Received November 10, 2019, accepted November 24, 2019, date of publication November 28, 2019, date of current version December 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2956557

HT: A Novel Labeling Scheme for k-Hop Reachability Queries on DAGs

MING DU[1], ANPING YANG[1], JUNFENG ZHOU[1], XIAN TANG[2], ZIYANG CHEN[3], AND YANFEI ZUO[4]

[1]School of Computer Science and Technology, Donghua University, Shanghai 201620, China
[2]School of Electronic and Electrical Engineering, Shanghai University of Engineering Science, Shanghai 201620, China
[3]Shanghai Lixin University of Accounting and Finance, Shanghai 201620, China
[4]Histo Pathology Diagnostic Center, Shanghai 200030, China

Corresponding authors: Junfeng Zhou (zhoujf@dhu.edu.cn) and Xian Tang (txianz@163.com)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB0309800, and in part by the Natural Science Foundation of China under Grant 61472339, Grant 61303040, Grant 61572421, and Grant 61272124.

ABSTRACT Given a directed acyclic graph (*DAG*), a k-hop reachability query $u \xrightarrow{?k} v$ is used to answer whether there exists a path from u to v with length $\leq k$. Answering k-hop reachability queries is a fundamental graph operation and has been extensively studied during the past years. Considering that existing approaches still suffer from inefficiency in practice when processing large graphs, we propose a novel labeling scheme, namely *HT*, to accelerate k-hop reachability queries answering. *HT* uses a constrained 2hop distance label to maintain the length of shortest paths between a set of hop nodes and other nodes, and for the remaining reachability information, *HT* uses a novel topological level to accelerate graph traversal. Further, we propose to enhance *HT* by two optimization techniques. The experimental results show that compared with the state-of-the-art approaches, *HT* works best for most graphs when answering k-hop reachability queries with small index size and reasonable index construction time.

INDEX TERMS Graph data management, reachability queries processing, k-hop reachability.

I. INTRODUCTION

Given a directed graph, a reachability query $u? \rightsquigarrow v$ asks whether there exists a path from u to v. Answering reachability queries on directed graph is a basic operator for a variety of databases, such as XML and RDF, and network applications, such as social and biological networks, which has been extensively studied during the past decades [1]–[14]. In practice, besides asking whether u can reach v, users may want to know the strength of the influence of one node on another, which is usually measured as the length of the path connecting the two nodes. For example, in a wireless network, the information may get lost after a few hops. For another example, in social networks, users may be interested to know whether two persons are connected within a few hops. In these situations, the query becomes a k-hop reachability query, which asks whether there exists a path from u to v with length $\leq$ k, denoted as $u \xrightarrow{?k} v$, and traditional reachability query can be seen as a special case of k-hop query where $k = \infty$. Further, there are many applications where the data graphs are *DAG*s, instead of directed graphs. For example, the paper citation relationship in DBLP[1] can be naturally modeled as a *DAG*. For another example, the Gene Ontology of the UniProt project[2] is also a *DAG* [6]. Although researchers have proposed various approaches [15]–[17] for answering k-hop reachability queries on directed graphs, they are not tailored for *DAG*s, and still suffer from inefficiency when the underlying data graphs are *DAG*s. With the continuing increase of the graph size, it is practically challenging to efficiently answer k-hop reachability queries online with reasonable offline index construction time and index size, especially for applications where k-hop reachability queries answering is intensively involved [18], [19] with the underlying *DAG*s.

To answer a k-hop reachability query, one way is pre-computing the shortest distance between each pair of nodes, such that we can answer it in $O(1)$ time with a single look-up. Another option is performing graph traversal, such as depth-first search (*DFS*) or breadth-first search (*BFS*) to get the answer. The former suffers from unaffordable index

The associate editor coordinating the review of this manuscript and approving it for publication was Donghyun Kim.

[1]https://dblp.uni-trier.de/
[2]https://www.uniprot.org/

size and index construction time, and cannot scale to large graphs, while the latter suffers from the worst query performance. During the past years, researchers made trade-off between the above two extremes and proposed various approaches.

Generally speaking, the state-of-the-art approaches can be classified into two categories: *Label-Only* [15] and *Label+G* [16], [17], [20]. By *Label-Only*, it means that based on the index, the shortest distance between two nodes can be got by comparing their labels. By *Label+G*, it means that the index covers only a part of shortest distance information, and we need to perform graph traversal, if we cannot get the answer directly from the index. Even though both kinds of approaches are much more efficient than the naive approaches, they still suffer from inefficiency in practice for large graphs. On one hand, the *Label-Only* approaches need to maintain all the shortest distance information, thus suffer from large index for both directed graphs and *DAG*s. And they are inefficient when the given k-hop reachability query $u \xrightarrow{?k} v$ is unreachable, due to that they need to scan the whole node label to get the result. On the other hand, for *Label+G* approaches, they have different performance. *kReach* [16], [17] aims at processing k-hop reachability queries on general directed graphs. It constructs a k step index to answer k-hop reachability queries and suffers from large index if k is large. As a comparison, *BFSI-B* [20] is the first and unique approach until now that targets processing k-hop reachability queries on *DAG*s. It uses topological orders to quickly prune unreachable queries, and uses interval of a *BFS* spanning tree to facilitate reachable queries answering. It was shown in [20] that when processing k-hop reachability queries on *DAG*s, *BFSI-B* performs better than *kReach*. However, since a spanning tree interval covers only a small part of reachable nodes, its performance degenerates quickly when the number of satisfied queries increases.

Considering that answering k-hop reachability queries on *DAG*s can find its own applications in practice and existing approaches cannot work well, in this paper, we focus on efficiently answering k-hop reachability queries on *DAG*s. We propose a novel *Label+G* labeling scheme, namely *HT*, that utilizes the properties of *DAG*s to efficiently answer k-hop reachability queries with reasonable index construction time and index size. The basic idea is to further increase the coverage of node label by efficiently constructing a compact index that captures shortest distance between most nodes, such that many k-hop reachability queries can be answered without graph traversal. For the remaining queries, we answer them by performing graph traversal, during which *HT* label is used to largely reduce the search space. Compared with existing *Label-Only* approaches, we do not need to maintain all shortest distance information, therefore *HT* achieves smaller index size. Compared with existing *Label+G* approaches, our index covers much more shortest distance information, therefore *HT* achieves better query performance. Our contributions are as follows.

1) We first propose a new labeling scheme *HT*, which consists of two kinds of labels. The first is partial 2hop distance label, which takes a few nodes as hop nodes to construct 2hop distance label, in order to capture a certain percent of reachability information and the shortest distance between most nodes and these hop nodes, such that to reduce the search space of k-hop reachability queries answering. The second is two complementary topological levels to help us determine whether the given query is unreachable.
2) We propose several optimization techniques, based on which we propose a k-hop reachability queries processing algorithm to efficiently answer k-hop reachability queries.
3) We conduct rich experiment on real datasets. The experimental results show that our approach works best on most datasets for queries answering with small index size and reasonable index construction time.

II. BACKGROUND AND RELATED WORK

A. PRELIMINARIES

Given a directed acyclic graph (*DAG*) $G = (V, E)$, where V is the set of nodes and E the set of edges. Similar as [10], we use $in_G(u) = \{v|(v, u) \in E\}$ to denote the set of in-neighbors of u in G, and $out_G(u) = \{v|(u, v) \in E\}$ the set of out-neighbors of u. We use $in^*_G(u)$ to denote the set of nodes in G that can reach u where $u \notin in^*_G(u)$, and $out^*_G(u)$ the set of nodes in G that u can reach where $u \notin out^*_G(u)$.

A topological sorting on G is a mapping $t : V \to X$, such that for $\forall(u, v) \in E$, we have $t(u) < t(v)$ [10]. Here, $t(u)$ is the topological order of u in X. A topological sorting on G can be done in linear time $O(|V| + |E|)$ [21] to get topological order.

We use $d(u, v)$ to denote the shortest distance between u and v, i.e., length of the shortest path from u to v.

Problem Statement: Given a *DAG* G and a k-hop reachability query $u \xrightarrow{?k} v$, return TRUE if there exists a simple directed path from u to v with no more than k edges, otherwise FALSE.

B. RELATED WORK

The state-of-the-art approaches on k-hop reachability queries answering can be classified into two categories according to the coverage of shortest distance: *Label-Only* [8], [15] and *Label+G* [16], [17], [20], as discussed in below.

Label-Only Approaches: The main idea of these approaches is by constructing a certain index, such that a given k-hop reachability query can be answered using just index without graph traversal.

