TF-Label: a Topological-Folding Labeling Scheme for Reachability Querying in a Large Graph

James Cheng, Silu Huang, Huanhuan Wu, Ada Wai-Chee Fu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{jcheng, slhuang, hhwu, adafu}@cse.cuhk.edu.hk

ABSTRACT

Reachability querying is a basic graph operation with numerous important applications in databases, network analysis, computational biology, software engineering, etc. Although many indexes have been proposed to answer reachability queries, most of them are only efficient for handling relatively small graphs. We propose TF-label, an efficient and scalable labeling scheme for processing reachability queries. TF-label is constructed based on a novel topological folding (TF) that recursively folds an input graph into half so as to reduce the label size, thus improving query efficiency. We show that TF-label is efficient to construct and propose efficient algorithms and optimization schemes. Our experiments verify that TF-label is significantly more scalable and efficient than the state-of-the-art methods in both index construction and query processing.

1. INTRODUCTION

A **reachability query** asks whether there exists a path from one vertex to another vertex in a directed graph. Reachability querying is one of the fundamental operations in directed graphs. It has a wide range of applications such as processing recursive queries in data and knowledge base management, querying associations and logical reasoning in Web and Semantic Web graphs, pattern matching in graphs and XML documents, analyzing the biological function of genes, checking connections in geographic navigation systems, social network analysis, ontology querying, program analysis, and many more.

Reachability querying has been extensively studied in the past [1, 2, 3, 4, 5, 6, 10, 11, 12, 14, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27]. In recent years, there is a shift of interest to handle large graphs. The more recent works [6, 18, 19, 24, 27] have highlighted the applications of reachability querying in large graphs such as Web graphs, Semantic Web and RDF graphs, social networks, large XML databases, etc., and more efforts have been given to the development of scalable methods for answering reachability queries.

As pointed out in [18], most existing methods can only handle relatively small graphs with tens to hundreds of thousands vertices and edges. For processing larger graphs, these methods are either too costly in index construction or in query processing (more discussion in Section 9), which severely limits their application to real world graphs.

For graphs with millions of vertices and edges, only a few methods can process them with reasonably good efficiency [19, 24, 27]. For larger graphs with tens of millions of vertices and edges, the only known method that attains reasonable indexing and querying efficiency is the recently proposed *backbone* structure [18]. A reachability query, where a vertex s can reach another vertex t, can be answered by (1) first finding all backbone vertices B_s that can be reached from s and all backbone vertices B_t that can reach t, and then (2) check whether any vertex in B_s can reach any vertex in B_t. Any existing method can be applied to the backbone graph to process Step (2), and querying is generally faster since the backbone can be significantly smaller than the original graph.

Although the backbone is used as a general framework (called *SCARAB* [18]) to further improve the scalability of any method (including ours) for processing reachability queries, an efficient and scalable method itself is still most crucial for query performance for the two main reasons (both verified in our experiments). First, SCARAB itself may not be scalable to large graphs. Second, the backbone of a large graph may still be too large for existing methods.

We propose an efficient and scalable labeling scheme, which can process large graphs that cannot be handled by SCARAB and other existing methods. Given the labels of s and t, i.e., a set of vertices that are reachable from s and can reach t respectively, we can answer whether s can reach t efficiently by simply intersecting their labels (same as [14]). We highlight the main idea of our method as follows.

We propose a novel data structure, called **topological folding** (**TF**), based on which we develop our labeling scheme, **TF-label**. Given a directed graph, we can convert it into a directed acyclic graph (DAG) by condensing each strongly connected component (SCC) in the graph into a super node. Reachability queries can be answered on the DAG since all vertices are reachable from each other within an SCC. We define a *topological structure* $\mathcal{T}$ for the DAG. TF is intuitively a structure obtained by folding $\mathcal{T}$ into half each time, which essentially implies a great reduction in the label size as labeling is processed in $O(\lg \ell)$ levels instead of a total of ℓ levels in $\mathcal{T}$. Then, we apply a labeling technique, inspired by the work of [16], on the TF structure to construct labels for answering reachability queries.

We summarize the main contribution of our work as follows.

- We propose an efficient and scalable TF-based labeling scheme for reachability query processing.
- We establish the formal correctness proof which reveals various important properties of the TF structure and our labeling scheme.

- We propose optimization techniques such as special handling of high-degree vertices to further improve the scalability of our method.
- We propose efficient algorithms for constructing the TF structure and then the labels from the TF, as well as the optimization techniques.
- Our experiments on a wide spectrum of real and synthetic datasets verify that TF-label achieves competitive indexing performance and significantly better query performance than the state-of-the-art methods [18, 19, 24, 27]. In many cases, TF-label is an order to several orders of magnitude faster in query processing. We also show that TF-label is more scalable and has stable performance with the change in various graph properties.

The rest of the paper is organized as follows. We first give some basic notations and problem definition in Section 2. Then, through Sections 3 to 7 we present the details of TF and TF-label with their design and algorithms. We evaluate the performance of TF-label in Section 8. Finally, we discuss related work in Section 9 and conclude the paper in Section 10.

2. NOTATIONS/PROBLEM DEFINITION

Given a directed graph $\mathcal{G}$, a **reachability query** asks whether there is a path from a vertex u to another vertex v in $\mathcal{G}$. We assume $u \neq v$ as it is trivial to process $u = v$. Formally, a *directed edge*, or simply an *edge* (since all edges are directed in this paper), from u to v is denoted by (u, v). A *path* P from v_1 to v_p in $\mathcal{G}$ is defined by $P = \langle v_1, \ldots, v_p \rangle$ such that (v_i, v_{i+1}) is an edge in $\mathcal{G}$ for $1 \leq i < p$. We use $u \rightarrow v$ to indicate that u can reach v (or v is reachable from u), and $u \nrightarrow v$ to indicate that u cannot reach v.

Given any two vertices u and v in a *strongly connected component* (*SCC*) of $\mathcal{G}$, u can always reach v. With this observation, existing methods first compute a compressed graph, $G = (V_G, E_G)$, of $\mathcal{G}$ as follows: the set of vertices V_G of G is the set of SCCs of $\mathcal{G}$, and a directed edge is created in G from one SCC C_1 to another SCC C_2 if there exists a directed edge (v_1, v_2) in $\mathcal{G}$, where v_1 is a vertex in C_1 and v_2 is a vertex in C_2. Then, a reachability query is answered by checking whether there is a path from C_u to C_v in G, where $C_u, C_v \in V_G$, u is a vertex in C_u and v is a vertex in C_v.

The compressed graph G created above is in fact a *directed acyclic graph* (*DAG*). Thus, for simplicity, we call G the DAG of $\mathcal{G}$ in this paper. Since the SCCs of $\mathcal{G}$ can be computed efficiently [15], we follow the convention of existing methods and assume that the input to our algorithm is the DAG of the input directed graph.

Given a DAG, $G = (V_G, E_G)$, we define the set of *in-neighbors* (*out-neighbors*) of a vertex $v \in V_G$ as $nb_{in}(v, G) = \{u : (u, v) \in E_G\}$ ($nb_{out}(v, G) = \{u : (v, u) \in E_G\}$), and the *in-degree* (*out-degree*) of v as $deg_{in}(v, G) = |nb_{in}(v, G)|$ ($deg_{out}(v, G) = |nb_{out}(v, G)|$).

Problem definition. We study the following problem: given a DAG $G = (V_G, E_G)$, compute a set of vertex labels (also called an index) for processing reachability queries, i.e., given $s, t \in V_G$, the query whether s can reach t can be efficiently answered using the labels of s and t.

3. TOPOLOGICAL FOLDING

Through Sections 3 to 6, we present our main indexing scheme, called **TF-label**, which is designed based on a novel **topological folding** scheme of the DAG of a directed graph. We first present the concept of topological folding in this section.

3.1 Basic Topological Folding

Given a DAG $G = (V_G, E_G)$, we start by assigning each vertex in G a topological level number as follows.

DEFINITION 1 (TOPOLOGICAL LEVEL NUMBER). *Given a DAG $G = (V_G, E_G)$, the **topological level number** of a vertex $v \in V_G$, denoted by $\ell(v, G)$, is defined as follows:*

- *If $nb_{in}(v, G) = \emptyset$: $\ell(v, G) = 1$;*
- *Else: $\ell(v, G) = \max\{(\ell(u, G) + 1) : u \in nb_{in}(v, G)\}$.*

*The **topological level number** of G, denoted by $\ell(G)$, is given by $\ell(G) = \max\{\ell(v, G) : v \in V_G\}$.*

Since G is a DAG, it is easy to see that every vertex $v \in V_G$ has *exactly one* topological level number, which can be derived from a topological ordering of the DAG.

Given the topological level number, we now define the *topological levels* of a DAG and state an important property that will be used in the definition of topological folding later on.

DEFINITION 2 (TOPOLOGICAL LEVELS). *A DAG $G = (V_G, E_G)$ consists of t* topological levels *of vertices, denoted by $\{L_1(G), \ldots, L_t(G)\}$, where $t = \ell(G)$, and $L_i(G) = \{v : v \in V_G, \ell(v, G) = i\}$ for $1 \leq i \leq t$.*

LEMMA 1. *Each topological level $L_i(G)$ of a DAG G, for $1 \leq i \leq \ell(G)$, is an* independent set *of G.*

PROOF. $L_i(G)$ is an independent set of G if $\forall u, v \in L_i(G)$, $(u, v) \notin E_G$ and $(v, u) \notin E_G$. Suppose to the contrary if $(u, v) \in E_G$ or $(v, u) \in E_G$, then we have either $\ell(u, G) < \ell(v, G)$ or $\ell(v, G) < \ell(u, G)$, contradicting the fact that $u, v \in L_i(G)$, i.e., $\ell(u, G) = \ell(v, G) = i$. □

To clearly illustrate the concepts, for now let us assume that the DAG G only has edges going from vertices in $L_i(G)$ to vertices in $L_{i+1}(G)$, and there is no edge going from any vertex in $L_i(G)$ to a vertex in $L_j(G)$ where $j > i + 1$ (we will handle such edges in Section 3.2). We call such a DAG a ***k*-partite DAG**, where $k = \ell(G)$. Figure 1(a) shows an example of a k-partite DAG where $k = 6$.

We define a *topological folding scheme* that recursively folds up G by taking away half of the levels, as follows.

