analyze scalability of *Min-Forest*.

3 Construction of *Min-Forest*

Our study is motivated by a list of tree-based approaches, and we propose *Min-Forest* consisted by tree-shape subgraphs to cover a DAG G.

Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are two trees in $G = (V, E)$, we use $T_1 \cap T_2$ to denote the intersection of Tree T_1 and Tree T_2 with vertices and edges, and $T_1 \cup T_2$ to denote the union of T_1 and T_2 with vertices and edges. We use E_{T1} to denote the edge set of T_1, $E_{T1 \cup T2}$ to denote the union of edges of T_1 and T_2, and $E_{T1 - T2}$ to denote the complement of edges in T_1 and T_2. We define *Non-Forest Edge* based on the above terminology.

Definition 1 (Forest). Given a DAG $G = (V, E)$, and $(T_1, T_2, \ldots, T_n)$ are divided multiple trees by deleting some edges from G, where $T_i \cap T_j = \varnothing (i \neq j, i, j \in (1, n))$, and $F_G = T_1 \cup T_2 \cup \ldots \cup T_n = (V^*, E^*)$ $(V^* = V, E^* \in E \wedge E^* = E_{T1 \cup T2 \cup \ldots \cup Tn})$ is called the **Forest** of G, and $S = E - E^* = \{(u, v) | (u, v) \in E \,\&\&\, (u, v) \notin E^*\}$ is called ***Non-Forest* Edge Set**.

As an example, Fig. 2 represents a decomposed Forest F_G from DAG G in Figs. 1, and 2(a) and (b) show two trees of T_1 and T_2 in F_G. ***Non-Forest* Edge Set** is generated in the decomposition process of G, $S = E - E_{T1 \cup T2} = \{(4, 6), (4, 7), (5, 9), (6, 9), (7, 10), (9, 8), (9, 15), (10, 12), (10, 13), (11, 10), (12, 13), (12, 16), (13, 16), (14, 13), (15, 16), (16, 17)\}$.

There may be several possible forests as the results when converting a DAG to Forest, and different forests may contain different number of trees. We know the worst result is that each vertex in the original DAG is converted into a tree in the forest, so the number of trees of the forest is the number of vertices in the original DAG. Therefore, we propose *Min-Forest* to define as the least number of trees in Forest as possible while converting the original DAG to Forest.

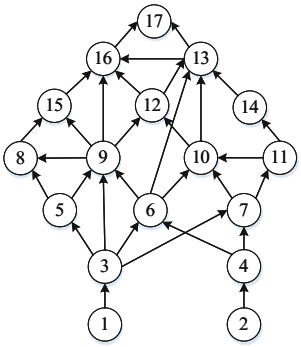


Fig. 1. DAG *G*.

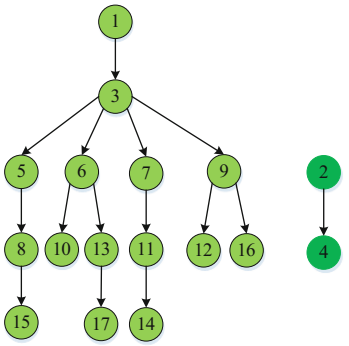


Fig. 2. Decomposed Forest F_G from *G*.

Lemma 1 (*Min-Forest* Criterion). Given DAG *G*, if the number of vertices with in-degree 0 in *G* is *N*, the minimal number of trees in *Min-Forest F* is *N*.

Proof: The vertices with in-degree 0 can only be the root vertices in the tree. Suppose there are *N* vertices with in-degree 0, then there are at least *N* root vertices when converting *G* to *Min-Forest*. That is, the number of trees in *Min-Forest* is *N*.

From Lemma 1, the converting process from *G* to *Min-Forest* is as following: (1) Traverse *G* to find out *N* vertices with in-degree 0, (2) and then delete incoming edges from the vertices with in-degree more than 1 and just keep one incoming edge, to ensure in-degree of each vertex is no more than 1, (3) finally, we get *Min-Forest FG* with *N* trees and *E** edges, and the set of deleted edges S, where $E = E^* + S$.

According the converting process, we design the algorithm of *Min-Forest* Construction, and we concern the scalability and space-saving of the algorithm.

Algorithm 1: ConstructMinForest(*G*)

Parameter: *G* is the DAG

Parameter: (*srcId*, *dstId*) is the data type of edge

```
1: savedEdgeRDD ← G.aggregateMessages(
2:   sendMsg ←{triplet =>triplet.sendToDst(srcId, dstId)}   //map
3:   mergeMsg ← {(a, b) => if(a.Id < b.Id) a else b}        //reduce
4: )
5: .map( _ => (srcId, dstId))
6: .map( _ => Edge(srcId, srcId))                          //create edge object
7: G ← Graph.fromEdges(savedEdgeRDD, 1)
```

Using Algorithm 1, we get the converted *Min-Forest* with *Tree1* and *Tree2* in Fig. 2 from the original DAG *G* in Fig. 1. We can also get *Non-Forest* Edge set during this converting process. Two sets of edges are generated by executing *Map* operation twice, including *E*-the edge set of *G* and *E**-the edge set of the converted *Min-Forest*, and then *Non-Forest* Edge set is the complement of *E* and *E**.

Algorithm 1 of constructing *Min-Forest* has two advantages. The first is that the forest structure based on tree structure helps to increase the reachability query. The

second is that the integrated functions of Spark can filter out all the isolated vertices automatically, which reduce the difficultly of dealing with large-scale graph data.