Following this idea, a naive approach is directly maintaining the shortest distance between all pairs of nodes, such that a given k-hop reachability query can be answered by a single look-up. Although it is simple and efficient, it suffers from huge index with space complexity $O(|V|^2)$ and cannot scaled to large graphs. Considering this problem, [15] proposed to construct *PLL* index, which is an efficient distance index based on 2hop [22], where each node u is associated with two distance labels, in-label $L_{in}(u)$ and out-label $L_{out}(u)$. $L_{in}(u)(L_{out}(u))$ consists of a set of tuples, where each tuple $[v, d(v, u)]([v, d(u, v)])$ consists of a node v that can reach (be reached by) u and the shortest distance from v to u (u to v). Given the 2hop distance label, the computation of shortest distance between u and v is transformed into computation of the smallest summation of the shortest distance from u and v to a hop node that connects them, as shown by Equation 1, where $C_{in}(v) = \{u|[u, d(u, v)] \in L_{in}(v))\}$ and $C_{out}(v) = \{u|[u, d(v, u)] \in L_{out}(v))\}$.

$$d(u, v) = \min_{w \in C_{out}(u) \bigcap C_{in}(v)} (d(u, w) + d(w, v)) \qquad (1)$$

Based on the shortest distance $d(u, v)$, a k-hop reachability query $u \overset{?k}{\longrightarrow} v$ returns TRUE if $d(u, v) \leq k$, otherwise returns FALSE indicating that there does not exist a path from u to v with length $\leq k$.

Even though *PLL* can answer k-hop reachability query without graph traversal, it suffers inefficiency from two aspects: (1) large index size and index construction time, due to maintaining the length of all the shortest path information, (2) long query time for unreachable queries due to scanning the whole label to find that there does not exist a path connecting the two nodes of a given k-hop reachability query.

Label+G Approaches: Considering that existing *Label-Only* approaches suffer from large index size and index construction time, researchers proposed to make improvements by maintaining partial distance index to make tradeoff. The basic idea is: constructing a compact index with linear or sub-linear time, based on which we can answer many k-hop reachability queries in constant time, and for queries that need to be answered during graph traversal, we use the index to reduce the search space. According to their pruning power, there are three kinds of approaches as discussed below.

The first kind of approaches [1]–[11] are the ones to answer traditional reachability queries. We can answer unreachable queries by these approaches. For reachable queries, however, we need to further check whether they satisfy the requirement of k step by graph traversal. Therefore, although this kind of approaches has comparatively small index and index construction time, when the number of reachable queries increases, their performance degrade significantly.

The second kind of approaches [16], [17] construct a partial index based on set cover. The basic idea is to first construct a node cover, then compute the transitive closure of these nodes and maintain the shortest distance value between each pair of nodes. Based on this index, a k-hop reachability query can be answered efficiently if both or either one of the two query nodes are in the node sets. The main problem of this approach lies in that the index is constrained by k's value, which lacks scalability in practice. When the k in the given query is larger than the k in the index, it may need to afford expensive graph traversal.

The third kind of approach [20] uses two topological orders proposed in [4] to quickly prune many unreachable queries. For other queries, the authors proposed a *BFS* based spanning tree, based on which each node is assigned a level and an interval. Here, interval is used to check whether two nodes have ancestor-descendant relationship in the spanning tree, while level is used to check whether they have a distance smaller than k. The problem of this approach is that the coverage of interval label is small, which covers only a small part of reachable queries, therefore the query performance degenerates when the number of satisfied queries increases.

In summary, *Label-Only* approaches do not need to perform expensive graph traversal, while they suffer from large index and index construction time, and their query performance degenerates when the number of unreachable queries increases. On the contrary, *Label+G* approaches, such as *BFSI-B* [20], can construct index quickly with smaller index size, while for query time, their performance degenerates when the number of reachable queries increases.

III. THE *HT* LABELING SCHEME

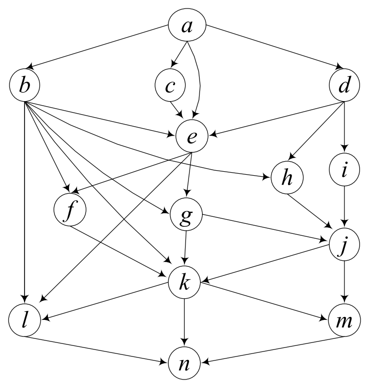
Considering that existing approaches cannot perform well for both reachable and unreachable queries, we propose a new *Label+G* labeling scheme, namely *HT*, which aims at efficiently answering k-hop reachability queries with smaller index size and index construction time. Here, *HT* means that it assigns each node two kinds of labels: (1) a partial 2$\underline{h}$op distance label, and (2) two $\underline{t}$opological levels. We discuss the details in below.

TABLE 1. The complete 2hop distance label of *G* in Figure 1(a).

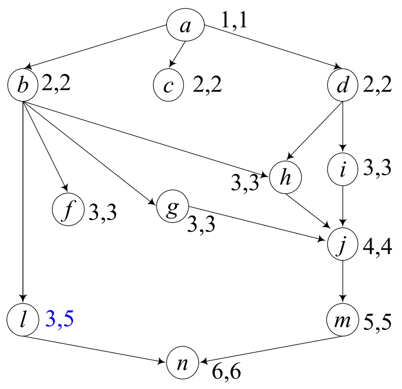
nodes	$L_{in}(u)$	$L_{out}(u)$
a	[a,0]	[a,0],[d,1],[b,1],[e,1],[k,2]
b	[b,0]	[b,0],[e,1],[k,1]
c	[a,1],[c,0]	[c,0],[e,1]
d	[d,0]	[d,0],[e,1],[j,2]
e	[e,0]	[e,0]
f	[b,1],[e,1],[f,0]	[f,0],[k,1]
g	[b,1],[e,1],[g,0]	[j,1],[k,1],[g,0]
h	[b,1],[d,1],[h,0]	[j,1],[k,2],[h,0]
i	[d,1],[i,0]	[i,0],[j,1],[k,2]
j	[b,2],[e,2],[j,0]	[k,1],[j,0]
k	[e,2],[k,0]	[k,0]
l	[b,1],[e,1],[k,1],[l,0]	[l,0]
m	[e,3],[k,1],[j,1],[m,0]	[m,0]
n	[e,2],[k,1],[l,1],[n,0]	[n,0]

A. PARTIAL 2HOP DISTANCE LABEL

To give an intuitive comparison before introducing our partial 2hop distance label, we begin by giving the complete 2hop distance label in Table 1 for G in Figure 1(a). The complete 2hop distance label is constructed as follows: (1) It first sorts all nodes based on their rank values, where the rank value of each node u can be measured by either one of its degree, betweenness centrality or closeness centrality. Here, we use $(|in_G(u)| + 1) \times (|out_G(u)| + 1)$ to rank nodes, which was shown performed well in practice [15]. (2) In each



(a) $DAG\,G$



(b) reduced $DAG\,G'$

node	$L_{in}(u)$	$L_{out}(u)$
a		$[e,1],[k,2]$
b		$[e,1],[k,1]$
c		$[e,1]$
d		$[e,1]$
e	$[e,0]$	$[e,0]$
f	$[e,1]$	$[k,1]$
g	$[e,1]$	$[k,1]$
h		$[k,2]$
i		$[k,2]$
j	$[e,2]$	$[k,1]$
k	$[e,2],[k,0]$	$[k,0]$
l	$[e,1],[k,1]$	
m	$[e,3],[k,1]$	
n	$[e,2],[k,1]$	

(c) partial 2hop distance label

FIGURE 1. **Illustration of a sample *DAG* (a), the reduced *DAG* G' (b) and the partial 2hop distance label (c), where in G', the two integers beside each node u is u's topological level and its sinking topological level.**

iteration, it picks the node u with the highest rank value, and performs forward *BFS* and backward *BFS* to compute the shortest distance between u and nodes being visited. If the distance between u and each visited node v is less than the distance computed by 2hop distance label constructed so far, we add an entry $[u, d(u, v)]$ to v's in-label, if v is visited when performing forward *BFS*; otherwise we add $[u, d(v, u)]$ to v's out-label.

It was shown in [14] that for 2hop distance label, nodes with large degree have smaller label size but catch most reachability information, and the majority part of label size is contributed by nodes with small degree.

Example 1: If we construct 2hop distance label using e and k, the result is shown in Figure 1(c). By comparing Table 1 and Figure 1(c), we know that using 2 nodes, we catch 79% reachable information, but the label size is only 40% to the complete 2hop distance label. We further show the reachability ratio of k hop nodes on four real graphs in Figure 2, from which we know that even though reachability ratio various for different graphs, for each graph, they will change little when $k \geq 8$. □

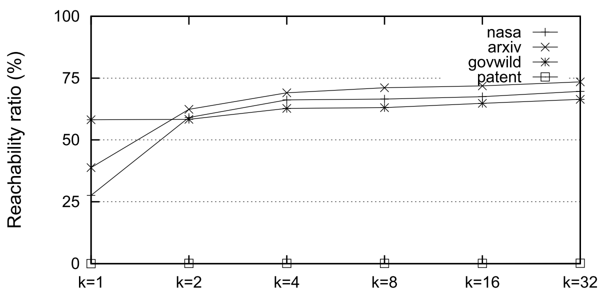


FIGURE 2. **Coverage of reachability queries for k hop nodes.**

Based on the above observation, we propose to construct a *partial* 2hop distance label, which maintains most reachability relationship between nodes with small label size. Here, "partial" means that we construct 2hop distance label using a part of, rather than all nodes. Let reachability ratio be the number of pairs of reachable nodes that can be answered using partial 2hop label over all the reachable pairs of the given *DAG*, the number of nodes used for partial 2hop distance label construction is determined by λ, which is a threshold to control the ratio of reachability information. That is, we will construct the 2hop label until the reachability ratio $\geq \lambda$.

