Reachability Querying: Can It Be Even Faster?

Jiao Su[†‡], Qing Zhu[†], Hao Wei[‡], and Jeffrey Xu Yu[‡]

[†]*Renmin University of China, Beijing, China;* [‡]*The Chinese University of Hong Kong, Hong Kong*
{jiaosu,hwei,yu}@se.cuhk.edu.hk, zq@ruc.edu.cn

Abstract—As an important graph operator, reachability query has been extensively studied over decades, which is to check whether a vertex can reach another vertex over a large directed graph G with n vertices and m edges. The efforts made in the reported studies have greatly improved the query time of answering reachability queries online, while reducing the offline index construction time to construct an index with a reasonable size given the approach taken, where an entry in an index for a vertex is called a label of the vertex. Among all the work, the recent development of *IP* (Independent Permutation) employs randomness using k-min-wise independent permutations to process reachability queries, and shows the advantages for both query time and index construction time. In this paper, we propose a new Bloom filter Labeling, denoted as *BFL*. We show that the probability to answer reachability queries by *BFL* can be bounded, and *BFL* has high pruning power to answer more reachability queries directly. We give algorithms and analyze the pruning power of *BFL*. We conduct extensive studies using 19 large datasets. We show that *BFL* with an interval label performs best in the index construction time for all 19 cases, and performs best in query time for 16 out of 19 cases.

Index Terms—Reachability Query, Bloom Filter

1 Introduction

Graph processing becomes even important given a large number of emerging real graphs from collaborator networks, bibliographic networks, online social networks, etc, which are large in size. Among all the graph operators, one important operator is reachability query to check if a vertex can reach another vertex in a large graph. The applications, as given in [28], include semantic web (*RDF*), concept subsumption in ontologies (i.e., Gene Ontology), biological networks, and online social networks. And such reachability queries are also supported in *SPARQL*. Due to the importance of reachability checking, it has been extensively studied as can be observed from the number of studies in the literature [1], [11], [10], [19], [8], [25], [4], [22], [9], [5], [16], [15], [27], [29], [23], [6], [12], [7], [14], [20], [24], [26]. The efforts made in the reported studies have greatly improved the query time of answering reachability queries online, while reducing the offline index construction time to construct an index with a reasonable size given the approach taken, where an entry in an index for a vertex is called a label of the vertex. Following [26], the existing approaches are categorized into Label-Only and Label+G. The Label-Only includes *Chain-Cover* [11], [5], *Tree-Cover* [1], *Dual-Label* [25], *2-Hop* [10], *Path-Tree* [16], *3-Hop* [15], bit-vector [23], *TF-Label* [7], *HL* [14], and *DL* [14]. They only need to answer reachability queries using the index constructed for a graph G, without accessing the graph G. The Label+G includes *Tree+SSPI* [4], *GRIPP* [22], *GRAIL* [27], *Ferrari* [20], *Feline* [24] and *IP* [26]. The Label+G approaches are designed to reduce the offline index construction time taken by the Label-Only approaches, due to the fact that the Label-Only approaches are to compress transitive closure (TC) with a high overhead. In other words, Label+G approaches are designed to answer reachability queries as much as possible using the index constructed, and conduct depth-first search (*DFS*) over G when the index cannot be used to answer alone.

In a summary, all the approaches balance the offline and online costs. For the offline cost, it is the index construction time and the index size; for the online cost, it is the query processing time. For the Label-Only approaches, the early attempts are to compress TC to reduce the index size, where the index construction time can be high to deal with large denser graphs. The recent attempts such as *TF-Label*, *HL*, and *DL*, are to reduce the index construction time, but the index sizes of such approaches cannot be bounded, and therefore the query processing time cannot be bounded given the query time is related to the index size. Different from the Label-Only approaches, the Label+G approaches are proposed to construct an index in both linear time and linear space. However, such linear approaches come with a cost. The Label+G approaches may take long query time, since they may need to conduct *DFS* to search the destination vertex in a graph.

Among all the work, the recent development of *IP* (Independent Permutation) [26] is unique in the sense that it is the first to employ randomness for reachability queries. We introduce *IP* in brief below. *IP* takes reachability testing from a vertex u to a vertex v as set-containment testing. Intuitively, if a vertex u can reach a vertex v, then u must be able to reach all the vertices that v can reach. Let $Out(x)$ denote the set of all vertices that a vertex x can reach in a graph. If u can reach v, then $Out(v) \subseteq Out(u)$. To answer a reachability query from u to v is to test whether $Out(v) \subseteq Out(u)$. *IP* checks $Out(v) \subseteq Out(u)$ by checking $Out(v) \not\subseteq Out(u)$ instead, which is to check if there is at least one element in $Out(v)$ that is not in $Out(u)$. Since the cost of checking set-containment is high, instead of getting an exact answer for testing $Out(v) \not\subseteq Out(u)$, *IP* is designed to get an answer with the probability guarantee based on k-min-wise independent permutations. By k-min-wise independent permutations, an *IP* label of a vertex u consists of two sub-labels, one for $Out(u)$ and one for $In(u)$, where $In(u)$ is the set of all vertices that can reach u in a graph. Both sub-labels keep up to k smallest numbers by a permutation. As reported in [26], among 19 large real and synthetic datasets being tested, *IP*+ performs best in 14 out of 19 for the index construction time, and performs best in 10 out of 19 for query time, where *IP+* is to use *IP* with two additional labels to early stop *DFS* at run-time.

The key challenge is to process reachability queries even faster, given all the efforts that have already been made over years to improve the efficiency to a level that a 1 million reachability queries can be answered in 20 milliseconds or one single reachability query can be answered with less than 20 nanoseconds. The main contributions of this work are summarized below.