4 Interval Labeling of *Min-Forest*

In Sect. 3, we get *Min-Forest* with trees and *Non-Forest* Edge set when converting the original *G* to *Min-Forest*. We will introduce how to label each vertex with a 3-tuple to cover these two kinds of information by *Min-Forest* in this section.

The beginning two elements of 3-tuple cover the position of vertex in *Min-Forest*, which help to answer the reachability query among trees in *Min-Forest*, and the last element cover the connection information between trees in *Min-Forest*. These three elements of 3-tuple can compress the full transitive closure of *G* to answer the reachability query of the original *G*.

4.1 Interval Label of Vertex in *Min-Forest*

In this section, we present how to assign the vertex with the interval label of the beginning two elements. Similar to Path-Tree approach [8], we also perform a Depth-First Search (DFS) to create an *X* label for each vertex, which denotes the tree ID in the order of the *Min-Forest* by DFS, and create a *Y* label, which denotes the branch ID in the whole *Min-Forest*. By utilizing the interval label (*X*, *Y*), we can easily answer the reachability query among the *Min-Forest*.

A. The DFS order of Min-Forest

We assign *X* label of the interval label (*X*, *Y*) for each vertex by DFS algorithm. The procedure of the algorithm is as following: (1) Find the vertices with in-degree 0 in *Min-Forest*, which are root vertices of the trees in *Min-Forest*. For example, the vertices in the set of {1, 2} are root vertices with in-degree 0, (2) and then perform DFS traversal from the root vertex set sequentially, until all vertices in *Min-Forest* are visited, (3) finally, order all vertices in *Min-Forest* by DFS traversal order, and then label the order as their *X*. As for the isolated vertices deleted during the process of generating *Min-Forest*, we label 0 as their *X* (Fig. 3).

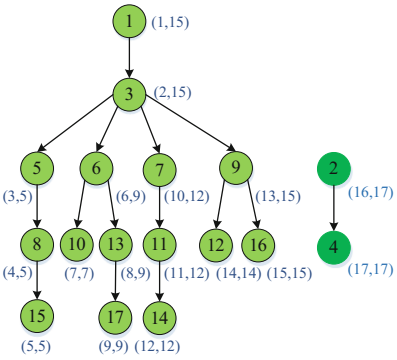


Fig. 3. Interval label of *Min-Forest*.

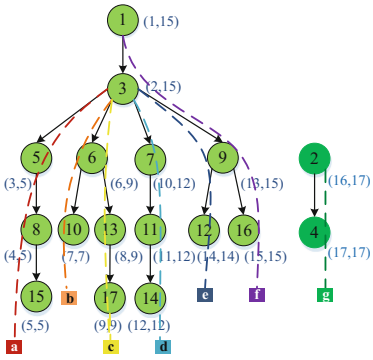


Fig. 4. Branches of *Min-Forest*.

Lemma 2. For any two vertices u, v in *Min-Forest*, if u can reach v, then $u.X < v.X$.

Proof: Clearly, if u can reach v, DFS traversal will visit u earlier than v, and it turns to u only after visiting all v's neighbors. So, if u can reach v, $u.X < v.X$ based on DFS, but not vice versa.

B. The Branch Order of Min-Forest

Using X of interval label (X, Y), it can answer the reachability query between root vertex and its child vertices, but cannot answer the query between the child vertices below the same root vertex. Therefore, we assign Y of interval label (X, Y) to the vertex as the branch order of *Min-Forest* to solve this kind of reachability queries. The branches are just like the tree branches from the root, and these help to label different branches.

Definition 2 (Branch of *Min-Forest*). A *branch* is a subdivision of *Min-Forest* F_G that starts at the root vertex and explores vertices as far as possible along each edge until the leaf vertex with out-degree 0, and this subdivision path formed is a *branch* of F_G.

As shown in Fig. 4, the branches of *a*, *b*, *c*, *d*, *e*, *f*, *g* are seven branches of *Min-Forest* F_G. We observe that different branches never join back up together, each root vertex or father vertex may belong to different branches, and each leaf vertex can only belongs to one branch. For example, root vertex *6* belongs to branch *b* and branch *c*, but leaf vertex *16* only belongs to branch *b*. Therefore, we design a post-order traversal algorithm for fast assigning the branch order of Y to each vertex in *Min-Forest*.

The procedure of the algorithm is as following: (1) Initial the branch order of Y for each vertex. We initial the DFS order as the branch order for the leaf vertex with out-degreee 0, and initial 0 as the branch order for non-leaf vertex with out-degree more than 0, (2) and then post-order traverse vertices in *Min-Forest*, that is, it always first visit the child vertices from left to right, and then visit the father vertices of them, (3) finally, label the branch order of Y for each vertex during post-order traversal. If the branch order of father vertex is less than that of its child vertices, then it is updated with the branch order of its child.

For example, vertex *6* belongs to branch *b* and branch *c*, and it is the father of vertex *10* and vertex *13*, with the branch order *7* and *9* respectively, so vertex *6* is assigned with the branch order of *9*, which is the maximal branch order of vertex *10* and vertex *3*. From Fig. 4, we also note that the vertices below the same father vertex belong to different branches, and their interval label (X, Y) are mutually exclusive. The branch order of root vertex is always labeled with the maximal branch order among all its child vertices, and its interval label (X, Y) is larger than that of its child vertices. The *Pregel* iterative algorithm in Cloud Platform of Spark can easily label branch order of vertices of *Min-Forest*.