DEFINITION 3 (TOPOLOGICAL FOLDING (TF)). *Given a $\ell(G)$-partite DAG $G = (V_G, E_G)$, the **topological folding (TF)** of G is a set of DAGs, $\mathbb{G} = \{G_1, G_2, \ldots, G_f\}$, where each $G_i = (V_{G_i}, E_{G_i})$ is defined as follows:*

- *$V_{G_1} = V_G$ and for $2 \leq i \leq f$, $V_{G_i} = \bigcup_{1 \leq j \leq \lfloor \ell(G_{i-1})/2 \rfloor} L_{2j}(G_{i-1})$;*
- *For $1 \leq i \leq f$, E_{G_i} is a set of edges with which G_i is a $\ell(G_i)$-partite DAG and $\forall u, v \in V_{G_i}$, $u \rightarrow v$ in G_i if and only if $u \rightarrow v$ in G.*

*The **topological folding number**, or **TF number**, of G, denoted by $tf(G)$, is given by $tf(G) = f = |\mathbb{G}| = \lfloor \log_2 \ell(G) \rfloor + 1$.*

Intuitively, TF folds each G_i into half (i.e., taking away half of the levels together with their vertices) to obtain G_{i-1}, starting from $G_1 = G$ to G_f which has only one level and cannot be folded any more. Hence, we have the name "topological folding".

To correctly process reachability queries, it is necessary for the edge sets E_{G_i} to maintain the reachability of the vertices. To efficiently process reachability queries, we also want each E_{G_i} to be as small as possible. Thus, the construction of E_{G_i} is an optimization problem, which is expensive to solve for a large graph, since we need to first collect the set of all paths connecting each vertex to another and then select a subset of paths with minimum number of edges while keeping the original reachability.

Although an optimal solution is costly, for the purpose of reachability indexing we find that a simple and efficient solution based on the following lemma is possible.

LEMMA 2. *Let $G = (V_G, E_G)$ be a $\ell(G)$-partite DAG and $\mathbb{G} = \{G_1, G_2, \ldots, G_{tf(G)}\}$ be a topological folding of G. For $2 \le i \le tf(G)$, $V_{G_{i-1}} \backslash V_{G_i}$ is an* independent set *of G_{i-1}.*

PROOF. According to Lemma 1, each $L_j(G_{i-1})$ for $1 \le j \le \ell(G_{i-1})$ is an independent set of G_{i-1}. According to the definition of $\mathbb{G}$, $V_{G_1} = V_G$ and for $2 \le i \le tf(G)$, $V_{G_{i-1}} \backslash V_{G_i}$ are the vertices at all the odd levels of G_{i-1}. Since each G_{i-1} is a $\ell(G_{i-1})$-partite DAG, the union of the vertices at all the odd levels of G_{i-1} is clearly an independent set of G_{i-1}. □

With Lemma 2, a simple way to construct the edge sets E_{G_i} is given as follows.

- $E_{G_1} = E_G$;
- For $2 \le i \le tf(G)$, E_{G_i} is constructed from G_{i-1} as follows: for each $v \in L_j(G_{i-1})$, where j is odd, create a new edge in E_{G_i} from each in-neighbor (if any in G_{i-1}) of v to each out-neighbor (if any in G_{i-1}) of v.

LEMMA 3. *The edge sets E_{G_i} constructed above give a valid topological folding $\mathbb{G}$ of a $\ell(G)$-partite DAG $G = (V_G, E_G)$.*

PROOF. First, each G_i is a $\ell(G_i)$-partite DAG since each edge in E_{G_i} only goes from $L_j(G_i)$ to $L_{j+1}(G_i)$, for $1 \le j \le \ell(G_i)$. Second, reachability from each vertex to another is maintained because each $u_{in} \in L_{j-1}(G_{i-1})$ is connected to each $u_{out} \in L_{j+1}(G_{i-1})$ by an edge in E_{G_i} if the edges (u_{in}, v) and (v, u_{out}) exist in G_{i-1}, where $v \in L_j(G_{i-1})$ and j is odd. □

Note that the correctness of the proof of Lemma 3 also depends on the validity of Lemma 2, because if any edge (u, v), where $u, v \in V_{G_{i-1}} \backslash V_{G_i}$, exists in G_{i-1}, then the reachability established in the proof of Lemma 3 will not be valid.

The following example illustrates the idea of topological folding.

EXAMPLE 1. *Figure 1 shows the topological folding of a 6-partite DAG G ($\ell(G) = 6$). G_2 is constructed from G_1 by adding edges $(c, f), (d, f)$, and (f, h), and then removing all vertices in the odd levels of G_1. Next, odd level vertices of G_2 are removed to form G_3.*

3.2 Dealing with Cross-Level Edges

In Section 3.1 we introduced the basic concepts and structure of topological folding of a DAG and some of its essential properties. However, the DAG G of a real world directed graph is rarely $\ell(G)$-partite. On the contrary, there can be many **cross-level edges** in G, i.e., there can be edges from vertices in $L_i(G)$ to vertices in $L_j(G)$, where $1 \le i < i + 1 < j \le \ell(G)$, as shown in Figure 2.

To deal with these cross-level edges in the DAG, we observe that each DAG G_i in a topological folding $\mathbb{G}$ need not be $\ell(G_i)$-partite, but only need the following essential properties to be maintained

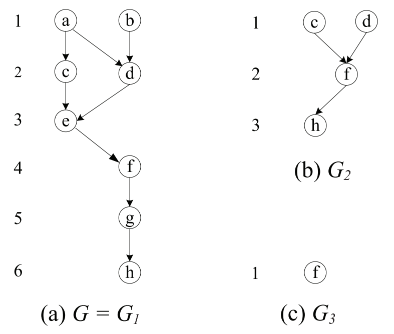


Figure 1: Topological folding

in each G_i: **(1)** *the set of vertices to be removed from G_i is an independent set of G_{i-1} for $2 \le i \le tf(G)$*; and **(2)** *$\forall u, v \in (V_{G_i} \cap V_G)$, $u \to v$ in G_i if and only if $u \to v$ in G.*

To construct each G_i that satisfies the above two properties, we devise a transformation scheme for G_{i-1}, for $2 \le i \le tf(G)$, with which we construct the corresponding **transformed topological folding** as follows:

Procedure 1. TRANSFORMED TF CONSTRUCTION:

1. $G_1 = G$, and set $i = 1$;
2. Initialize $G_i^*{=}G_i$, then do the following three steps in order:
   - 2.1. For $1 \le j \le \ell(G_i^*)$ and j is odd, for each $v \in L_j(G_i^*)$: Let $U = (L_k(G_i^*) \cap nb_{out}(v, G_i^*))$, where $k > j{+}1$. If $U \ne \emptyset$, then add a **dummy vertex** w to $L_{j+1}(G_i^*)$, add a new edge set $\{(w, u_{out}) : u_{out} \in U\}$ and a new edge (v, w) to $E_{G_i^*}$, and remove the edge set $\{(v, u_{out}) : u_{out} \in U\}$ from $E_{G_i^*}$.
   - 2.2. For $1 \le j \le \ell(G_i^*)$ and j is odd, for each $v \in L_j(G_i^*)$: Let $U = (L_k(G_i^*) \cap nb_{in}(v, G_i^*))$, where $k < j - 1$ and k is even. If $U \ne \emptyset$, then add a **dummy vertex** w to $L_{j-1}(G_i^*)$, add a new edge set $\{(u_{in}, w) : u_{in} \in U\}$ and a new edge (w, v) to $E_{G_i^*}$, and remove the edge set $\{(u_{in}, v) : u_{in} \in U\}$ from $E_{G_i^*}$.
   - 2.3. For $1 \le j \le \ell(G_i^*)$ and j is odd, for each $v \in L_j(G_i^*)$: add a new edge set $\{(u_{in}, u_{out}) : u_{in} \in (L_{j-1}(G_i^*) \cap nb_{in}(v, G_i^*)), u_{out} \in (L_{j+1}(G_i^*) \cap nb_{out}(v, G_i^*))\}$ to $E_{G_i^*}$.
3. If $\ell(G_i^*) > 1$, initialize $G_{i+1} = G_i^*$, and remove all vertices at odd levels of G_{i+1} together with all edges incident to them; then, set $i = i + 1$ and go to Step 2. Otherwise, return the **transformed topological folding** $\mathbb{G}^* = \{G_1^*, \ldots, G_{tf(G)}^*\}$ and quit.

Note that Step 2.2 ignores all Level-k in-neighbors of v if k is odd, because for this case a dummy vertex must have been created at an even level in Step 2.1, and is thus also handled in Step 2.2.

Also note that we do not increase the number of levels in any G_i or G_i^*, and hence $tf(G)$ is still defined in the same way as in Definition 3. We also define the *TF number* of a vertex as follows.

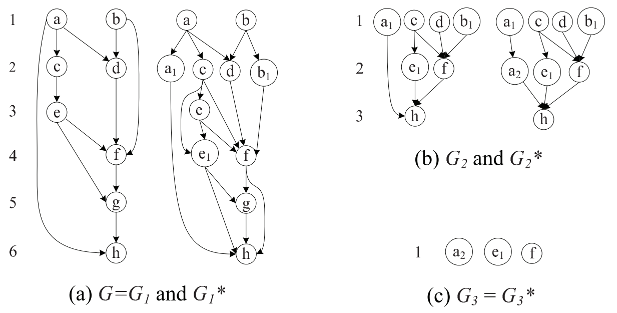
DEFINITION 4 (TOPOLOGICAL FOLDING NUMBER). *Let $G = (V_G, E_G)$ be a DAG, $\mathbb{G}^* = \{G_1^*, \ldots, G_{tf(G)}^*\}$ be the transformed topological folding of G, and let V^* be the set of dummy* *vertices created in* $\mathbb{G}^*$. *The* TF number *of a vertex* $v \in (V_G \cup V^*)$, *denoted by* $tf(v)$, *is given by* $tf(v) = \max\{i : v \in V_{G_i^*}\}$.

The TF number *of* G *is given by* $tf(G) = |\mathbb{G}^*| = \lfloor \log_2 \ell(G) \rfloor + 1$. *Also note that* $tf(G) = \max\{tf(v) : v \in V_G\}$.

We illustrate the concept using the following example.

EXAMPLE 2. *Figure 2 shows the transformed topological folding of a DAG. The DAG* G *in Figure 2(a) contains a number of cross-level edges:* $(a, h), (b, f), (d, f), (e, g)$. *By Procedure 1, we first transform* $G = G_1$ *to* G_1^*. *At level 1, Step 2.1 is executed, we add dummy vertex* a_1 *for* a, *and add edges* (a, a_1) *and* (a_1, h), *then edge* (a, h) *is removed; similarly, we add* b_1, (b, b_1) *and* (b_1, f), *and remove* (b, f). *Next consider level 3,* e_1 *is added for* e, *and we add* $(e, e_1), (e_1, g)$, *and remove* (e, g). *At Step 2.3, we add* (c, e_1) *and* (c, f). *Finally for level 5, at Step 2.3, we add* (e_1, h) *and* (f, h). *Thus, we have constructed* G_1^*, *i.e., the figure on the right in Figure 2(a). Note that in* G_1^*, *the vertices at all the odd levels are independent of each other. At Step 3 these vertices are removed, and we obtain* G_2, *as shown in Figure 2(b). Repeating the process, we obtain* G_2^* *and* G_3, *while* G_3^* *is simply the same as* G_3.