- First, we propose a Bloom-filter labeling, denoted as *BFL*, and give the algorithms to further improve the performance of *IP* using every bit to prune.
- Second, we study the *BFL* pruning power to answer reachability queries by *BFL* directly. By directly, we mean that there is no need to conduct *DFS*. The pruning power is one minus *BFL* false positive rate, and a small false postive rate implies that more reachability queries can be answered by *BFL* directly. We show that the false positive rate can be bounded.
- Third, we further give the conditions under which the false positive rate can be controlled using vertices mergning, and show the *BFL* pruning power over vertices merging is high. Comparing to the up-to-date *IP* technique [26] which uses up to k smallest numbers by a permutation where a number is represented as an integer, *BFL* has higher pruning power to answer reachability queries directly using every bit in *BFL*. It is worth noting that an integer can be used to represent many bits, even though a number in *IP* has two times of power over a single bit used in *BFL* proposed in this work.
- Finally, we conduct extensive experimental studies to test our *BFL* approach with an interval label. We compare with the up-to-date approaches including *IP*, using 19 large real datasets used in the existing work. We confirm that *BFL* is the best in index construction time in all 19 cases, and is the best in query time for 16 out of 19 cases.

The remainder of the paper is organized as follows. The related work is given in Section 2. We discuss the preliminaries in Section 3, We discuss the new Bloom filter Label in Section 4, give the algorithms to compute *BFL* in Section 5, and give the algorithm to answer reachability queries in Section 6. We confirm the efficiency of *BFL* in extensive experimental studies using 19 large real and synthetic datasets, and report our findings in Section 7. We conclude this work in Section 8.

2 Related Work

The research issue for answering $\mathsf{Reach}(u, v)$ is to find a trade-off between the online processing cost and the offline processing cost. The best online processing cost is $O(1)$ if the transitive closure (TC) of a graph G is maintained at the expense of the space complexity, $O(n^2)$, since it needs to maintain all pairs for a graph with n vertices. It is impractical since TC is too large to be maintained. The largest online processing cost without any preprocessing or any index built is $O(n + m)$, since it needs to traverse a graph with n vertices and m edges in the worst case, which is known to be impractical to deal with large graphs.

All the reported studies aim to find a trade-off among three main costs: (a) query time which is the time to answer a reachability query online, (b) construction time which is the time to construct an index offline, and (c) the index size which is the space needed to maintain the index. As indicated by the three costs, the main idea behind all the reported studies is to build an index, where an entry in the index for a vertex u in G is called a label, denoted as $\mathsf{label}(u)$, and to answer reachability queries using labels. As categorized in [26], there are two main approaches, namely Label-Only and Label+G. The Label-Only approach by name is to answer reachability queries using the labels computed only, where the labels are computed by compressing TC offline. The Label+G approach is to make use of the labels computed as much as possible, but it may need to traverse G. The main idea behind Label+G is to reduce the time to construct an effective index, since the computing cost to compress TC can be very high. A survey can be found in [29].

	Query Time	**Index Size**	**Construction Time**
BFS/DFS	$O(n+m)$	$O(1)$	$O(1)$
TC [21], [23]	$O(1)$	$O(n^2)$	$O(nm)$
Chain-Cover [5]	$O(\log k)$	$O(kn)$	$O(n^2 + kn\sqrt{k})$
Path-Tree [16]	$O(\log^2 k)$	$O(kn)$	$O(km)$ or $O(nm)$
Tree-Cover [1]	$O(\log n)$	$O(n^2)$	$O(nm)$
Dual-Label [25]	$O(1)$	$O(n+t^2)$	$O(n+m+t^3)$
2-Hop [10]	$O(m^{1/2})$	$O(nm^{1/2})$	$O(n^3 \cdot \vert TC\vert)$
3-Hop [15]	$O(\log n + k)$	$O(kn)$	$O(kn^2 \cdot \vert Con(G)\vert)$
TF-Label [7]	–	–	$O(T)$
HL [14]	–	–	$O(H)$
DL [14]	–	–	$O(n(n+m)L)$
TOL [31]	–	–	$O(TOL)$
Tree+SSPI [4]	$O(m-n)$	$O(n+m)$	$O(n+m)$
GRIPP [22]	$O(m-n)$	$O(n+m)$	$O(n+m)$
GRAIL [27]	$O(k)$ or $O(n+m)$	$O(kn)$	$O(k(n+m))$
Ferrari [20]	$O(k)$ or $O(n+m)$	$O((k+s)n)$	$O(k^2 m + S)$
Feline [24]	$O(1)$ or $O(n+m)$	$O(n)$	$O(n\log n + m)$
IP [26]	$O(k)$ or $O(knr^2)$	$O((k+h)n)$	$O((k+h)(m+n))$
BFL (ours)	$O(s)$ or $O(sn+m)$	$O(sn)$	$O(s(m+n))$

TABLE 1
Time and Space Complexity

Table 1 summarizes the three main costs for the existing work. We discuss the factors in the complexities for the approaches given in Table 1 in brief, where the outlines of the approaches listed in Table 1 can be found in [26]. For all approaches, n and m are the number of vertices and the number of edges in G, respectively.