Algorithm 2 of *BranchVisit* shows the algorithm for assigning Y label of interval label (X, Y) for each vertex in *Min-Forest* by post-order traversal. Figure 4 shows the Y label based on the post-order traversal algorithm and the branch order of isolated vertices is labeled with 0.

Algorithm 2: BranchVisit (*adjList*, *N*, *startVertices*)

```
1:  visited(0) to visited(N-1) ← false
2:  branchArray ← null
3:  foreach vertex v
4:    if ( outDegree = 0)
5:      branchArray(v) ← v.X  // X is DFS order
6:    else
7:      branchArray(v) ← 0
8:  for ( i ← 0 to startVertices.length )
9:    if ( visited(startVertices(i)) = false )
10:     LRD( startVertices(i) )
11:   end if
12: end for
  Procedure LRD(v)
1:  if ( adjList(v) is null )
2:    for ( i ← 0 to adjList(v).size )
3:      v' ← adjList(v)(i)
4:      if ( visited(v') = false )
5:        LRD(v')
6:      end if
7:    end for
8:    visited(v) ← true
9:    for ( j ← 0 to adjList(v).size )
10:     if ( v.Y < adjList(v)(j).Y )  //Y is Branch order
11:       v.Y ← adjList(v)(j).Y
12:     end if
13:   end for
14:  else
15:   visited(v) ← true
16:  end if
```

Lemma 3. For any two vertices u, v in *Min-Forest*, if u can reach v, then $u.Y \geq v.Y$.

Proof: (Case 1:) if u only belongs to the same branch of v belongs to, then $u.Y = v.Y$. (Case 2:) if u also belongs to different branches besides the branch of v belongs to, u is the root vertex with the maximal branch order of all its child vertices based on the post-order traversal in Algorithm 2, then $u.Y \geq v.Y$. Combining both Case 1 and Case 2, we prove our result.

4.2 Connectivity Between Vertices in *Min-Forest*

We map the connectivity between vertices in *Min-Forest* to a two-dimensional space, according to the interval label (X, Y) of vertices. As shown in Fig. 6, X-axis represents vertex's DFS order, and Y-axis represents the vertex's branch order in *Min-Forest*. As we know, for any two vertices u, v in *Min-Forest*, if u can reach v, then $u.X < v.X$ && $u.Y \geq v.Y$, based on Lemmas 2 and 3, that is, v is located at the lower right of

u corresponding to the two-dimensional space. Therefore, we use two-dimensional map of *Min-Forest* to express the interval label (X, Y) of each vertex, and it also reflects the reachability between vertices. This approach is similar to the labeling approach of Path-Tree [8].

Lemma 4. For any two vertices u, v in *Min-Forest*, u can reach v if and only if $u.X < v.X \wedge u.Y \geq v.Y$.

Proof: First, we easily prove $u \rightarrow v \Rightarrow u.X < v.X \wedge u.Y \geq v.Y$ based on Lemmas 2 and 3. Second, we prove $u.X < v.X \wedge u.Y \geq v.Y \Rightarrow u \rightarrow v$. (Case 1:) if $u.Y = v.Y$, then u and v are on the same branch of *Min-Forest*, and we will visit u before v in DFS traversal because u is the ancestor of v ($u.X < v.X$), so we have $u \rightarrow v$. (Case 2:) This can be proved by contradiction. Let us assume u cannot reach v. Then if $u.Y > v.Y$, u is the root vertex in *Min-Forest* or u and v belongs to two different branches but under the same father vertex. However, if u is the root vertex in *Min-Forest*, then u can reach v obviously, this contradicts the assumption. If u and v are under the same father vertex, and $u.Y > v.Y$ means that it first visits v and then u according to post-order traversal in Algorithm.2, then we get $v.X < u.X$, a contradiction. Combining both cases 1 and 2, we prove our result (Fig. 5).

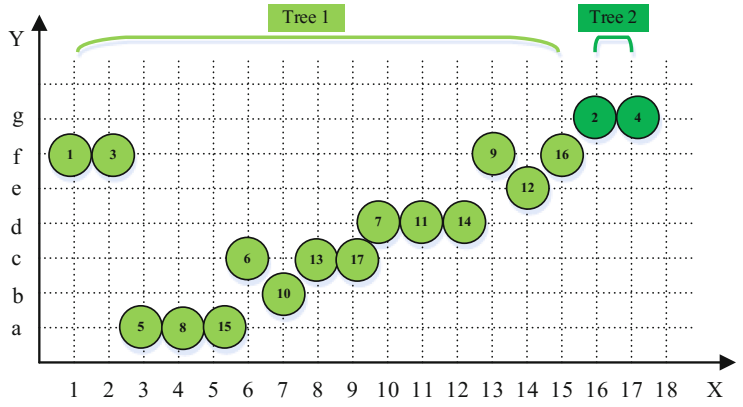


Fig. 5. Branches of *Min-Forest*.

5 Connectivity Labeling for Vertices in Trees of *Min-Forest*

In Sect. 4, we introduce how to judge the reachability of vertices among trees in *Min-Forest*. However, we cannot answer the reachability query of any two vertices in the original DAG, because we delete some edges as *Non-Forest* edges from the original

DAG when constructing *Min-Forest*, which weakens the connectivity of the original DAG. We will build the connections that *Non-Forest* edges break in this section.