By Definition 4, $tf(v) = 1$ *for* $v \in \{a, b, e, g\}$ *since their last occurrence is in* G_1^*. *Similarly,* $tf(v) = 2$ *for* $v \in \{a_1, c, d, b_1, h\}$, $tf(v) = 3$ *for* $v \in \{a_2, e_1, f\}$, *and* $tf(G) = 3$.


(a) G=G_1 and G_1*

(b) G_2 and G_2*

(c) G_3 = G_3*

Figure 2: Transformed topological folding

One concern in the process of Procedure 1 is that many dummy vertices and edges may be created. We will handle these cases in Sections 5 and 6. In fact, G_i^* (or G_i) is also not useful for reachability processing and hence deleted after the labeling process.

The following lemma are important in establishing the correctness of our method for reachability query answering in Section 4.1.

LEMMA 4. *Let* $\mathbb{G}^* = \{G_1^*, \ldots, G_{tf(G)}^*\}$ *be the transformed topological folding of a DAG* $G = (V_G, E_G)$. *Let* G_i *be the graph from which* G_i^* *is transformed. Then,* **(1)** $V_{G_{i-1}^*} \backslash V_{G_i}$ *is an independent set of* G_{i-1}^* *for* $2 \leq i \leq tf(G)$; *and* **(2)** $\forall u, v \in (V_{G_i} \cap V_{G_i^*})$, *where* $1 \leq i \leq tf(G)$, $u \to v$ *in* G_i^* *if and only if* $u \to v$ *in* G_i; *and* **(3)** $\forall u, v \in (V_{G_i^*} \cap V_{G_j^*})$, *where* $j = i - 1$ *and* $1 < i \leq tf(G)$, $u \to v$ *in* G_i^* *if and only if* $u \to v$ *in* G_j^*.

PROOF. We first prove **(1)**. According to Procedure 1, we obtain G_i by removing the odd levels of G_{i-1}^*, i.e., $V_{G_{i-1}}^* \backslash V_{G_i}$. Since there is no edge from a vertex to another vertex at the same level in G_{i-1}^*, each level of G_{i-1}^* is an independent set of G_{i-1}^*. For any edge that goes from u at an odd level to v at another odd level, the edge is removed from G_{i-1}^* and a dummy vertex is created to preserve the connection from u to v. Thus, for any $u, v \in V_{G_{i-1}}^* \backslash V_{G_i}$, (u, v) does not exist in G_{i-1}^*.

Next we prove **(2)**. From G_i to G_i^*, Procedure 1 either converts a cross-level edge to a path with a middle dummy vertex or adds an edge from an in-neighbor to an out-neighbor of an odd-level vertex in G_i. Thus, in both cases, **(2)** is true.

Lastly, we prove **(3)**. According to Procedure 1, all the cross-level edges in G_j are removed from G_j^* and hence a vertex w at L_k of G_j^*, where $1 \leq k \leq \ell(G_j^*)$ and k is odd, has only in-neighbors at L_{k-1} (if any) and out-neighbors at L_{k+1} (if any). Since Procedure 1 creates an edge from every in-neighbor of w to every out-neighbor of w, we have $u \to v$ in G_i if and only if $u \to v$ in G_j^* for any $u, v \in (V_{G_i} \cap V_{G_j^*})$, which together with **(2)** implies **(3)**. □

Note that by a recursive analysis on **(3)** of Lemma 4, we can actually prove a stronger lemma that shows $u \to v$ in G_i^* if and only if $u \to v$ in G_j^*, for all $u, v \in (V_{G_i^*} \cap V_{G_j^*})$, where $1 \leq j < i \leq tf(G)$ (instead of $j = i - 1$ as in **(3)** of Lemma 4).

4. LABELING AND QUERY ANSWERING

In this section, we present our TF-based labeling scheme and discuss reachability query answering using the labels.

4.1 The Labeling Scheme

The label of a vertex is defined as follows.

DEFINITION 5 (VERTEX LABEL). *Let* $G = (V_G, E_G)$ *be a DAG,* $\mathbb{G}^* = \{G_1^*, \ldots, G_{tf(G)}^*\}$ *be the transformed topological folding of* G, *and let* V^* *be the set of dummy vertices created in* $\mathbb{G}^*$. *The* ***in-label*** *and* ***out-label*** *of a vertex* $v \in (V_G \cup V^*)$, *denoted by* $label_{in}(v)$ *and* $label_{out}(v)$, *are defined as follows:*

- $label_{in}(v)$: *(1)* $v \in label_{in}(v)$, *and (2) for any* $u \in label_{in}(v)$, $nb_{in}(u, G_{tf(u)}^*) \subset label_{in}(v)$.
- $label_{out}(v)$: *(1)* $v \in label_{out}(v)$, *and (2) for any* $u \in label_{out}(v)$, $nb_{out}(u, G_{tf(u)}^*) \subset label_{out}(v)$.

Intuitively, we add to $label_{in}(v)$ and $label_{out}(v)$ recursively the in-neighbors and out-neighbors in the folding graph G_i^* of each vertex u currently in $label_{in}(v)$ and $label_{out}(v)$, where $i = tf(u)$.

The following property between a vertex and its in-neighbors/out-neighbors shows that, in constructing the labels for a vertex, we only go for reachable vertices with higher TF number and ignore all other reachable vertices. This is a crucial design principle of our labeling scheme that leads to *a significant reduction on the label size (compared with transitive closure), since each vertex has* $O(\ell(G))$ *levels of reachable vertices, but only* $O(\lg \ell(G))$ *levels of reachable vertices with higher TF number.*

LEMMA 5. *If* $w \in nb_{in}(u, G_{tf(u)}^*)$ *or* $w \in nb_{out}(u, G_{tf(u)}^*)$, *then* $tf(w) > tf(u)$.

PROOF. Since w is in $G_{tf(u)}^*$, we have $tf(w) \geq tf(u)$. However, $tf(w) = tf(u)$ implies that both w and u are in an independent set of $G_{tf(u)}^*$, which contradicts the fact that the edge (u, w) or (w, u) exists in $G_{tf(u)}^*$. Thus, $tf(w) \neq tf(u)$ and $tf(w) > tf(u)$. □

We use the following example to illustrate the labeling scheme.

EXAMPLE 3. *Consider the labeling for vertex* a. *Initially,* a *is added to* $label_{in}(a)$ *and* $label_{out}(a)$. *Since* $tf(a) = 1$ *and* $nb_{in}(a, G_1^*) = \emptyset$, *we finalize* $label_{in}(a) = \{a\}$. *Next, since* $nb_{out}(a, G_1^*) = \{a_1, c, d\}$, $\{a_1, c, d\}$ *are added to* $label_{out}(a)$. *Since* a_1 *has an out-neighbor* a_2 *in* $G_{tf(a_1)}^* = G_2^*$, *we add* a_2 *to* $label_{out}(a)$. *We also add* $\{e_1, f\}$ *to* $label_{out}(a)$ *for*

$nb_{out}(c, G_2^*) = \{e_1, f\}$ and $nb_{out}(d, G_2^*) = \{f\}$. The vertices $\{a_2, e_1, f\}$ have TF number of 3 but they have no out-neighbor in G_3^*, and hence the labeling for a is completed. The labels for all vertices are shown in Table 1.

vertex	$label_{out}$	$label_{in}$
a	{a, a$_1$, c, d, e$_1$, f}	{a}
b	{b, b$_1$, d, f}	{b}
e	{e, e$_1$, f}	{c, e}
g	{g, h}	{e$_1$, f, g}
a$_1$	{a$_1$, a$_2$}	{a$_1$}
c	{c, e$_1$, f}	{c}
d	{d, f}	{d}
b$_1$	{b$_1$, f}	{b$_1$}
h	{h}	{a$_2$, e$_1$, f, h}
a$_2$	{a$_2$}	{a$_2$}
e$_1$	{e$_1$}	{e$_1$}
f	{f}	{f}

Table 1: Labeling for the example in Figure 2

4.2 Reachability Querying using Labels

We now discuss how we use the vertex labels to process reachability queries. Given two vertices s and t in G, we ask whether s can reach t, the query answer is given by the following equation.

$$s \to t = \begin{cases} \texttt{true}, & \text{if } label_{out}(s) \cap label_{in}(t) \neq \emptyset; \\ \texttt{false}, & \text{if } label_{out}(s) \cap label_{in}(t) = \emptyset. \end{cases} \quad (1)$$

We give an example of reachability query processing as follows.

EXAMPLE 4. *Consider the example in Figure 2, the labeling is shown in Table 1. Suppose the query is to ask whether c can reach h: since $label_{out}(c) \cap label_{in}(h) = \{e_1, f\}$, the answer is* `true`. *Now consider whether a can reach b: since $label_{out}(a) \cap label_{in}(b) = \emptyset$, the answer is* `false`.

Lemmas 6-9 and Theorem 1 establish the correctness of reachability query answering by Equation (1). The lemmas themselves also reveal important properties and the design of the TF structure, and hence how TF labeling works for reachability query answering.

LEMMA 6. *Given a path $P = \langle u_1, \ldots, u_\alpha \rangle$ in any graph in $\mathbb{G}^*$, there exists a sequence of vertices $S = \langle u_1 = v_1, \ldots, v_\beta = u_\alpha \rangle$ such that for $1 \leq i < \beta$:* **(1)** *the edge (v_i, v_{i+1}) is in G_j^* where $j = \min(tf(v_i), tf(v_{i+1}))$; and* **(2)** *the sequence S is maximal, i.e., no sub-sequence can be inserted between any v_i and v_{i+1} such that the resultant sequence also satisfies* **(1)**.

PROOF. The path P implies that there exists a sequence $S = \langle u_1, S_1, u_2, S_2, \ldots, u_{\alpha-1}, S_{\alpha-1}, u_\alpha \rangle$, where each S_i for $1 \leq i < \alpha$ is constructed (according to Procedure 1) as follows.

If $\ell(u_i, G_j^*) = \ell(u_{i+1}, G_j^*) + 1$, where $j = \min(tf(u_i), tf(u_{i+1}))$, then either u_i or u_{i+1} will be removed in G_{j+1} and hence S_i must be an empty set. In this case, we have (u_i, u_{i+1}) in G_j^*.

Otherwise, (u_i, u_{i+1}) is a cross-level edge in G_j, where $j = \min(tf(u_i), tf(u_{i+1}))$, then S_i is a sequence of dummy vertices. Assume $j = tf(u_i)$ (the case $j = tf(u_{i+1})$ can be processed similarly). To preserve the reachability from u_i to u_{i+1} in G_j, at least one dummy vertex w must be created in G_j^* together with the edges (u_i, w) and (w, u_{i+1}). Thus, we have the edge (u_i, w) in G_j^*. If (w, u_{i+1}) is still a cross-level edge in $G_{j'}$, where $j' = \min(tf(w), tf(u_{i+1}))$, then another dummy vertex is to be created in $G_{j'}^*$ to preserve the reachability from w to u_{i+1} in G_j^*. A recursive expansion in this way gives the sub-sequence $S_i' = \langle u_i = w_1, w_2, \ldots, w_{\gamma-1}, w_\gamma = u_{i+1} \rangle$, where $S_i = \langle w_2, \ldots, w_{\gamma-1} \rangle$, and for $1 \leq k < \gamma$, (w_k, w_{k+1}) in G_j^* and $j = \min(tf(w_k), tf(w_{k+1}))$. S_i' is ensured to be maximal if the above recursive expansion is executed until no more sub-sequence can be generated.