First, we discuss Label-Only approaches. *Chain-Cover* [11] is to compress TC by chain decomposition to find a minimal number of pair-wise disjoint chains to represent DAG, where k in the complexity is the minimal number of chains in DAG. *Path-Tree* [13] decomposes a DAG G into a set of pair-wise disjoint paths, where k is the number of paths computed in *Path-Tree*. *Tree-Cover* [1] covers TC using an optimal spanning tree, where a label is a set of intervals. The reachability from u to v is decided by whether the interval of v is contained in the interval of u. *Dual-Label* [25] is defined for sparse graphs, where a spanning tree can be effectively used to answer most reachability queries. In addition to the labels on the spanning tree, *Dual-Label* also deals with those non-tree edges which are not the edges on the spanning tree. In the complexity, t is the number of non-tree edges, and $t \ll n$ for a sparse graph. *2-Hop* [10] labels a vertex in G using two subsets of vertices, and compresses TC in a way as to deal with the set cover problem, the labels are assigned approximately, since the set cover problem is known as NP-hard. An approximate (greedy) algorithm is proposed [10]. Several attempts have been made to speed up computing of *2-Hop* [19], [8], [9]. *3-Hop* [15] aims at improving *2-Hop* by further making use of the idea behind *Chain-Cover*. In the complexity, k is the number of chains, and $Con(G)$ is the transitive closure contour. To improve *3-Hop*, *Path-Hop* [3] uses a tree structure to replace the chain decomposition used in *3-Hop*. *TF-Label* (topological folding) [7] is to compute *2-Hop* labels using topological level. It represents a DAG G as a sequence of DAGs, where the first $G_0 = G$, and G_i is constructed based on its previous DAG G_{i-1}, where the topological level is for every DAG. The bound T in Table 1 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, $\ell(G)$ is the max level of G, and G_i^* is the transformed graph of G_i. The bounds for query time and index size are unknown mainly because *TF-Label* is to improve the efficiency of *2-Hop* labels computing, rather than minimizing the index size. *HL* (Hierarchical Labeling) and *DL* (Distribution Labeling) [14] are proposed over the general framework *SCARAB* [12]. *HL* represents a graph G as a sequence of reachability backbones. The initial reachability backbone is the graph G itself, and in the sequence the i-th backbone, B_i, is the backbone of the (i-1)-th backbone, B_{i-1}, in the sequence. The time complexity H in Table 1 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. *DL* is based on the order of vertices to compute *2-Hop* labels following the breadth-first search (*BFS*) forward/backward. For the time complexity shown in Table 1, L is the maximal labeling size. Like *TF-Label*, both *HL* and *DL* are designed to reduce the cost of computing *2-Hop* labels, and are not designed to reduce the complexities for the index size and the query time. *TOL* proposes a framework which summarizes *TF-Label* and *DL* [31]. Liked *DL*, it computes *2-Hop* labels based on a total order and its time complexity is $O(TOL) = O(\sum_{k=1}^{n}(|E_k| + k|V_k|))$ where V_k and E_k is the number of visited nodes and edges when computing the node in the k-th iteration. It further investigates how to get a good total order which reduces the index size. Dynamic maintenance algorithm of *2-Hop* label is also proposed based on the total order.

Second, we discuss Label+G approaches. Both *Tree+SSPI* [4] and *GRIPP* [22] use an interval label for every vertex over a single spanning tree, and depth-first-search (*DFS*) to traverse when needed. *GRAIL* [27], [28] randomly generates k spanning trees by *DFS*, instead of single spanning tree. The query time is either $O(k)$ when the labels can be used to answer directly, or $O(n+m)$ when *DFS* is needed. Like *GRAIL*, *Ferrari* [20] uses up to k intervals for every vertex of G. Unlike *GRAIL*, the k intervals are computed over an optimal spanning tree [1] using some approximation techniques. In Table 1, for the complexities of *Ferrari*, S is the time complexity to find the top-s largest degree vertex. *Feline* [24] is inspired by *Dominance Graph Drawing*, and uses two topological orders to label every vertex. The second topological order is computed based on the first topological order to maximize the coverage of unreachable pairs. The query vertex pair is answered as unreachable pairs iff they cannot satisfy either of these two topological orders. If their topological orders fail to answer a reachability query, *Feline* performs *DFS* like other Label-G approaches. *IP* [26] is based on k-min-wise independent permutation. Together with *IP*, it also uses two additional labels, namely, the level label and the huge-vertex label, where the level label is used to stop *DFS* early as used in *GRAIL* and *TF-Label*, and the huge-vertex label is used to handle high outdegree vertices. The size of a huge-vertex label for a high outdegree vertex is limited by h. In Table 1, r is the R-ratio, which indicates the percentage of reachability queries that end up to be $u \leadsto v$, which is known to be very small [26]. Also, I/O efficiency for reachability queries is studied in [30].

3 Preliminaries

We discuss it following the similar notations used in [26]. We model a directed graph as $G = (V, E)$ where V represents a set of vertices and E represents a set of edges (ordered pairs of vertices) of G. 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 $\mathsf{path}(u, v) = (v_1, v_2, \ldots, v_p)$ where (v_i, v_{i+1}) is an edge in E, for $1 \leq i < p$, $u = v_1$, and $v = v_p$. 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 $Suc(u)$ and $Pre(u)$ to denote the direct successors and direct predecessors of u, such as $Suc(u) = \{v \mid (u, v) \in E\}$ and $Pre(u) = \{v \mid (v, u) \in E\}$, respectively. The indegree and the outdegree of a vertex u is denoted as $deg^-(u) = |Pre(u)|$ and $deg^+(u) = |Suc(u)|$.

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

In this paper, we assume G is a directed acyclic graph (DAG), as assumed in the existing work. This is because the reachability in G can be completely represented by its 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 S_i to another vertex in SCC S_j. All reachability queries, $\mathsf{Reach}(u, v)$, in G can be answered by $\mathsf{Reach}(S_i, S_j)$ in the corresponding DAG, assuming u and v are in S_i and S_j, respectively. When two vertices are in the same SCC, they are reachable from each other by SCC definition. Note that DAG can be computed in $O(n+m)$ time efficiently.

4 Bloom Filter Labeling

In this paper, we propose a new labeling approach, called *BFL* (Bloom filter Labeling), denoted as $\mathcal{L}_{bf}(u)$ for a vertex u in G, where $\mathcal{L}_{bf}(u)$ consists of two sets of integers, $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$. Both $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$ are a subset of $\{1, 2, \ldots, s\}$, where s is a user-given number, and both can be saved in a bit vector of $\lceil s/8 \rceil$ bytes. As a hash-based approach, like *IP* [26], the main idea behind *BFL* is $u \leadsto v$ if u can reach all the vertices that v can reach and all vertices that can reach u can also reach v, denoted as both $Out(v) \subseteq Out(u)$ and $In(u) \subseteq In(v)$. This also means $u \not\leadsto v$ if $Out(v) \not\subseteq Out(u)$ or $In(u) \not\subseteq In(v)$. By *BFL*, we aim at achieving the following three altogether.