Example 2: If we set $\lambda = 70\%$, then for G in Figure 1(a), the partial 2hop distance label is shown in Figure 1(c), where two nodes, i.e., k and e, are used to construct 2hop distance label. □

Hereafter, we call nodes that are used to construct 2hop distance label as the hop nodes. We have that after constructing the partial 2hop distance label, the hop nodes can be removed to simplify the input *DAG*, since the shortest distance between any node and hop node is maintained by the partial 2hop distance label. For example, after deleting k and e from Figure 1(a), we get the reduced graph G' shown in Figure 1(b).

Based on the partial 2hop distance label, a k-hop reachability query can be answered in two steps: (1) we get a distance from the partial 2hop distance label, if it is less than k, then we return the answer directly, otherwise (2) we perform graph traversal on the largely simplified graph to quickly get the final answer.

Example 3: Given a k-hop reachability query $f \xrightarrow{?2} n$, using the partial 2hop distance label in Figure 1(c), we know that the shortest distance from f and n through k is 2, thus we directly return TRUE for the query. If the given query is $f \xrightarrow{?1} n$, then by the partial 2hop distance label, we still can not say that there does not exist a path from f to n with one step. We need to perform graph traversal from f in G'. Here, for G' in Figure 1(b), f does not have out-neighbors, which means that we do not need to perform graph traversal anymore. Therefore, we know immediately that the result is FALSE. □

For graph traversal operation, we further propose a novel topological level. we discuss the details in below.

B. SINKING TOPOLOGICAL LEVEL

A topological level for each node u is the length of the longest path from all nodes without in-neighbors. Given the reduced *DAG* G', we assign each node u a topological level $l(u)$, as shown by Equation 2, then we know that u cannot reach v, if $l(u) \geq l(v)$. In this way, we can quickly prune many unreachable queries.

$$l(u) = \begin{cases} 1, & in_G(u) = \emptyset \\ \max\{l(v) + 1 | v \in in_G(u)\}, & \text{otherwise} \end{cases} \quad (2)$$

Example 4: For G' in Figure 1(b), the topological level of each node u is shown by the first integer beside u. For k-hop reachability query $b \overset{?2}{\longrightarrow} c$, we know that b cannot reach c due to $l(b) = 2 = l(c)$, thus we can directly return FALSE. However, if the k-hop reachability query is $l \overset{?2}{\longrightarrow} m$, we cannot tell that the result is FALSE by topological level due to that $l(l) = 3 < l(m) = 5$. □

For this problem, we propose to further enhance the pruning power of topological level by assigning each node a sinking topological level, denoted by Equation 3.

$$l_s(u) = \begin{cases} l(u), & out_G(u) = \emptyset \\ \min\{l_s(v) - 1 | v \in out_G(u)\}, & \text{otherwise} \end{cases} \quad (3)$$

The second topological level is constructed based on the first one. Intuitively, it is used to make the difference of topological levels between two nodes as small as possible. Consider the *DAG* in Figure 3, where the first (second) integer on the right of f and g is (sinking) topological level. It can be seen that Equation 2 assigns each node a topological level as small as possible, and Equation 3 makes the topological level as large as possible based on Equation 2. We have the following result.

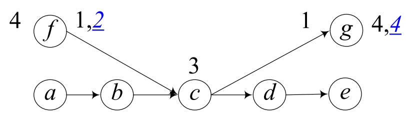


FIGURE 3. Illustration of the sinking topological level.

Theorem 1: Given two nodes u and v, if u can reach v, then we know that $l(v) - l(u) \geq l(v) - l_s(u) \geq d(u, v)$.

Proof 1: We first prove $l(v) - l(u) \geq l(v) - l_s(u)$. Assume that $l(v) - l(u) < l(v) - l_s(u)$ holds, we know that $l_s(u) < l(u)$ holds. However, according to Equation 2, $\forall v \in out_G(u)$, $l(v) - l(u) \geq 1$. Therefore, according to Equation 3, $l_s(u) - l(u) \geq 0$.

We then prove $l(v) - l_s(u) \geq d(u, v)$. First, even though $l_s(u) \geq l(u)$, if u can reach v, the length of the longest path from u to v is still bounded by $l_s(v) - l(u)$. As $d(u, v)$ is the length of the shortest path from u to v, we know that $l(v) - l_s(u) \geq d(u, v)$ holds. □

According to Theorem 1, we can use topological level and its sinking topological level to quickly prune more unreachable queries.

Example 5: For k-hop reachability query $l \overset{?2}{\longrightarrow} m$ on G' in Figure 1(b), by sinking topological level, we can immediately tell that l cannot reach m, due to that $l_s(l) = 5 = l(m)$. □

It is worth noting that similar to Equation 2, we can assign each node a topological level following the reversed direction of edges. For example, for the *DAG* in Figure 3, following the reversed direction of edges, we assign the topological level of g and f as 1 and 4, respectively. Even though, for k-hop reachability query $f \overset{?2}{\longrightarrow} g$, we cannot tell the result by reversed topological level due to that the difference between their topological levels is 3, which is greater than 2. However, using sinking topological level, we can return the result immediately due to that $l(g) - l_s(f) = 4 - 2 = 2$.

IV. INDEX CONSTRUCTION

As shown by Algorithm 1, we construct the index in two steps: we first generate the 2hop partial distance label by calling Algorithm 2, then generate the two topological levels by calling Algorithm 3.

```
Algorithm 1 genIndex($G = (V, E)$, $\lambda$)
1 generate 2hop partial distance label w.r.t. $\lambda$ by calling
  Algorithm 2.
2 generate topological level label by calling Algorithm 3.
```

A. PARTIAL 2HOP DISTANCE LABEL CONSTRUCTION

Similar to [14], [15], we sort all nodes by $(|in^*_G(u)| + 1) \times (|out^*_G(u)| + 1)$. Then in each iteration, we pick a node u with the highest rank value as the hop node, and perform forward and backward *BFS* from u to add the distance entry to the label of nodes being visited. During this process, we terminate the graph traversal in advance based on the following Theorems.

Theorem 2: When processing u during 2hop distance label construction, if the visited node v is a hop node processed before u, then we do not need to visit other nodes from v.

Proof 2: As shown by Figure 4, assume that hop node v is processed before u, and by the 2hop distance label constructed by v, $d(x, v) = d_1$ and $d(v, u) = d_2$. Therefore, we know that the length of the shortest path from x to u through v is $d_1 + d_2$.

When processing u, we need to perform backward *BFS* from u. Obviously, when encountering v, we know that the shortest distance from u to v is still d_2. Therefore, even if we do not stop graph traversal in v, the shortest distance from x to

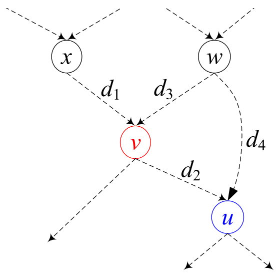


FIGURE 4. Illustration of the stop conditions for 2hop distance label construction.

Algorithm 2 genPart2Hoplabel($G = (V, E)$, λ)

```
1  compute the size of transitive closure $\mathcal{T}$ of $G$ by [23]
2  $N \leftarrow 0$        /*the number of reachable pairs got*/
3  sort all nodes by $(|in^*_G(u)| + 1) \times (|out^*_G(u)| + 1)$ in decreasing order
4  for each (unvisited node $u \in V$) do
5     $c_f = c_b = 1$     /*$c_f$ and $c_b$ are the number of visited nodes of $u$*/
6     add $[u, 0]$ to $L_{in}(u)$ and $L_{out}(u)$
7     push all $[v, 1]$ into $Q$, where $v \in out_G(u)$
8     while ($\neg$ isEmpty($Q$)) do
9        $[v, d] \leftarrow$ removeHead($Q$)
10       if ($d(u, v) > d$) then     /*$d(u, v)$ is computed by Equation 1*/
11          add $[u, d]$ to $L_{in}(v)$
12          $c_f \leftarrow c_f + 1$
13          for each (unvisited $w \in out_G(v)$) do
14             addTail($Q$, $[w, d + 1]$)
15          endfor
16       endif
17    endwhile
18    push all $[v, 1]$ into $Q$, where $v \in in_G(u)$
19    while ($\neg$ isEmpty($Q$)) do
20       $[v, d] \leftarrow$ removeHead($Q$)
21       if ($d(u, v) > d$) then     /*$d(u, v)$ is computed by Equation 1*/
22          add $[u, d]$ to $L_{out}(v)$
23          $c_b \leftarrow c_b + 1$
24          for each (unvisited $w \in in_G(v)$) do
25             addTail($Q$, $[w, d + 1]$)
26          endfor
27       endif
28    endwhile
29    $N \leftarrow N + c_f \times c_b - 1$
30    if ($\frac{N}{\mathcal{T}} > \lambda$) then return
31 endfor
```

u through v is still $d_1 + d_2$ based on 2hop distance label of u. Hence, we can safely terminate the graph traversal in v. □

Theorem 3: When processing u during 2hop distance label construction, for any node w being visited, if the length of the shortest path between u and w is greater than the length of the path computed by 2hop distance label constructed so far, then we do not need to visit other nodes from w.

