Reachability Querying: An Independent Permutation Labeling Approach

Hao Wei, Jeffrey Xu Yu, Can Lu
Chinese University of Hong Kong
Hong Kong, China
{hwei,yu,lucan}@se.cuhk.edu.hk

Ruoming Jin
Kent State University
Kent, OH, USA
jin@cs.kent.edu

ABSTRACT

Reachability query is a fundamental graph operation which answers whether a vertex can reach another vertex over a large directed graph G with n vertices and m edges, and has been extensively studied. In the literature, all the approaches compute a label for every vertex in a graph G by index construction offline. The query time for answering reachability queries online is affected by the quality of the labels computed in index construction. The three main costs are the index construction time, the index size, and the query time. Some of the up-to-date approaches can answer reachability queries efficiently, but spend non-linear time to construct an index. Some of the up-to-date approaches construct an index in linear time and space, but may need to depth-first search G at run-time in $O(n + m)$. In this paper, as the first, we propose a new randomized labeling approach to answer reachability queries, and the randomness is by independent permutation. We conduct extensive experimental studies to compare with the up-to-date approaches using 19 large real datasets used in the existing work and synthetic datasets. We confirm the efficiency of our approach.

1. INTRODUCTION

Reachability query is one of the fundamental graph operations to answer whether a vertex can reach another vertex over a large directed graph. The real applications that need this operation are many and can be found among online social networks, biological networks, ontology, transportation networks, etc. The reachability query has been extensively studied over a decade [1, 16, 13, 23, 11, 28, 7, 26, 12, 8, 21, 20, 29, 27, 9, 10, 19, 24, 31], and the early work can be traced back to 1989 to compute transitive closure (TC) over a graph. However, it is still an unsolved question whether we can do faster to answer reachability queries online over even larger and/or even denser graphs with the possible minimum cost (time/space) for offline precomputing and preparation.

The main idea behind the approaches in the literature is to compute a label for every vertex in a graph G by precomputing offline. This is known as index construction, because it is to construct an index to maintain all the labels computed to vertices of a graph. The index construction needs time and space to be done, and the quality of the labels computed offline will affect the query time for answering reachability queries online. In this work, we classify all the existing works in the literature into two categories. One is called Label-Only. By the name, the approaches in this category only use the labels computed to answer reachability queries, and include *Chain-Cover* [16, 8], *Tree-Cover* [1], *Dual-Label* [28], *2-Hop* [13], *Path-Tree* [21], *3-Hop* [20], *PWAH8* [27], *TF-Label* [10], *HL* [19], and *DL* [19]. The other is called Label+G. By the name, the approaches in this category use labels computed where possible, and conduct depth-first-search (*DFS*) at run-time, if the reachability queries cannot be answered using the labels only. Such approaches include *Tree+SSPI* [7], *GRIPP* [26], *GRAIL* [29], and *Ferrari* [24].

All the approaches take a different way to balance the three main costs, namely, the index construction time, the index size, and the query time. The Label+G approaches construct an index in linear (time/space). By linear we mean it in terms of the number of vertices (n) plus the number of edges (m) of a graph G. The up-to-date Label+G approaches are *GRAIL* and *Ferrari*. However, the Label+G approaches may take long query time when it needs *DFS* to search the destination vertex over a large graph in $O(n + m)$. On the other hand, there are two main ways taken by the Label-Only approaches. (1) Some works aim at constructing a small index by compressing TC, because it leads to a small index, and therefore reduces query time [1, 16, 13, 28, 11, 12, 8, 21, 20, 27]. (2) Some works aim at constructing an index fast. Different from the Label+G approaches, none of the Label-Only approaches can construct an index in linear time, and the index size by an Label-Only approach can be either non-linear or be small but cannot be bounded. The up-to-date Label-Only approaches are all from (2) including *TF-Label*, *HL*, and *DL*.

The main contributions of this work are summarized below. First, different from all the existing approaches, as the first, we study a new approach which employs randomness. With the randomness introduced, we can construct an index fast, and answer reachability queries efficiently. Second, we propose a novel labeling, denoted as *IP*, based on independent permutation [3]. We discuss the ideas, give the algorithms, and show the bounds. Third, we propose two additional labels which can be computed in linear time and used to reduce the search cost when *DFS* is needed at run-time. Finally, we conduct extensive experimental studies to compare with the up-to-date approaches using 19 large real datasets used in the existing work and synthetic datasets. We confirm the efficiency of our approach.

The remainder of the paper is organized as follows. We discuss the preliminaries and the problem in Section 2, and discuss the related work in Section 3. We give the main ideas of *IP* labels in Section 4 followed by the discussion on the algorithms to compute *IP* labels in Section 5. We also discuss two additional labels to

Proceedings of the VLDB Endowment, Vol. 7, No. 12

be used to together with *IP* at run-time in Section 6. We give our algorithm to answer reachability queries in Section 7. We report our experimental studies using large real and synthetic datasets in Section 8, and conclude our paper in Section 9.

2. PRELIMINARIES

We model a directed graph as $G = (V, E)$ where $V(G)$ represents a set of vertices and $E(G)$ represents a set of edges (ordered pairs of vertices) of G. We may simply use V and E if the context is obvious. The numbers of vertices and edges in G are denoted as $n = |V|$ and $m = |E|$, respectively. A path from a vertex u to a vertex v is defined as $\mathsf{path}(u, v) = (v_1, v_2, \cdots, v_p)$ where (v_i, v_{i+1}) is an edge in E, for $1 \leq i < p$, $u = v_1$, and $v = v_p$. The length of $\mathsf{path}(u, v)$ is the number of edges in the path. The distance between two vertices, u and v, denoted as $\mathsf{dst}(u, v)$, is the shortest path distance (length) from u to v in G. A vertex u is said to reach v if there exists a path from u to v over G. In the following, we use $Out(u)$ to denote the entire set of vertices that u can reach including u itself, and use $In(u)$ to denote the entire set of vertices in which every vertex can reach u including u itself. In addition, we use $N_O(u)$ and $N_I(u)$ to denote the out-neighbors and in-neighbors of u, such as $N_O(u) = \{v \mid (u, v) \in G\}$ and $N_I(u) = \{v \mid (v, u) \in G\}$, respectively. The in-degree, out-degree, and the degree of a vertex u is denoted as, $d_I(u) = |N_I(u)|$, $d_O(u) = |N_O(u)|$, and $d(u) = d_I(u) + d_O(u)$.

A *reachability query*, denoted as $\mathsf{Reach}(u, v)$, is to answer if u can reach v over G. We use $u \rightsquigarrow v$ to denote u can reach v (reachable), and $u \not\rightsquigarrow v$ otherwise.

In this paper, like the existing works, we assume G is a directed acyclic graph (DAG). This is because any directed graph G can be condensed into a DAG, where a vertex in DAG represents a strongly connected component (SCC) in G, and an edge (S_i, S_j) in DAG represents that there is at least one edge from a vertex in an SCC that S_i represents to a vertex in another SCC that S_j represents. Therefore, $\mathsf{Reach}(u, v)$ in G can be answered by $\mathsf{Reach}(S_i, S_j)$ in the corresponding DAG, if u and v belong to S_i and S_j, respectively. Note that u and v are reachable, $u \rightsquigarrow v$ and $v \rightsquigarrow u$, if u and v co-exist in the same SCC in G.

The problem we study in this paper is to answer $\mathsf{Reach}(u, v)$ online when G is very large. It is commonly known that the online breadth/depth-first search (*BFS*/*DFS*) algorithms cannot answer $\mathsf{Reach}(u, v)$ for a large graph G efficiently, because both have the time complexity in $O(m + n)$. On the other hand, although it is possible to answer $\mathsf{Reach}(u, v)$ in $O(1)$ by maintaining the edge transitive closure (TC) of G, the space complexity for maintaining TC is $O(n^2)$, which is too large for a very large graph G, with the time complexity $O(nm)$ to compute TC.

3. RELATED WORKS

In general, the main idea to answer $\mathsf{Reach}(u, v)$ online is to construct an index offline, where every entry in the index keeps a label computed for a vertex $u \in G$, denoted $\mathsf{label}(u)$. $\mathsf{Reach}(u, v)$ can be answered either by Label-Only approach or Label+G approach. The former is to answer $\mathsf{Reach}(u, v)$ by $\mathsf{label}(u)$ and $\mathsf{label}(v)$ only, and the latter is to answer $\mathsf{Reach}(u, v)$ by $\mathsf{label}(u)$ and $\mathsf{label}(v)$ with the possibility of accessing G if needed. There are three main costs, namely, query time (the time to answer online), construction time (the time to construct an index offline), and the index size (the space needed to maintain the index). Here, as indicated in many works, the query time is in the range between $O(1)$ with TC as an index and $O(n + m)$ without any index, the index size is between $O(n^2)$ with TC and $O(1)$ without an index computed, and the time complexity for index construction can be very high, when it aims at compressing TC to minimal. A survey can be found in [31]. Table 1 summarizes the three main costs for the existing work.

	Query Time	Index Size	Construction Time
BFS/*DFS*	$O(n+m)$	$O(1)$	$O(1)$
TC [25, 27]	$O(1)$	$O(n^2)$	$O(nm)$
Chain-Cover [8]	$O(\log k)$	$O(kn)$	$O(n^2 + kn\sqrt{k})$
Tree-Cover [1]	$O(\log n)$	$O(n^2)$	$O(nm)$
Dual-Label [28]	$O(1)$	$O(n + t^2)$	$O(n + m + t^3)$
Path-Tree [21]	$O(\log^2 k)$	$O(kn)$	$O(km)$ or $O(nm)$
2-Hop [13]	$O(m^{1/2})$	$O(nm^{1/2})$	$O(n^3 \cdot \lvert TC\rvert)$
3-Hop [20]	$O(\log n + k)$	$O(kn)$	$O(kn^2 \cdot \lvert Con(G)\rvert)$
TF-Label [10]	–	–	$O(T)$
HL [19]	–	–	$O(H)$
DL [19]	–	–	$O(n(n+m)L)$
Tree+SSPI [7]	$O(m-n)$	$O(n+m)$	$O(n+m)$
GRIPP [26]	$O(m-n)$	$O(n+m)$	$O(n+m)$
GRAIL [29]	$O(k)$ or $O(n+m)$	$O(kn)$	$O(k(n+m))$
Ferrari [24]	$O(k)$ or $O(n+m)$	$O((k+s)n)$	$O(k^2m + S)$
IP+ (ours)	$O(k)$ or $O(knr^2)$	$O((k+h)n)$	$O((k+h)(m+n))$

Table 1: Time and Space Complexity ($T = \sum_{1 \leq i \leq \log \ell(G)} \sum_{v \in V(G_i^*) \setminus V(G_{i+1})} h(v)$, $H = \sum_{1 \leq i < h} (\sum_{v \in V(B_i) \setminus V(B_{i+1})} g(v))$)

Label-Only: Several approaches proposed aim at finding a way to compress TC, because a smaller index size implies labels computed for vertices are smaller, which leads to a better query time. Jagadish [16] first introduces chain decomposition, which is also known as *Chain-Cover*, to compress TC by finding a minimal number of pair-wise disjoint chains to represent DAG. Here, a chain is a sequence of $(v_1, v_2, \cdots, v_c)$ in which $v_i \rightsquigarrow v_j$ if $i \leq j$. In G, every vertex v is assigned to a pair of (c_i, p_j) where c_i is the chain id v belongs to and p_j is the position of v in c_i, and $\mathsf{label}(u)$ is a set of such pairs. By *Chain-Cover*, $u \rightsquigarrow v$ if there is a pair (c_i, p_j) in $\mathsf{label}(u)$, such that v is in the chain c_i and its position is $\geq p_j$. Let k be the minimal number of chains in DAG, the query time of *Chain-Cover* is $O(\log k)$. Chen and Chen propose algorithms in [8] to compute *Chain-Cover* to construct an index of $O(kn)$ in $O(n^2 + kn\sqrt{k})$ time. Agrawal et al. propose *Tree-Cover* [1] that covers TC using an optimal spanning tree, based on which a vertex, v, is assigned to an interval $[s, e]$, where e is the postorder of v, and s is the smallest postorder of v's descendants. Here, $\mathsf{label}(u)$ in *Tree-Cover* is a set of intervals, and $u \rightsquigarrow v$ if the interval of v is fully contained in an interval in $\mathsf{label}(u)$. The query time is $O(\log n)$, but its index size is $O(n^2)$ and the construction time is $O(nm)$, which are both high. Wang et al. propose *Dual-Label* [28] for a sparse graph. By *Dual-Label*, $\mathsf{label}(v)$ has two parts. One is a single interval for answering reachability over a spanning tree of G, and the other is to deal with the transitive closure over the non-tree edges of G. Let the number of non-tree edges be t. The transitive closure is $O(t^2)$ for $t \ll n$ over a sparse graph G. *Dual-Label* can achieve $O(1)$ query time, since it works as to maintain TC. *Dual-Label* is constructed in $O(n+t^2)$ time and in $O(n+m+t^3)$ space. When G becomes denser, t will approach n. Jin et al. propose *Path-Tree* [18], which decomposes a DAG G into a set of pair-wise disjoint paths. *Path-Tree* shares the similar ideas used in *Chain-Cover*. But, unlike *Chain-Cover*, *Path-Tree* uses paths instead of chains. The query time is $O(\log^2 k)$ where k is the number of paths computed in *Path-Tree*. *Path-Tree* can have a smaller index size than *Chain-Cover*. van Schaik and de Moor propose a bit-vector approach to compress TC [27]. Even though many approaches make use of a spanning tree to construct an index using intervals (a pair of numbers), the construction time is non-linear and cannot deal with larger and denser graphs.