From the definitions of Forest, *Min-Forest* and *Non-Forest* Edge Set, we know $E = E_F + S$, where *Non-Forest* Edge Set of S break the connectivity of the original DAG G. We define two concepts to study the connectivity besides *Min-Forest*: Start-Vertex Set of *Non-Forest* Edge (*SVS*) and Nearest Ancestor Vertex of *Non-Forest* Edge (*NAV*).

Definition 3 (*SVS*). Start-Vertex Set for *Non-Forest* Edge (*SVS*) is the staring vertex set for each ending vertex existing *Non-Forest* edge to connect them in the original DAG. As for the ending vertex v, $SVS_v = \{U | \ u_i, \ u_i \in V^* \ \&\& \ (u_i, v) \in S\}$.

We can get SVS of each vertex from *Non-Forest* Edge Set. Figure 6 shows *Non-Forest* edges of the original G by dotted lines. For instance, $SVS_8 = \{9\}$, and $SVS_9 = \{5, 6\}$. In particular, if there is no starting vertex for a *Non-Forest* edge v, we record SVS_v= *Null*.

Definition 4 (*NAV*). Nearest Ancestor Vertex of *Non-Forest* Edge (*NAV*) is the nearest ancestor vertex for each ending vertex existing *Non-Forest* edge to connect them in the tree of *Min-Forest* FG. Suppose the vertex set $\{u_1, \ldots, u_i, \ldots, u_n\} \rightarrow v$, where $u_i \in V^*$ && $SVS_{ui} \neq \varnothing$, $i \in [1, n]$, $n = |V^*| - 1$, then

$$NAV_v = \begin{cases} MAX\{u_1,\ldots,u_i,\ldots,u_n\}, i \in [1,n] \\ 0, \ i = 0 \end{cases} \tag{1}$$

For instance, $NAV_{15} = 8$, and $NAV_{11} = 7$. For any vertex v without *NAV*, we record *NAVv* = *Null*.

We can find *NAV* of each vertex by DFS traversal or BFS traversal with the same $O(n' + m')$ time, where n' and m' are the number of vertices and edges in *Min-Forest*. However, the practical results from later experiments show that DFS traversal is better than BFS traversal, especially to the large-scale data, because BFS traversal may result in pop operation and push operation consciously for some vertices. Therefore, we design an algorithm to find out *NAV* based on DFS traversal. The procedure of the algorithm is as following: (1) First, find out the ending vertices connected to the *Non-Forest* edges according to *Non-Forest* Edge Set. We know the connectivity of these ending vertices is weakened by the deleted *Non-Forest* edges they originally connect to. Figure 6 shows the deleted *Non-Forest* edges in red of the original *G*. Figure 7 shows the ending vertices connected to the *Non-Forest* edges in red of *Min-Forest*, (2) and then find out *NAV* of the ending vertex connecting to *Non-Forest* edge in *Min-Forest* based on DFS traversal. If the father vertex of vertex v is just the starting vertex connecting to a *Non-Forest* edge, then we record this father vertex as *NAV* of vertex v, that is, *NAV* of vertex v is its father vertex, else *NAV* of vertex v is its father's *NAV* recursively, that is, *NAV* of vertex v is its father's *NAV*. In particular, if

vertex v does not exist *NAV*, that means this vertex's ancestor vertices does not connect to any *Non-Forest* edges, then we record $NAVv = 0$.

From the above, we can see that vertex u may reach vertex v from one branch of the trees in *Min-Forest*, or through deleted *Non-Forest* edges. Although the deleted *Non-Forest* edges decrease reachability between vertices, the reachability also exists by using *SVS* and *NAV*. Therefore, reachability query ($u \rightarrow v$? $u, v \in V$) can be answered by the following 3-step judges: (1) check the scopes of interval label (X, Y) for u and v, (2) and then check the reachability from u to any vertex in $SVSv$, (3) and then check the reachability for u to $NAVv$.

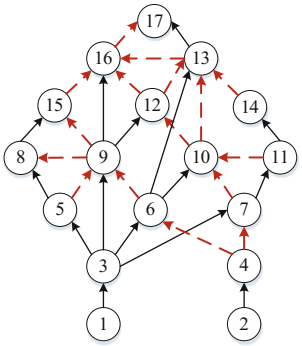


Fig. 6. Deleted edges of *Min-Forest* (Color figure online)

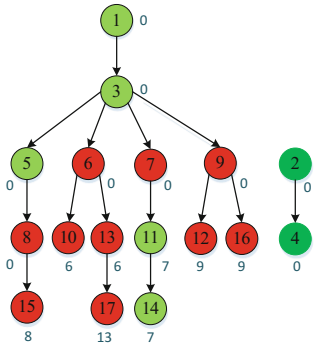


Fig. 7. *NAV* of *Min-Forest* (Color figure online)

For example, if there is a reachability query ($5 \rightarrow 9$?), we first check whether vertex 5 and vertex 9 are not on the same branch in *Min-Forest* from Fig. 7. If no, we cannot immediately conclude that vertex 5cannot reach vertex 9, because there actually exists an edge (5, 9) connecting vertex 5 and 9 in Fig. 6, which is just deleted when constructing *Min-Forest*. Therefore, we should use SVS_9 to judge whether there exist starting vertices connecting to vertex 9. From Fig. 6, we get $SVS_9 = \{5, 6\}$, so we can conclude $5 \rightarrow 9$.