By relabeling the vertices, we obtain $S = \langle u_1 = v_1, \ldots, v_\beta = u_\alpha \rangle$ such that S satisfies both **(1)** and **(2)**. □

Lemma 6 is used to show that a sequence of vertices S with a special property (as specified in the lemma) exists for a path P in any graph in $\mathbb{G}^*$. The existence of such a sequence is essential in proving the correctness of Lemma 9 and hence Theorem 1.

LEMMA 7. *Given a sequence of vertices $S = \langle s = v_1, \ldots, v_\beta = t \rangle$, where for $1 \leq i < \beta$, the edge (v_i, v_{i+1}) is in G_j^* where $j = \min(tf(v_i), tf(v_{i+1}))$: if s and t are both in some graph $G_\phi^* \in \mathbb{G}^*$, then $s \to t$ in G_ϕ^*.*

PROOF. First, each edge (v_i, v_{i+1}) in G_j^* implies $v_i \to v_{i+1}$ in G_j^*. We can derive the reachability from v_1 to v_β in G_ϕ^* as follows.

Consider the vertex $v_i \in S$ where $tf(v_i) < \phi$ and $tf(v_i) \leq tf(v)$ for all $v \in S \backslash \{v_i\}$. If v_i exists in S, then according to Procedure 1, v_{i-1} must be connected to v_{i+1} in $G_{tf(v_i)}^*$ in order to preserve the the reachability from v_{i-1} to v_{i+1} via v_i. Thus, removing v_i from S we still have $v_{i-1} \to v_{i+1}$ in G_j^*, where $j = \min(tf(v_{i-1}), tf(v_{i+1}))$. We repeat the above process with $S = S \backslash \{v_i\}$ until we have $tf(v) \geq \phi$ for all remaining vertices v in S, and let $S' = \langle s = v_1, \ldots, v_{\beta'} = t \rangle$ be the new sequence obtained at the end of this process. We continue with S' as follows.

Consider the vertex $v_i \in S'$ that is not in G_ϕ^* and $tf(v_i) \geq tf(v)$ for all $v \in S' \backslash \{v_i\}$. If v_i exists in S', then we have $v_{i-1} \to v_i$ in $G_{tf(v_{i-1})}^*$ and $v_i \to v_{i+1}$ in $G_{tf(v_{i+1})}^*$. Since v_i is not in G_ϕ^* and $tf(v_i) > \phi$, v_i is a dummy vertex and v_i preserves the reachability from v_{i-1} to v_{i+1} in G_j^*, where $j = \min(tf(v_{i-1}), tf(v_{i+1}))$. Thus, removing v_i from S' we still have $v_{i-1} \to v_{i+1}$ in G_j^*. We repeat the above process with $S' = S' \backslash \{v_i\}$ until all the remaining vertices are in G_ϕ^*. Let $S'' = \langle s = v_1, \ldots, v_{\beta''} = t \rangle$ be the new sequence obtained at the end of this process.

Note that both s and t are still in S'' since s and t are in G_ϕ^*. According to the derivation process, we have $v_i \to v_{i+1}$ in G_ϕ^* for $1 \leq i \leq \beta''$, from which we have $s = v_1 \to \cdots \to v_{\beta''} = t$. Thus, $s \to t$ in G_ϕ^*. □

Lemma 7 reveals an important reachability relation between vertices in a sequence as defined in Lemma 6. This reachability relation is also crucial in the proofs of Lemmas 8 and 9.

LEMMA 8. *Given two vertices $s, t \in V_G$, if there exists a vertex $x \in label_{out}(s) \cap label_{in}(t)$, then $s \to t$ in G.*

PROOF. Let us first assume that $x \neq s$ and $x \neq t$. Then, according to Definition 5, if $x \in label_{out}(s)$, there exists a vertex $u \in label_{out}(s)$ such that $x \in nb_{out}(u, G_{tf(u)}^*)$. Moreover, $u \in label_{out}(s)$ in turn implies that there exists $u' \in label_{out}(s)$ such that $u \in nb_{out}(u', G_{tf(u')}^*)$. Thus, we obtain a sequence $S_{out} = \langle s = u_1, \ldots, u_\alpha = x \rangle$, where for $1 \leq i < \alpha$ the edge (u_i, u_{i+1}) is in $G_{tf(u_i)}^*$. Similarly, we obtain another sequence $S_{in} = \langle x = v_\beta, \ldots, v_1 = t \rangle$, where for $1 \leq i < \beta$ the edge (v_{i+1}, v_i) is in $G_{tf(v_i)}^*$. According to Lemma 5, $tf(u_i) < tf(u_{i+1})$ for $1 \leq i < \alpha$ and $tf(v_i) < tf(v_{i+1})$ for $1 \leq i < \beta$. Thus, according to Lemma 7, the sequence $S = \langle s = u_1, \ldots, u_\alpha = x = v_\beta, \ldots, v_1 = t \rangle$ implies that $s \to t$ in G_1^*, and hence $s \to t$ in $G = G_1$ by Lemma 4. If $x = t$, then

$t \in label_{out}(s)$ gives the sequence $S = \langle s = u_1, \ldots, u_\alpha = x = t\rangle$, which implies that $s \rightarrow t$ in G. And similarly for $x = s$. □

The following lemma proves the reverse statement of Lemma 8.

LEMMA 9. *Given two vertices $s, t \in V_G$, if $s \rightarrow t$ in G, then there exists a vertex $x \in label_{out}(s) \cap label_{in}(t)$.*

PROOF. We show that if $s \rightarrow t$ in G, then there exists a sequence of vertices $S = \langle s, \ldots, t\rangle$ such that there is a vertex x in S, where $x \in label_{out}(s)$ and $x \in label_{in}(t)$.

First, $s \rightarrow t$ in G implies that there is a path $P = \langle s = \ldots, t\rangle$ in G_1^* (by Procedure 1 and Lemma 4). According to Lemma 6, there exists a sequence $S = \langle s = w_1, \ldots, w_\gamma = t\rangle$ such that for $1 \leq i < \gamma$, the edge (w_i, w_{i+1}) is in G_j^* where $j = \min(tf(w_i), tf(w_{i+1}))$, and S is maximal.

Next, we show that there exists a unique vertex x in S such that $tf(x) > tf(w)$ for all $w \in S\backslash\{x\}$. It is trivially true that there exists x such that $tf(x) \geq tf(w)$ for all $w \in S\backslash\{x\}$. To remove the '$=$' sign, suppose to the contrary that there exists another vertex x' such that $tf(x') = tf(x) = j$, which implies that x and x' are both in G_j^*. Assume, without loss of generality, that x appears before x' in S. Then, $tf(x') = tf(x) = j$ implies that x and x' are both in an independent set of G_j^* according to Lemma 4. The independence between x and x' implies that either (1) $x \nrightarrow x'$ or (2) x reaches x' via some other vertex x'' in G_j^* such that $tf(x'') > tf(x)$. For (1), it is a contradiction since $x \rightarrow x'$ in G_j^* according to Lemma 7. For (2), we have the path $P' = \langle x, \ldots, x'', \ldots, x'\rangle$ in G_j^* and by Lemma 6 we can obtain another sequence $S' = \langle x, \ldots, x'', \ldots, x'\rangle$ from P', which contradicts to the fact that S is maximal.

We complete the proof by showing that the unique vertex x, where $tf(x) > tf(w)$ for all $w \in S\backslash\{x\}$, is in both $label_{out}(s)$ and $label_{in}(t)$. Let $S = \langle s = u_1, \ldots, u_\alpha = x = v_\beta, \ldots, v_1 = t\rangle$. We first consider the sub-sequence $\langle s = u_1, \ldots, u_\alpha = x\rangle$. If $s = u_1 = u_\alpha = x$, then $x \in label_{out}(s)$ by Definition 5. If $\alpha > 1$, for each u_i, we find the first u_j, where $1 \leq i < j \leq \alpha$, such that $tf(u_i) < tf(u_j)$. Such a u_j must exist since there is at least one vertex u_α where $tf(u_i) < tf(u_\alpha)$. Moreover, $u_i \rightarrow u_j$ in $G^*_{tf(u_i)}$ according to Lemma 7. Thus, (u_i, u_j) is an edge in $G^*_{tf(u_i)}$ because otherwise, u_i reaches u_j in $G^*_{tf(u_i)}$ via some other vertex u_k, which contradicts to the fact that S is maximal.

Thus, we obtain a sequence $\langle s = u'_1, \ldots, u'_{\alpha'} = x\rangle$, where $tf(u'_i) < tf(u'_{i+1})$ and (u'_i, u'_{i+1}) is an edge in $G^*_{tf(u'_i)}$ for $1 \leq i < \alpha'$. According to Definition 5, $s = u'_1 \in label_{out}(s)$, $u'_2 \in label_{out}(s)$ since $u'_1 \in label_{out}(s)$ and $u'_2 \in nb_{out}(G^*_{tf(u'_1)})$, ..., $u'_{i+1} \in label_{out}(s)$ since $u'_i \in label_{out}(s)$ and $u'_{i+1} \in nb_{out}(G^*_{tf(u'_i)})$, ..., $x = u'_{\alpha'} \in label_{out}(s)$ since $u'_{\alpha'-1} \in label_{out}(s)$ and $u'_\alpha \in nb_{out}(G^*_{tf(u'_{\alpha'-1})})$. Finally, a similar analysis shows that $x \in label_{in}(t)$. □

We note that the sequence S in the proof of Lemma 9 may not be unique, but we only need to show the existence of one such sequence for the proof.

The following theorem proves the correctness of reachability query answering by vertex labels.

THEOREM 1. *Given a reachability query whether a vertex $s \in V_G$ can reach another vertex $t \in V_G$, the answer given by Equation 1 is correct.*

PROOF. The proof follows directly from Lemmas 8 and 9. □

5. REMOVING DUMMY VERTICES

The vertex labels constructed in Section 4 contain dummy vertices, which may take up a lot of space and incur extra processing in query answering. In this section, we propose a new label with all dummy vertices removed.

According to Procedure 1, a dummy vertex w is created only as either an out-neighbor of u or an in-neighbor of v for a cross-level edge (u, v). If w is created as an out-neighbor of u (or an in-neighbor of v), then u (or v) is called the **in-source vertex** (or **out-source vertex**) of w, denoted by $src(w) = u$ (or $src(w) = v$). If $src(w) = v$ is a vertex in G, i.e., v is not a dummy vertex, then v is called the **root vertex** of w, denoted by $rt(w)$. In general, we have $rt(w) = src(src(\cdots src(w) \cdots))$.