Cohen et al. in [13] propose *2-Hop* label. A $\mathsf{label}(u)$ consists of $L_{out}(u)$ and $L_{in}(u)$, where $L_{out}(u)$ is a subset of vertices that u can reach ($L_{out}(u) \subseteq Out(u)$) and $L_{in}(u)$ is a subset of those vertices that can reach u ($L_{in}(u) \subseteq In(u)$). *2-Hop* compresses TC. By *2-Hop*, $u \rightsquigarrow v$ if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. Computing the optimal *2-Hop* for G is known to be NP-hard [13], since it is a set-cover problem. In [13], an approximate (greedy) algorithm is proposed. Several heuristic approaches are proposed to compute *2-Hop* [23, 11, 12]. Furthermore, Jin et al. propose *3-Hop* [20] to improve *2-Hop* by utilizing the idea of the chain decomposition as used in *Chain-Cover*. It takes $O(\log n + k)$ query time, but it takes $O(kn^2 \cdot |Con(G)|)$ time to construct *3-Hop* with the space complexity of $O(kn)$, where k is the number of chains and $Con(G)$ is the transitive closure contour (Definition 3 in [20]). Cai et al. propose *Path-Hop* [5] to replace the chain decomposition with a tree structure to improve *3-Hop*.

Cheng et al. [10] propose *TF-Label* (topological folding) to compute *2-Hop* labels for a DAG G using topological level. A vertex in G has a level assigned, denoted as $\ell(u, G)$, where the min level is 1, and the max level is $\ell(G)$. Given the levels, $V(G)$ can be represented as a disjoint set of vertices, such that $V(G) = \cup_{i=1}^{\ell(G)} L_i(G)$, where $L_i(G)$ is the set of vertices at the level i. Furthermore, G can be represented as a sequence of DAGs, such that $\mathbb{G} = (G_1, G_2, \cdots G_f)$, for $f = \lfloor \log_2 \ell(G) \rfloor + 1$. Here, $G_1 = G$, and G_i for $i > 1$ is constructed as follows: $V(G_i) = \cup_{1 \leq j \leq \lfloor \ell(G_{i-1})/2 \rfloor} L_{2j}(G_{i-1})$ and $E(G_i)$ is a set of edges, (u, v), over $V(G_i)$ if $u \rightsquigarrow v$ in the original G and $\ell(v, G_i) = \ell(u, G_i) + 1$ (non-cross-edge). For example, suppose a DAG G has 6 levels $\ell(G) = 6$. Then, $G_1 = G$. G_2 is a DAG with 3 levels consisting of all vertices at the level 2, 4, and 6 of G_1, and all non-cross-edges. G_3 is a DAG with 1 level. Since G_i may have cross-edges from a vertex at the level i to a vertex at level j for $j > i + 1$, *TF-Label* transforms G_i to G_i^* such all edges in G_i^* are from a vertex at the level i to a vertex at the level i+1 by adding dummy vertices. The topological folding reduces the computational cost to compute *2-Hop* labels. The bound for index construction time given [10] is $O(\sum_{1 \leq i \leq \log \ell(G)} \sum_{v \in V(G_i^*) \setminus V(G_{i+1})} h(v))$, where $h(\cdot)$ is the cost of computing *2-Hop* label for v. It is worth noting that the time complexity of *TF-Label* is non-linear, because it needs to merge *2-Hop* labels for a vertex v in G_{i+1} to G_i. Also because *TF-Label* aims at computing *2-Hop* labels efficiently instead of attempting to minimizing the index size, there is no tight bound for the index size, and the time complexity for query time depends on the index size.

Jin et al. [17] propose *SCARAB* as a general framework to represent a reachability backbone, denoted as a graph $B(V, E)$, for a graph G. The reachability backbone B represents every $u \rightsquigarrow v$ that are reachable in G, if u and v exist in B, given $V(B) \subseteq V(G)$. With the reachability backbone B, $\mathsf{Reach}(u, v)$ for u and v in G can be answered. That is, $u \rightsquigarrow v$, if there exists a pair of local neighbor vertices, u' and v', such that $\mathsf{dst}(u, u') \leq \epsilon$, $u' \rightsquigarrow v'$ in B, and $\mathsf{dst}(v', v) \leq \epsilon$. Any algorithms can be used to construct an index over B which is smaller than G. In [19], Jin et al. further propose *HL* (Hierarchical Labeling) over *SCARAB*. Informally, a graph G can be represented as a sequence of reachability backbones, $\mathbb{B} = (B_0, B_1, B_2, \cdots, B_h)$, where $B_0 = G$ and B_{i+1} is the reachability backbone of B_i, for $i > 0$. With *HL*, the *2-Hop* labels computed for B_{i+1} can be merged to B_i. The time complexity of *HL* construction time is $O(\sum_{1 \leq i < h} (\sum_{v \in V(B_i) \setminus V(B_{i+1})} g(v))$, where h is the number of backbones, and $g(\cdot)$ is the cost of computing *2-Hop* label for v. Also, in [19], Jin et al. propose *DL* (Distribution Labeling). It is based on a list of vertices in an order $\mathbf{V} = (v_1, v_2, \cdots, v_n)$. Given $\mathbf{V}$, the *2-Hop* labels are computed by breadth-first search (*BFS*) for every vertex v_j twice: *BFS*-forward following the direction of edges and *BFS*-backward following the reversed direction of edges, with an early stop condition. First, we assign v_1 for every vertex in $Out(v_1)$ and $In(v_1)$. That is, $L_{in}(w) = \{v_1\}$ for every $w \in Out(v_1)$, and $L_{out}(u) = \{v_1\}$ for every $u \in In(v_1)$. Second, consider *BFS* of v_i in $\mathbf{V}$ for $i > 1$. v_i will be inserted into $L_{in}(v_i)$. Assume the *BFS*-forward reaches v_j from v_i, v_i will be inserted into $L_{in}(v_j)$ if $L_{in}(v_j) \cap L_{out}(v_i) = \emptyset$, otherwise *BFS*-forward will not continue from v_j. In a similar way, assume the *BFS*-backward reaches v_j from v_i, v_i will be inserted into $L_{out}(v_j)$ if $L_{out}(v_j) \cap L_{in}(v_i) = \emptyset$, otherwise *BFS*-backward will not continue from v_j. The early stop condition makes the *2-Hop* labels for G compact. The time complexity of *DL* is $O(n(n+m)L)$ where L is the maximal labeling size. However, like *TF-Label*, *HL* and *DL* aim at reducing the construction time to compute *2-Hop* labels, and the index size and the query time are not tightly bounded.

Label+G: *Tree+SSPI* [7] and *GRIPP* [26] are two works that use an interval label for every vertex over a spanning tree, and attempt to reduce depth-first-search (*DFS*) time if needed at run-time. The query time for *Tree+SSPI* and *GRIPP* are $O(m - n)$, and both construction time and index size are $O(n + m)$.

Yildirim et al. propose *GRAIL* [29, 30], which randomly generates k *DFS* spanning trees to cover G, which can significantly reduce query time than that of using a single spanning tree. The label of a vertex is k intervals, $\mathsf{label}(v) = (I_1, I_2, \cdots, I_k)$, where the j-th interval $I_j = [s_j, e_j]$ is computed by the j-th *DFS*. Let $\mathsf{label}(v) \subseteq \mathsf{label}(u)$ if all intervals I_j of v are contained in I_j of u computed in the same *DFS*, and $\mathsf{label}(v) \not\subseteq \mathsf{label}(u)$ otherwise. *GRAIL* can only answer $u \not\rightsquigarrow v$ over G if $\mathsf{label}(v) \not\subseteq \mathsf{label}(u)$. But, *GRAIL* needs to do *DFS* from u to reach v at run-time, if $\mathsf{label}(v) \subseteq \mathsf{label}(u)$. The *DFS* from u does not need to continue at a vertex w, if $\mathsf{label}(v) \not\subseteq \mathsf{label}(w)$. The query time is either $O(k)$ using the label only or $O(n + m)$ when it needs to do *DFS*.

Seufert et al. propose *Ferrari* [24]. Like *GRAIL*, *Ferrari* computes up to k intervals for every vertex over an optimal spanning tree computed by [1]. Let $\mathsf{label}(u) = (I_1, I_2, \cdots)$ be the set of intervals for u. It is worth noting that by *Tree-Cover* the number of intervals for a vertex cannot be bounded, but in order to control the index size *Ferrari* only assigns up to k intervals for a vertex. Therefore, some intervals in $\mathsf{label}(u)$ are approximate intervals, because they cover certain interval that is not supposed to be covered to correctly answer $\mathsf{Reach}(u, v)$. *Ferrari* is to minimize the size of approximate intervals. This is done by computing up to k intervals of u with the intervals computed for all children of u using dynamic programming. The *Ferrari* constructed is not optimal. The time complexity for construction is $O(k^2 m + S)$, where S is the time complexity of finding the top-s largest degree vertex for seed based pruning and constructing the index. Its index size is $O((k + s)n)$, where s is the number of seeds added to every vertex label.

4. A NEW LINEAR LABELING

As discussed in Section 3, in the category of the Label-Only approaches, the approaches that aim at compressing TC, like *2-Hop* and *3-Hop*, incur high construction cost and large index size, which affects query time. *Chain-Cover* and *Tree-Cover* use a disjoint set of chains and a tree cover, respectively, but they cannot deal with large dense graphs. *Dual-Label* can reach $O(1)$ query time in theory, but it is for a considerably sparse graph. The up-to-date approaches in this category are *TF-Label*, *HL*, and *DL*. They aim at reducing the construction time, but the construction time is non-

linear, which will incur high construction time for large and dense graphs. In addition, both index size and query time cannot be tightly bounded.

In comparison with the Label-Only approaches, in the category of Label+G, all approaches are linear regarding construction time and index size. Here, linear is on the basis of the graph size (the number of vertices plus the number of edges). Given the linear construction time and index size, the query time for *Tree+SSPI* and *GRIPP* is $O(m-n)$ and the query time for *GRAIL* and *Ferrari* can be up to $O(n+m)$, which means it cannot deal with large dense graphs. In [10, 19], the authors indicate that *TF-Label*, *HL*, and *DL* outperform *GRAIL*, in terms of query time in practice.

The main idea: In this paper, we propose a novel labeling approach, called *IP* (Independent Permutation), and test set containment with probability guarantee, where the label of a vertex is a set. Let $\mathcal{A}$ and $\mathcal{B}$ be two subsets of a set $\mathcal{V}$. There are two ways to check whether $\mathcal{B}$ is contained in $\mathcal{A}$. One is to check $\mathcal{B} \subseteq \mathcal{A}$, and the other is to check $\mathcal{B} \not\subseteq \mathcal{A}$. Both are time consuming if an exact answer is needed for large sets. We take the latter approach, and answer a reachability query by finding whether there is at least one element in one set that is not contained in the other for $\mathcal{B} \not\subseteq \mathcal{A}$. Our approach is based on randomness with high probability guarantee where the guarantee is ensured by independent permutation. With the probability guarantee, we can answer a large number of $\mathcal{B} \not\subseteq \mathcal{A}$ tests with 100% probability guarantee by *IP*. For those we cannot answer by *IP* with 100% probability guarantee, we do *DFS* online. It is important to note that the randomness we use is to minimize the probability of doing *DFS*, and the randomness is based on min-wise independent permutations [4]. We further improve the probability guarantee by min-wise independent permutations with a new k-min-wise independent permutations proposed in this work.