Theorem 1. Given vertex u and vertex v in the *Min-Forest* FG with the original DAG G, u can reach v if and only if (1) u can reach v in F_G, *i.e.*, $u.X < v.X \wedge u.Y \geq v.Y$, (2) u can reach v by one of vertices in $SVSv$ directly or indirectly, (3) u can reach v by $NAVv$ indirectly.

Proof: The proposition of $u \rightarrow v$ is equivalent to the proposition that u can reach v or v's ancestors, and we can check v's ancestors by the following: (1) v's ancestors are direct ancestors of v in *Min-Forest* by Definition 2, that is, $u \rightarrow v$ is equivalent to $u.X < v.X \wedge u.Y \geq v.Y$; (2) v's ancestors are vertices in the set of $SVSv$ by Definition 3, that is, $u \rightarrow v$ is equivalent to $u \rightarrow w$, $w \in SVSv$; (3) v's ancestor is $NAVv$ by Definition 4, that is, $u \rightarrow v$ is equivalent to $u \rightarrow NAVv$.

In conclusion, we can answer reachability query by the interval label (X, Y) of *Min-Forest*, *SVS* or *NAV*, so we construct vertex index by Theorem 1, and build index

for each vertex by 3-tuple (X, Y, $NAVv$) and $SVSv$. Table 1 lists the index for each vertex in Fig. 1, and the construction time for index is $O(n' + m)$.

Table 1. DAG index for vertices in DAG G of Fig. 1

Node Id	(X, Y, NAV_v)	$SVSv$
1	(1, 15, 0)	*Null*
2	(16, 17, 0)	*Null*
3	(2, 15, 0)	*Null*
4	(17, 17, 0)	*Null*
5	(3, 5, 0)	*Null*
6	(6, 9, 0)	{4}
7	(10, 12, 0)	{4}
8	(4, 5, 0)	{9}
9	(13, 15, 0)	{5, 6}
10	(7, 7, 6)	{7, 11}
11	(11, 12, 7)	*Null*
12	(14, 14, 9)	{10}
13	(8, 9, 6)	{10, 12, 14}
14	(12, 12, 7)	*Null*
15	(5, 5, 8)	{9}
16	(15, 15, 9)	{12, 13, 15}
17	(9, 9, 13)	{16}

6 Reachability Query of *Min-Forest* Approach

To answer reachability query between two vertices of u and v, *Min-Forest* query processing presents a 3-step querying approach, and it answers whether u can reach v after the following checking:

(1) Whether the interval label (X, Y) of u includes that of v by *Min-Forest*, that is, to check whether $u.X < v.X \wedge u.Y \geq v.Y$.
(2) Whether u can reach w in $SVSv$, where w is one of ancestor vertices in $SVSv$. This step is an iterative process to find ancestor vertices for the vertices in $SVSv$.
(3) Whether u can reach v by $NAVv$.

Vertex u can reach vertex v if and only if (1) or (2) or (3) holds.

We design an efficient reachability query algorithm with $O(d)$, where d is the number of deleted edges for all vertices, and we know that $d \ll n$ (n is the number of vertices in DAG). The reachability query algorithm is described in Algorithm 3.

Algorithm 3 Query(u, v)

```
1:   flag ← false
2:   if ( getForestQuery(u, v) = true)
3:     flag ← true
4:   else
5:     if ( getDeletedInEdgeQuery (u, v) = true)
6:       flag ← true
7:     else
8:       flag ← getFatherVertexQuery(u, v)
9:     end if
10:  end if
   Procedure getForestQuery(u,v)
1:   flag ← false
2:   if ((u.X < v.X && u.Y >= v.Y) || u = v )
3:     flag ← true
4:   end if
5:   return flag
   Procedure getDeletedInEdgeQuery(u,v)
1:   flag ← false
2:   i ← 0
3:   while ( i < deletedInEdges.length && flag = false)
4:     flag = getQuery(u, deletedInEdges(v)(i))
5:     i ← i + 1
6:   end while
7:   return flag
   Procedure getFatherVertexQuery(u,v)
1:   flag ← false
2:   if (nearestAncestors(v) != 0)
3:     flag = getQuery(u, nearestAncestors (v))
4:   end if
5:   return flag
```

7 Reachability Query of *Min-Forest* Approach

We conducted an extensive set of experiments in order to evaluate the performance of *Min-Forest* in comparison with state-of-art reachability approaches. We also focused on three important measures for reachability query: query time, index size and construction time.

7.1 Reachability Query

Our experiments are conducted by a Dell desktop computer equipped with 4 Intel Core i5 CPUs at 3.20 GHz, and 36 gigabyte of main memory. We use OS CentOS7/Linux with Cloud Platform of Spark using version 2.1.0. We study other reachability algorithms implemented by C++, such as Grail [12], Path-Tree [8], and BFL [15], and in real environment we implement *Min-Forest* approach using Scala languagewith version 2.11.8 on Spark GraphX with version 2.1.0 on local mode, which is a special standalone cluster mode of Spark for computing, namely stand-alone mode.