- **Linear Construction Time**: The construction time is linear with the size of graph $(n+m)$. As bit vector is used to implement the operation over sets, the constant factor is much smaller than the other linear approaches.
- **Linear Index Size**: The index size is linear with the number of vertices.
- **Efficient Query Processing**: The querying processing time is in $O(s)$, with a few of queries need to do *DFS*.

We give the definition of the Bloom filter [2] on which *BFL* is defined.

Definition 4.1: Given a set of elements, X, a function over X is called a hash function if it randomly maps an element to a number in $\{1, 2, \ldots, s\}$, denoted as $h : X \rightarrow \mathbb{Z}$. Let $\mathcal{P}(X)$ be a power set of X. A function over a power set is called a Bloom filter function, denoted as $\mathcal{B} : \mathcal{P}(X) \rightarrow \mathcal{P}(\mathbb{Z})$, such that $\mathcal{B}(S) = \{h_i(x) \mid x \in S, 1 \leq i \leq t\}$, where $\{h_1, h_2, \ldots, h_t\}$ is a set of independent hash functions. □

Bloom filter is designed to determine whether an element belongs to a certain set. Given a set of elements, S, if we have $h_i(x) \in \mathcal{B}(S)$ for all $1 \leq i \leq t$, then we can claim that x is in S; otherwise we can claim that x cannot be in S. For the latter case, since $h_i(x) \in \mathcal{B}(S)$ for all $x \in S$ and $1 \leq i \leq t$, the claim is always true by definition. However, for the former case, the claim is not always true. For example, suppose we have a hash function $h_1(\cdot)$ which hashes a number into either 1 or 2, for $s = 2$. Let $X = \{1, 2, 3\}$, and assume $h_1(1) = 2$, $h_1(2) = 2$, and $h_1(3) = 1$. Then, consider whether 2 is in the set of $S = \{1, 3\}$. By the Bloom filter, since $\mathcal{B}(S) = \{1, 2\}$ for $h_1(1) = 2$ and $h_1(3) = 1$, and $h_1(2) \in \mathcal{B}(S)$, it answers yes, which is wrong. It is known that when testing whether an element belongs to a set by Bloom filter, there cannot be any false negative, but there can be false positive. In other words, Bloom filter has a 100% recall rate, and a non-zero false positive rate. The approximation of the false positive is given in the following equation.

$$\Pr(h_i(x) \in \mathcal{B}(S)) \approx \left(1 - e^{-t \cdot |S|/s}\right)^t \tag{1}$$

In this work, we study how to answer negative reachability queries, $u \not\rightsquigarrow v$, by checking $Out(v) \not\subseteq Out(u)$ or $In(u) \not\subseteq In(v)$ using *BFL* directly, since the percentage of reachability queries that end up to be reachable ($u \rightsquigarrow v$) is very small [26]. Let S and T be the sets representing $Out(u)$ and $Out(v)$, respectively. Our focus is on containment test between two sets, S and T, instead of membership test between an element and a set. We prove $\mathcal{B}(T) \subseteq \mathcal{B}(S)$, if $T \subseteq S$. We study the pruning power by *BFL*, which is $1 - p$, where p is the *BFL* false positive rate. The non-zero *BFL* false positive rate, p, implies that some negative reachability queries, $u \not\rightsquigarrow v$, cannot be answered by *BFL* only. By treating the *BFL* false positive rate as the probability of $\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S))$, we analyze the pruning power by studying the probability that *BFL* can be used to answer negative reachability queries directly. In Section 4.1, let s be a number, and let $\alpha = |S|$ and $\beta = |T \setminus S|$. We show that the false positive rate of p by *BFL* can be bounded in terms of α, β, and s (refer to Eq. (8)). In order to achieve such bound, we point out the condition that must be held is $\alpha \leq cs$ for $c = \ln \beta - \ln(\ln \frac{1}{p})$. In Section 4.2, to ensure $\alpha \leq cs$, we discuss vertices merging to merge subsets of vertices to representative vertices. Such vertices merging reduces the number of vertices needed in S and T, but introduces additional false positive rate. We further give the bound of such false positive rate by *BFL* over vertices merging (Eq. (10)). In Section 4.3, we show the false positive rate of *BFL* over vertices merging outperforms *IP* by TC zero-cover.

4.1 Bloom Filter Label

In our approach, we use one single hash function for the Bloom filter, $h_1(\cdot)$ or simply $h(\cdot)$. We prove $\mathcal{B}(T) \subseteq \mathcal{B}(S)$, if $T \subseteq S$ in Theorem 4.1.

Theorem 4.1: *Let S and T be two subsets of X and $\mathcal{B} : \mathcal{P}(X) \rightarrow \mathcal{P}(\mathbb{Z})$ be a Bloom filter function. If $T \subseteq S$, then $\mathcal{B}(T) \subseteq \mathcal{B}(S)$.*

Proof: Let $b \in \mathcal{B}(T)$, by Definition 4.1, there exists a $u \in T$ such that $b = h(u)$. Since $T \subseteq S$, we have $u \in S$ for all $u \in T$. Thus, there exists a $u \in S$ such that $b = h(u)$ for all $b \in \mathcal{B}(T)$, i.e., $b \in \mathcal{B}(S)$ for all $b \in \mathcal{B}(T)$. Therefore, $\mathcal{B}(T) \subseteq \mathcal{B}(S)$. □