Before introducing min-wise independent permutations followed by our k-min-wise independent permutations, we show how $u \rightsquigarrow v$ is connected to $\mathcal{A}$ and $\mathcal{B}$ in *IP*. This is based on our observation that $u \rightsquigarrow v$, if $Out(v) \subseteq Out(u)$ or $In(u) \subseteq In(v)$. In other words, if u can reach all the vertices that v can reach, or all vertices that can reach u can also reach v, then $u \rightsquigarrow v$. In a similar manner, $u \not\rightsquigarrow v$, if $Out(v) \not\subseteq Out(u)$ or $In(u) \not\subseteq In(v)$. Therefore, $\mathcal{A}$ and $\mathcal{B}$ denote $Out(u)$ and $Out(v)$ respectively, or $In(v)$ and $In(u)$ respectively. Below, we consider $\mathcal{A}$ and $\mathcal{B}$ as a subset of $\mathcal{V}$ which is a set of n numbers, $\mathcal{V} = \{0, 1, 2, \cdots, n-1\}$, representing all vertices in $V(G)$ for $n = |V(G)|$.

Min-wise Independent Permutations: We briefly review the min-wise independent permutations as given [4]. Given a set of numbers, Ω, a permutation of Ω is a bijection from Ω to itself, denoted as $\pi : \Omega \rightarrow \Omega$. It is important to note that $\pi(x)$, for $x \in \Omega$, is unique in $\pi(\Omega)$. Consider $\mathcal{V}$ as a set of numbers of size n. Let $\mathcal{V}_n$ be the set of all permutations of $\mathcal{V}$. $\mathcal{F} \subseteq \mathcal{V}_n$ is min-wise independent, if for any $X \subseteq \mathcal{V}$ and any $x \in X$, when π is given to choose uniformly and randomly from $\mathcal{F}$, then the following holds:

$$\mathsf{Pr}(\min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|} \tag{1}$$

This says that, for any $x \in X$, the probability that $\pi(x)$ is the smallest number in $\pi(X)$ is $1/|X|$. Based on the min-wise independent permutations, Min-Hash [3] is designed to compute Jaccard similarity of two sets, $\mathcal{A}$ and $\mathcal{B}$, $\mathsf{Jacard}(\mathcal{A}, \mathcal{B}) = |\mathcal{A} \cap \mathcal{B}|/|\mathcal{A} \cup \mathcal{B}|$ as $\mathsf{Pr}(\min\{\pi(\mathcal{A})\} = \min\{\pi(\mathcal{B})\})$

In this work, our focus is on set containment, $\mathcal{B} \subseteq \mathcal{A}$. We know that $\mathcal{B} \subseteq \mathcal{A}$ if $\mathcal{A} = \mathcal{A} \cup \mathcal{B}$. Let $\min\{X\}$ indicates the smallest number of a set of numbers, X, we have

$$\mathsf{Pr}(\min\{\pi(\mathcal{A} \cup \mathcal{B})\} = \min\{\pi(\mathcal{B} \setminus \mathcal{A})\}) = \frac{|\mathcal{B} \setminus \mathcal{A}|}{|\mathcal{A} \cup \mathcal{B}|} = 1 - \frac{|\mathcal{A}|}{|\mathcal{A} \cup \mathcal{B}|} \tag{2}$$

By min-wise independent permutations, any number in $\mathcal{A} \cup \mathcal{B}$ is equally likely to have the smallest number in $\pi(\mathcal{A} \cup \mathcal{B})$, e.g., $\min\{\pi(\mathcal{A} \cup \mathcal{B})\}$, and $\min\{\pi(\mathcal{A} \cup \mathcal{B}\}) = \min\{\pi(\mathcal{B} \setminus \mathcal{A})\}$ is true, if and only if $\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\}$. Hence, we have

$$\mathsf{Pr}(\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\}) = 1 - \frac{|\mathcal{A}|}{|\mathcal{A} \cup \mathcal{B}|} \tag{3}$$

Because $\mathcal{B} \subseteq \mathcal{A}$ if $|\mathcal{A}| = |\mathcal{A} \cup \mathcal{B}|$, there does not exist any possible permutation π by which $\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\}$, as given in Eq. (3). Therefore, the condition of $\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\}$ can be used to conclude $\mathcal{B} \not\subseteq \mathcal{A}$. However, there are cases that $\mathcal{B} \not\subseteq \mathcal{A}$ is true, when $\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\}$ is not true. As can be seen from Eq. (3), the probability for $\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\}$ to be false is non-zero, $\frac{|\mathcal{A}|}{|\mathcal{A} \cup \mathcal{B}|}$. In other words, when $\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\}$ is not true, it can be possible that either $\mathcal{B} \subseteq \mathcal{A}$ or $\mathcal{B} \not\subseteq \mathcal{A}$. Here, our problem to answer $u \not\rightsquigarrow v$ by $\mathcal{B} \not\subseteq \mathcal{A}$ becomes a problem on how to increase the probability for $\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\}$ to be true.

k-min-wise: In order to increase $\mathsf{Pr}(\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\})$, for testing $\mathcal{B} \subseteq \mathcal{A}$, we propose to use top-$k$ smallest numbers instead of the top-1 smallest number as used in min-wise independent permutation. By independent permutation π, we define $\min_k\{\pi(X)\}$ as a subset of $\pi(X)$ containing up to k smallest numbers, such as $\min_k\{\pi(X)\} = (\pi(x_1), \pi(x_2), \cdots)$, where $\pi(x_i) < \pi(x_j)$ if $i < j$, and $|\min_k\{\pi(X)\}| \leq k$, Here, by k-min-wise ($\min_k$), we mean $\pi(x_i) < \pi(x_j)$ for any $\pi(x_i) \in \min_k\{\pi(X)\}$ and any $\pi(x_j) \in \pi(X) \setminus \min_k\{\pi(X)\}$.

To deal with $\mathcal{B} \subseteq \mathcal{A}$, we define an order ($\preceq$) between $\min_k\{\pi(\mathcal{A})\}$ and $\min_k\{\pi(\mathcal{B})\}$, such that $\min_k\{\pi(\mathcal{A})\} \preceq \min_k\{\pi(\mathcal{B})\}$ if every $\pi(b_i) \in \min_k\{\pi(\mathcal{B})\} \setminus \min_k\{\pi(\mathcal{A})\}$ is greater than the largest number in $\min_k\{\pi(\mathcal{A})\}$. We use $\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\}$ otherwise. We prove $\mathcal{B} \not\subseteq \mathcal{A}$, if $\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\}$ in Theorem 4.1.

Theorem 4.1: *Let $\mathcal{A}$ and $\mathcal{B}$ be a subset of $\mathcal{V} = \{0, 1, 2, \cdots, n-1\}$, and π be randomly and uniformly chosen from $\mathcal{V}_n$ (the set of all permutations of $\mathcal{V}$). Then, $\mathcal{B} \not\subseteq \mathcal{A}$, if $\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\}$.*

Proof Sketch: Consider the case when $\mathcal{B} \subseteq \mathcal{A}$. We have $\pi(\mathcal{B}) \subseteq \pi(\mathcal{A})$. Because the k smallest numbers in $\min_k\{\pi(\mathcal{A})\}$ are taken from $\pi(\mathcal{A})$ or in other words taken from $\pi(\mathcal{A} \cup \mathcal{B})$, there cannot exist $\pi(b_i)$ in $\min_k\{\pi(\mathcal{B})\} \setminus \min_k\{\pi(\mathcal{A})\}$ that is smaller than the largest number in $\min_k\{\pi(\mathcal{A})\}$. Therefore, if $\mathcal{B} \subseteq \mathcal{A}$, we have $\min_k\{\pi(\mathcal{A})\} \preceq \min_k\{\pi(\mathcal{B})\}$. Hence, if $\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\}$, then $\mathcal{B} \not\subseteq \mathcal{A}$. □

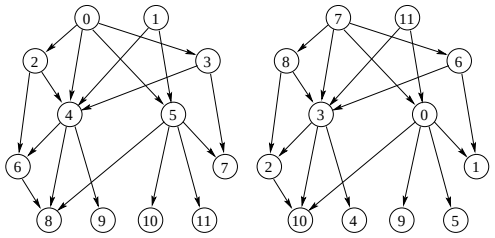
Next, like Eq. (3), we give the probability for $\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\}$ in Theorem 4.2.

Theorem 4.2: *Let $\mathcal{A}$ and $\mathcal{B}$ be a subset of $\mathcal{V} = \{0, 1, 2, \cdots, n-1\}$, and π be randomly and uniformly chosen from $\mathcal{V}_n$ (the set of all permutations of $\mathcal{V}$). Assume $|\mathcal{A}| = p$, $|\mathcal{A} \cup \mathcal{B}| = q$, and $|\min_k\{\pi(\mathcal{A})\}| = k_A$ for $k_A \leq k$.*

$$\mathsf{Pr}(\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\}) = 1 - \frac{p!(q-k_A)!}{q!(p-k_A)!} \tag{4}$$

Proof Sketch: For $\mathcal{B} \subseteq \mathcal{A}$, $\mathsf{Pr}(\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\}) = 0$ because $p = q$. We focus on the cases for $\mathcal{B} \not\subseteq \mathcal{A}$, provided $\mathcal{X} = \mathcal{A} \cup \mathcal{B}$.

Recall that for a given k, "up to k" is for the cases that the size of a set can be less than k. In general, $|\mathcal{A}|$, $|\mathcal{B}|$, and $|\mathcal{X}|$ can be possible $\leq k$. We use k_A, k_B, and k_X, which are a max possible number $\leq k$, for $|\min_k\{\pi(\mathcal{A})\}| = k_A$, $|\min_k\{\pi(\mathcal{B})\}| = k_B$, and $|\min_k\{\pi(\mathcal{X})\}| = k_X$, because $k_A \leq |\mathcal{A}|$, $k_B \leq |\mathcal{B}|$, and $k_X \leq |\mathcal{X}|$. We also know $k_A \leq k_X$ and $k_B \leq k_X$ because $\mathcal{X}$ is the union



(a) DAG G (b) A permutation π

Figure 1: A permutation π

Vertex	$\mathcal{L}_{out}$	$\mathcal{L}_{in}$	Vertex	$\mathcal{L}_{out}$	$\mathcal{L}_{in}$
v_0	{0,1,2,3,4}	{7}	v_6	{2,10}	{2,3,6,7,8}
v_1	{0,1,2,3,4}	{11}	v_7	{1}	{0,1,6,7,11}
v_2	{2,3,4,8,10}	{7,8}	v_8	{10}	{0,2,3,6,7}
v_3	{1,2,3,4,6}	{6,7}	v_9	{4}	{3,4,6,7,8}
v_4	{2,3,4,10}	{3,6,7,8,11}	v_{10}	{9}	{0,7,9,11}
v_5	{0,1,5,9,10}	{0,7,11}	v_{11}	{5}	{0,5,7,11}

Table 2: *IP* Label for DAG G given in Fig. 1

of $\mathcal{A}$ and $\mathcal{B}$. For simplicity, we use $\min_{k_A}\{\pi(\mathcal{A})\}$, $\min_{k_B}\{\pi(\mathcal{B})\}$, and $\min_{k_X}\{\pi(\mathcal{X})\}$, to indicates $\min_k\{\pi(\mathcal{A})\}$, $\min_k\{\pi(\mathcal{B})\}$, and $\min_k\{\pi(\mathcal{X})\}$, whose sizes are k_A, k_B, and k_X.

By Theorem 4.1, we know that if all the smallest numbers in $\min_{k_A}\{\pi(\mathcal{A})\}$ are the top-k_A smallest numbers in $\min_{k_X}\{\pi(\mathcal{X})\}$, such that $\min_{k_A}\{\pi(\mathcal{A})\} = \min_{k_A}\{\pi(\mathcal{X})\}$, then $\min_k\{\pi(\mathcal{A})\} \preceq \min_k\{\pi(\mathcal{B})\}$. In a similar way, it is also true $\min_k\{\pi(\mathcal{A})\} = \min_{k_A}\{\pi(\mathcal{X})\}$ if $\min_k\{\pi(\mathcal{A})\} \preceq \min_k\{\pi(\mathcal{B})\}$.

Recall that π is equally likely to be any permutation in the set of all permutations of $\mathcal{X}$, denoted $\mathcal{X}_n$. We consider the probability of $\min_{k_A}\{\pi(\mathcal{A})\} = \min_{k_A}\{\pi(\mathcal{X})\}$. There are $\frac{q!}{(q-k_A)!}$ permutations in total to have $\min_{k_A}\{\pi(\mathcal{X})\}$. Among such $\frac{q!}{(q-k_A)!}$ permutations, there are $\frac{p!}{(p-k_A)!}$ for $\min_{k_A}\{\pi(\mathcal{A})\} = \min_{k_A}\{\pi(\mathcal{X})\}$ in total. As a result, the probability that $\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\}$ is $1 - \frac{p!(q-k_A)!}{q!(p-k_A)!}$. □

In practice, k_A is up to k which is small and is user-controlled. p ($= |\mathcal{A}|$) and q ($= |\mathcal{A} \cup \mathcal{B}|$) can be large considering a large graph G. We have $k_A \ll p \leq q$. When it is the case, the probability that $\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\}$ can be represented as

$$1 - \frac{p(p-1)\cdots(p-k_A+1)}{q(q-1)\cdots(q-k_A+1)} \approx 1 - (\frac{p}{q})^{k_A} \quad (5)$$

Here, $(\frac{p}{q})^{k_A}$ can be very small, even for a relatively small k_A. Comparing Eq. (4) with Eq. (3), $\mathsf{Pr}(\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\})$ is much larger than $\mathsf{Pr}(\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\})$, and is equal to $\mathsf{Pr}(\min\{\pi(\mathcal{A})\} > \min\{\pi(\mathcal{B})\})$ when $k_A = 1$.