As for dataset size, we consider dataset with less than 500,000 vertices as small dataset, and others as large dataset. As for dataset density, we consider dataset with the ratio of edges to nodes below 1.5 as sparse dataset, and others as density dataset. Therefore, we divide 22 datasets into 4 categories based on dataset size and dataset density which are Small Sparse Datasets, Small Dense Datasets, Large Sparse Datasets and Large Dense Datasets. We use the real graph datasets listed in Tables 2, 3, 4 and 5, which are the benchmark sets for recent reachability research. We get these graph datasets from the URI of https://code.google.com/arc-hive/p/grail/downloads, which is provided by Dr. Wei Hao of Chinese University of Hong Kong.

In Tables 2, 3, 4 and 5, columns of $|V|$, $|E|$ and $|E|/|V|$ record the number of vertices, edges, and ratio of edges to vertices in the original DAG, respectively. Columns of $|V_F|$ and $|E_F|$ record the number of vertices and edges in the *Min-Forest* converted from the original DAG, and the column of $|E_d|$ records the number of deleted *Non-Forest* edges. We order all four groups of datasets by the vertex number in datasets.

Table 2. Small sparse datasets.

DataSet	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	$\|V_F\|$	$\|E_F\|$	$\|E_d\|$
kegg	3617	4395	1.22	2619	2436	1472
amaze	3710	3947	1.06	2333	2152	1448
nasa	5605	6538	1.17	5605	5604	933
xmark	6080	7051	1.16	6080	6079	946
vchocyc	9491	10345	1.09	9491	9490	653
mtbrv	9602	10438	1.09	9602	9601	644
anthra	12499	13327	1.07	12495	12493	611
ecoo	12620	13575	1.08	12620	12619	731
agrocyc	12684	13657	1.07	12684	12683	725
human	**38811**	**39816**	**1.01**	**38811**	**38810**	**766**

Table 3. Small dense datasets.

DataSet	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	$\|V_F\|$	$\|E_F\|$	$\|E_d\|$
arxiv	6000	66707	11.12	5389	5039	61668
yago	6642	42392	6.38	1857	1466	40926
go	6793	13361	1.97	6777	6729	6632
pubmed	9000	40028	4.45	7437	6391	33637
citeseer	10720	44258	4.13	7598	6148	38110

Table 4. Large sparse datasets.

DataSet	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	$\|V_F\|$	$\|E_F\|$	$\|E_d\|$
citeseer	693947	312282	0.45	142749	80450	231832
unip_22 m	1595444	1595442	1	78562	39286	1556156
unip_100 m	16087295	16087293	1	2834057	1488335	14598958
unip_150 m	25037600	25037598	1	6329139	3387543	21650055

Table 5. Large dense datasets.

DataSet	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	$\|V_F\|$	$\|E_F\|$	$\|E_d\|$
cit-patents	3774768	16518947	4.38	3567328	3258983	13259964
citeseerx	**6540401**	**15011260**	**2.3**	**6452911**	**5973250**	**9038010**
go-uniprot	6967956	34770235	4.99	31833	21953	34769339

7.2 Query Time, Index Size and Construction Time

We use four groups of real datasets to verify the efficiency of *Min-Forest* approach. In this section, we report these four groups of experiment results to address query time, index size and construction time.

Min-Forest approach is implemented by *Spark GraphX*, which is different from other approaches implemented by STL C++. We mainly compare our *Min-Forest* approach with the state-of-the-art reachability approaches including Grail, Path-Tree and BFL+, which generally perform well as analyzed in [12, 15]. All experiments in [12] are performed on machine with x86_64 Dual Core AMD Opteron (tm) Processor 870, 32 GB RAM with Linux OS, and all experiments in [15] are performed on machine with 3.60 GHz Intel Core i7-4790 CPU, 32 GB RAM with Linux OS, so both of the machines perform better than ours.

Tables 6, 7, 8 and 9 show query time, construction time and index size for four groups of datasets. As for small sparse and dense datasets, we compare *Min-Forest* with Grail and PT, and as for large datasets, we compare with Grail and BFL+.

Table 6. Query time (ms), index size and construction time (ms) on small sparse datasets.

Dataset	Query time			Construction time			Index size
	Min-Forest	Grail	Path-Tree	Min-Forest	Grail	Path-Tree	Min-Forest
kegg	6.30E−03	1063	7.1	26	3.8	939	5.89
amaze	3.60E−03	764	7	18	3.8	818	5158
nasa	5.52E−04	26.5	7.8	6	6.3	126	6538
xmark	2.20E−04	79	8.2	16	7.5	263	7026
vchocyc	3.18E−04	49.6	7.2	132	12	201	10144
mtbrv	3.60E−04	49	7.2	135	12	208	10246
anthra	2.97E−04	49	8.5	240	16	268	13110
ecoo	2.99E−04	56	8	292	16	276	13351
agrocyc	3.47E−04	57	8	253	16.1	279	26093
human	**2.86E−04**	**80**	**14**	**4548**	**72**	**822**	**39577**

Query Time

As for sparse graphs in Tables 6 and 8, we note query time of *Min-Forest* approach is on average about 10^{-4} ms, not only for small sparse graphs, but also for large sparse graphs. However the query time of state-of-the-art reachability approaches at present is about 10 ms for sparse graphs.

Table 7. Query time (ms), index size and construction time (ms) on small dense Datasets.

Dataset	Query time			Construction time			Index size
	Min-Forest	Grail	Path-Tree	Min-Forest	Grail	Path-Tree	Min-Forest
arxiv	1.22E−01	575	24.4	12	21.7	9639	67668
yago	3.50E−04	46.9	13.8	7	18.2	512	47568
go	9.99E−04	51.4	11.6	9	9.5	220.9	13425
pubmed	2.92E−03	75.5	22.1	14	43.9	774	42637
citeseer	2.60E−03	82.6	24.5	11	43.1	751.5	48830

Table 8. Query time (ms), index size and construction time (ms) on large sparse datasets.