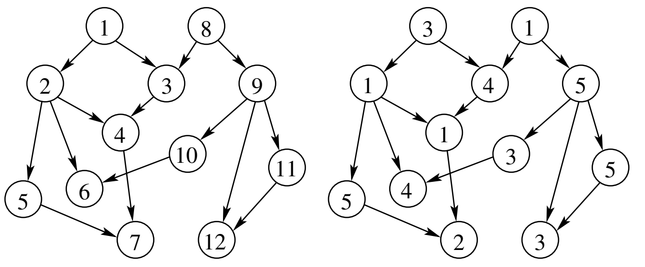

(a) DAG G (b) $h(v)$ for $v \in G$

Fig. 1. An Example

Vertex	$\mathcal{L}_{out}(u)$	$\mathcal{L}_{in}(u)$
v_1	$\{1, 2, 3, 4, 5\}$	$\{3\}$
v_2	$\{1, 2, 4, 5\}$	$\{1, 3\}$
v_3	$\{1, 2, 4\}$	$\{1, 3, 4\}$
v_4	$\{1, 2\}$	$\{1, 3, 4\}$
v_5	$\{2, 5\}$	$\{1, 3, 5\}$
v_6	$\{4\}$	$\{1, 3, 4, 5\}$
v_7	$\{2\}$	$\{1, 2, 3, 4, 5\}$
v_8	$\{1, 2, 3, 4, 5\}$	$\{1\}$
v_9	$\{3, 4, 5\}$	$\{1, 5\}$
v_{10}	$\{3, 4\}$	$\{1, 3, 5\}$
v_{11}	$\{3, 5\}$	$\{1, 5\}$
v_{12}	$\{3\}$	$\{1, 3, 5\}$

TABLE 2
$\mathcal{L}_{bf}(u)$ of vertices in Fig. 1

We show that we can answer reachability queries with the help of Bloom filter function $\mathcal{B}$ in Theorem 4.2.

Theorem 4.2: *Let $X = V$, $S = Out(u)$ and $T = Out(v)$, and $\mathcal{B} : \mathcal{P}(V) \rightarrow \mathcal{P}(\mathbb{Z})$ be a Bloom filter function. If $u \rightsquigarrow v$, then $\mathcal{B}(T) \subseteq \mathcal{B}(S)$.*

Proof: $Out(u)$ and $Out(v)$ are defined as the vertices u can reach and the vertices v can reach respectively. By $u \rightsquigarrow v$, it means u can reach v. It is obvious that every vertex, that v can reach, is reachable from u. Thus $Out(v) \subseteq Out(u)$, $T \subseteq S$. By Theorem 4.1, $\mathcal{B}(T) \subseteq \mathcal{B}(S)$ □

Theorem 4.2 holds if we replace $Out(u)$ and $Out(v)$ with $In(v)$ and $In(u)$ respectively in the statement.

For a vertex $u \in G$ and a given s, we define Bloom filter label (or *BFL* for short) of a vertex u, denoted by $\mathcal{L}_{bf}(u)$, as a pair of sets that are subsets of $\{1, 2, \ldots, s\}$, $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$. Here, $\mathcal{L}_{out}(u)$ keeps the image of $Out(u)$ under a Bloom filter function $\mathcal{B}$, such as $\mathcal{L}_{out}(u) = \mathcal{B}(Out(u))$, and $\mathcal{L}_{in}(u)$ keeps the image of $In(u)$ under a Bloom filter function $\mathcal{B}$ (possibly different from the one used in $\mathcal{L}_{out}$), such as $\mathcal{L}_{in}(u) = \mathcal{B}(In(u))$. To answer $\mathsf{Reach}(u, v)$, we have (a) $u \rightsquigarrow v$ if $u = v$, (b) $u \not\rightsquigarrow v$ if $\mathcal{L}_{out}(v) \not\subseteq \mathcal{L}_{out}(u)$ or $\mathcal{L}_{in}(u) \not\subseteq \mathcal{L}_{in}(v)$ by the Bloom filter labels only following Theorem 4.2, and (c) otherwise checking $\mathsf{Reach}(w, v)$ for all $w \in Suc(u)$ recursively following *DFS* during which $u \rightsquigarrow v$ if $w \rightsquigarrow v$ since $u \rightsquigarrow w$ holds by *DFS*.

Example 4.1: A DAG G is shown in Fig. 1. The numbers in Fig. 1 identify the 12 vertices in $V = \{v_1, v_2, \ldots, v_{12}\}$. Let $s = 5$. Given a hash function $h(\cdot)$, assume $h(v_1) = 3$, $h(v_2) = 1$, $h(v_3) = 4$, $h(v_4) = 1$, $h(v_5) = 5$, $h(v_6) = 4$, $h(v_7) = 2$, $h(v_8) = 1$, $h(v_9) = 5$, $h(v_{10}) = 3$, $h(v_{11}) = 5$ and $h(v_{12}) = 3$. Fig. 1(b) shows $h(v_i)$ for every $v_i \in V$. Table 2 shows *BFL* for G in Fig. 1(a). We show four reachability queries.