The *IP* Labeling: By k-min-wise independent permutations, for a vertex $u \in G$, for a given k, we define $\mathsf{label}(u)$ as a pair of sets, $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$. $\mathcal{L}_{out}(u)$ keeps up to k smallest numbers by the permutation π for the set of $Out(u)$, denoted as $\mathcal{L}_{out}(u) = \min_k\{\pi(Out(u))\}$. $\mathcal{L}_{in}(u)$ keeps up to k smallest number by the same permutation π for the set of $In(u)$, denoted as $\mathcal{L}_{in}(u) = \min_k\{\pi(In(u))\}$. By Theorem 4.1, to answer $\mathsf{Reach}(u, v)$, $u \not\rightsquigarrow v$ if $\mathcal{L}_{out}(u) \succ \mathcal{L}_{out}(v)$ or $\mathcal{L}_{in}(v) \succ \mathcal{L}_{in}(u)$ by the labels only. However, it needs to check by *DFS* online otherwise whether $u \rightsquigarrow v$ or $u \not\rightsquigarrow v$ since the probability given in Eq. (4) is non-zero.

Example 4.1: A DAG G is shown in Fig. 1(a). The numbers in Fig. 1(a) identify the 12 vertices, $V(G) = \{v_0, v_1, \cdots, v_{11}\}$. With a permutation π, which we will show how to compute, we assume the result is $\pi(0) = 7$, $\pi(1) = 11$, $\pi(2) = 8$, $\pi(3) = 6$, $\pi(4) = 3$, $\pi(5) = 0$, $\pi(6) = 2$, $\pi(7) = 1$, $\pi(8) = 10$, $\pi(9) = 4$, $\pi(10) = 9$, and $\pi(11) = 5$. Fig. 1(b) shows the $\pi(v_i)$ for every $v_i \in V(G)$. Let $k = 5$. Table 2 shows *IP* labels of G given in Fig. 1. We show four reachability queries. (Q_1) consider $\mathsf{Reach}(v_2, v_7)$. We have $\mathcal{L}_{out}(v_2) = \{2, 3, 4, 8, 10\}$ and $\mathcal{L}_{out}(v_7) = \{1\}$. Since $1 \in \mathcal{L}_{out}(v_7)$, $1 \notin \mathcal{L}_{out}(v_2)$, and $1 <$ the largest number 10 in $\mathcal{L}_{out}(v_2)$, we have $\mathcal{L}_{out}(v_2) \succ \mathcal{L}_{out}(v_7)$. By Theorem 4.1, we have $Out(v_7) \not\subseteq Out(v_2)$, and therefore $v_2 \not\rightsquigarrow v_7$. ($Q_2$) consider $\mathsf{Reach}(v_3, v_2)$. We have $\mathcal{L}_{out}(v_3) = \{1, 2, 3, 4, 6\}$, $\mathcal{L}_{out}(v_2) = \{2, 3, 4, 8, 10\}$. By definition, $\mathcal{L}_{out}(v_3) \preceq \mathcal{L}_{out}(v_2)$. However, since $\mathcal{L}_{in}(v_2) = \{7, 8\}$ and $\mathcal{L}_{in}(v_3) = \{6, 7\}$, we have $\mathcal{L}_{in}(v_2) \succ \mathcal{L}_{in}(v_3)$. Because $In(v_3) \not\subseteq In(v_2)$, the answer of $\mathsf{Reach}(v_3, v_2)$ is $v_3 \not\rightsquigarrow v_2$. ($Q_3$) consider $\mathsf{Reach}(v_1, v_8)$. We have $\mathcal{L}_{out}(v_1) \preceq \mathcal{L}_{out}(v_8)$ and $\mathcal{L}_{in}(v_8) \preceq \mathcal{L}_{in}(v_1)$. We cannot answer it by *IP* labels only, and *DFS* is needed over G (Fig. 1(a)). We start *DFS* from v_1. Supposed the next vertex to be visited is v_4. We have to further *DFS*, because $\mathcal{L}_{out}(v_4) \preceq \mathcal{L}_{out}(v_8)$ and $\mathcal{L}_{in}(v_8) \preceq \mathcal{L}_{in}(v_4)$. Suppose the next to be visited is v_8, we can conclude that $v_1 \rightsquigarrow v_8$. (Q_4) consider $\mathsf{Reach}(v_1, v_3)$. We have $\mathcal{L}_{out}(v_1) \preceq \mathcal{L}_{out}(v_3)$ and $\mathcal{L}_{in}(v_3) \preceq \mathcal{L}_{in}(v_1)$, we cannot answer it by *IP* labels only, and *DFS* is needed. We start *DFS* from v_1, which has the two children v_4 and v_5. For v_4, we cannot do *DFS* further, because $v_4 \not\rightsquigarrow v_3$ for $\mathcal{L}_{out}(v_4) \succ \mathcal{L}_{out}(v_3)$ and $\mathcal{L}_{in}(v_3) \succ \mathcal{L}_{in}(v_4)$. For v_5, we cannot do *DFS* further neither, because $v_5 \not\rightsquigarrow v_3$ for $\mathcal{L}_{out}(v_5) \succ \mathcal{L}_{out}(v_3)$ and $\mathcal{L}_{in}(v_3) \succ \mathcal{L}_{in}(v_5)$.

The value of p/q: Reconsider Eq. (4) (or Eq. (5)). Here, p and q imply either $|Out(u)|$ and $|Out(u) \cup Out(v)|$ respectively, or $|In(v)|$ and $|In(v) \cup In(u)|$ respectively. Since $p \leq q$, if $q - p$ is larger, p/q becomes smaller, and $\mathsf{Pr}(\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\})$ becomes larger. Below, we show that the difference between p and q is not small. We discuss the case for $p = |Out(u)|$, and $q = |Out(u) \cup Out(v)|$. The same can be applied to the $In(v)$ and $In(v) \cup In(u)$.

First, as observed in many works, given a DAG G, the percentage of reachability queries ($\mathsf{Reach}(u, v)$) that are answered negatively, $u \not\rightsquigarrow v$, over all possible reachability queries is over 90%. On the other hand, the reachability queries that are answered positively, $u \rightsquigarrow v$, over all possible reachability queries is very small, which we call reachability-ratio (or simply R-ratio), denoted as r. We conducted testing to confirm r is very small. We generate large DAGs by fixing the number of vertices to be 10 million vertices and increasing the average degree from 2 to 8. For each DAG G, we sample 100 million vertex pairs, u and v, to estimate the reachability ratio of r for G. The R-ratio for the average degree (2, 3, 4, 5, 6, 7, 8) is (9.0E-7, 6.2E-6, 3.8E-5, 2.17E-4, 1.24E-3, 5.68E-3, 1.68E-2). As a summary, for sparse DAGs, when the average degree ≤ 4, the R-ratio is smaller than 0.0001, and for dense DAGs, when the average degree is 8, the R-ratio is below 0.017 still.

Second, given the R-ratio (r) for a DAG G, for two vertices u and v, the expected value of $|Out(u)|$ and $|Out(v)|$ is nr, where n is the number of vertices in G. If u cannot reach v, the number of the common vertices between $Out(u)$ and $Out(v)$ is expected to be

$$|Out(u) \cap Out(v)| = |Out(u)| \times \frac{|Out(v)|}{|V(G)|} = nr \cdot \frac{nr}{n} = nr^2 \quad (6)$$

Then, the percentage of $|Out(u) \cap Out(v)|/|Out(u)|$ becomes $nr^2/nr = r$. This implies that q is much larger than p given $q = |Out(u) \cup Out(v)|$ and $p = |Out(u)|$. This fact shows $\mathsf{Pr}(\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\})$ is large.

The number of vertices to be visited by *DFS*: Given $\mathsf{Reach}(u, v)$ and assume that we cannot answer by *IP* labels, and *DFS* is needed even though $u \not\rightsquigarrow v$. Suppose we visit a vertex w by *DFS*. Because u can reach w by *DFS*, we have $|Out(w)| < |Out(u)|$ and $|In(u)| < |In(w)|$ due to $Out(w) \subseteq Out(u)$ and $In(u) \subseteq In(w)$. While *DFS* along some path toward v further, $|Out(w)|$ will become significantly smaller than $|Out(u)|$ and $|In(u)|$ will become significantly smaller than $|In(w)|$. Accordingly, the following is true.

$$\frac{|Out(w)|}{|Out(w) \cup Out(v)|} < \frac{|Out(u)|}{|Out(u) \cup Out(v)|} \quad (7)$$

$$\frac{In(v)|}{|In(v) \cup In(w)|} < \frac{|In(v)|}{|In(v) \cup In(u)|} \quad (8)$$

As a result, if u cannot reach v, during *DFS*, the vertex w we visit will become more unlikely to reach v by Eq. (4). Given this, the length of *DFS* is supposed not to large. We give a lemma below.

Lemma 4.1: *Given* $\mathsf{Reach}(u, v)$*, assume DFS is needed even though* $u \not\rightsquigarrow v$*. Consider when a vertex* w*, as a descendant of* u*, is visited by DFS towards* v*. The followings are true.*

$$\mathsf{Pr}(\mathcal{L}_{out}(u) \succ \mathcal{L}_{out}(v)) < \mathsf{Pr}(\mathcal{L}_{out}(w) \succ \mathcal{L}_{out}(v))$$
$$\mathsf{Pr}(\mathcal{L}_{in}(v) \succ \mathcal{L}_{in}(u)) < \mathsf{Pr}(\mathcal{L}_{in}(v) \succ \mathcal{L}_{in}(w)) \quad (9)$$

□

Proof Sketch: Since w is a descendant of u, we have $|Out(w)| < |Out(u)|$ and $|In(u)| < |In(w)|$ because $Out(w) \subseteq Out(u)$ and $In(u) \subseteq In(w)$. By Eq. (7) and Eq. (8), we have $\frac{|Out(w)|}{|Out(w) \cup Out(v)|} < \frac{|Out(u)|}{|Out(u) \cup Out(v)|}$ and $\frac{In(v)|}{|In(v) \cup In(w)|} < \frac{|In(v)|}{|In(v) \cup In(u)|}$. By Eq. (4) (or its simplified version Eq. (5)), it shows that if $\frac{p}{q}$ decreases, $\mathsf{Pr}(\min_k\{\pi(\mathcal{A})\} \succ \min_k\{\pi(\mathcal{B})\})$ will become larger accordingly. Therefore the probability of $\mathcal{L}_{out}(u) \succ \mathcal{L}_{out}(v)$ becomes smaller than the probability of $\mathcal{L}_{out}(w) \succ \mathcal{L}_{out}(v)$ and the probability of $\mathcal{L}_{in}(v) \succ \mathcal{L}_{in}(u)$ becomes smaller than the probability of $\mathcal{L}_{in}(v) \succ \mathcal{L}_{in}(w)$. □

Lemma 4.1 suggests that the probability becomes higher for a descendant of u, w, to answer the reachability queries negatively. It implies that it is more likely to answer the reachability query by *IP* labels in *DFS* search.

We analyze $\mathsf{Pr}(\mathcal{L}_{out}(u) \succ \mathcal{L}_{out}(v)) < \mathsf{Pr}(\mathcal{L}_{out}(w) \succ \mathcal{L}_{out}(v))$ in Eq. (9), where the probability $\mathsf{Pr}(\cdot)$ is given as Eq. (5) for simplicity. Here, w is a descendant of u. Let $p_u = |Out(u)|$ and $q_u = |Out(u) \cup Out(v)|$. By Eq. (5), $\mathsf{Pr}(\mathcal{L}_{out}(u) \succ \mathcal{L}_{out}(v)) \approx 1 - (p_u/q_u)^k$. Let $p_w = |Out(w)|$ and $q_w = |Out(w) \cup Out(v)|$. By Eq. (7), we know $p_w/q_w < p_u/q_u$, and we denote p_w/q_w as $\alpha \cdot p_u/q_u$ for $\alpha < 1$. Hence, $\mathsf{Pr}(\mathcal{L}_{out}(w) \succ \mathcal{L}_{out}(v)) \approx 1 - \alpha^k \cdot (p_u/q_u)^k$, which is significantly larger than $\mathsf{Pr}(\mathcal{L}_{out}(u) \succ \mathcal{L}_{out}(v))$. Furthermore, if we apply the same analysis to a vertex ω which is a child of w in a similar. That is, Let $p_\omega = |Out(\omega)|$ and $q_\omega = |Out(\omega) \cup Out(v)|$, and assume p_ω/q_ω is $\beta \cdot p_w/q_w$ for $\beta < 1$. Then, $\mathsf{Pr}(\mathcal{L}_{out}(\omega) \succ \mathcal{L}_{out}(v)) \approx 1 - (\alpha\beta)^k \cdot (p_u/q_u)^k$. The same analysis can be applied to $\mathcal{L}_{in}$. As discussed, while *DFS* search becomes deeper, it is much more likely to answer the reachability queries negatively, and therefore, it can stop in an early stage.