Dataset	Query time			Construction time			Index size
	Min-Forest	Grail	BFL+	Min-Forest	Grail	BFL+	Min-Forest
citeseer	4.79E−04	94.9	14.5	145	413	51	925779
unip_22 m	3.89E−04	132.3	14.7	273	595	61	3151600
unip_100 m	4.58E−04	186.1	14.4	2465	7472	857	30686253
unip_150 m	4.67E−04	183	14.7	4926	12083	1538	46687655

Table 9. Query time (ms), index size and construction time (ms) on large dense datasets.

Dataset	Query time			Construction time			Index size
	Min-Forest	Grail	BFL+	Min-Forest	Grail	BFL+	Min-Forest
cit-patents	7.30E−01	1579.9	257.671	3355	61911.9	1375	17034732
citeseerx	**5.3**	**12496.6**	**29.543**	**15875**	**19836**	**1022**	**15578411**
go-uniprot	3.82E−04	194.1	17.458	2487	32678.7	955	23645698

As for dense graphs in Tables 7 and 9, we divide datasets of them into two groups. The first group are datasets of *go*, *yago* and *go-uniprot*, and the second group are datasets of *arxiv*, *citeseer*, *pubmed*, *citeseerx* and *cit-patents* for better query we use two query methods, including the original query method described in algorithm 3 for the first group datasets and the improved query method by restriction of query access for the second datasets. We observe that the query time of the first group is also on average about 10^{-4} ms and the second group's is less than 5.3 ms.

Above all, the query time of our *Min-Forest* approach can be several orders of magnitude faster than other algorithms.

Index Size

We label for each vertex in *Min-Forest* with interval label of (X, Y), which denotes tree *ID* and branch *ID* of the vertex. We also assign the connectivity labels for each vertex in *Min-Forest*, including *SVS* and *NAV*. Therefore, the index size for *Min-Forest* approach is $|V| + |E_d|$, where V is the set of all vertices in *Min-Forest*, and E_d the deleted edge set from the original edge set E when constructing *Min-Forest*.

Note that $|E_d|$ is positively correlated to $|E|$, and we know $|E_d| < |E|$, so we get the index size as:

$$Index\,Size = |V| + \alpha|E|(0 < \alpha < 1) \tag{2}$$

We observe that the index size is positive correlative to $|V|$ and $|E|$, with the coefficient of 1 and α, respectively. Therefore, the index size of *Min-Forest* approach is not large and has good scalability.

Construction Time
We note that construction time increase with increasing density of datasets. As for small sparse datasets, construction time of Min-Forest is almost less than that of PT but almost 11 times higher than that of Grail. However, as for small dense datasets construction time of ours is always 23 times less than that of PT and always less than that of Grail. As for large datasets, construction time of Min-Forest is almost 2.1 times less than that of Grail and 2.4 times higher than that of BFL+, which can keep about 75 percent the pruning power even in the densest datasets. However, we know Min-Forest is several orders of magnitude faster than BFL+ on query time.

Above all, the construction time of our *Min-Forest* approach is not bad and is good scalable in dense graph and large graph.

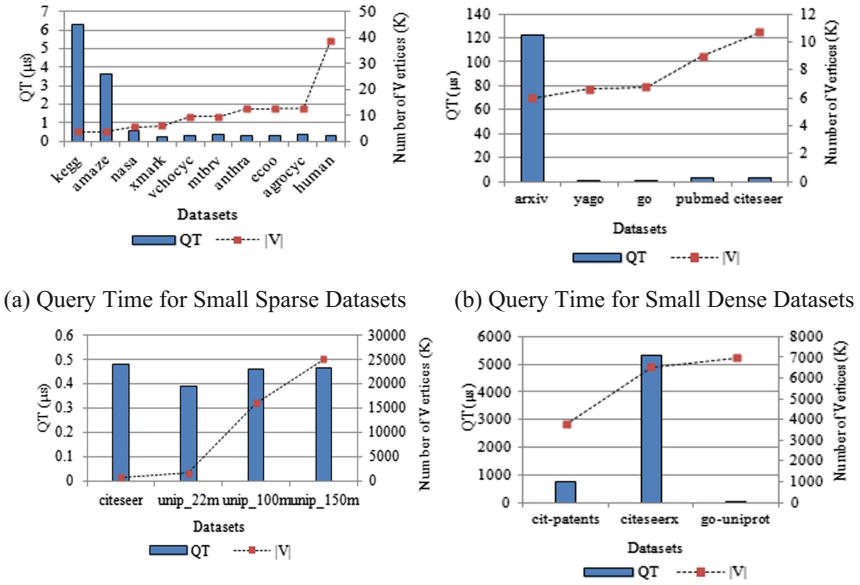
7.3 Scalability

In this section, we study the scalability for sparse graphs and dense graphs from the aspects of query time, according to the experiments results shown in Tables 2, 3, 4, 5, 6, 7, 8 and 9. We show the experiment results of query time for four kinds of datasets in Fig. 8.