- (Q_1) $\mathsf{Reach}(v_3, v_9)$: We have $\mathcal{L}_{out}(v_3) = \{1, 2, 4\}$ and $\mathcal{L}_{out}(v_9) = \{3, 4, 5\}$. Since $3 \in \mathcal{L}_{out}(v_9)$, $3 \notin \mathcal{L}_{out}(v_3)$, we have $\mathcal{L}_{out}(v_9) \not\subseteq \mathcal{L}_{out}(v_3)$, and then $v_3 \not\rightsquigarrow v_9$ by Theorem 4.2.
- (Q_2) $\mathsf{Reach}(v_1, v_{11})$: We have $\mathcal{L}_{out}(v_1) = \{1, 2, 3, 4, 5\}$, $\mathcal{L}_{out}(v_{11}) = \{3, 5\}$. So $\mathcal{L}_{out}(v_{11}) \subseteq \mathcal{L}_{out}(v_1)$. However, since $\mathcal{L}_{in}(v_1) = \{3\}$ and $\mathcal{L}_{in}(v_{11}) = \{1, 5\}$, we have $\mathcal{L}_{in}(v_1) \not\subseteq \mathcal{L}_{in}(v_{11})$. By Theorem 4.2, $v_1 \not\rightsquigarrow v_{11}$.
- (Q_3) $\mathsf{Reach}(v_1, v_5)$: We have $\mathcal{L}_{out}(v_5) \subseteq \mathcal{L}_{out}(v_1)$ and $\mathcal{L}_{in}(v_1) \subseteq \mathcal{L}_{in}(v_5)$. We cannot answer it by *BFL* only. *DFS* is needed over G (Fig. 1(a)). We start *DFS* from v_1. Supposed the next vertex to be visited is v_2. We have to continue *DFS*, because $\mathcal{L}_{out}(v_5) \subseteq \mathcal{L}_{out}(v_2)$ and $\mathcal{L}_{in}(v_2) \subseteq \mathcal{L}_{in}(v_5)$. Suppose the next to be visited is v_5, we can conclude that $v_1 \rightsquigarrow v_5$.
- (Q_4) $\mathsf{Reach}(v_2, v_3)$: We have $\mathcal{L}_{out}(v_3) \subseteq \mathcal{L}_{out}(v_2)$ and $\mathcal{L}_{in}(v_2) \subseteq \mathcal{L}_{in}(v_3)$, we cannot answer it by *BFL* labels only, and *DFS* is needed. We start *DFS* from v_2, which has the three direct successors, v_4, v_5 and v_6. For v_4, there is no need to continue *DFS*, because $v_4 \not\rightsquigarrow v_3$ for $\mathcal{L}_{out}(v_4) \not\subseteq \mathcal{L}_{out}(v_3)$ and $\mathcal{L}_{in}(v_3) \not\subseteq \mathcal{L}_{in}(v_4)$. For the similar reason, there is no need to continue *DFS* for the other two nodes, v_5 and v_6. We have $v_2 \not\rightsquigarrow v_3$.

We have showed how to use the condition of $\mathcal{B}(T) \not\subseteq \mathcal{B}(S)$ to conclude $T \not\subseteq S$. But, the condition of $\mathcal{B}(T) \subseteq \mathcal{B}(S)$ cannot conclude $T \subseteq S$. Like using Bloom filter to test membership, using Bloom filter to test containment also has a 100% recall rate (by Theorem 4.2) and a non-zero false positive rate. The smaller false positive rate we have, the better query processing performance we get. To analyze the false positive rate when testing containment between two sets, we introduce a probability distribution p.

Lemma 4.1: *Given a Bloom filter function, $\mathcal{B} : \mathcal{P}(X) \to \mathcal{P}(\mathbb{Z})$, where a hash function used randomly maps a number into a number in $\{1, 2, \cdots, s\}$. Let S be a subset of X, where the size of S is $\alpha = |S|$. The probability $p(i)$ that $\mathcal{B}(S)$ contains exactly i elements is as follows.*

$$p(i) = \frac{\left\{ {\alpha \atop i} \right\} \cdot \binom{s}{i} \cdot i!}{s^{\alpha}} \tag{2}$$

where $\binom{s}{i}$ is the number of ways to choose i distinct elements from a set of s elements (known as binomial coefficients), and $\left\{ {\alpha \atop i} \right\}$ is the number of ways to partition a set of α elements into i nonempty subsets (known as Stirling numbers of the second kind). □

Proof: Let $\mathcal{H} = \{h_1, h_2, \cdots\}$ be the set of all possible hash functions that hash X elements into $\{1, 2, \cdots, s\}$. The size of $\mathcal{H}$ is $s^{|X|}$. The probability that $\mathcal{B}(S)$ contains exactly i elements, indicated by $p(i)$, is the number of hash functions that hash S into i distinct numbers over all possible hash functions.

$$p(i) = \frac{\text{\# of hash functions that end up } |\mathcal{B}(S)| = i}{s^{|X|}}$$

Recall a hash function h_i is defined over X not over S, and $S \subseteq X$. In other words, the number (#) of hash functions that end up $|\mathcal{B}(S)| = i$ is multiplication of two parts. The first part is related to how h_i hashes S, and the second part is related to how h_i hashes $X \setminus S$. Here, the number of hash functions for the second part is $s^{|X \setminus S|}$. In the following we focus on the first part, and $p(i)$ can be rewritten as follows.

$$p(i) = \frac{\text{\# of hash functions that end up } |\mathcal{B}(S)| = i \text{ over } S}{s^{\alpha}}$$

where $\alpha = |S|$ and $s^{\alpha} = s^{|X|}/s^{|X \setminus S|}$. To consider the first part is to consider hash functions over S. On one hand, there are in total $\left\{ {a \atop i} \right\}$ ways to partition S into i nonempty subsets, where $\left\{ {a \atop i} \right\}$ is the Stirling number of the second kind (or Stirling partition number). On the other hand, there are $\binom{s}{i}$ ways to select i numbers from $\{1, 2, \ldots s\}$ in total. And for every partition there are $i!$ ways to map an element in the partition to a certain number. Thus, there are $\left\{ {a \atop i} \right\} \cdot \binom{s}{i} \cdot i!$ different hash functions over S that map S into i distinct numbers. The lemma is proved. □

Based on the probability distribution given in Lemma 4.1, we give the false positive rate in testing whether T is a subset of S, $\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S))$, in Theorem 4.3.