Reconsider (Q_4) in Example 4.1 over G as shown in Fig. 1(a), $Out(v_1) = \{v_1, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$, $In(v_1) = \{v_1\}$ and $Out(v_3) = \{v_3, v_4, v_6, v_7, v_8, v_9\}$, $In(v_3) = \{v_0, v_3\}$. Therefore, $\mathsf{Pr}(\mathcal{L}_{out}(v_1) \succ \mathcal{L}_{out}(v_3)) = \frac{1}{2}$, $\mathsf{Pr}(\mathcal{L}_{in}(v_3) \succ \mathcal{L}_{in}(v_1)) = \frac{2}{3}$. The vertex v_1 has two children, v_4 and v_5. Suppose it conducts *DFS* to either of them. $\mathsf{Pr}(\mathcal{L}_{out}(v_4) \succ \mathcal{L}_{out}(v_3)) = \frac{14}{15}$, $\mathsf{Pr}(\mathcal{L}_{in}(v_3) \succ \mathcal{L}_{in}(v_4)) = \frac{9}{10}$ and $\mathsf{Pr}(\mathcal{L}_{out}(v_5) \succ \mathcal{L}_{out}(v_3)) = \frac{125}{126}$, $\mathsf{Pr}(\mathcal{L}_{in}(v_3) \succ \mathcal{L}_{in}(v_5)) = \frac{5}{6}$. The probability becomes larger while *DFS* search.

Algorithm 1 VS (G)

1: **for each vertex** $v_i \in V(G)$ **do**
2: $\quad \pi(i) \leftarrow i$;
3: **for** i **from** $n-1$ **downto** 1 **do**
4: $\quad j \leftarrow \mathsf{Random}(i)$;
5: $\quad$ swap $\pi(i)$ and $\pi(j)$;

Algorithm 2 *IP-Construct* (G, k)

1: Compute the topological order for all vertices in G;
2: VS (G);
3: **for each vertex** $u \in V(G)$ **do**
4: $\quad \mathcal{L}_{out}(u) \leftarrow \{\pi(u)\}$;
5: $\quad \mathcal{L}_{in}(u) \leftarrow \{\pi(u)\}$;
6: **for each vertex** u in the topological order **do**
7: $\quad$ **for each vertex** $w \in N_I(u)$ **do**
8: $\quad\quad$ maintain the top-k smallest numbers in $\mathcal{L}_{in}(u)$ among $\mathcal{L}_{in}(u) \cup \mathcal{L}_{in}(w)$;
9: **for each vertex** u in the reverse topological order **do**
10: $\quad$ **for each vertex** $w \in N_O(u)$ **do**
11: $\quad\quad$ maintain the top-k smallest numbers in $\mathcal{L}_{out}(u)$ among $\mathcal{L}_{out}(u) \cup \mathcal{L}_{out}(w)$;

5. COMPUTING *IP* LABELS

We discuss computing *IP*. First, we discuss the algorithm to compute a permutation (π), which assigns every $u \in V(G)$ a unique permutation number $\pi(u)$ in $O(n)$. Then, we give an algorithm to compute *IP* based on the permutation in $O(k(m+n))$.

5.1 Computing Permutation

To compute a permutation, we adopt the Knuth shuffle algorithm [22], denoted as KS, which is also known as the Fisher-Yates shuffle algorithm [15]. The KS algorithm is unbiased, and has the equal chance to generate every possible permutation, which meets the requirement for min-wise independent permutation technique. It is important to us that the complexity of the algorithm is $O(n)$. We show our VS algorithm in Algorithm 1, which is based on the KS algorithm. The VS algorithm first assigns an initial permutation number in $[0, n-1]$ to every vertex v_i in G, where $n = |V(G)|$ (line 1-2). Then, in a for loop (line 3-5), from $i = n-1$ to 1, it generates a random number j uniformly in the range of $[0, i]$ (line 4), and swaps $\pi(i)$ and $\pi(j)$ between i and j (line 5). The time complexity of the VS algorithm is $O(n)$. We can see that the probability of $\pi(i) = x$, for any $i, x \in [0, n\text{-}1]$, is equal to the probability that x has not been randomly chosen for exchange in the first n-i-1 iterations multiplies the probability that x is chosen for exchange in the $(n\text{-}i)$-th iteration. That is,

$$\mathsf{Pr}(\pi(i) = x) = \frac{n-1}{n} \times \frac{n-2}{n-1} \times \cdots \frac{i+1}{i+2} \times \frac{1}{i+1} = \frac{1}{n}$$

Therefore, for any $i \in [0, n\text{-}1]$, $\pi(i)$ has the equal chance to be any value in $[0, n\text{-}1]$. The VS algorithm (Algorithm 1) is equally likely to generate any possible independent permutation.

5.2 *IP* Computing

We discuss how to compute *IP*. Consider a vertex u, its $\mathcal{L}_{out}(u)$ can be computed by all $\mathcal{L}_{out}(w)$ in its out-neighbors ($\in N_O(u)$). This is because $Out(u) = \bigcup_{w \in N_O(u)} Out(w)$. Therefore, $\mathcal{L}_{out}(u)$ must be the subset of $\bigcup_{w \in N_O(u)} \mathcal{L}_{out}(w)$. In other words, $\mathcal{L}_{out}(u)$ consists of the top-k smallest numbers of $\bigcup_{w \in N_O(u)} \mathcal{L}_{out}(w)$. In a similar way, its $\mathcal{L}_{in}(u)$ can be computed by all $\mathcal{L}_{in}(w)$ in its in-neighbors ($\in N_I(u)$). This is because $In(u) = \bigcup_{w \in N_I(u)} In(w)$. Therefore, $\mathcal{L}_{in}(u)$ must be the subset of $\bigcup_{w \in N_I(u)} \mathcal{L}_{in}(w)$. In

other words, $\mathcal{L}_{in}(u)$ consists of the top-k smallest numbers from $\bigcup_{w \in N_I(u)} \mathcal{L}_{in}(w)$. Based on this idea, we design an algorithm called *IP-Construct* to compute *IP* labels for vertices in G.

The *IP-Construct* is shown in Algorithm 2. First, we compute the topological order for all vertices in G (line 1), and compute the permutation (π) by calling the VS algorithm (line 2). First, for every vertex $u \in V(G)$, it assigns $\pi(u)$ in $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$ (line 3-5). Below, we maintain $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$ in ascending order. It is worth mentioning that the size of $\mathcal{L}_{out}(u)$ and the size of $\mathcal{L}_{in}(u)$ is up to k. Second, we visit vertices following the topological order (line 6-8), and compute $\mathcal{L}_{in}(u)$ from its in-neighbors (line 7-8). To update $\mathcal{L}_{in}(u)$ by one additional in-neighbor w, it selects the top-k smallest numbers from two lists, $\mathcal{L}_{in}(u)$ and $\mathcal{L}_{in}(w)$, which are both sorted in ascending order. Finally, we visit vertices following the reverse topological order (line 9-11), and compute $\mathcal{L}_{out}(u)$ from its out-neighbors (line 10-11) in a similar way.

Construction time and Index size: To compute *IP* by Algorithm 2, it takes $O(m+n)$ to compute the topological order (line 1), and $O(n)$ to compute the permutation (Algorithm 1) (line 2). The initialization of $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$ for every $u \in G$ (line 3-5) is $O(n)$. Then, it takes $O(k(n+m))$ time to compute $\mathcal{L}_{in}(u)$ for every $u \in G$ (line 6-8). This is because it takes $O(k)$ to update a vertex u's $\mathcal{L}_{in}(u)$ using $\mathcal{L}_{in}(w)$ for an in-neighbor w of u (line 8) by merging the two sorted lists of size up to k, and such update is to be done for every of m edges when accessing all n vertices in G. In a similar way, it takes $O(k(n+m))$ time to compute $\mathcal{L}_{out}(u)$ for every $u \in G$ in line 11. Overall, the time complexity is $O(k(n+m))$. Our algorithm is more efficient than *GRAIL* even though both are $O(k(n+m))$. In our algorithm, we only need to scan G twice. In *GRAIL*, it needs to scan G k times to generate k spanning trees. The *IP* index size is at most $2kn$, because $|\mathcal{L}_{out}(u)| \leq k$ and $|\mathcal{L}_{in}(u)| \leq k$ for every $u \in G$. Comparing with *GRAIL*, our index size is up to $2kn$, whereas the index size for *GRAIL* is $2kn$.

6. TWO ADDITIONAL LABELS

In this section, we discuss two additional labels together to be used with *IP* labels. The two additional labels are used to reduce the *DFS* search cost.

6.1 A Level Label

In order to early stop in *DFS* for answering $\mathsf{Reach}(u, v)$, we introduce a level label for every vertex u, denoted as $\mathcal{L}_{level}(u)$. The $\mathcal{L}_{level}(u)$ consists of two parts, $\mathcal{L}_{up}(u)$ and $\mathcal{L}_{down}(u)$, and are defined in Eq. (10) and Eq. (11).

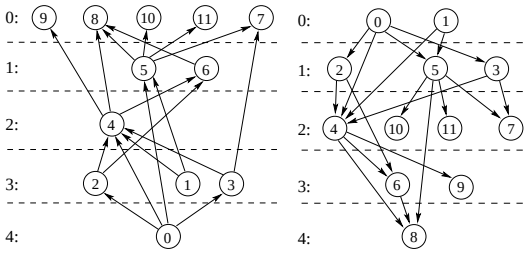
$$\mathcal{L}_{up}(u) = \begin{cases} 0 & \text{if } |N_O(u)| = 0 \\ 1 + \max\limits_{v \in N_O(u)} \{\mathcal{L}_{up}(v)\} & \text{otherwise} \end{cases} \quad (10)$$

$$\mathcal{L}_{down}(u) = \begin{cases} 0 & \text{if } |N_I(u)| = 0 \\ 1 + \max\limits_{v \in N_I(u)} \{\mathcal{L}_{down}(v)\} & \text{otherwise} \end{cases} \quad (11)$$

Since we have already computed the topological order, it takes $O(n+m)$ to compute the level labels for vertices in G. Based on Eq. (10) and Eq. (11), we have the following theorem about the level labels, which helps to prune unnecessary search paths in *DFS* effectively.

Theorem 6.3: *Given two vertices u and v in G. If u can reach v, for $u \neq v$, then both $\mathcal{L}_{up}(u) > \mathcal{L}_{up}(v)$ and $\mathcal{L}_{down}(u) < \mathcal{L}_{down}(v)$ must be true.*

Proof Sketch: Based on Eq. (10), $\mathcal{L}_{up}(u)$ must be larger than $\mathcal{L}_{up}(w)$ for $w \in N_O(u)$, and be larger than any descendant of u. If u can reach v, then $N_O(u) \neq \emptyset$ and v is a descendant of u. This leads to the conclusion that $\mathcal{L}_{up}(u) > \mathcal{L}_{up}(v)$. Also, based on Eq. (11), $\mathcal{L}_{down}(v)$ must be larger than $\mathcal{L}_{down}(\omega)$ for $\omega \in N_I(u)$, and be larger than any ascendant of v. If u can reach v, then $N_I(v) \neq \emptyset$ and u is an ascendant of v. This leads to the conclusion that $\mathcal{L}_{down}(u) < \mathcal{L}_{down}(v)$. □



(a) $\mathcal{L}_{up}(u)$ for $u \in G$ (b) $\mathcal{L}_{down}(u)$ for $u \in G$

Figure 2: Level Labels for G in Fig. 1(a)

Example 6.2: Consider graph G in Fig. 1(a), the level labels of G are shown in Fig. 2. Consider $\mathsf{Reach}(v_1, v_0)$ and $\mathsf{Reach}(v_1, v_2)$. As shown in Fig. 2(a), v_1 cannot reach v_0 nor v_2, because $\mathcal{L}_{up}(v_0) = 4$ and $\mathcal{L}_{up}(v_2) = 3$ are not smaller than $\mathcal{L}_{up}(v_1) = 3$. Consider $\mathsf{Reach}(v_3, v_5)$. From Fig. 2(a), it seems v_3 can reach v_5, because $\mathcal{L}_{up}(v_3) > \mathcal{L}_{up}(v_5)$. However, From Fig. 2(b), we have $\mathcal{L}_{down}(v_5) = \mathcal{L}_{down}(v_3)$. Therefore, v_3 cannot reach v_5.