With the definition of in-source/out-source vertices and root vertices, we define a new vertex label as follows.

DEFINITION 6 (VERTEX LABEL WITHOUT DUMMIES). *Let $f(u)$ be a function such that $f(u) = rt(u)$ if u is a dummy vertex, and $f(u) = u$ otherwise. The new labels of a vertex $v \in V_G$, denoted by $label2_{in}(v)$ and $label2_{out}(v)$, are defined as follows:*

- $label2_{in}(v) = \{f(u) : u \in label_{in}(v)\}$.
- $label2_{out}(v) = \{f(u) : u \in label_{out}(v)\}$.

Intuitively, $label2_{in}(v)$ is obtained by replacing every dummy vertex u in $label_{in}(v)$ with $rt(u)$, and similarly for $label2_{out}(v)$.

For all $v \in V_G$, $|label2_{in}(v)| \leq |label_{in}(v)|$ and $|label2_{out}(v)| \leq |label_{out}(v)|$, since there can be multiple dummy vertices with the same root vertex and/or the root vertex may already exist in the set. Thus, compared with $label$, $label2$ reduces index storage space and improves querying efficiency.

The following lemma and theorem prove the correctness of query answering using $label2$.

LEMMA 10. *Given $s, t \in V_G$, **(1)** if $x \in label_{out}(s)$ and $rt(x) \notin label_{out}(s)$, then $s \rightarrow rt(x)$ in G; and **(2)** if $x \in label_{in}(t)$ and $rt(x) \notin label_{in}(t)$, then $rt(x) \rightarrow t$ in G.*

PROOF. We first prove **(1)**. From the proof of Lemma 8, $x \in label_{out}(s)$ implies a sequence $S = \langle s = u_1, \ldots, u_\alpha = x\rangle$, where for $1 \leq i < \alpha$ the edge (u_i, u_{i+1}) is in $G^*_{tf(u_i)}$. Since x is a dummy vertex, according to Procedure 1 there exists another sequence $S_2 = \langle rt(x) = v_1, \ldots, v_{\beta-1} = src(x), v_\beta = x\rangle$, where for $1 \leq i < \beta$: either the edge (v_i, v_{i+1}) is in $G^*_{tf(v_i)}$ if $rt(x)$ is an in-source vertex, or (v_{i+1}, v_i) is in $G^*_{tf(v_i)}$ if $rt(x)$ is an out-source vertex.

If $rt(x)$ is an in-source vertex, then we construct the proof as follows. Let $y = x$. Start from $i = \alpha - 1$ to $i = 2$, we reassign $y = u_i$ if $u_i = src(y)$ (note that $i \neq 1$ since $s = u_1 = rt(x)$ contradicts $rt(x) \notin label_{out}(s)$). Let $\langle s = u_1, \ldots, u_{\alpha'} = y\rangle$ be the sub-sequence such that $u_{\alpha'-1} \neq src(y)$. According to Procedure 1, $u_{\alpha'-1}$ is an in-neighbor of $rt(x)$ so that $u_{\alpha'-1}$ is also connected to v_2 in $G^*_{tf(rt(x))}$ to preserve the reachability from $u_{\alpha'-1}$ to $rt(x)$'s cross-level out-neighbors (now via v_2). Note that v_2 may not be in $label_{out}(s)$, i.e., S, because v_2 may not be an out-neighbor of $u_{\alpha'-1}$ in $G^*_{tf(u_{\alpha'-1})}$, i.e., $tf(v_2) < tf(u_{\alpha'-1})$. Thus, we have the sequence $\langle s = u_1, \ldots, u_{\alpha'-1}, rt(x)\rangle$, where $(u_{\alpha'-1}, rt(x))$ in $G^*_{tf(rt(x))}$, from which we have $s \rightarrow rt(x)$ in G by Lemma 7.

If $rt(x)$ is an out-source vertex, then we have $\langle s = u_1, \ldots, u_\alpha = x = v_\beta, v_{\beta-1} = src(x), \ldots, v_1 = rt(x)\rangle$. Again, by Lemma 7 we have $s \rightarrow rt(x)$ in G.

Similarly we can prove **(2)**. □

THEOREM 2. *Given a reachability query whether a vertex $s \in V_G$ can reach another vertex $t \in V_G$, the answer given by Equation 1 with "$label$" replaced by "$label2$" is correct.*

PROOF. Let $X = label_{out}(s) \cap label_{in}(t)$ and $X2 = label2_{out}(s) \cap label2_{in}(t)$. We show that **(1)** if $X \neq \emptyset$, then $X2 \neq \emptyset$, and **(2)** if $X = \emptyset$, then $X2 = \emptyset$.

We first prove **(1)**. If $X \neq \emptyset$, then either (i) $\exists x \in X$, x is not a dummy vertex, or (ii) $\forall x \in X$, x is a dummy vertex. For (i), x is also in $X2$ according to Definition 6 and hence $X2 \neq \emptyset$. For (ii), $rt(x)$ is in $X2$ and hence $X2 \neq \emptyset$.

We now prove **(2)**. Suppose to the contrary that $X2 \neq \emptyset$, which must be caused by the replacement of some dummy vertex x by $rt(x)$, i.e., $rt(x) \in X2$ for some dummy vertex x. We have the following possible cases:

(i) If $x \in label_{out}(s)$ and $rt(x) \notin label_{out}(s)$: then we have $rt(x) \in label2_{out}(s)$ as a replacement of x. Thus, by Lemma 10, we have $s \to rt(x)$ in G.

Otherwise, $rt(x)$ is originally in $label_{out}(s)$ since $rt(x) \in X2$. Thus, we have $rt(x) = s$, or $s \to rt(x)$ in G by Lemma 8 since $rt(x) \in label_{out}(s)$ and $rt(x) \in label_{in}(rt(x))$.

(ii) If $x \in label_{in}(t)$ and $rt(x) \notin label_{in}(t)$: then similarly as (i) we have $rt(x) \to t$ in G by Lemma 10.

Otherwise, similarly as (i) we have either $rt(x) = t$, or $rt(x) \to t$ in G.

For every combination of the cases in (i) and (ii) above, we have $s \to t$ in G, which implies $X \neq \emptyset$ by Lemma 9 and thus a contradiction. Therefore, we have our result that $X = \emptyset$ implies $X2 = \emptyset$.

Given **(1)** and **(2)**, the correctness of the theorem follows directly from Theorem 1. □

The following example illustrates the concept of $label2$.

EXAMPLE 5. *Table 2 shows the labeling of the same graph in Example 3 with dummy vertices removed. In Table 1, we have $label_{out}(b) = \{b, b_1, d, f\}$, but $label2_{out}(b) = \{b, d, f\}$ in Table 2 since $rt(b_1) = b$ already exists in $label_{out}(b)$. For $label_{out}(c) = \{c, e_1, f\}$ in Table 1, we replace dummy vertex e_1 with $rt(e_1) = e$ and obtain $label2_{out}(c) = \{c, e, f\}$ in 2. Similarly, we obtain $label2$ for all other vertices in G.*

vertex	$label2_{out}$	$label2_{in}$
a	{a, c, d, e, f}	{a}
b	{b, d, f}	{b}
e	{e, f}	{c, e}
g	{g, h}	{e,f,g}
c	{c, e, f}	{c}
d	{d, f}	{d}
h	{h}	{a,e,f,h}
f	{f}	{f}

Table 2: Removing dummy vertices from the labels in Table 1

6. HANDLING HIGH-DEGREE VERTICES

In the construction of G^*_{i+1} from G^*_i, or G^*_i from G_i, many new edges may be created to connect the in-neighbors of a vertex v to v's out-neighbors. Although such connections are necessary to preserve reachability after v is removed, the construction is costly in the presence of high-degree vertices since the number of edges created is given by $(deg_{in}(v, G_i) * deg_{out}(v, G_i))$. The following example illustrates the problem caused by high-degree vertices.

EXAMPLE 6. *Consider the example in Figure 3(a), f is a high-degree vertex with $deg_{in}(f, G_1) * deg_{out}(f, G_1) = 3*5 = 15$. By Procedure 1, f is removed at the first iteration and we need to add many edges in order to maintain reachability in G_2 as shown in Figure 3(b). In the DAG of many real graphs, often we have a few vertices with very high degree (these vertices normally correspond to giant SCCs in the original directed graph). For example, in the p2p dataset, we have a vertex v with $deg_{in}(v, G_1) = 43562$ and $deg_{out}(v, G_1) = 366$. Such high-degree vertices will take up a lot of space in the intermediate graphs and hence incur a significant amount of extra processing in the overall labeling process.*

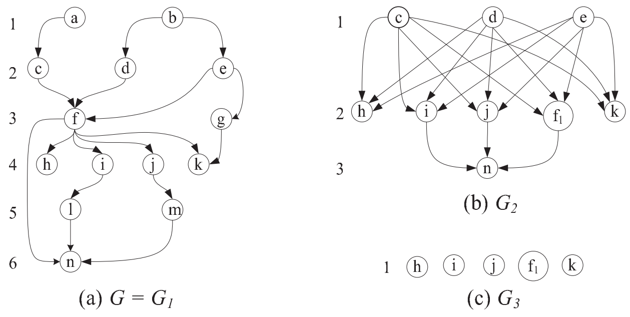


Figure 3: Problem caused by high-degree vertices

Here we propose a method to address this problem. For simplicity, in the subsequent discussion we focus on handling high-degree vertices in $G_1 = G$, but we remark that the method applies to other G_i in the same way.

Given a vertex $v \in V_G$, we define the set of vertices that are reachable from v as $reach_{out}(v, G) = \{u : v \to u\}$, and the set of vertices that can reach v as $reach_{in}(v, G) = \{u : u \to v\}$. Let H be the set of top-k high-degree vertices defined as follows: $\forall h \in H$ and $v \in V_G \backslash H$, $(deg_{in}(h, G) * deg_{out}(h, G)) \geq (deg_{in}(v, G) * deg_{out}(v, G))$. We may set k as the h-index value of a graph [8, 9].

We propose a new vertex label of a vertex $v \in V_G$, denoted by $label3_{in}(v)$ and $label3_{out}(v)$, which have *dummy vertices removed* as in Section 5 and *high-degree vertices handled* as follows:

1. For each $h \in H$, $label3_{in}(h) = \{h\}$ and $label3_{out}(h) = \{h\}$.
2. For each $v \in V_G \backslash H$, initialize $label3_{in}(v) = \{h : h \in H, v \in reach_{out}(h, G)\}$ and $label3_{out}(v) = \{h : h \in H, v \in reach_{in}(h, G)\}$.
3. Remove all vertices in H, together with all edges incident to them, from G. Let G' be the remaining graph.
4. For each $v \in V_{G'}$ (i.e., $v \in V_G \backslash H$), construct $label2_{in}(v)$ and $label2_{out}(v)$ from G' as discussed in Sections 3-5.
5. For each $v \in V_G \backslash H$, $label3_{in}(v) = label2_{in}(v) \cup label3_{in}(v)$ and $label3_{out}(v) = label2_{out}(v) \cup label3_{out}(v)$.