Theorem 4.3: *Given a Bloom filter function, $\mathcal{B} : \mathcal{P}(X) \to \mathcal{P}(\mathbb{Z})$, where a hash function used randomly maps a number into a number in $\{1, 2, \cdots, s\}$. Let S and T be two subsets of X, where $\alpha = |S|$ and $\beta = |T \setminus S|$. Then we have*

$$\begin{aligned} \Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S)) &= \sum_{i=1}^{s} p(i) \cdot \left(\frac{i}{s}\right)^{\beta} \\ &= \sum_{i=1}^{s} \frac{\left\{ {\alpha \atop i} \right\} \cdot \binom{s}{i} \cdot i!}{s^{\alpha}} \cdot \left(\frac{i}{s}\right)^{\beta} \end{aligned} \tag{3}$$

Proof: It is obvious that

$$\begin{aligned} & \Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S)) \\ = & \sum_{i=1}^{s} \Pr(|\mathcal{B}(S)| = i) \cdot \Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S) \mid |\mathcal{B}(S)| = i) \\ = & \sum_{i=1}^{s} p(i) \cdot \Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S) \mid |\mathcal{B}(S)| = i) \end{aligned}$$

Here $p(i)$ is given by Lemma 4.1. Following the hash function definition, we also have

$$\begin{aligned} & \Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S) \mid |\mathcal{B}(S)| = i) \\ = & \Pi_{x \in T \setminus S} \Pr(h(x) \in \mathcal{B}(S) \mid |\mathcal{B}(S)| = i) \\ = & \Pi_{x \in T \setminus S} \frac{i}{s} = \left(\frac{i}{s}\right)^{\beta} \end{aligned}$$

Then, the theorem is proved. □

Consider (Q_1) in Example 4.1. Let $S = Out(v_3) = \{3, 4, 7\}$ and $T = Out(v_9) = \{6, 9, 10, 11, 12\}$. We have $\alpha = 3$ and $\beta = 5$. By Theorem 4.3, $Pr(\mathcal{L}_{out}(v_9) \subseteq \mathcal{L}_{out}(v_3)) = \sum_{i=1}^{5} \frac{\left\{ {3 \atop i} \right\} \cdot \binom{5}{i} \cdot i!}{5^3} \cdot \left(\frac{i}{5}\right)^5 \approx 4.23\%$. That is to say, we have 95.77% chance to answer $Reach(v_3, v_9)$ by looking at $\mathcal{L}_{out}(v_3)$ and $\mathcal{L}_{out}(v_9)$ only.

Next, we will give an approximation of false positive rate, since the exact false positive rate is hard to analyze.

Lemma 4.2: *Given a Bloom filter function, $\mathcal{B} : \mathcal{P}(X) \to \mathcal{P}(\mathbb{Z})$, where a hash function used randomly maps a number into a number in $\{1, 2, \cdots, s\}$. Let S be a subset of X, where the size of S is $\alpha = |S|$. The expected size of $\mathcal{B}(S)$, $E[|\mathcal{B}(S)|]$, is $s(1 - (1 - \frac{1}{s})^{\alpha})$.* □

Proof: Let $X_1, X_2, \ldots, X_s$ be random variables such that $X_i = 1$ if $i \in \mathcal{B}(S)$ and $X_i = 0$ otherwise. Then $E[|\mathcal{B}(S)|] =$

$E[\sum_{i=1}^{s} X_i] = \sum_{i=1}^{s} E[X_i]$. Since $E[X_i] = Pr(i \in \mathcal{B}(S)) = 1 - Pr(\forall x \in S, h(x) \neq i) = 1 - (1 - \frac{1}{s})^{\alpha}$, $E[|\mathcal{B}(S)|] = \sum_{i=1}^{s} E[X_i] = s(1 - (1 - \frac{1}{s})^{\alpha})$. □

By applying Azuma's Inequality [17], we have the following theorem.

Theorem 4.4: *Given a Bloom filter function, $\mathcal{B} : \mathcal{P}(X) \to \mathcal{P}(\mathbb{Z})$, where a hash function used randomly maps a number into a number in $\{1, 2, \cdots, s\}$. Let S be a subset of X, where the size of S is $\alpha = |S|$. Then we have*

$$\begin{aligned} \sum_{|i - E[|\mathcal{B}(S)|]| > \lambda} p(i) &= Pr(||\mathcal{B}(S)| - E[|\mathcal{B}(S)|]| > \lambda) \\ &\leq 2e^{-2\lambda^2/s} \end{aligned} \quad (4)$$

Theorem 4.4 suggests that the size of $\mathcal{B}(S)$ is concentrated around its expected value, $E[|\mathcal{B}(S)|]$, which is $s(1 - (1 - \frac{1}{s})^{\alpha})$. In other words, this says that the probability of the difference between $\mathcal{B}(S)$ and $E[|\mathcal{B}(S)|]$ to be greater than a value λ is bounded by a small number. With the bound by Theorem 4.4, we can give an approximation of false positive rate.

Theorem 4.5: *Given a Bloom filter function, $\mathcal{B} : \mathcal{P}(X) \to \mathcal{P}(\mathbb{Z})$, where a hash function used randomly maps a number into a number in $\{1, 2, \cdots, s\}$. Let S and T be two subsets of X, where $\alpha = |S|$ and $\beta = |T \setminus S|$. Then we have*

$$\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S)) \approx (1 - e^{-\alpha/s})^{\beta} \quad (5)$$

$$\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S)) \approx e^{-\frac{\beta}{e^{\frac{\alpha}{s}}}} \quad (6)$$

Proof: In Theorem 4.3, we have $\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S)) = \sum_{i=1}^{s} p(i) \cdot (\frac{i}{s})^{\beta}$. Since the size of $\mathcal{B}(S)$ is concentrated around its expected value, as shown in Theorem 4.4, we have $\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S)) \approx (\frac{E[|\mathcal{B}(S)|]}{s})^{\beta} = (1 - (1 - \frac{1}{s})^{\alpha})^{\beta}$. Since $1 - x \approx e^{-x}$ for $0 < x \ll 1$, we have $(1 - (1 - \frac{1}{s})^{\alpha})^{\beta} \approx (1 - e^{-\alpha/s})^{\beta}$ and $(1 - e^{-\alpha/s})^{\beta} \approx e^{-\frac{\beta}{e^{\frac{\alpha}{s}}}}$. □

With the approximation given by Theorem 4.5, $\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S))$ is approximated as $(1 - e^{-\alpha/s})^{\beta}$ by (Eq. (5)), and $\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S))$ can be further approximated as $e^{-\frac{\beta}{e^{\frac{\alpha}{s}}}}$ by (Eq. (6)). In order to control the probability of $\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S))$ to be less than or equal to a given p, we have the following based on Eq. (6).

$$e^{-\frac{\beta}{e^{\frac{\alpha}{s}}}} \leq p \quad (7)$$

By rewriting Eq. (7), we have

$$s \geq \frac{\alpha}{\ln\beta - \ln(\ln\frac{1}{p})} \quad (8)$$

Here, Eq. (8) suggests that the inequality needs to be held in order to have the probability less than a given p. Let $c = \ln\beta - \ln(\ln\frac{1}{p})$. Eq. (8) suggests that in order to control the probability to be less than p, α much be less than cs ($\alpha \leq cs$) where c is a small constant.