Proof 3: As shown by Figure 4, assume that hop node v is processed before u, and by the 2hop distance label constructed by v, $d(w, v) = d_3$ and $d(v, u) = d_2$. When performing backward *BFS* from u, and assume that the length of the shortest path from w to u is d_4 satisfying that $d_4 > d_3 + d_2$, then we know that we do not need to add u to w's out-label. Because in this case, we know that the path from w to u through v is shorter than d_4. And for every node that can reach w, we do not visit them either when performing backward *BFS* from u. Similarly, when performing forward *BFS* from u, we also have this conclusion. □

Algorithm 3 genTopolevel($G' = (V', E')$)

```
1  for all nodes $u$, $l(u) \leftarrow 0$, $l_s(u) \leftarrow \infty$
2  perform a topological sorting to get a topological order
3  for each ($u$ processed in ascending topological order) do
4     if ($in_G(u) = \emptyset$) then $l(u) = 1$
5     else
6        for each ($v \in out_G(u)$) do
7           $l(v) \leftarrow \max\{l(u) + 1, l(v)\}$
8        endfor
9     endif
10 endfor
11 for each ($u$ processed in descending topological order) do
12    if ($out_G(u) = \emptyset$) then $l_s(u) = l(u)$
13    else
14       for each ($v \in in_G(u)$) do
15          $l_s(v) \leftarrow \min\{l_s(u) - 1, l_s(v)\}$
16       endfor
17    endif
18 endfor
```

Based on Theorems 2 and 3, we can significantly reduce the 2hop distance label size.

Example 6: For G in Figure 1(a), the first processed hop node is e. After processing it, we process the second hop node k. When processing k by performing backward *BFS*, according to Theorem 2, we can terminate the expansion when encountering e. When visiting d by performing backward *BFS* from k, according to Theorem 3, we can terminate the expansion on d, due to that by the 2hop distance label of e, we find the distance of the path from d to k through e is not larger than the distance from d to k by backward *BFS*. □

The Algorithm: We use Algorithm 2 to compute the partial 2hop distance label, which works in three steps: (1) In line 1, it computes the size of transitive closure $\mathcal{T}$ of G by the approach proposed in [23]. (2) In line 3, it sorts all nodes based on degree to get a node processing order. (3) In lines 4-31, for each processed node u, it first performs forward *BFS* in lines 7-17 to add u to in-label of its reachable nodes. It then performs backward *BFS* in lines 18-28 to add u to out-label of nodes that can reach u. Then, we compute the number of covered reachable node pairs in line 29, and if we reach the required ratio λ, we terminate algorithm 2 in line 30.

Example 7: Consider constructing partial 2hop distance label for G in Figure 1(a). Assume that $\lambda = 0.7$. By line 1, we know that $\mathcal{T} = 89$. The first processed node is e. After processing it, $c_f = 8$ and $c_b = 5$, then we know $\frac{N}{\mathcal{T}} = (5 \times 8 - 1)/89 = 44\% < \lambda$. The second processed node is k. After processing it, we know that $c_f = 4$ and $c_b = 8$, than we know that $N = 39 + 4 \times 8 - 1 = 70$. Then, we terminate the partial 2hop distance label construction, due to that $\frac{N}{\mathcal{T}} = 70/89 = 79\% \geq \lambda = 0.7$. The partial 2hop distance label of G in Figure 1(a) w.r.t. $\lambda = 0.7$ is shown in Figure 1(c). □

It is worth noting that in line 10 of Algorithm 2, we need to compute $d(u, v)$ according to Equation 1, which is a set intersection operation and requires that both $L_{out}(u)$ and $L_{in}(v)$ are sorted by node ID. To accelerate index construction and reduce the index size, we use the following techniques to make acceleration, as shown in below.

R1 (Avoiding the Sorting Operations): We notice that the result of the set intersection between $L_{out}(u)$ and $L_{in}(v)$ is the length of the shortest path from u to v through a hop node. We do not need to know exactly which hop node it is. Therefore, we use the following heuristics to avoid the sorting operation when a node is added into $L_{out}(u)$ and $L_{in}(v)$.

Rule-1: For each node u and any entry $[v, d]$ in its in-label and out-label, we only need to replace v's ID with v's processing order.

Example 8: In Figure 1(c), we can replace e by 1 for all entries denoting that e is the first processed node. Similarly, we can replace k by 2 denoting that k is the second processed node. In this way, when any entry needs to be added into an in-label or out-label, we do not need to do sorting operation, instead, we directly add this entry at the end of the label. □

R2 (Reducing the Size of Index): On one hand, the number of processed hop nodes is usually very small to construct 2hop distance label. On the other hand, for a real graph, the distance is also very small. Therefore, we do not need to maintain an entry by two integers. We can reduce the index size using the following heuristics.

Rule-2: Each entry $[u, d]$ of in-label and out-label can be represented using one integer. The higher half is used to represent u's processing order, and the lower half is used to represent d.

Based on Rule-2, we use one integer to maintain each entry $[u, d]$, and the index size can be reduced into half of its original size.

Analysis: We consider the time complexity of Algorithm 2. The cost of line 1 is $O(w_{avg} \times |E|)$, where w_{avg} is the average number of processed nodes for all nodes. The cost of line 3 is $O(|V| \log |V|)$. For each node processed in lines 5-30 of Algorithm 2, the cost is $O((|V|+|E|) \times L)$, where L is the length of the longest node label. Therefore, given the constraints of λ, the cost of lines 4-31 is $O(c \times (|V|+|E|) \times L)$, where c is the number of processed hop nodes. Usually in practice, c is very small compared with $|V|$. Therefore, the time complexity of Algorithm 2 is $O(w_{avg} \times |E| + |V| \log |V| + c \times (|V|+|E|) \times L)$.

B. TOPOLOGICAL LEVEL LABEL CONSTRUCTION

After constructing partial 2hop distance label, we remove all hop nodes to simplify the given *DAG*. For example, for G in Figure 1(a), if we take e and k as hop nodes, and construct the partial 2hop distance label shown in Figure 1(c), we can safely remove them and get the reduce graph G' shown in Figure 1(b).

Based on the reduced graph, we compute topological levels using Algorithm 3. In line 2, we perform topological sorting to get a topological order. In lines 3-10, we visit nodes in *ascending* topological order and assign each node a topological level as Equation 2. In line 11-18, we visit nodes in *descending* topological order to assign each node the sinking topological level as Equation 3. For example, for G' in Figure 1(b), after performing Algorithm 3, the topological level and the sinking topological level for each node u are shown as the second pair of integers beside u.

We consider the time complexity of Algorithm 3. The cost of line 1 is $O(|V| + |E|)$. The cost of performing topological sorting in line 2 is $O(|V| + |E|)$ [21]. In lines 6-8, we visit all out-neighbors of u once, thus the cost of lines 3-10 is $O(|V|+|E|)$. similarly, the cost of lines 11-18 is $O(|V|+|E|)$. Therefore, the time complexity of Algorithm 3 is $O(|V|+|E|)$. Together with Algorithm 2, the time complexity of Algorithm 1 for index construction is $O(w_{avg} \times |E| + |V| \log |V| + c \times (|V| + |E|) \times L)$, and the space complexity is $O(L|V|)$.

We show the comparison of time and space complexities with the state-of-the-art *Label-Only* approach *PLL* and *Label+G* approach *BFSI-B* in Table 2. For time complexity, as c and L are small in practice, our *HT* usually has a comparable performance with *BFSI-B* (shown by the experimental results). As a comparison, *PLL* suffers from larger time complexity due to processing all nodes to get the complete 2hop distance label. For space complexity, all the three algorithms need to maintain all labels during index construction, therefore the space complexities are $O(\mathcal{L}|V|)$, $O(h|V|)$ and $O(L|V|)$ for *PLL*, *BFSI-B* and *HT*, respectively.