The following theorem proves the correctness of reachability query answering using *label3* obtained from the above steps.

THEOREM 3. *Given a reachability query whether a vertex $s \in V_G$ can reach another vertex $t \in V_G$, the answer given by Equation 1 with "$label$" replaced by "$label3$" is correct.*

PROOF. First, we show that if $s \to t$ in G, i.e., there exists a path $P = \langle s, \ldots, t\rangle$ in G, then the answer returned is `true`.

1. If P contains no vertex in H, then P must be in the remaining graph G'. Thus, query answering using "$label2$", which is constructed from G' and contained in "$label3$", returns `true` as proved in Theorem 2.

2. If P contains at least one vertex $h \in H$, then we must have $h \in label3_{out}(s)$ and $h \in label3_{in}(t)$. Thus, the answer returned is `true`.

Next, we show that if $s \nrightarrow t$ in G, then the answer returned is `false`. Suppose to the contrary that the answer is `true`, i.e., $\exists x \in (label3_{out}(s) \cap label3_{in}(t))$.

1. If $x \in H$, then we have $s \in reach_{in}(x, G)$ and $t \in reach_{out}(x, G)$, assuming that $x \neq s$ and $x \neq t$. Thus, we have $s \to x$ and $x \to t$ in G, which implies $s \to t$ in G. Now if $x = s$ or $x = t$, then $t \in reach_{out}(x = s, G)$ or $s \in reach_{in}(x = t, G)$, which again implies $s \to t$ in G. In each case, the result contradicts to the fact that $s \nrightarrow t$ in G.

2. If $x \notin H$, then $x \in label2_{out}(s)$ and $x \in label2_{in}(t)$, which implies $s \to t$ in G' by Theorem 2. Since G' is a subgraph of G, we have $s \to t$ in G, which is a contradiction.

□

The following example further illustrates the idea.

EXAMPLE 7. *Consider the example in Figure 3. We first obtain* $reach_{in}(f, G) = \{a, b, c, d, e\}$ *and* $reach_{out}(f, G) = \{h, i, j, k, l, m, n\}$. *Then, we initialize* $label3$ *for the vertices:* $label3_{out}(v) = \{f\}$ *for each* $v \in \{a, b, c, d, e\}$, *and* $label3_{in}(v) = \{f\}$ *for each* $v \in \{h, i, j, k, l, m, n\}$. *Then, we remove* f *and all edges incident to* f, *which gives the graph as shown in Figure 4(a). Next we construct the TF and then* $label2$ *from the DAG in Figure 4(a). Finally, we merge* $label2$ *and* $label3$ *to obtain the final* $label3$ *as shown in Table 3(a).*

Compared with $label2$ *computed for the graph in Figure 3(a), which is shown in Table 3(b),* $label3$ *is considerably smaller. The example also reveals that after removing the high-degree vertices, the graph becomes much easier to handle.*

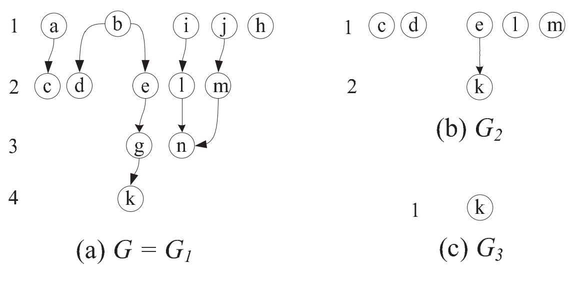


Figure 4: Topological folding with high-degree vertex removed

7. ALGORITHM AND COMPLEXITY

In this section, we discuss the algorithmic and complexity issues of our proposed method. Our method consists of two main phases, namely, the pre-processing or indexing phase and the query processing phase. Query processing is just an intersection of two sets which terminates as soon as the first common element is found and thus the complexity is bounded by the label size. The pre-processing phase includes computing the DAG from an input directed graph, topological sorting of the resulting DAG, construction of the transformed TF structure, and the label construction. The steps before labeling are either simple or have been presented in sufficient details. We therefore focus our discussion on the labeling algorithm here.

	$label3_{out}$	$label3_{in}$
a	{a,c,f}	{a}
b	{b,d,e,f,k}	{b}
c	{c,f}	{c}
d	{d,f}	{d}
e	{e,f,k}	{e}
f	{f}	{f}
g	{g,k}	{e,g}
h	{h}	{f,h}
i	{i,l}	{f,i}
j	{j,m}	{f,j}
k	{k}	{f,k}
l	{l}	{f,l}
m	{m}	{f,m}
n	{n}	{f,l,m,n}

(a)

	$label2_{out}$	$label2_{in}$
a	{a,c,f,h,i,j,k}	{a}
b	{b,d,e,f,h,i,j,k}	{b}
c	{c,f,h,i,j,k}	{c}
d	{d,f,h,i,j,k}	{d}
e	{e,f,h,i,j,k}	{e}
f	{f,h,i,j,k}	{c,d,e,f}
g	{g, k}	{e,g}
h	{h}	{h}
i	{i}	{i}
j	{j}	{j}
k	{k}	{k}
l	{l,n}	{i,l}
m	{m,n}	{j,m}
n	{n}	{f,i,j,n}

(b)

Table 3: Labeling for G in Figure3(a): (a) $label3$; (b) $label2$

Algorithm 1: Labeling($\mathbb{G}^* = \{G_1^*, \ldots, G_{tf(G)}^*\}$)

```
1  Let $V_{G_i} = \emptyset$, where $i = tf(G) + 1$;
2  for $i = 1, ..., tf(G)$ do
3      foreach $v \in (V_{G_i^*} \backslash V_{G_{i+1}})$ do
4          $label_{in}(v) \leftarrow \{v\} \cup \{u : (u, v) \in G_i^*\}$;
5          $label_{out}(v) \leftarrow \{v\} \cup \{u : (v, u) \in G_i^*\}$;

6  for $i = tf(G), ..., 1$ do
7      foreach $v \in (V_{G_i^*} \backslash V_{G_{i+1}})$ do
8          foreach $u \in label_{in}(v)$ do
9              $label_{in}(v) \leftarrow label_{in}(v) \cup label_{in}(u)$;
10         foreach $u \in label_{out}(v)$ do
11             $label_{out}(v) \leftarrow label_{out}(v) \cup label_{out}(u)$;

12 return $label_{in}(v)$ and $label_{out}(v)$ for all vertices $v$;
```

We propose an efficient top-down algorithm to construct the vertex labels defined in Definition 5. As shown in Algorithm 1, Lines 1-5 initializes $label_{in}(v)$ and $label_{out}(v)$ for each vertex v to contain the in-neighbors and out-neighbors of v in $G_{tf(v)}^*$. Note that for each $v \in (V_{G_i^*} \backslash V_{G_{i+1}})$, $tf(v) = i$ since v no longer exists in G_{i+1}. Line 1 is introduced so that $(V_{G_i^*} \backslash V_{G_{i+1}}) = V_{G_i^*}$ when $i = tf(G)$ in Lines 3 and 7, since $G_{tf(G)+1}$ does not really exist.

Lines 6-11 performs a top-down operation starting at the highest level of the TF structure. At each level i, for each vertex $v \in (V_{G_i^*} \backslash V_{G_{i+1}})$, we simply include the in-label (out-label) of v's in-neighbors (out-neighbors) in $label_{in}(v)$ ($label_{out}(v)$).

The correctness of Algorithm 1 follows from Definition 5 and Lemma 5. While the algorithm does not remove dummy vertices, we discuss how it can be handled with little additional overhead, as inspired by the following lemma.

LEMMA 11. *For any vertex* $v \in V_G$ *and any* $G_i^* \in \mathbb{G}^*$, *at most two dummy vertices will be created in* G_i^* *whose root vertex is* v.

PROOF. According to Procedure 1, initially we may create one dummy vertex u_{out} as an out-neighbor of v and/or another dummy vertex u_{in} as an in-neighbor of v. And u_{out} and u_{in} must be created in $G^*_{tf(v)}$. At most one dummy vertex (let it be w_{out}) will be created as an out-neighbor of u_{out} since all incoming edges of u_{out} are not cross-level edges by construction. And w_{out} must be created in G^*_j, where $j = tf(u_{out})$. Similarly, at most one dummy vertex will be created as an out-neighbor of w_{out}, and so on. A similar analysis applies to u_{in} and thus in any $G^*_i \in \mathbb{G}^*$, we have at most two dummy vertices created whose root vertex is v. $\square$

If v is the root vertex of any dummy vertex and v is the in-source vertex, then Lemma 11 implies the existence of a unique sequence $S_{out} = \langle v = u_1, \dots, u_\alpha \rangle$, where u_{j-1} is the in-source vertex of u_j for $1 < j \le \alpha$; thus, we can use only two labels, $label_{in}(u_j)$ and $label_{out}(u_j)$, to keep the labels for all dummy vertices u_j at each level $i = tf(u_j)$ in Lines 6-11 of Algorithm 1. Similarly, the same strategy applies to another unique sequence if v is the root vertex of a set of dummy vertex and v is the out-source vertex. Thus, in the top-down labeling process, in total we maintain at most four labels for each vertex $v \in V_G$ for all dummy vertices created with v as their root vertex.

Next we analyze the complexity of the pre-processing phase. Computing the DAG takes linear time in the size of the input directed graph. Given the DAG $G = (V_G, E_G)$, topological sorting takes $O(|V_G| + |E_G|)$ time. Then, we apply Procedure 1 to construct the TF structure, which takes $O(\lg \ell(G))$ iterations of Steps 2 and 3. At the i-th iteration, we need $O(\sum_{v \in V_{G^*_i}} (deg_{in}(v, G^*_i) * deg_{out}(v, G^*_i)))$ time for the construction. From Lemma 11, $|V_{G^*_i}| \le 2|V_{G_i}|$ and the degree of a dummy vertex w is bounded by that of $src(w)$. The total time complexity is given by $C1 = O(\sum_{1 \le i \le \lg \ell(G)} \sum_{v \in V_{G_i}} (deg_{in}(v, G_i) * deg_{out}(v, G_i)))$. The complexity of Algorithm 1, together with dummy vertex handling, is bounded by $C2 = O(\sum_{1 \le i \le \lg \ell(G)} \sum_{v \in (V_{G^*_i} \setminus V_{G_{i+1}})} (\sum_{u \in nb_{in}(v, G^*_i)} label_{in}(u) + \sum_{u \in nb_{out}(v, G^*_i)} label_{out}(u)))$. Both $C1$ and $C2$ depend on the characteristics of the input DAG, especially the vertex degree. Both $C1$ and $C2$ can be significantly reduced by removing the set of high-degree vertices H, which takes $O(|H|(|V_{G_i}| + |E_{G_i}|))$ time to remove H and add $h \in H$ to the labels of other vertices as discussed in Section 6.