The level labels are effective as we will show in our experimental study, but are not new. *GRAIL* [30] uses a similar technique like $\mathcal{L}_{up}(u)$, and *TF-Label* [10] uses $\mathcal{L}_{down}(u)$. The difference between the *TF-Label* and ours, in terms of $\mathcal{L}_{down}(u)$ is as follows. We use $\mathcal{L}_{down}(u)$ to early stop in *DFS* at run-time, whereas *TF-Label* uses to construct *2-Hop* labels.

6.2 A Huge-Vertex Label

A key factor that affects the performance of *DFS* at run-time is how to deal with those vertices that have large out-degree. We call such a vertex as a huge-vertex denoted as *HV*. With the existence of such huge-vertices, in particular, in power-law graphs [14], when it performs *DFS* starting from a huge-vertex at run-time, the time consumption will become large, because its out-degree is high, and the size of its out-neighbors is large. There are too many possible paths to search in *DFS*. Take $\mathsf{Reach}(v_0, v_{11})$ as an example, and consider *DFS* search of G in Fig. 1(a). v_0 itself is a relative huge vertex in comparison with other vertices in this small graph G, so as two of its children v_4 and v_5. If the *DFS* search order is (v_0, v_2, v_6, v_8, v_4, v_9, v_3, v_7, v_5, v_{10}, v_{11}), then it will search nearly the entire G before finally answering this reachability query. There are certain huge-vertices in many real graphs that we need to deal with to reduce the run-time cost.

We propose a new simple but effective label, called *HV-Label*, and denoted as $\mathcal{L}_{hv}(v)$ for every vertex $v \in G$. Here, $\mathcal{L}_{hv}(v)$ contains up to top-h largest *HV* vertices, $\{u\}$, if u can reach v and the out-degree of u is larger than μ where $n = |V(G)|$ and μ is a user-given threshold, such that $\mathcal{L}_{hv}(v) = \{u \mid u \rightsquigarrow v \wedge d_O(u) > \mu\}$ for $|\mathcal{L}_{hv}(u)| \leq h$.

Answering with *HV-Label*: With *HV-Label*, we can decide $w \rightsquigarrow v$, if $w \in \mathcal{L}_{hv}(v)$. In other words, suppose that we *DFS* search to a huge-vertex w for answering $\mathsf{Reach}(u, v)$. If $w \in \mathcal{L}_{hv}(v)$, then we can answer $u \rightsquigarrow v$ immediately without any needs of *DFS*. If w is not a huge-vertex of v ($w \notin \mathcal{L}_{hv}(v)$) and $|\mathcal{L}_{hv}(v)| < h$, $w \not\rightsquigarrow v$. In addition, if w is a huge-vertex but $w \notin \mathcal{L}_{hv}(v)$ and $d_O(w)$ is larger than some out-degree of the vertices in $\mathcal{L}_{hv}(v)$, we

Vertex	$\mathcal{L}_{hv}(v_i)$	Vertex	$\mathcal{L}_{hv}(v_i)$
v_0	{0}	v_6	{0,4}
v_1	$\emptyset$	v_7	{0,5}
v_2	{0}	v_8	{0,5}
v_3	{0}	v_9	{0,4}
v_4	{0,4}	v_{10}	{0,5}
v_5	{0,5}	v_{11}	{0,5}

Table 3: Huge-Vertices for G shown in Fig. 1(a)

Algorithm 3 *HV-Construct* (G, h, μ)

```
1: Compute topological order for all vertices in G;
2: for each vertex u ∈ V(G) do
3:     if d_o(u) > μ then
4:         L_hv(u) ← {u};
5:     else
6:         L_hv(u) ← ∅;
7: for each vertex u in the topological order do
8:     for each vertex w in N_I(u) do
9:         maintain L_hv(u) by taking up to h vertices that have the largest
           out-degree from L_hv(u) ∪ L_hv(w);
```

can answer $w \not\rightsquigarrow v$ by definition.

Example 6.3: Consider G shown in Fig. 1(a), where $n = 12$. Let $\mu = 2$ and $h = 2$. A vertex is a huge-vertex if its out-degree is larger than 2. For G shown in Fig. 1(a), the huge-vertices are v_0, v_4 and v_5. The *HV-Label*s computed for G are shown in Table 3. Consider a reachability query $\mathsf{Reach}(v_0, v_{11})$. Here, v_0 is a huge-vertex and $0 \in \mathcal{L}_{hv}(v_{11})$, we know $v_0 \rightsquigarrow v_{11}$. Consider another reachability query $\mathsf{Reach}(v_0, v_1)$. We can answer $v_0 \not\rightsquigarrow v_1$, because v_0 is a huge-vertex and $\mathcal{L}_{hv}(v_1)$ is empty. Also, we can answer $\mathsf{Reach}(v_5, v_6)$ as $v_5 \not\rightsquigarrow v_6$ using *HV-Label*s. Here, v_5 is a huge-vertex and $\mathcal{L}_{hv}(v_6) = \{0, 4\}$. By *HV-Label*, we know that $5 \notin \mathcal{L}_{hv}(v_6)$ but the out-degree of v_5 is larger than the out-degree of v_4, and therefore, the conclusion of $v_5 \not\rightsquigarrow v_6$ can be made.

The algorithm to compute *HV-Label*s for a graph G is given Algorithm 3, called *HV-Construct*. The idea to construct *HV-Label*s is very similar to the *IP-Construct* algorithm. The time complexity for Algorithm 3 is $O(h(n + m))$ and the space is $O(hn)$.

We discuss the two user-given thresholds h and μ below. Here, h determines the size of *HV-Label*, $\mathcal{L}_{hv}(u)$, for every vertex in G, and μ determines what vertices are huge-vertices to be selected. Since h is for a few of huge-vertices, it does not need to be large. Decreasing μ will make more vertices to be huge-vertices. Doing so will include really huge-vertices in *HV-Label*. However, it is not necessary to use a small μ. We suggest to use $\mu = 100$ after many testings. In other words, a vertex is a huge-vertices if its out-degree is > 100.

7. ANSWER $\mathsf{Reach}(u, v)$

We have discuss three labels, *IP* labels, *Level* labels, and *HV-Label*. For every vertex $u \in G$, the labels are $\mathcal{L}_{out}(u)$, $\mathcal{L}_{in}(u)$, $\mathcal{L}_{up}(u)$, $\mathcal{L}_{down}(u)$, and $\mathcal{L}_{hv}(u)$. Here, both $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$ are up to k numbers, $\mathcal{L}_{up}(u)$ and $\mathcal{L}_{down}(u)$ are fixed to 2 numbers, and $\mathcal{L}_{hv}(u)$ is up to h numbers. In total, the max size of labels are $2k + h + 2$ for a vertex.

We process a reachability query $\mathsf{Reach}(u, v)$ in a way as given in the *IP+* algorithm (Algorithm 4). First, it checks the trivial case whether $u = v$, and answers $u \rightsquigarrow v$ if $u = v$ (line 1). Second, it checks *IP* labels, and answers $u \not\rightsquigarrow v$ if either $\mathcal{L}_{out}(u) \succ \mathcal{L}_{out}(v)$ is true or $\mathcal{L}_{in}(v) \succ \mathcal{L}_{in}(u)$ is true (line 2). Third, it checks *Level* labels, and answers $u \not\rightsquigarrow v$ if either $\mathcal{L}_{up}(u) \leq \mathcal{L}_{up}(v)$ is true or $\mathcal{L}_{down}(u) \geq \mathcal{L}_{down}(v)$ is true (line 3). Forth, it uses *HV-Label* to answer in line (4-7) when u is a huge-vertex ($d_O(u) > \mu$). There are three cases. (a) It answers $u \rightsquigarrow v$ if the huge-vertex u is maintained in $\mathcal{L}_{hv}(v)$ (line 5). (b) It answers $u \not\rightsquigarrow v$ if $|\mathcal{L}_{hv}(v)| < h$ (line 6), where the condition $u \notin \mathcal{L}_{hv}(v)$ is true due to line 5. (c) It answers $u \not\rightsquigarrow v$ if the out-degree of u is larger than the out-degree of some vertex in $\mathcal{L}_{hv}(v)$ (line 7). Fifth, if none of the above is true, it will conducts *DFS* to search the vertices that have not been visited before (line 8-10). Finally, it will answer $u \not\rightsquigarrow v$ if it is impossible to answer positively.

Algorithm 4 *IP+* (G, (u, v))

```
1: if u = v then return u ⇝ v;
2: if L_out(u) ≻ L_out(v) ∨ L_in(v) ≻ L_in(u) then return u ↛ v;
3: if L_up(u) ≤ L_up(v) ∨ L_down(u) ≥ L_down(v) then return u ↛ v;
4: if d_O(u) > μ then
5:     if u ∈ L_hv(v) then return u ⇝ v;
6:     if |L_hv(v)| < h then return u ↛ v;
7:     if d_O(u) > d_o(w) for some w ∈ L_hv(v) then return u ↛ v;
8: for each w ∈ N_O(u) do
9:     if w has not been visited then
10:        if IP+ (G, (w, v)) is w ⇝ v then return u ⇝ v;
11: return u ↛ v;
```

Dataset	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	d_{avg}	R-ratio (r)
citeseer	693,947	312,282	0.450	3.20E-6
email	231,000	223,004	0.965	5.06E-2
LJ	971,232	1,024,140	1.054	2.13E-1
mapped-100K	2,658,702	2,660,628	1.000	1.56E-6
mapped-1M	9,387,448	9,440,404	1.005	7.00E-7
uniprotenc22m	1,595,444	1,595,442	0.999	1.45E-6
uniprotenc100m	16,087,295	16,087,293	0.999	1.60E-7
uniprotenc150m	25,037,600	25,037,598	0.999	1.30E-7
web	371,764	517,805	1.392	1.48E-1
wiki	2,281,879	2,311,570	1.013	8.14E-3
yago	16,375,503	25,908,132	1.582	1.00E-6
twitter	18,121,168	18,359,487	1.013	7.39E-2
web-uk	22,753,644	38,184,039	1.678	1.50E-1
citeseerx	6,540,399	15,011,259	2.295	4.07E-4
patent	3,774,768	16,518,947	4.376	2.36E-3
go-uniprot	6,967,956	34,770,235	4.990	3.64E-6
govwild	8,022,880	23,652,610	2.948	7.20E-5
dbpedia	3,365,623	7,989,191	2.374	2.47E-2
HostLink	12,754,590	26,669,293	2.091	4.48E-2

Table 4: Large Real Graphs

Query Time: We discuss the time complexity of Algorithm 4 for answering $\mathsf{Reach}(u, v)$. The time complexity is $O(k)$, because both labels used are sorted, when *IP* labels over u and v can be directly used to answer the reachability query without the needs of *DFS*. Next, we consider the cases when *DFS* is needed. First, suppose the answer of $\mathsf{Reach}(u, v)$ is $u \rightsquigarrow v$. Then, the time complexity is related to the vertices that can be possibly on some paths from u to v. The number of such vertices is $|Out(u) \cap In(v)|$, which is nr^2, in a similar way as obtained in Eq. (6), where n is the number of vertices and r is the R-ratio of the graph G. Since it consumes $O(k)$ in every vertex visited, the time complexity is $O(knr^2)$. It does not take the *HV-Label* into consideration, which can be effectively used to reduce the query time. Second, suppose the answer of $\mathsf{Reach}(u, v)$ is $u \not\rightsquigarrow v$. As discussed above, the *IP* labels can be effectively used to terminate *DFS* with the assistance of *Level* labels and *HV-Label* during *DFS*. The *DFS* search will be terminated after a small number of vertices to be visited, which we consider as a constant (refer to Eq. (9) in Lemma 4.1). Such a number is much smaller than n and it is not related to m. It is much less than but can be bounded by $O(knr^2)$.