Query time
As for the sparse datasets, we observe from Fig. 8(a) and (c) that query time doesn't increase with the number of vertices increasing, but decrease rapidly at first and tend to be stable then. And the stable value of query time is about 10^{-4} ms no matter small sparse datasets and large sparse datasets, of which the range of number of vertices is 5605 to 25037600. From it, we can see the scalability of *Min-Forest* approach is very good for the sparse datasets.

As for the dense datasets, query time isn't always linear growth with the number of vertices increasing. In terms of small dense datasets, the change trend of query time is decrease quickly at begin and slowly linear growth with the number of vertices after. As for large dense datasets, except exceptional data *citeseerx,* the query time is a downward trend with the number of vertices. Above all, the scalablity of *Min-Forest* for dense datasets is still good.

In summary, our experimental results on query time indicate that *Min-Forest* approach is scalable, especially in sparse datasets, since query time is always about 10^{-4} ms for sparse datasets, of which the range of number of vertices is 5605 to 25037600.



(a) Query Time for Small Sparse Datasets

(b) Query Time for Small Dense Datasets

(c) Query Time for Large Sparse Datasets

(d) Query Time for Large Dense Datasets

Fig. 8. Query time in four kinds of datasets

8 Conclusion

In this paper, we propose *Min-Forest* approach to solve large-scale reachability queries in large graphs. We present a 4-tuple labeling scheme to construct index of original DAG, with two tuples of interval labels for vertices in the same tree and two tuples of connecting labels for vertices in the different trees. We design algorithms for our *Min-Forest* approach by Scala and implement them on the Cloud Platform of Spark, which also help to speed up reachability query in large graphs. Our experiment results on four kinds of real datasets demonstrate that *Min-Forest* approach have the fastest query time and comparable index construct time compared with the state-of-art approaches, including Grail, Path-Tree and BFL+. Furthermore, the query time and index construction time of our approach are linear for both sparse graphs and dense graphs, and it performs quite well when graph are large and dense, so it is scalable and applicable to large-scale datasets. In the future, we plan to apply *Min-Forest* approach to the dynamic large graphs, and we will further study the reachability problem in query reasoning by *Min-Forest* approach.

Acknowledgement. This work is sponsored by China Scholarship Council, and supported by the National Natural Science Foundation of China under Grant No. 61472450 and No. 61572525.

References

1. Cheng, B., Zhai, Z., Zhao, S., Chen, J.: LSMP: a lightweight services mashup platform for ordinary-users. IEEE Commun. Mag. **55**(4), 116–122 (2017)
2. Cheng, B., Li, C., Chen, J.: A web services discovery approach based on interface underlying semantics mining. IEEE Trans. Knowl. Data Eng. **29**(5), 950–962 (2017)
3. Bang-Jensen, J., Gutin, G.: Digraphs: Theory, Algorithms and Applications, 2nd edn. Springer, Heidelberg (2008). https://doi.org/10.1007/978-1-4471-3886-01
4. Linking Open Data (2018). http://www.w3.org/wiki/SweoIG/TaskFores/CommunityProjects/Lin-kingOpenData. Accessed 8 Mar 2018
5. Jagadish, H.V.: A compression technique to materialize transitive closure. ACM Trans. Database Syst. **15**(4), 558–598 (1990)
6. Chen, Y., Chen, Y.: An efficient algorithm for answering graph reachability queries. In: 24th ICDE Proceedings, pp. 893–902. IEEE, New York (2008)
7. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In: ACM SIGMOD Proceedings, pp. 253–262. ACM, New York (1989)
8. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: answering graph reachability queries in constant time. In: 22th ICDE Proceedings, pp. 1–12. IEEE, New York (2006)
9. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: 8th SIGMOD Proceedings, pp. 595–608. ACM, New York (2008)
10. Jin, R., Ruan, N., Xiang, Y., Wang, H.: Path-tree: an efficient reachability indexing scheme for large directed graphs. ACM Trans. Database Syst. **36**(1), 73–84 (2011)
11. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM J. Comput. **32**(5), 1338–1355 (2003)
12. Jin, R., Xiang, Y., Ruan, N., Fuhry, D.: 3-hop: a high-compression indexing scheme for reachability query. In: 9th ACM SIGMOD Proceedings, pp. 813–826. ACM, New York (2009)
13. Cai, J., Poon, C.K.: Path-hop: efficiently indexing large graphs for reachability queries. In: 10th CIKM Proceedings, pp. 119–128. ACM, New York (2010)
14. Yildirim, H., Chaoji, V., Zaki, M.J.: GRAIL: a scalable index for reachability queries in very large graphs. VLDB J. **21**(4), 509–534 (2012)
15. Seufert, S., Anand, A., Bedathur, S., Weikum, G.: FERRARI: flexible and efficient reachability range assignment for graph indexing. In: 13th ICDE Proceedings, pp. 1009–1020. IEEE, New York (2013)
16. Jin, R., Ruan, N., Dey, S., Xu, J.Y.: SCARAB: scaling reachability computation on large graphs. In: 12th SIGMOD Proceedings, pp. 169–180. ACM, New York (2012)
17. 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)
18. Spark GraphX Homepage. http://spark.apache.org/graphx/. Accessed 12 Dec 2017
19. Yang, L., Yang, L., Niu, J., Hu, Z., Long, J., Zheng, M.: A semantic data parallel query method based on hadoop. In: Cellary, W., Mokbel, Mohamed F., Wang, J., Wang, H., Zhou, R., Zhang, Y. (eds.) WISE 2016. LNCS, vol. 10041, pp. 396–404. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48740-3_29