8. EXPERIMENTAL EVALUATION

We implemented our method, **TF-label**, in C++ (all source codes will be made available). We compare TF-label with the following state-of-the-art methods for processing reachability queries: **PathTree** [19], **GRAIL** [27], **PWAH8** [24], **ScaPathTree** and **ScaGRAIL**. ScaPathTree and ScaGRAIL are the application of PathTree and GRAIL in the **SCARAB** framework [18], i.e., first computing the backbone of the input DAG and then applying PathTree or GRAIL for reachability querying (more details in Section 1). Though in theory any existing method can be applied in SCARAB, we were not able to do so for PWAH8 and TF-label due to unfamiliarity with their system. ScaPathTree and ScaGRAIL were provided by the authors of [18].

All source codes of the methods we compare with are the latest version provided by their authors, and all were implemented in C++ and compiled using the same gcc compiler as TF-label. We ran all experiments on a computer with an Intel 3.3 GHz CPU, 16GB RAM, and running Ubuntu 11.04 Linux OS.

8.1 Performance on Real Datasets

We first evaluate the performance of our method on real-world datasets from a wide spectrum of domains. As shown below, the first set of 7 datasets are from 3 different domains, while the second set of 5 datasets are from 5 different domains. We want to examine the differences in the spectrum of datasets that our method can handle versus those of existing methods.

Real datasets. We used the following 7 large real datasets that are used in [18, 27] for scalability test: `citeseer`, `citeseerx` and `cit-patent` (`patent`) are citation networks, in which non-leaf vertices have an average out-degree of 10 to 30; `go-uniprot` is the joint graph of Gene Ontologyterm and the annotations from the UniProt database (*www.uniprot.org*), which is the universal protein resource; `uniprot22m`, `uniprot100m` and `uniprot150m` are the subsets of the complete RFG graph of UniProt.

We also used 5 real datasets from Stanford Large Network Dataset Collection. We selected one large directed graph from each of the following categories: `email-EuAll` (`email`) from communication networks, `soc-LiveJournal1` (`LJ`) from social networks, `p2p-Gnutella31` (`p2p`) from Internet peer-to-peer networks, `web-Google` (`web`) from Web graphs, and `wiki-talk` (`wiki`) from Wikipedia networks. In addition, `cit-patent` from citation networks is already included in the first 7 graphs. Detailed descriptions of the datasets can be found in (*snap.stanford.edu/data*).

Table 4 lists the number of vertices and edges in the original directed graph, $\mathcal{G}$, as well as in the DAG G of $\mathcal{G}$, respectively. We do not show $|V_{\mathcal{G}}|$ and $|E_{\mathcal{G}}|$ for the datasets obtained from [27] since the authors did not provide these numbers. Note that existing methods for reachability querying assume that the input is a DAG. We also show the topological level number of G, $\ell(G)$, as well as the average degree of the vertices (denoted by d_{avg}) in G.

Table 4: Real datasets (K $= 10^3$)

Dataset	$\lvert V_{\mathcal{G}}\rvert$	$\lvert E_{\mathcal{G}}\rvert$	$\lvert V_G\rvert$	$\lvert E_G\rvert$	$\ell(G)$	d_{avg}
`citeseer`	–	–	694K	312K	13	0.45
`citeseerx`	–	–	6540K	15011K	59	2.30
`go-uniprot`	–	–	6968K	34770K	21	4.99
`patent`	–	–	3775K	16519K	32	4.38
`uniprot22m`	–	–	1595K	1595K	4	1.00
`uniprot100m`	–	–	16087K	16087K	9	1.00
`uniprot150m`	–	–	25038K	25038K	10	1.00
`email`	265K	420K	231K	223K	7	0.97
`LJ`	4848K	68994K	971K	1024K	24	1.05
`p2p`	63K	148K	48K	55K	14	1.14
`web`	876K	5105K	372K	518K	34	1.39
`wiki`	2394K	5021K	2282K	2312K	8	1.01

Indexing Performance. We first report indexing performance results, but remark that (online) query performance should be the more important performance indicator, provided that (offline) indexing performance is reasonable. We report the index construction time (total elapsed time in seconds) in Table 5. The shortest time for each dataset is highlighted in **bold**.

For the datasets from [27], GRAIL has the best performance and the performance of ScaGRAIL is close to that of GRAIL. The indexing time of TF-label is comparable to that of PWAH8 for most datasets. For `citeseerx` and `patent`, TF-label is 135 and 8.5 times faster than PWAH8. Compared with ScaPathTree, our method is from a few times to 74 times faster. ScaPathTree was not able to obtain the results for `citeseerx` and `patent`, while PathTree can only run on `citeseer`.

For the datasets from the Stanford Collection, TF-label is the best for indexing all the datasets. TF-label is about twice faster than

Table 5: Index construction time (in sec)

	TF-label	PathTree	ScaPathTree	GRAIL	ScaGRAIL	PWAH8
citeseer	**0.73**	26.76	1.60	0.79	0.98	0.76
citeseerx	63.60	–	–	**7.80**	15.43	8597.02
go-uniprot	47.49	–	724.67	**13.95**	16.60	52.46
patent	162.44	–	–	**7.24**	36.23	1380.76
uniprot22m	2.27	–	10.26	2.10	**2.09**	**2.09**
uniprot100m	40.29	–	1301.71	27.25	28.94	**24.10**
uniprot150m	55.48	–	4107.77	43.86	48.22	**41.07**
email	**0.10**	–	0.61	0.26	0.26	166.98
LJ	**0.55**	–	31.93	1.08	1.17	–
p2p	**0.03**	2.16	0.13	0.04	0.04	1.40
web	**0.40**	–	11.12	0.41	0.62	1559.91
wiki	**0.96**	–	–	2.54	2.35	–

GRAIL and ScaGRAIL on average, and up to orders of magnitude faster than PWAH8, PathTree and ScaPathTree. We note that we did not specifically pick these datasets, but rather simply selected one large graph from each category of directed graphs (we did leave out two categories because the DAGs of these graphs are too small, for which most existing methods will be efficient enough). Therefore, the result shows that our method is able to perform well for graphs from various domains.

Table 6 reports the index size (in MB). For the 3 `uniprot` datasets, TF-label is from about 3 to 10 times smaller than all other methods. For `citeseer`, TF-label is only worse than PathTree, but much better than the other methods. But for `citeseerx`, `patent` and `go-uniprot`, TF-label is much larger. However, for the second set of 5 datasets, TF-label is much smaller in all cases except `p2p` for which it is larger than PathTree.

Table 6: Index or label size (in MB)

	TF-label	PathTree	ScaPathTree	GRAIL	ScaGRAIL	PWAH8
citeseer	2	**1**	28	11	28	7
citeseerx	1524	–	–	**100**	285	149
go-uniprot	431	–	403	**106**	387	244
patent	4732	–	–	**58**	206	5334
uniprot22m	**6**	–	68	24	67	19
uniprot100m	**77**	–	685	246	673	209
uniprot150m	**132**	–	1071	382	1049	349
email	**0.9**	–	10	4	10	2
LJ	**4**	–	41	15	41	–
p2p	0.2	**0.1**	2	0.7	2	0.2
web	**3**	–	16	6	16	4
wiki	**9**	–	–	35	95	–

Overall, the results of indexing time and index size show that our method is very competitive in indexing performance, especially for the datasets from the Stanford Collection. In fact, only GRAIL and ScaGRAIL are able to beat TF-label for indexing a few datasets. However, next we will show that GRAIL and ScaGRAIL are significantly slower in query processing than TF-label for all datasets.

Query Performance. We randomly generate 1 million queries for each dataset and Table 7 reports the total time taken to run the queries (the shortest time for each dataset is highlighted in **bold**).

The result clearly shows that TF-label outperforms all other methods in all cases except for `p2p`, for which TF-label is comparable with PWAH8. ScaGRAIL can run on all datasets, but is from about 2 to 32 times slower than TF-label. ScaPathTree and PWAH8 are also significantly slower than TF-label, and they cannot scale to run on a number of datasets. GRAIL is up to orders of magnitude slower than TF-label and PathTree cannot scale for processing most of the datasets.

Another important feature of TF-label is that it has stable good-performance for all datasets, unlike the other methods which are slow for processing some datasets. For example, ScaGRAIL is particularly slow in processing `web`, for which ScaPathTree and PWAH8 perform reasonably well. Similarly, ScaPathTree is slow in processing `uniprot150m` and PWAH8 is slow in processing `patent`. Such a stable performance from TF-label is important for handling datasets from various application domains.

Table 7: Total query processing time (in milli-sec)

	TF-label	PathTree	ScaPathTree	GRAIL	ScaGRAIL	PWAH8
citeseer	**6**	98	85	174	63	112
citeseerx	**160**	–	–	18861	684	187
go-uniprot	**48**	–	142	365	109	449
patent	**419**	–	–	6726	1240	14593
uniprot22m	**34**	–	115	259	97	210
uniprot100m	**79**	–	198	407	155	275
uniprot150m	**95**	–	862	433	183	294
email	**14**	–	124	6715	93	146
LJ	**51**	–	207	3741919	999	–
p2p	12	22	36	9192	24	**11**
web	**49**	–	196	436682	1548	142
wiki	**39**	–	–	457529	139	–

We also emphasize that TF-label can be further applied in the SCARAB framework, as do ScaGRAIL and ScaPathTree, to improve the performance. Thus, our result is impressive since TF-label even significantly outperforms the existing methods applied in SCARAB. In the next experiment, we show that TF-label scales well where all existing methods, including SCARAB, cannot scale, for both indexing and querying.

8.2 Scalability and Effects of Various Graph Properties

We use synthetic datasets to control the different properties of the DAG graph and hence assess their effects on the performance of our method, for both efficiency and scalability.

Synthetic datasets. We consider three important properties of the DAG graph: (1) *the number of vertices* (V_G), (2) *the average vertex degree* (d_{avg}), and (3) *the number of topological levels ($\ell(G)$)*. We generate three categories of datasets as follows (let $M = 10^6$):

(C1) Fix $d_{avg} = 3$ and $\ell(G) = 7$, then: set $V_G = 5M$, $10M$, $20M$, $40M$ and $80M$, respectively.

(C2) Fix $V_G = 1M$ and $\ell(G) = 7$, then: set $d_{avg} = 10$, 20, 30, 40 and 50, respectively.

(C3) Fix $V_G = 1M$ and $d_{avg} = 3$, then: set $\ell(G) = 3$, 7, 15, 31 and 63, respectively.