Dataset	*GRAIL*	*GRAIL**	*ScaGRAIL*	*PWAH8*	*TF-Label*	*HL*	*DL*	*Ferrari-G*	*IP+*
citeseer	0.959	0.635	2.611	0.530	0.780	1.108	0.589	0.537	**0.37**
email	0.303	0.166	1.067	0.225	0.113	0.289	0.147	0.194	**0.09**
LJ	1.299	0.695	5.367	1.052	0.628	1.319	0.667	0.887	**0.54**
mapped-100K	3.106	2.032	9.231	**0.527**	2.307	4.422	2.272	2.748	1.64
mapped-1M	12.334	7.299	35.229	**2.830**	13.049	17.347	7.736	9.986	6.12
uniprotenc22m	2.451	1.206	6.790	1.485	2.223	2.220	1.038	0.919	**0.84**
uniprotenc100m	33.214	13.658	90.518	17.543	39.848	26.297	12.889	14.022	**12.46**
uniprotenc150m	58.242	23.820	145.576	29.169	58.529	44.712	22.280	24.292	**18.96**
web	0.495	0.340	1.900	0.740	0.454	0.717	0.370	0.449	**0.25**
wiki	2.886	1.329	7.335	**0.459**	1.011	2.878	1.375	1.927	1.22
yago	27.489	21.353	85.837	**12.723**	16.905	40.886	21.121	35.977	13.35
twitter	32.323	13.162	—	**11.864**	15.291	27.645	13.719	19.972	12.44
web-uk	44.031	17.854	—	248.814	—	1081.390	24.240	26.927	**17.46**
citeseerx	23.170	23.730	61.188	19.976	91.877	131.962	12.045	19.792	**7.54**
patent	21.404	23.498	82.217	1098.786	238.422	—	135.628	41.729	**9.00**
go-uniprot	44.557	39.342	70.277	38.668	64.501	27.591	18.277	40.365	**9.68**
govwild	29.237	28.300	62.963	30.520	139.247	32.230	18.584	19.924	**8.45**
dbpedia	17.503	16.099	45.570	7.771	12.152	10.926	4.741	6.236	**3.75**
HostLink	58.279	63.388	158.537	42.759	303.723	33.533	16.815	24.760	**11.89**

Table 5: Index Construction Time on Large Real Graphs (in second)

Dataset	*GRAIL*	*GRAIL**	*ScaGRAIL*	*PWAH8*	*TF-Label*	*HL*	*DL*	*Ferrari-G*	*IP+*
citeseer	10.588	18.530	11.941	6.565	**1.789**	7.713	7.145	13.607	9.440
email	3.525	6.168	4.365	2.342	**0.851**	2.693	2.586	5.792	3.664
LJ	14.820	25.935	18.785	9.644	**3.956**	11.677	11.227	18.096	19.112
mapped-100K	40.569	70.995	51.112	**10.715**	17.185	39.276	34.424	30.766	53.899
mapped-1M	143.241	250.672	180.650	**38.078**	73.933	137.352	116.089	109.546	171.253
uniprotenc22m	24.345	42.603	30.675	18.635	**6.294**	18.620	18.467	45.646	24.494
uniprotenc100m	245.473	429.577	310.668	208.622	**76.721**	204.739	197.182	460.261	251.150
uniprotenc150m	382.043	668.576	486.188	349.251	**131.920**	336.998	318.504	716.331	394.966
web	5.672	9.927	7.615	4.330	**2.560**	5.894	4.848	8.269	6.952
wiki	34.818	60.933	43.940	8.987	**8.858**	26.325	26.203	26.622	51.992
yago	249.870	437.273	315.017	134.646	**98.831**	290.393	223.766	312.701	250.567
twitter	276.507	483.887	—	95.566	**70.111**	211.143	202.480	237.106	384.894
web-uk	347.193	607.587	—	**260.100**	—	4520.436	697.646	458.302	442.890
citeseerx	249.497	399.194	303.017	148.781	1523.491	1531.189	**117.542**	186.650	151.018
patent	143.996	230.393	250.212	5334.114	4731.991	—	625.282	205.195	**137.865**
go-uniprot	265.806	425.290	399.486	244.026	430.527	291.748	247.852	378.774	**184.683**
govwild	306.049	489.678	409.899	304.061	3122.743	319.489	**191.040**	402.808	193.963
dbpedia	128.388	205.421	159.639	59.897	**52.189**	85.040	53.299	106.202	94.275
HostLink	486.549	778.478	580.031	**115.435**	5670.271	269.332	201.445	229.391	369.603

Table 6: Index Size of Large Real Graphs (in MB)

8. EXPERIMENTAL STUDIES

We conduct extensive experimental studies, and report our findings in this section. We denote our approach as *IP+* (Algorithm 2), which uses *IP* labels with two additional *Level* labels and *HV-Label*. And we compare *IP+* with the state-of-the-art reachability approaches including *GRAIL* [29], *GRAIL** [30], *ScaGRAIL* [17, 29], *PWAH8* [27], *TF-Label* [10], *HL* [19], *DL* [19], and *Ferrari* [24]. Here, *GRAIL* and *Ferrari* are two state-of-art Label+G approaches, and their index size and construction time are in linear. We test *Ferrari* using the *Ferrari-G* index given in [24], because *Ferrari-G* is scalable to handle massive-scale graphs. *GRAIL** is the improved version of *GRAIL* [30] and *ScaGRAIL* is the implementation of *GRAIL* in the *SCARAB* framework [17].[1] We also test the Label-Only approaches. *PWAH8* is the state-of-the-art transitive closure compression approach. And *TF-Label* [10], *HL* [19], and *DL* [19] are the three state-of-the-art Label-Only approaches based on *2-Hop* labels. We use the source codes provided by the authors to test the existing approaches. The source code of all approaches is implemented in C++ and compiled by G++ 4.8.1. All experiments are performed on machine with 2.67GHz Intel Xeon X5550 CPU, 24GB RAM and running Linux.

The three measures of the testing are: index construction time, index size, and query time. Programs that run $\geq$ 24 hours or exceed the memory limit (24GB) will be terminated, and the results will be shown "—" in the tables or "INF" in the figures.

Real Datasets: We use all large datasets used in the recent works [29, 30, 17, 27, 10, 19, 24]. Here, citeseer, citeseerx (`citeseerx.ist.psu.edu`), and patent (`snap.stanford.edu`) are 3 citation networks in which the out-degree of the non-leaf vertices is about 10 to 30. go-uniprot is the joint graph of Gene Ontology terms with the annotations file from the the universal protein resource database UniProt (`www.uniprot.org`). uniprotenc22m, uniprotenc100m, and uniprotenc150m are subgraphs of the complete RDF graph of UniProt. mapped-100K and mapped-1M are two datasets used in [17, 19]. email is a DAG of the communication network email-EuAll. LJ is a DAG of social network soc-LiveJournal1. web is the DAG of web graph web-Google. wiki is a DAG of wiki-talk, got from Wikipedia. These 4 datasets are from `snap.stanford.edu`. govwild is a large RDF graph from `govwild.hpi-web.de` and is transformed into the corresponding DAG. yago is a DAG of a large RDF representing a knowledge

[1] We can also implement our *IP+* in the *SCARAB* framework. We will leave it as our future work.

Dataset	*GRAIL*	*GRAIL**	*ScaGRAIL*	*PWAH8*	*TF-Label*	*HL*	*DL*	*Ferrari-G*	*IP+*
citeseer	363.738	96.722	108.564	138.810	**7.869**	60.558	59.351	67.833	24.503
email	12495.148	111.984	150.430	165.464	**19.341**	38.145	36.005	101.753	38.135
LJ	4267759	216.862	1141.400	171.929	66.592	65.465	**65.049**	94.538	86.988
mapped-100K	361.348	180.750	62.324	55.395	78.988	91.895	87.881	**23.931**	32.030
mapped-1M	413.987	225.407	79.515	65.853	99.754	106.026	103.001	**27.083**	44.186
uniprotenc22m	416.848	69.703	225.292	250.250	41.326	72.286	71.939	51.734	**21.791**
uniprotenc100m	720.053	139.863	411.032	327.161	99.315	110.354	109.584	112.616	**48.091**
uniprotenc150m	820.249	186.325	448.566	344.848	119.164	123.067	119.618	116.351	**54.205**
web	573719	254.526	1766.090	157.571	62.971	57.815	**57.206**	147.893	86.664
wiki	682422	76.472	161.378	55.983	45.719	77.611	78.795	**26.817**	27.033
yago	609.651	191.759	167.131	134.714	119.745	119.136	135.862	137.366	**88.854**
twitter	—	239.108	—	100.933	102.923	108.730	104.698	82.212	**79.285**
web-uk	—	492.230	—	216.287	—	227.611	**146.429**	214.857	253.082
citeseerx	28774.346	410.155	723.276	274.382	230.318	187.795	111.329	131.534	**101.444**
patent	5912.227	6005.880	1371.050	24252.523	579.454	—	**310.007**	7038.440	2278.440
go-uniprot	499.505	69.349	132.072	596.555	55.279	167.387	153.214	313.300	**34.577**
govwild	719.494	360.247	179.312	506.232	254.785	138.939	128.199	295.432	**112.990**
dbpedia	119909	383.253	416.931	291.059	113.774	101.615	108.982	200.239	**92.571**
HostLink	8693750	464.266	5433.720	185.109	264.938	119.398	138.803	184.410	**117.339**

Table 7: Query Time on Large Real Graphs (in millisecond)

graph. twitter is a DAG of the social network graph crawled from `twitter.com` collected by [6]. web-uk is a DAG of a web graph dataset collected by [2]. We also add two new datasets. dbpedia is the DAG of the knowledge graph DBpedia (`dbpedia.org`). HostLink is the DAG of the latest 100 million host links extracted from the host link graph `data.webarchive.org.uk`. All the real graphs are DAGs, and the basic information is given in Table 4. In Table 4, $|V(G)|$ and $|E(G)|$ are the number of vertices and the number of edges, d_{avg} is the average degree of a graph, and R-ratio (r) is the reachability-ratio discussed in Section 4. As shown in Table 4, most of these graphs are tree-like graphs whose d_{avg} is very closed to 1. We classify the real graphs into two classes: sparse graphs ($d_{avg} < 2$) and dense graphs ($d_{avg} \geq 2$). The 6 dense graphs are citeseerx, patent, go-uniprot, govwild, dbpedia, and HostLink. The R-ratio of reachable pairs is very small, far below 1% in most cases, which is consistent with our discussion in Section 4. The following 5 datasets, email, LJ, web, twitter, and web-uk, have relatively high R-ratio, because they have some huge-vertices. The neighbors of the huge-vertices nearly cover the entire vertex set of the graphs.

Synthetic datasets: We generate large DAGs with 10 million vertices and an average degree from 2 to 8 using the same graph generation algorithm used in *GRAIL* [29]. We first randomly create 10 million vertices and do a random ordering for these vertices. Then we randomly pick up two distinct vertices from the whole vertex set and create an edge which points from a lower order vertex to a higher order vertex. By repeating the process, we create the predefined number of edges.

Parameters: The 3 Label+G approaches have parameters to control their index size. The Label+G approaches get better performance when setting the parameters small for sparse graphs and larger for dense graphs. For *IP+*, we set $k = 2$ and $h = 2$ for sparse graphs and $k = 5$ and $h = 5$ for dense graphs, and $\mu = 100$ by default. By $\mu = 100$, a vertex u with $d_O(u)$ greater than μ is considered as a huge-vertex. For *Ferrari-G*, it is set as $k = 2$ for sparse graphs with $s = 2$ seeds as an additional index for seed based pruning, and $k = 5, s = 5$ for dense graphs. Similarly, we set $k = 2$ for *GRAIL* for sparse graphs and $k = 5$ for dense graphs. It is worth noting that the exact index sizes consumed by *IP+* and *Ferrari-G* can be bounded but cannot be fully controlled. By setting the similar parameters, the three *IP+*, *Ferrari-G*, and *GRAIL* will consume the similar index space.

8.1 Performance on Large Real Graphs

We report the index construction time, index size, and query time for large real graphs in Table 5, Table 6 and Table 7. The best results among are highlighted in **bold** font.

Index construction: Table 5 reports the index construction time. *IP+* is the fastest in almost all datasets. *IP+* cannot reach the best results in some other sparse graphs whose d_{avg} is close to 1, whereas *PWAH8* shows good performance in these sparse graphs. For other sparse graphs and dense graphs, *IP+* performs the best compared with others. The index construction time of *2-Hop* approaches is about 2 to 25 times of that of our approach over the dense graphs, and the index construction of *PWAH8* is on average 3 times slower than *IP+* over the dense graphs. *HL* fails its index construction in the dense graph patent, whereas *TF-Label* fails its index construction in the largest graph web-uk. The three versions of *GRAIL* approaches and *Ferrari-G* perform significantly slower than *IP+* in the index construction. For the dense graphs, *IP+* is twice faster than *GRAIL* approaches and *Ferrari-G*. In the large graphs twitter and web-uk, *ScaGRAIL* fails to compute their reachability backbone.