TABLE 2. The comparison of time and space complexities for index construction, where $\mathcal{L}$ is the length of the longest node label of *PLL*, and *h* is the number of integers for each node.

algorithm	time complexity	space complexity
PLL	$O(\|V\|(\|V\| + \|E\|)\mathcal{L})$	$O(\mathcal{L}\|V\|)$
BFSI-B	$O(\|V\|(\log\|V\|) + \|E\|)$	$O(h\|V\|)$
HT	$O(w_{avg}\|E\| + \|V\|\log\|V\| + c(\|V\| + \|E\|)L)$	$O(L\|V\|)$

V. OPTIMIZATION

We make optimizations from two aspects: (1) enhancing the pruning power of answering unreachable queries, and (2) early stop condition. We discuss the details below.

O1 (Enhancing the Pruning Power for Unreachable Queries): Existing work [4] showed that topological orders can be used to quickly answer many unreachable queries. We notice that after removing hop nodes, the graph can be largely simplified, and the topological order can be more effective to answer unreachable queries. Different with [4] that uses two reversed topological orders to answer unreachable queries on the original *DAG*s, we use four topological orders on the reduced *DAG*s. Here, the aim is not just "$1 + 1 = 2$", but "$1 + 1 > 2$". In our approach, the four topological orders w.r.t. every node u are divided into two pairs, the first one is the same as [4], denoted as $t_1(u)$ and $t_2(u)$. The second pair of mutually reversed topological orders are computed based on the reversed direction of edges, denoted as $t_3(u)$ and $t_4(u)$. Given a reachability query asking whether u can reach v, we can tell that u cannot reach v, if $\exists i \in [1, 4]$, such that $t_i(u) > t_i(v)$. In this way, we can

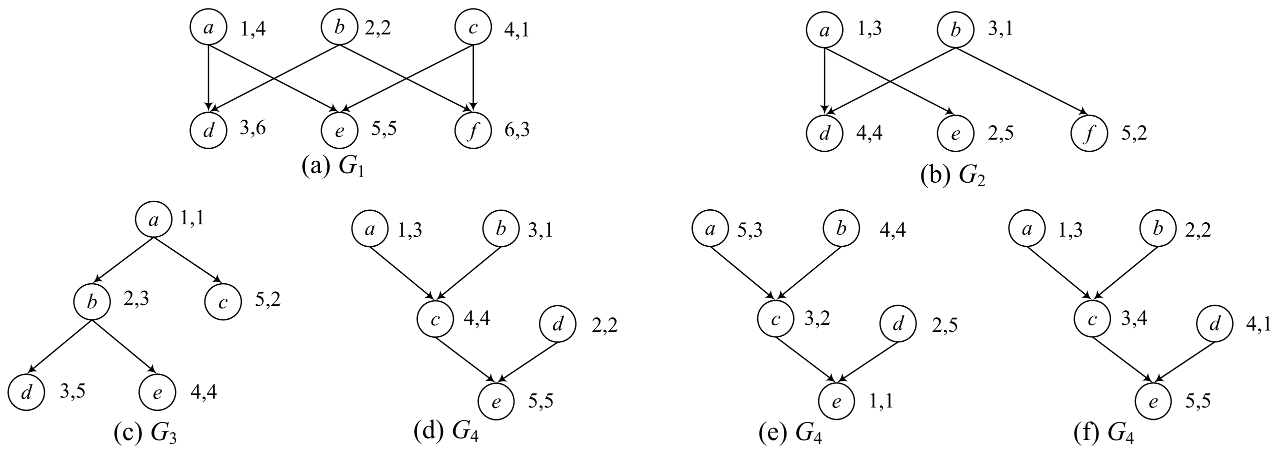


FIGURE 5. **Illustration of the benefits of using the first optimization, where the two integers beside each node are its two topological orders.**

answer more unreachable queries. We show the benefits by the following example.

Example 9: Consider the *DAG* known as S_3^0 graph in Figure 5(a), it is not possible to use two topological orders to avoid false positives [4]. For instance, we cannot tell that b cannot reach e using the two topological orders in Figure 5(a) due to that $t_1(b) = 2 < t_1(e) = 5 \wedge t_2(b) = 2 < t_2(e) = 5$. However, if we take c as a hop node, and construct partial 2hop distance label of c, then c can be safely removed. After that, we have the simplified *DAG* G_2 shown in Figure 5(b). Based on G_2, we compute the two topological orders for each node. It is easy to verify that now we can tell that e cannot reach b, due to that $t_1(b) = 3 > t_1(e) = 2$.

Further, it was shown in [10], [11] that if the given *DAG* is a tree where the root node has no in-neighbors, then two topological orders can answer all unreachable queries, such as the *DAG* G_3 in Figure 5(c). However, if the root node of the tree has no out-neighbors, such as the *DAG* G_4 in Figure 5(d), we cannot guarantee that all unreachable queries can be answered using two topological orders. For instance, we cannot tell that d cannot reach c in G_4 due to that $t_1(d) = 2 < t_1(c) = 4 \wedge t_2(d) = 2 < t_2(c) = 4$. To this problem, we compute the second pair of topological orders following the reversed direction of edges, i.e., we process a tree with root node without out-neighbors same as the tree with root node without in-neighbors, as shown by G_4 in Figure 5(e). Then, we can answer all unreachable queries using the corresponding two topological orders. But in this case, the testing of whether u cannot reach v becomes whether $t_i(u) < t_i(v)$. To make it consistent with the original case, for each topological order x, we replace it by $|V|+1-x$, then we have the two topological orders shown in Figure 5(f), and now we can check whether u cannot reach v by testing whether $\exists i$, such that $t_i(u) > t_i(v)$. □

It is worth noting that this optimization does not change the time and space complexities of index construction, due to that the four topological orders can be got with time $O(|V| + |E|)$ and space $O(|V|)$ [10], [11].

O2 (Early Stop Condition): When using Equation 1 to compute the distance between u and v, we do not need to get length of the shortest path, instead, we take k as a parameter, and when finding that the length of a path is not larger than k, we immediately terminate the set intersection operation, and return TRUE.

Example 10: Consider the k-hop reachability query $a \xrightarrow{?2} l$ on G in Figure 1. If we use Equation 1, we need to first compute the set intersection of $C_{out}(a) \cap C_{in}(l) = \{e, k\}$ based on the partial 2hop distance label in Figure 1(c), then compute the length of the shortest path based on hop nodes, i.e., $d(a, l) = \min\{d(a, e) + d(e, l), d(a, k) + d(k, l)\} = \min\{2, 3\} = 2$. Finally, we return TRUE due to that we find a path with length ≤ 2.

As a comparison, by taking $k = 2$ as a parameter, when we find the first result e of the set intersection, we directly compute the length of the path from a to l through e. Since the length of the path is 2, we immediately terminate the set intersection operation and return TRUE without checking other hop nodes. □

VI. QUERIES ANSWERING

Given a k-hop reachability query $u \xrightarrow{?k} v$, we use Algorithm 4 to return the answer. If $u = v$, we return TRUE in line 1. In line 2, we compute the distance $d(u, v)$ between u and v using Equation 1 and the early stop condition, and return TRUE if $d(u, v) \leq k$. In line 3, if u or v is a hop node, then we know that $d(u, v)$ computed in line 2 is the length of the shortest path from u to v, and return FALSE immediately. In line 4, we use the topological level to prune unreachable queries, and we use topological orders in line 5 to find more unreachable queries. If we still cannot get the result so far, in lines 6-14, we perform graph traversal recursively to get the answer, where we always choose the node with smaller degree to make expansion, as shown by line 6. It is worth noting that based on the topological level we got, it is easy to verify that the larger the difference of topological levels between two nodes, the longer the distance between

Algorithm 4 *HT* (u, v, k)

```
1  if ($u = v$) then return TRUE
2  if ($d(u, v) \leq k$) then return TRUE     /*by Equation 1*/
3  if ($u$ or $v$ is a hop node) then return FALSE
4  if ($l_s(u) \geq l(v)$) then return FALSE
5  if ($t_1(u) > t_1(v) \vee t_2(u) > t_2(v) \vee t_3(u) > t_3(v) \vee t_4(u) >$
   $t_4(v)$) then return FALSE
6  if ($|out_G(u)| < |in_G(u)|$) then return
7    for each ($w \in out_G(u)$) do
8      if ($HT(w, v, k-1)$) then return TRUE
9    endfor
10 else
11   for each ($w \in in_G(v)$) do
12     if ($HT(u, w, k-1)$) then return TRUE
13   endfor
14 endif
15 return FALSE
```

TABLE 3. Statistics of real datasets.