For the generation of a DAG G with $|V_G|$ vertices, $|\ell(G)|$ levels, and average degree d_{avg}, we first create $|V_G|$ vertices and distribute them to the $|\ell(G)|$ levels. Then, for each vertex v at each level i, where $1 < i < |\ell(G)|$, we add one edge from a vertex selected randomly at level $i-1$ to v, and add edges from v to $(d_{avg} - 1)$ randomly selected vertices at level $j > i$ in G. To test query performance, we randomly generate 1 million queries for each dataset.

Effect of number of vertices. Figure 5 reports the performance results of processing the (C1) datasets, where we vary the number of vertices $|V_G|$ from $5M$ to $80M$ ($M = 10^6$).

For index construction, TF-label is significantly faster than all other methods except GRAIL. Compared with GRAIL, TF-label is slower when $|V_G| \leq 20M$, but is 3 times faster when $|V_G| \geq 40M$. When $|V_G| = 80M$, all other methods failed (we terminated GRAIL after it took two orders of magnitude longer time than ours). PWAH8 could only handle $5M$ vertices, while PathTree failed even with $5M$ vertices (thus not shown in Figure 5). Moreover, ScaPathTree and ScaGRAIL also cannot scale well, since SCARAB failed to construct the backbone for such large datasets.

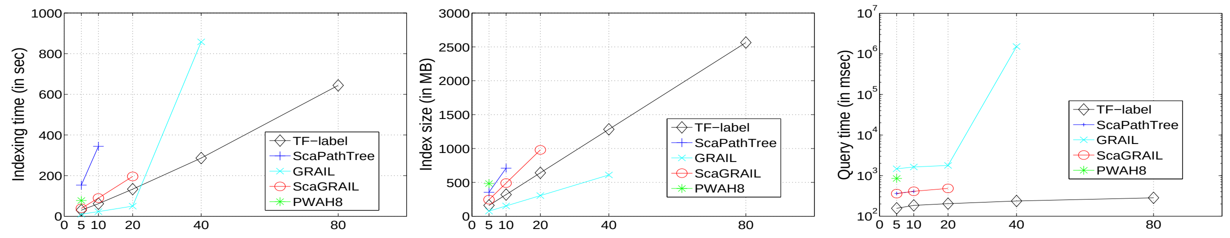


Figure 5: Performance on varying number of vertices: from $5M$ to $80M$ ($M = 10^6$)

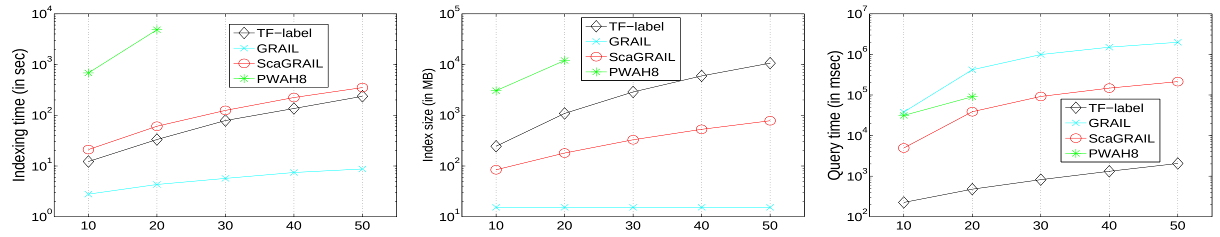


Figure 6: Performance on varying average degree: from 10 to 50

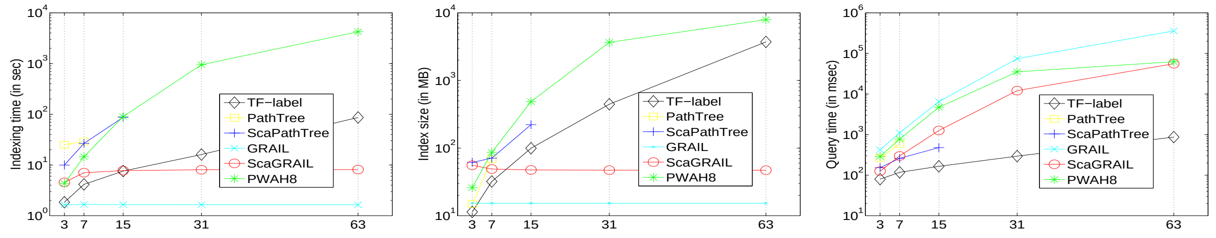


Figure 7: Performance on varying topological level number: from $(2^2 - 1)$ to $(2^6 - 1)$

The index size of TF-label is about twice that of GRAIL, and is 1.5 to 3 times smaller than that of the other methods (for the datasets they can handle).

For query processing, TF-label is again significantly faster than all the other methods. Moreover, we also see that GRAIL is the slowest and is over an order of magnitude slower than TF-label. When $|V_G| = 40M$, GRAIL is 6400 times slower than TF-label.

Overall, TF-label is shown to be much more scalable than the existing methods with the increase in the number of vertices, i.e., also in the graph size. The results also show that the indexing performance of TF-label scales linearly with the increase in the graph size, but remains reasonably stable in query performance. The reason that query time does not increases much when the graph size increases is because the average label size remains stable, which can be observed as the index size increases only linearly.

Effect of average vertex degree. Figure 6 reports the performance results of processing the (C2) datasets, where we vary the average vertex degree from 10 to 50.

The results show that both PathTree and ScaPathTree cannot scale to process datasets with average degree of even 10 (thus not shown in Figure 6). PWAH8 can only process datasets with average degree up to 20, and is up to two orders of magnitude worse than TF-label in both indexing and query performance.

TF-label is about twice faster than ScaGRAIL but is significantly slower than GRAIL in indexing, while the index size of TF-label is also much larger. However, for the more critical online query performance, both ScaGRAIL and GRAIL are too slow. ScaGRAIL is about two orders of magnitude slower and GRAIL is three orders of magnitude slower than TF-label in query processing for most of the cases.

As an index without reasonable query performance is not really useful, we can conclude that TF-label is the only method shown to be scalable with the increase in average vertex degree. TF-label scales linearly when average degree increases.

Effect of number of topological levels. Figure 7 reports the performance results of processing the (C3) datasets, where we vary the number of topological levels from $(2^2 - 1)$ to $(2^6 - 1)$ (which means that $tf(G)$ ranges from 2 to 6).

For index construction, TF-label is from a few times to 60 times faster than PathTree, ScaPathTree, and PWAH8. TF-label is faster than ScaGRAIL for the level number up to 15, but is slower than both ScaGRAIL and GRAIL in other cases. But in these cases ScaGRAIL and GRAIL are too slow in query processing. The index size also shows a similar trend.

For query processing, TF-label significantly outperforms all the other methods in all cases. Especially when the level number increases to 15 or more, TF-label is an order to two orders of magnitude faster than the other methods.

The results also show that TF-label scales roughly linearly when the level number increases, while the other methods scale poorly especially for query processing.

9. RELATED WORK

A reachability query can be answered in $O(|V_G| + |E_G|)$ time by a BFS or DFS in the input graph G, or in $O(1)$ time by precomputing the transitive closure [22] in $O(|V_G||E_G|)$ time. Existing methods all strive to attain high online query efficiency with a low offline index construction cost.

The full transitive closure is often too large and hence various labeling or compression schemes have been processed to reduce the label size [1, 5, 6, 17, 19, 24, 25]. Although these methods achieve reasonable query efficiency, most of them have a high indexing cost and are not efficient enough for processing large graphs. As we discussed in Section 1, a backbone structure was proposed as a general framework [18] on which existing methods such as [19] can be applied to handle larger graphs. However, we show in Section 8 that the performance of our method is significantly better than the state-of-the-art methods [19, 27] applied in the backbone framework.

There is another category of methods that construct vertex labels by traversing the graph only [4, 23, 27], and hence have a relatively low index construction cost. While these methods can efficiently answer a subset of queries that are supported by the labels, in general a much larger subset of queries are not covered by the index and are very costly to process as it requires graph traversal.

There are also a number of methods [2, 3, 10, 11, 12, 20, 21] that can be considered as improvements over the 2-hop labels [14], which constructs $label_{in}(v)$ and $label_{out}(v)$ for each vertex v and queries are answered as in Equation (1). Unlike our method, these methods are all very costly to construct and cannot scale to large graphs.

Due to space limit, we cannot discuss every method in greater details. More detailed discussions on the above existing methods can be found in [6, 18, 27, 28].

This work is inspired by the work [16], where a hierarchical structure is proposed for processing shortest path distance queries. However, the application of the topological structure and the design of topological folding are unique. In particular, our TF structure has at most $\lg \ell(G)$ levels, which is small for real graphs, while the hierarchical structure in [16] can have many levels.

10. CONCLUSIONS

We introduced a novel and highly effective indexing scheme, TF-label, for reachability querying in large graphs. Based on an extensive set of experimental studies, we showed that TF-label has a very stable high performance in query processing, which is typically an order of magnitude faster than the best previous methods [18, 19, 24, 27], while TF-label also enjoys competitive indexing performance. To our knowledge TF-label is the only truly scalable method since known scalable methods suffer from slow query response time for graphs with large sizes, large average degrees or large number of topological levels, while TF-label stays efficient. The ability to handle a wide range of different graph properties also demonstrates the suitability of TF-label for processing graphs from various application domains.

A useful extension of the current work is to develop I/O-efficient algorithms to index graphs that cannot fit in main memory. Methods developed in [7, 13, 26] may be applied to achieve this task.

11. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD Conference*, pages 253–262, 1989.

[2] R. Bramandia, B. Choi, and W. K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *WWW*, pages 845–854, 2008.

[3] J. Cai and C. K. Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *CIKM*, pages 119–128, 2010.

[4] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on DAGs. In *VLDB*, pages 493–504, 2005.

[5] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, pages 893–902, 2008.

[6] Y. Chen and Y. Chen. Decomposing DAGs into spanning trees: A new way to compress transitive closures. In *ICDE*, pages 1007–1018, 2011.

[7] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD Conference*, pages 457–468, 2012.

[8] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD Conference*, pages 447–458, 2010.

[9] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.*, 36(4):21, 2011.

[10] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-reach: Who is in your small world. *PVLDB*, 5(11):1292–1303, 2012.

[11] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[12] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, pages 193–204, 2008.

[13] S. Chu and J. Cheng. Triangle listing in massive networks. *TKDD*, 6(4):17, 2012.

[14] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[15] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2001.

[16] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *PVLDB*.

[17] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[18] R. Jin, N. Ruan, S. Dey, and J. X. Yu. Scarab: scaling reachability computation on large graphs. In *SIGMOD Conference*, pages 169–180, 2012.

[19] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, 36(1):7, 2011.

[20] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD Conference*, pages 813–826, 2009.

[21] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *EDBT*, pages 237–255, 2004.

[22] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1-3):325–346, 1988.

[23] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD Conference*, pages 845–856, 2007.

[24] S. J. van Schaik and O. de Moor. A memory efficient

reachability data structure through bit vector compression. In *SIGMOD Conference*, pages 913–924, 2011.

[25] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.

[26] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[27] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.

[28] J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. 2010.