Index size: Table 6 shows the index size. For the dense graphs patent and go-uniprot, *IP+* constructs the smallest index, and the index size by *IP+* is closed to the best result in two other dense graphs citeseerx and govwild. But for other graphs, the Label+G approaches (*GRAIL*, *Ferrari-G*, and *IP+*) result in an index whose size is larger than those by the three *2-Hop* approaches (*TF-Label*, *HL*, *DL*) and *PWAH8*. This is because *GRAIL*, *Ferrari-G*, and *IP+* use $O(k)$ labels and the additional index for every vertex no matter how sparse the graph is in practice. In theory, the index size by either *GRAIL*, *Ferrari-G*, or *IP+* is in linear, whereas the *2-Hop* approaches (*TF-Label*, *HL*, and *DL*) and *PWAH8* may construct an unacceptably large index in dense graphs as showed in the patent dataset, where the index by *2-Hop* approaches is at least 3.5 times larger than the index *IP+* constructs.

Query time: We randomly generate queries such that every vertex pair will be selected with the same probability. For each dataset, we generate 1 million reachability queries. Table 7 shows the total query time taken to answer all the reachability queries generated. *IP+* performs the best in 5 sparse graphs among the 13 sparse graphs, and is comparable to the best results in other sparse graphs. *IP+* is at least one order of magnitude faster than *GRAIL* and also much faster than its two improved versions *GRAIL** and *Sca-*

GRAIL. *Ferrari-G* shares the similar query performance with *IP+* in sparse datasets, but significantly slower than *IP+* for dense graphs. For dense graphs, *IP+* wins the best in all dense graphs except patent. This is because that nearly a half of vertices in patent have at least 4 out-degree. When it needs to do *DFS*, it may need to visit more vertices until all branches are pruned or encountering the destination vertex. We believe that *IP+* in *SCARAB* can perform well in the patent dataset, because *ScaGRAIL* greatly improves the query time in patent compared with *GRAIL*.

8.2 Scalability Study on Synthetic Graphs

We conduct experimental studies on synthetic graphs with different density. Since *GRAIL** has better query and index construction performance than original version of *GRAIL*, we use *GRAIL** in the experiments. Fig. 3 shows the experimental results of different approaches on the synthetic graphs generated.

Fig. 3(a) and Fig. 3(b) show the index construction time and index size on graphs with different density. First, except for sparse graphs with the average degree 2, *IP+* always performs the best both on the index construction time and the index size. The index construction time and the index size of *IP+* only increases marginally while the graph density increases. Second, *PWAH8* and *2-Hop* approaches (*TF-Label*, *HL*, *DL*) do not show good scalability in large dense graphs. Their index construction time and index size increases exponentially when the average degree of the graph increases linearly. For the dense graphs, their memory usage exceeds the memory size of the system and thus fails to complete the index construction. *PWAH8* and *HL* cannot compute its labels successfully when the average degree ≥ 5, whereas *TF-Label* and *DL* fail their index construction for the graphs whose average degree is ≥ 6. Third, the index construction time of *GRAIL** is on average 2 times larger than that of *IP+*, and the index construction time of *Ferrari-G* is about 3 times larger than that of *IP+*. The index size of *GRAIL** and *Ferrari-G* is about two times of the index size of *IP+*. In summary, *IP+* has the best scalability.

We also randomly generate 1 million reachability queries over the synthetic graphs. The total query time to answer all reachability queries generated is shown in Fig. 3(c). The *2-Hop* approaches have the better performance in the datasets if they can construct the index given the time and space limited. The query time of *IP+* is comparable to the *2-Hop* approaches (*TF-Label*, *HL*, *DL*) in graphs with average degree ≤ 3. *IP+* is on average twice faster than *PWAH8*. Only *GRAIL**, *Ferrari-G*, and *IP+* can answer the query in dense graphs with average degree ≥ 6. And the query time of *IP+* is about a half of the query time of *GRAIL** and *Ferrari-G* in all graphs. For answering a single reachability query in the densest graph with an average degree 8, the query time of *IP+* is on average 0.75 millisecond, whereas *GRAIL** and *Ferrari-G* need more than 1 millisecond. *IP+* is the best approach among all approaches that can scale to large dense graphs.

8.3 *IP* label *vs* interval label

We show that only *IP+*, *GRAIL**, and *Ferrari-G* are scalable to handle large dense graphs. Unlike *GRAIL* and *Ferrari-G*, *IP* uses a *2-Hop*-like label, instead of interval labeling. To further study the query time of *IP* label vs interval label, we compare *IP* label, *GRAIL* label, and *Ferrari-G* index by excluding any additional index and any query optimization techniques. For fairness, we use similar index sizes for the three approaches. We conduct two testings.

First, we test the same 1 million random queries generated over the synthetic graph with an average degree 3 we used for the scalability testing. Index sizes for the three approaches are: 776MB for *IP* labels, 800MB for *GRAIL* labels, and 809MB for *Ferrari-G* index. We measure how many vertices they visit in answering the reachability queries generated. Fig. 4 shows the distribution of the number of visited vertices. 94.5% of the queries can be answered directly using *IP* labels while 92.3% of the queries can be answered directly using *GRAIL* labels and 87.3% of the queries can be answered directly using *Ferrari-G* index. It shows that *IP* label has high potential to answer without the needs of *DFS*. For the reachability queries that cannot be answered directly using labels, we can see that *IP* label also performs significantly better than *GRAIL* and *Ferrari-G*. Compared with the two interval labeling approaches, *IP* label is less likely to visit many vertices before finally answering a reachability query. The percentages of visiting more than 30 vertices using *IP*, *GRAIL*, and *Ferrari-G*, are 0.57%, 1.10%, 0.99%. The percentage of *IP* is about a half of either *GRAIL* or *Ferrari-G*. For reachability queries that need to visit more than 70 vertices, the ratio of *IP* label is larger than the ratio of *Ferrari-G* index but the ratio of *IP* label is still below 0.1%. The results shown are consistent with the experiments in which *IP* label usually has better query performance than *GRAIL* and *Ferrari-G*.

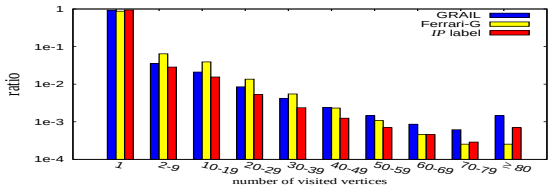
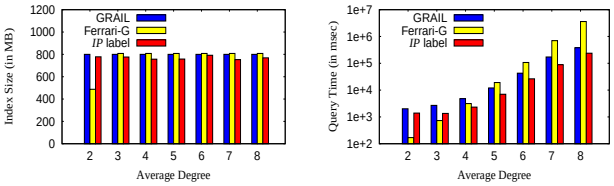


Figure 4: Distribution of the number of vertices visited



(a) Index Size (b) Query Time

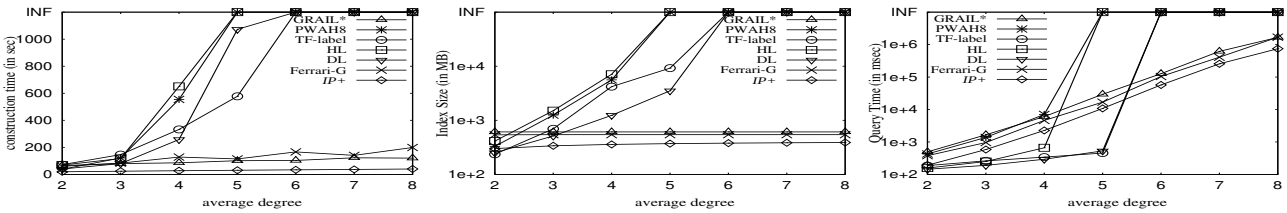
Figure 5: Equal Query Workload

Second, we test equal query workload with 500,000 reachable queries and 500,000 non-reachable queries sampled from TC over the same 7 synthetic graphs we used for the scalability testing (average degree is from 2 to 8). The index size and the query time are shown in Fig. 5. *Ferrari-G* performs well when the average degree is 2, where the index size cannot be larger by controlling the parameters provided in *Ferrari-G*. Overall, *IP* performs the best.

8.4 The *IP+* Label

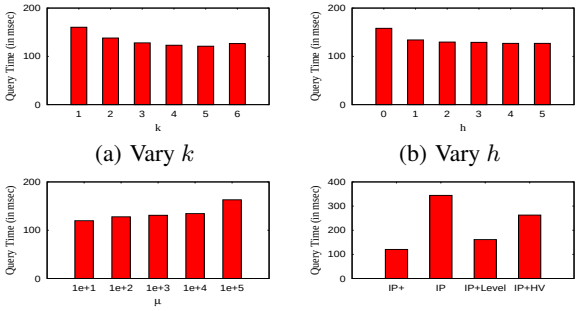
Here, first, we report how the 3 parameters, k, h, and μ, affect the query performance of *IP+*. Second, we report the effectiveness of the three labels used in *IP+*, *IP* label, *Level* label, and *HV-Label*. We conducted extensive testings by 1 million random queries generated. We report the results using the real graph govwild whose average degree is 2.95. As default, $k = 5$, $h = 5$, and $\mu = 100$.

Fig. 6(a) shows that a larger k can improve the query time, but it does not improve the query time when k is too large. Fig. 6(b) shows that *HV-Label* helps. A small h is sufficient because there is a few huge-vertices in a real graph. Fig. 6(c) shows that a smaller μ is sufficient to improve the query time. It confirms that $\mu = 100$ is reasonable. When μ becomes smaller, many vertices will be marked as a huge-vertex, which increase the index size. Fig. 6(d) shows the effectiveness of the labels used in *IP+*. We compare *IP+* with *IP*, *IP* plus *Level*, and *IP* plus *HV-Label*. With *IP* label only, it



(a) Construction Time (b) Index Size (c) Query Time

Figure 3: Query Synthetic graphs with average degree 2 to 8



(a) Vary k (b) Vary h

(c) Vary μ (d) Label Combinations

Figure 6: *IP+*

needs 344 millisecond to answer all queries. *Level* label and *HV-Label* improve the query time effectively.

9. CONCLUSIONS

In this paper, we propose a new *IP* labeling approach, which is the first one to explore the randomness to answer reachability queries. Like the up-to-date Label-Only approaches (*TF-Label*, *HL*, and *DL*), *IP* uses two sets of vertices. Unlike *TF-Label*, *HL*, and *DL*, *IP* uses set-containment instead of set-intersection. The fundamental difference behind answering $\mathsf{Reach}(u, v)$ is as follows. *TF-Label*, *HL*, and *DL* ensure $u \rightsquigarrow v$ using the labels, whereas *IP* is on the opposite aiming at $u \not\rightsquigarrow v$ by finding at least one vertex in one set that is not contained in the other. *IP* is effective given the small reachability-ratio for all large graphs. *IP* outperforms the up-to-date Label+G approaches, *GRAIL* and *Ferrari*. The randomness by independent permutation used in *IP* opens a new direction to study new labeling approaches, in order to further improve the query processing time while minimizing the index construction time/space.

ACKNOWLEDGEMENTS: The work was supported by the grant of the Research Grants Council of the Hong Kong SAR, China, No. 418512.

10. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. of SIGMOD'89*, 1989.

[2] P. Boldi, M. Santini, and S. Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2), 2008.

[3] A. Broder. On the resemblance and containment of documents. In *Proc. of SEQUENCES'97*, 1997.

[4] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proc. of STOC'98*, 1998.

[5] J. Cai and C. K. Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *Proc. of CIKM'10*, 2010.

[6] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *Proc. of ICWSM'10*, 2010.

[7] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. of VLDB'05*, 2005.

[8] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *Proc. of ICDE'08*, 2008.

[9] Y. Chen and Y. Chen. Decomposing dags into spanning trees: A new way to compress transitive closures. In *Proc. of ICDE'11*, 2011.

[10] 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. of SIGMOD'13*, 2013.

[11] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *Proc. of EDBT'06*, 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 *Proc. of EDBT'08*, 2008.

[13] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of SODA'02*, 2002.

[14] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM Computer Communication Review*, 29(4), 1999.

[15] R. A. Fisher, F. Yates, et al. *Statistical tables for biological, agricultural and medical research.* Oliver and Boyd, Edinburgh, 3rd edition, 1949.

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

[17] R. Jin, N. Ruan, S. Dey, and J. X. Yu. Scarab: scaling reachability computation on large graphs. In *Proc. of SIGMOD'12*, 2012.

[18] 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), 2011.

[19] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *PVLDB*, 6(14), 2013.

[20] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-HOP: A high-compression indexing scheme for reachability query. In *Proc. of SIGMOD'09*, 2009.

[21] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *Proc. of SIGMOD'08*, 2008.

[22] D. E. Knuth. The art of computer programming, volume 2: seminumerical algorithms, 1981.

[23] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex XML document collections. In *Proc. of EDBT'04*, 2004.

[24] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *Proc. of ICDE'13*, 2013.

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

[26] S. TrißI and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. of SIGMOD'07*, 2007.

[27] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *Proc. of SIGMOD'11*, 2011.

[28] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proc. of ICDE'06*, 2006.

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

[30] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: a scalable index for reachability queries in very large graphs. *VLDB Journal*, 21(4), 2012.

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