Dataset	$\|V\|$	$\|E\|$	d (avg. degree)
amaze	3,710	3,600	0.97
agrocyc	12,684	13,408	1.06
ecoo	12,620	13,350	1.06
vchocyc	9,491	10,143	1.07
kegg	3,617	3,908	1.08
xmark	6,080	7,025	1.16
mtbrv	9,602	10,245	1.07
nasa	5,605	6,537	1.17
yago	6,642	42,392	6.38
arxiv	6,000	66,707	11.12
email	231,000	223,004	0.97
uniprot100m	16,087,295	16,087,293	1.00
uniprot150m	25,037,600	25,037,598	1.00
wiki	2,281,879	2,311,570	1.01
LJ	971,232	1,024,140	1.05
web	371,764	517,805	1.39
10cit-Patent	1,097,775	1,651,894	1.50
05cit-Patent	1,671,488	3,303,789	1.98
citeseer	6,540,401	15,011,260	2.30
dbpedia	3,365,623	7,989,191	2.37
govwild	8,022,880	23,652,610	2.95
cit-Patents	3,774,768	16,518,947	4.38
go	6,967,956	34,769,339	4.99
10go-unip	469,526	3,476,397	7.40
twitter	18,121,168	18,359,487	1.01

two nodes on average. Therefore, when answering a k-hop reachability query, if graph traversal is needed, we first visit nodes that have the largest topological level.

VII. EXPERIMENT

In our experiment, we make comparison between our *HT* and the state-of-the-art approaches *BFSI-B* and *PLL*. We implemented all algorithms using C++ and compiled by G++ 6.2.0. All experiments were run on a PC with Intel(R) Core(TM) i5-3230M CPU @ 3.0 GHz CPU, 12 GB memory, and Ubuntu 18.04.1 Linux OS.

Datasets: Table 3 shows the statistics of 26 real datasets. Among these datasets, the first ten are small datasets ($|V| \leq 100,000$) and are downloaded from the same web page.[3] The following 15 datasets are large ones ($|V| > 100,000$). We call a graph a sparse graph if $d < 2$, otherwise a dense graph. These datasets are usually used in the recent works [1]–[9]. For large datasets, email[4] is an email network. As indicated by [6], go[3] and 10go-unip[5] (10go-uniprot) are the joint graphs of Gene Ontology terms and the annotations file from the UniProt[6] database. uniprot100m[3] (uniprotenc_100m) and uniprot150m[3] (uniprotenc_150m) are *DAG*s that are subgraphs of the RDF graph of UniProt, which contain many nodes without incoming edges and few nodes without outgoing edges. 10cit-Patent[5](10cit-Patent), 05cit-Patent[5] (05cit-Patent), cit-Patents[3] (cit-Patents) and citeseer[3] are all citation networks with out-degree of non-leaf nodes ranging from 10 to 30. wiki is the *DAG* of Wikipedia talk (communication) network wiki-Talk[4]. LJ is an online social network soc-LiveJournal1[4]. dbpedia[7] is a knowledge graph Dbpedia. govwild[8] is a large RDF graph. web is a web graph web-Google[8]. twitter[8] is a large-scale social network [24].

Among all these datasets, the first 8 small datasets and four large datasets, including email, wiki, LJ and web, are direct graphs initially. We transform each of them into a *DAG* by coalescing each strongly connected component into a node. Note that this can be done in linear time [25]. All other datasets are *DAG*s initially. The statistics in Table 3 are that of *DAG*s.

Workloads: We generate a workload containing 1,000,000 queries, where the number of reachable queries and unreachable queries is both 500,000. We generate unreachable queries by sampling node pairs with the same probability, where most queries are unreachable queries. We check whether they are unreachable by using existing reachability algorithms. Then we generate reachable queries as follows. We randomly select a node u and randomly select an out-neighbor v recursively until v has no out-neighbors. We take all pairs of u and v as reachable queries during this process until we reach the required number of reachable queries. The query time is the running time of a total of 1,000,000 queries.

A. IMPACTS OF OPTIMIZATIONS

Impacts of λ: As shown by Table 4, with the increase of λ, the index size and index construction time increases, but increases little. As a comparison, with the increase of λ, the query time is reduced. The reasons lie in the following aspects: (1) With only a few hop nodes, we can catch a big ratio of reachable information, therefore the index size and index construction time increase little. (2) With more covered reachable information, we can answer more k-hop reachability queries by 2hop distance label, therefore the query time can be reduced. In the following discussion, we set $\lambda = 80\%$.

[3]https://code.google.com/archive/p/grail/downloads

[4]http://snap.stanford.edu/data/index.html

[5]http://pan.baidu.com/s/1bpHkFJx

[6]http://www.uniprot.org/

[7]http://pan.baidu.com/s/1c00Jq5E

[8]https://code.google.com/p/ferrari-index/downloads/list

TABLE 4. Impacts of λ to index size (MB), index construction time (ms) and query time (ms).

Dataset	Index Size			Index Construction Time			Query Time ($k=5$)		
	$\lambda=20\%$	$\lambda=50\%$	$\lambda=80\%$	$\lambda=20\%$	$\lambda=50\%$	$\lambda=80\%$	$\lambda=20\%$	$\lambda=50\%$	$\lambda=80\%$
amaze	0.08	0.08	0.08	0.69	0.71	0.69	18	18	17
agrocyc	0.24	0.24	0.28	1.14	1.44	1.77	19	19	18
ecoo	0.24	0.24	0.28	0.85	1.86	1.48	19	19	19
vchocyc	0.18	0.18	0.21	0.64	1.26	1.44	18	18	17
kegg	0.08	0.08	0.08	0.31	0.33	0.61	23	24	22
xmark	0.13	0.13	0.13	0.74	0.58	1.08	35	35	31
mtbrv	0.19	0.19	0.21	1.55	0.94	1.37	22	21	21
nasa	0.11	0.11	0.11	0.88	0.66	0.69	38	38	36
yago	0.16	0.19	0.25	2.06	2.30	2.20	42	39	36
arxiv	0.13	0.14	0.15	3.98	4.09	4.52	760	646	571
email	5.05	5.05	5.05	24.93	22.56	23.22	53	56	50
uniprot100m	368	375	445	2,857	2,953	3,007	127	122	98
uniprot150m	573	590	709	4,653	4,895	5,040	159	137	116
wiki	52	52	52	220	222	228	74	75	64
LJ	22	22	22	123	130	133	59	60	56
web	8	8	8	101	105	111	70	72	63
10cit-Patent	22	25	32	601	635	641	137	132	105
05cit-Patent	34	35	41	1,492	1,584	1,604	213	198	152
citeseer	127	127	127	5,622	6,154	6,199	232	239	219
dbpedia	73	73	73	2,068	2,196	2,268	90	92	85
govwild	154	154	155	3,983	4,154	4,306	172	171	146
cit-Patents	90	109	140	9,491	10,081	10,346	3,153	3,036	2,727
go	170	217	275	5,979	6,358	6,405	201	189	179
10go-unip	13	16	20	504	551	549	239	228	208
twitter	409	409	409	2,755	2,934	3,032	93	93	90

TABLE 5. Impacts of optimization techniques (λ = 80%).

Dataset	Index Size (MB)			Index Construction Time (ms)			Query Time (ms, $k=5$)		
	HT-base	*HT*-1	*HT*	*HT*-base	*HT*-1	*HT*	*HT*-base	*HT*-1	*HT*
amaze	0.03	0.08	0.08	0.20	0.69	0.69	21	19	17
agrocyc	0.09	0.28	0.28	0.67	1.77	1.77	24	20	18
ecoo	0.09	0.28	0.28	0.76	1.48	1.48	24	20	19
vchocyc	0.06	0.21	0.21	0.39	1.44	1.44	23	19	17
kegg	0.03	0.08	0.08	0.19	0.61	0.61	27	24	22
xmark	0.04	0.13	0.13	0.42	1.08	1.08	40	35	31
mtbrv	0.06	0.21	0.21	0.40	1.37	1.37	26	22	21
nasa	0.03	0.11	0.11	0.47	0.69	0.69	49	39	36
yago	0.15	0.25	0.25	1.18	2.20	2.20	51	43	36
arxiv	0.06	0.15	0.15	3.20	4.52	4.52	1,652	591	571
email	1.53	5.05	5.05	13.08	23.22	23.22	79	56	50
uniprot100m	199	445	445	494	3,007	3,007	169	109	98
uniprot150m	327	709	709	719	5,040	5,040	203	128	116
wiki	17	52	52	90	228	228	114	75	64
LJ	7	22	22	60	133	133	97	62	56
web	3	8	8	47	111	111	110	72	63
10cit-Patent	15	32	32	32	641	641	130	114	105
05cit-Patent	15	41	41	463	1,604	1,604	235	176	152
citeseer	27	127	127	2,226	6,199	6,199	725	234	219
dbpedia	22	73	73	803	2,268	2,268	184	94	85
govwild	32	155	155	1,635	4,306	4,306	264	171	146
cit-Patents	83	140	140	3,650	10,346	10,346	6,092	2,913	2,727
go	168	275	275	1,030	6,405	6,405	263	209	179
10go-unip	13	20	20	93	549	549	279	229	208
twitter	132	409	409	1,232	3,032	3,032	162	95	90

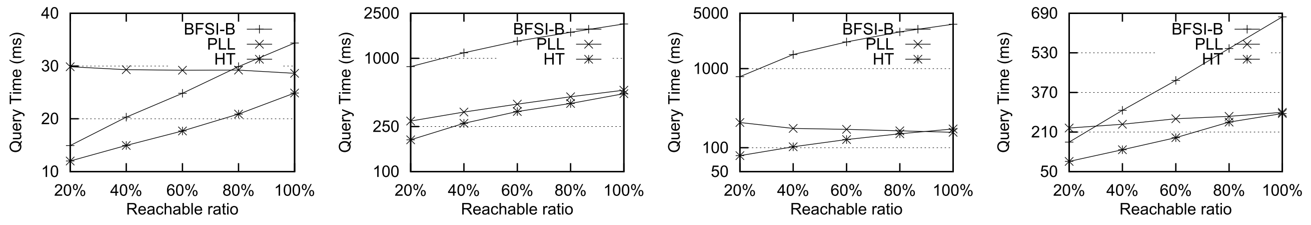
Impacts of Topo-Orders and Early Stop Condition: In Table 5, *HT*-base is our algorithm without using any optimization techniques, *HT*-1 is our algorithm using topo-orders to prune unreachable queries, and *HT* is our algorithm that uses both topo-order and early stop condition. From Table 5, we have the following observations.

First, for index size, since *HT*-base uses only partial 2hop distance label, it has the smallest index size. By using four topological orders for each node, the index size of *HT*-1 is larger than that of *HT*-base. Note that the difference between *HT*-1 and *HT* is that *HT* uses early stop condition, which does not affect the index size, therefore, the index size of *HT*-1 is equal to that of *HT*.

Second, for index construction time, *HT*-base consumes least time compared with *HT*-1 and *HT*, due to that it does not need to compute the four topological orders. *HT*-1 and

TABLE 6. Comparison of query time (ms, λ = 80%).

Dataset	$k=2$			$k=5$			$k=8$		
	BFSI-B	*PLL*	*HT*	*BFSI-B*	*PLL*	*HT*	*BFSI-B*	*PLL*	*HT*
amaze	25	30	**17**	26	29	**17**	26	31	**16**
agrocyc	30	35	**21**	30	32	**18**	30	34	**18**
ecoo	32	35	**21**	33	33	**19**	33	34	**18**
vchocyc	30	33	**19**	31	31	**17**	31	32	**17**
kegg	26	33	**21**	28	33	**22**	28	33	**21**
xmark	29	48	**27**	31.4	48	**30.9**	33	52	**27**
mtbrv	32	34	**22**	34	33	**21**	34	33	**20**
nasa	**30**	51	32	39	51	**36**	43	51	**31**
yago	70	58	**39**	93	57	**36**	80	58	**35**
arxiv	173	387	**140**	1,797	**394**	571	17,255	**392**	414
email	74	83	**50**	72	82	**50**	72	82	**47**
uniprot100m	167	155	**99**	166	152	**98**	144	151	**96**
uniprot150m	177	186	**121**	179	183	**116**	183	185	**118**
wiki	72	95	**64**	73	92	**64**	73	92	**54**
LJ	136	101	**59**	253	99	**56**	101	99	**55**
web	116	100	**67**	124	101	**63**	124	100	**56**
10cit-Patent	122	130	**107**	133	127	**105**	135	130	**108**
05cit-Patent	204	168	**158**	296	166	**152**	301	160	**137**
citeseer	466	**193**	338	1,613	**191**	219	3,811	192	**177**
dbpedia	272	146	**96**	321	148	**85**	327	146	**74**
govwild	**130**	213	143	160	213	**146**	144	211	**113**
cit-Patents	596	725	**551**	5,420	**720**	2,727	16,815	**727**	3,647
go	280	296	**209**	394	290	**179**	418	290	**151**
10go-unip	237	255	**195**	418	256	**208**	443	254	**177**
twitter	335	164	**89**	3,689	164	**90**	3,720	163	**83**



(a) amaze (b) arxiv (c) twitter (d) 10go-unip

FIGURE 6. Impacts of reachable ratio to query time on different datasets ($k = 5$).

HT have the same index construction time since they have the same index.

Finally, for query time, we can find that by using topo-order and early stop condition, the query time can be reduced significantly. The reasons lie in two aspects: (1) with topo-orders, we can quickly prune more unreachable queries, and (2) with early stop-condition, we can answer reachable queries more efficiently.

B. COMPARISON OF QUERY TIME

Table 6 shows the comparison of query time with the change of k. We have the following observations. For all the 25 real datasets, our *HT* algorithm works best on at least 22 datasets for all value of k, and the change of k has little affects on *HT*.

When $k = 2$ is small, we can see that *BFSI-B* can be more efficient than *PLL* on several datasets, such as arxiv, email, wiki and govwild. Even though, it is beaten by *PLL* on other datasets. When $k = 8$ becomes large, *BFSI-B* is much worse than *PLL* on several datasets due to the poor pruning power for reachable queries, such as on arxiv, citeseer, cit-Patents and twitter datasets.

It is worth noting that our approach *HT* is five times slower than *PLL* on cit-Patents dataset when $k = 8$. The reason lies in that our approach is a *Label+G* approach, and it cannot answer all reachable queries. For some queries, *HT* still needs to perform graph traversal. Even though, *HT* is 46 times faster than *PLL* on index construction and the index size of *PLL* is 12.5 times bigger than that of *HT*.

In Figure 6, we show the impacts of reachable ratio to query time on different datasets. Here, reachable ratio means that for a given query workload containing 1,000,000 queries, the ratio of the number of reachable queries. We select four datasets to show the results, where amaze is a small sparse graph, arxiv is a small dense graph, twitter is a large sparse graph and 10go-unip is a large dense graph. From Figure 6 we have the following observations. First, with the increase of reachable ratio, our *HT* always performs best. Second, when reachable ratio is small, *HT* works much better than *PLL*, and even when the reachable ratio equals 100%, *HT* still achieves similar performance as *PLL*. Third, *BFSI-B* could be better than *PLL* only if the reachable ratio is small, and with the increase of reachable ratio, the performance of *BFSI-B* degenerates significantly.

TABLE 7. Comparison of index construction time (ms) and index size (MB).

Dataset	Index Construction Time			Index Size		
	BFSI-B	*PLL*	*HT* ($\lambda = 80\%$)	*BFSI-B*	*PLL*	*HT* ($\lambda = 80\%$)
amaze	1.04	1.18	**0.69**	0.11	0.09	**0.08**
agrocyc	2.45	7.11	**1.77**	0.39	0.31	**0.28**
ecoo	2.62	3.37	**1.48**	0.39	0.31	**0.28**
vchocyc	1.93	2.51	**1.44**	0.29	0.24	**0.21**
kegg	0.74	1.07	**0.61**	0.11	0.09	**0.08**
xmark	**1.075**	4.69	1.082	0.19	0.25	**0.13**
mtbrv	1.85	4.34	**1.37**	0.29	0.24	**0.21**
nasa	1.16	4.99	**0.69**	0.17	0.22	**0.11**
yago	**1.82**	7.45	2.20	**0.20**	0.51	0.25
arxiv	**1.75**	232	4.52	0.18	2.68	**0.15**
email	28	78	**23**	7.05	5.23	**5.05**
uniprot100m	**2,604**	6,643	3,007	491	**394**	445
uniprot150m	**4,190**	12,033	5,040	764	**635**	709
wiki	281	617	**228**	70	53	**52**
LJ	141	360	**133**	30	23	**22**
web	**79**	223	111	11	10	**8**
10cit-Patent	**431**	964	641	34	**30**	32
05cit-Patent	**966**	2,383	1,604	51	59	**41**
citeseer	**3,401**	14,563	6,199	200	276	**127**
dbpedia	**1,226**	3,989	2,268	103	122	**73**
govwild	**2,405**	12,070	4,306	245	398	**155**
cit-Patents	**5,280**	476,502	10,346	**115**	1,749	140
go	**2,605**	15,749	6,405	**213**	533	275
10go-unip	**219**	1,453	549	**14**	47	20
twitter	**2,939**	6,728	3,032	553	417	**409**

C. INDEX CONSTRUCTION TIME AND INDEX SIZE

Table 7 shows the comparison of index construction time and index size, from which we can see that for index construction time, both *BFSI-B* and *HT* is smaller than that of *PLL*, and either *BFSI-B* or *HT* can be best on several datasets, no one can beat the other on all datasets. For index size, *HT* has the smallest index size on 17 datasets. As *PLL* needs to maintain all shortest path information, its index size is usually bigger than the other two and the index construction time is usually longer than the other two.

In summary, compared with *BFSI-B*, on one hand, our *HT* uses four topological orders on a reduced graph, while *BFSI-B* uses two topological orders on the original graph, therefore, *HT* can answer more unreachable queries. On the other hand, the partial 2hop distance label covers more reachability information than the spanning tree used by *BFSI-B*, therefore, it can answer more reachable queries. As shown by the experimental results on query time, *HT* works much better than *BFSI-B*, while at the same time, *HT* has comparable index size and index construction time compared with *BFSI-B*. Compared with *PLL*, *HT* is much more efficient when answering unreachable queries, and has the similar performance when answering reachable queries. More importantly, *HT* usually consumes less index construction time and has smaller index size. Therefore, *HT* has the best performance on average when processing k-hop reachability queries.

VIII. CONCLUSION

Considering that both existing approaches are inefficient when answering k-hop reachability queries on *DAG*s, we propose a new *Label+G* approach *HT* to efficiently answer k-hop reachability queries. We first propose a partial 2hop distance label to maintain most reachability information, such that to quickly answer many reachable queries. We then propose to use topological level and sinking topological level to quickly prune many unreachable queries. Besides, we further propose two optimization techniques to facilitate k-hop reachability queries answering. Compared with existing *Label-Only* approach *PLL*, our approach *HT* can answer unreachable queries more quickly. Compared with existing *Label+G* approach *BFSI-B*, our *HT* can be much more efficient when processing reachable queries. Our experimental results show that our approach achieves the best results on most datasets when answering k-hop reachability queries with smaller index size and reasonable index construction time. As an indication, for all the 25 real graphs, *HT* needs the least query time on 22,22 and 23 datasets compared with *BFSI-B* and *PLL*, when $k = 2$, 5 and 8, respectively.

REFERENCES

[1] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu, "TF-Label: A topological-folding labeling scheme for reachability querying in a large graph," in *Proc. SIGMOD*, 2013, pp. 193–204.

[2] R. Jin and G. Wang, "Simple, fast, and scalable reachability oracle," *J. Proc. VLDB Endowment*, vol. 6, no. 14, pp. 1978–1989, 2013.

[3] S. Seufert, A. Anand, S. Bedathur, and G. Weikum, "FERRARI: Flexible and efficient reachability range assignment for graph indexing," in *Proc. 29th IEEE Int. Conf. Data Eng. (ICDE)*, Brisbane, QLD, Australia, Apr. 2013, pp. 1009–1020.

[4] R. R. Veloso, L. Cerf, W. Meira, Jr, and M. J. Zaki, "Reachability queries in very large graphs: A fast refined online search approach," in *Proc. 17th Int. Conf. Extending Database Technol. (EDBT)*, Athens, Greece, Mar. 2014, pp. 511–522.

[5] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *J. Proc. VLDB Endowment*, vol. 7, no. 12, pp. 1191–1202, 2014.

[6] H. Yildirim, V. Chaoji, and M. J. Zaki, "GRAIL: A scalable index for reachability queries in very large graphs," *VLDB J.*, vol. 21, no. 4, pp. 509–534, 2012.

[7] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: A total order approach," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Snowbird, UT, USA, Jun. 2014, pp. 1323–1334.

[8] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida, "Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths," in *Proc. CIKM*, 2013, pp. 1601–1606.

[9] J. Su, Q. Zhu, H. Wei, and J. X. Yu, "Reachability querying: Can it be even faster?" *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 3, pp. 683–697, Mar. 2017.

[10] J. Zhou, S. Zhou, J. X. Yu, H. Wei, Z. Chen, and X. Tang, "DAG reduction: Fast answering reachability queries," in *Proc. ACM Int. Conf. Manage. Data SIGMOD Conf.*, Chicago, IL, USA, May 2017, pp. 375–390.

[11] J. Zhou, J. X. Yu, N. Li, H. Wei, Z. Chen, and X. Tang, "Accelerating reachability query processing based on *DAG* reduction," *VLDB J.*, vol. 27, no. 2, pp. 271–296, 2018.

[12] N. Sengupta, A. Bagchi, M. Ramanath, and S. Bedathur, "ARROW: Approximating reachability using random walks over web-scale graphs," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, Apr. 2019, pp. 470–481.

[13] X. Ke, A. Khan, and L. L. H. Quan, "An in-depth comparison of s-t reliability algorithms over uncertain graphs," *J. Proc. VLDB Endowment*, vol. 12, no. 8, pp. 864–876, 2019.

[14] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin, "Scaling distance labeling on small-world networks," in *Proc. Int. Conf. Manage. Data SIGMOD Conf.*, Amsterdam, The Netherlands, Jun./Jul. 2019, pp. 1060–1077.

[15] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, Jun. 2013, pp. 349–360.

[16] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu, "K-reach: Who is in your small world," *J. Proc. VLDB Endowment*, vol. 5, no. 11, pp. 1292–1303, 2012.

[17] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu, "Efficient processing of *k*-hop reachability queries," *VLDB J.*, vol. 23, no. 2, pp. 227–252, 2014.

[18] Y. Tao, Y. Li, and G. Li, "Interactive graph search," in *Proc. Int. Conf. Manage. Data SIGMOD Conf.*, Amsterdam, The Netherlands, Jun./Jul. 2019, pp. 1393–1410.

[19] S. Gurajada and M. Theobald, "Distributed set reachability," in *Proc. Int. Conf. Manage. Data SIGMOD Conf.*, San Francisco, CA, USA, Jun./Jul. 2016, pp. 1247–1261.

[20] X. Xie, X. Yang, X. Wang, H. Jin, D. Wang, and X. Ke, "BFSI-B: An improved k-hop graph reachability queries for cyber-physical systems," *Inf. Fusion*, vol. 38, pp. 35–42, Nov. 2017.

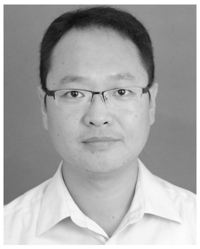
[21] K. Simon, "An improved algorithm for transitive closure on acyclic digraphs," *Theor. Comput. Sci.*, vol. 58, pp. 325–346, Jun. 1988.

[22] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," in *Proc. ACM-SIAM*, 2002, pp. 937–946.

[23] X. Tang, Z. Chen, K. Li, and X. Liu, "Efficient computation of the transitive closure size," *Cluster Comput.*, vol. 22, pp. 6517–6527, May 2019.

[24] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proc. 4th Int. Conf. Weblogs Social Media (ICWSM)*, Washington, DC, USA, May 2010, pp. 10–17.

[25] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.



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

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

ANPING YANG was born in Maanshan, Anhui, China, in 1994. He received the B.S. degree in computer and information from Anhui Polytechnic University, in 2017. He is currently pursuing the master's degree with the School of Computer Science and Technology, Donghua University, Shanghai.

His current research interests include query processing and optimization on graph data.

JUNFENG ZHOU was born in Xi'an, Shanxi, China, in 1977. He received the B.S. and M.S. degrees in computer science from Yanshan University, in 1999 and 2002, respectively, and the Ph.D. degree in computer applications from the Renmin University of China, Beijing, China, in 2009.

From 2009 to 2017, he was a Professor with Yanshan University. Since 2017, he has been a Professor with the School of Computer Science and Technology, Donghua University, Shanghai. His current research interests include information retrieval techniques, query processing on semi-structured data and graph data, and optimization.

XIAN TANG was born in Weifang, Shandong, China, in 1978. He received the B.S. and M.S. degrees from Yanshan University, in 2001 and 2006, respectively, all in computer science, and the Ph.D. degree in computer applications from the Renmin University of China, Beijing, China, in 2011.

From 2011 to 2017, he was a Lecturer with Yanshan University. Since 2017, he has been a Senior Lecturer with the School of Electronic and Electrical Engineering, Shanghai University of Engineering Science, Shanghai. His current research interests include query processing and optimization on graph data and semi-structured data and flash-based databases.

ZIYANG CHEN was born in Wuchang, Heilongjiang, China, in 1973. He received the B.S., M.S., and Ph.D. degrees from Yanshan University, China, in 1992, 2001, and 2008, respectively, all in computer science.

From 2008 to 2016, he was a Professor with Yanshan University. Since 2016, he has been a Professor with the Shanghai Lixin University of Accounting and Finance, Shanghai. His current research interests include computation theory, query processing, and optimization on graph data.

YANFEI ZUO was born in Kaifeng, Henan, China, in 1987. He received the B.S. degree in textile science and engineering from Jiangnan University, Wuxi, China, in 2009, and the M.S. degrees in computer science from Donghua University, Shanghai, China, in 2012.

He is currently the President of the Medical Big Data and AI Research and Development Center, Histo Pathology Diagnostic Center, Shanghai, China. His current research interests include medical big data analysis and pathology image analysis.