First, we discuss α and β regarding the R-ratio (Reachability-ratio) given in [26]. In [26], R-ratio is denoted as r, and is used to indicate the percentage of reachability queries that end up to be $u \rightsquigarrow v$. As shown in [26], the R-ratio is usually small. We show the R-ratio in Table 6 for the datasets tested. Given that

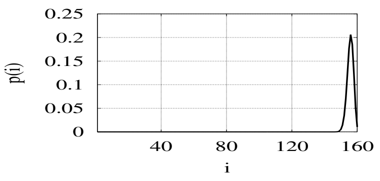


Fig. 2. Probability distribution of the size of $\mathcal{B}(S)$

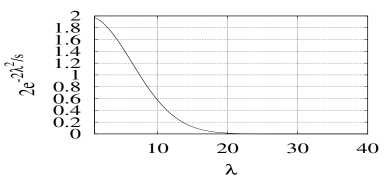


Fig. 3. The upper bound of $\sum_{|i - E[|\mathcal{B}(S)|]| < \lambda} p(i)$

a DAG G has a R-ratio r, $|Out(u)|$ and $|Out(v) \cap Out(u)|$ is about rn and r^2n respectively as explained in [26], where n is the number of vertices in G. In this work, since $\alpha = |Out(u)|$ and $\beta = |Out(v) - Out(u)|$, we have $\alpha = rn$ and $\beta = (r - r^2)n$. It is worth noting that we can assume $\alpha = \beta$, since $\alpha = rn$, $\beta = (r - r^2)n$, and r is very small.

As an example, we set $\alpha = 578$ and $\beta = 578$, based on the govwild dataset (Table 6), where $n = 8,022,880$ and $r = 7.20$E-5. Fig. 2 shows the probability distribution of the size of $\mathcal{B}(S)$ using Eq. (2) for $\alpha = 578$ and $s = 160$. Here, the value i in the x-axis is the size of a set S, and the expected size of $\mathcal{B}(S)$ is about 156 computed by Lemma 4.2. Following Lemma 4.2, Fig. 3 shows the probability of the difference between $\mathcal{B}(S)$ and $E[|\mathcal{B}(S)|]$ to be greater than a value λ is bounded by $2e^{-2\lambda^2/s}$.

Second, we can control the probability of $\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S))$ to be less than a given p regarding Eq. (8). For example, given $p = 1\%$, we let $\ln\beta \geq 2.53$ or $(r - r^2)n \geq 12.55$, when $s = \alpha = nr$ and $\beta = (r - r^2)n$. With $s = rn$, we can build an index using *BFL* in $O(sn) = O(rn^2)$ space complexity to achieve $p = 1\%$. In other words, we can support any small p that satisfies $c = \ln\beta - \ln(\ln\frac{1}{p}) \geq 1$ when $s = \alpha = rn$.

Third, we discuss the conditions that Eq. (8) is true in general. It is important to mention that Eq. (8) is true if $\alpha \leq s$ and $c \geq 1$. It is not difficult to ensure $c \geq 1$, since β is large. However, it may not be always true for $\alpha \leq s$. Fig. 4 shows that the exact and approximate false positive rates ($\Pr(\mathcal{B}(T) \subseteq \mathcal{B}(S))$) drop to nearly 0 from nearly 1 when s varies from 50 to 150, given $\alpha = \beta$. Here, in Fig. 4, Exact is computed by Eq. (3), and Appr.1 and Appr.2 are computed by Eq. (5) and Eq. (6), respectively. It suggests that the false positive rate will approach to zero when s is reasonable large. On the other hand, Fig. 5 shows the exact and approximate false positive rates change while varying α and β, given $s = 160$. As shown in Fig. 5, when α and β are greater than 600 ($> 3.75s$), the false positive rate becomes larger.

There is a need to deal with the case when $\alpha > s$, and there are two strategies. One is to make s large accordingly. However, to make $s \geq \alpha = nr$, the index size becomes unaffordable. The other is to reduce α to be handled in the *BFL* index. We take the second strategy. In the next section, we discuss vertices merging to be used together with *BFL*, to achieve a small false positive rate, by fixing s and using a small constant c to ensure $\alpha \leq cs$.

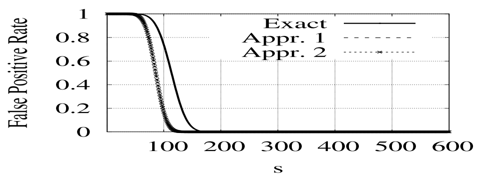


Fig. 4. The false positive rate by varying s

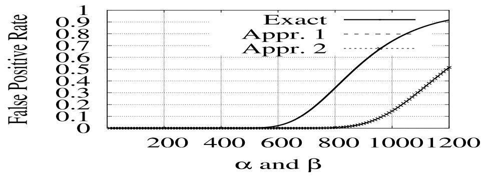


Fig. 5. They false positive rate by varying $\alpha = \beta$

4.2 Bloom Filter Label with Vertices Merging

In the previous section, in analyzing the approximation of false positive rate, we show that we have a small false positive rate when $\alpha \leq s$. In a similar way, we can show that we have a small false positive rate when $\alpha \leq cs$ for a small constant c, as shown in Eq. (8) for $c = \ln\beta - \ln(\ln\frac{1}{p})$. In this section, we discuss how to ensure $\alpha \leq cs$ by vertices merging, since in general α can be much greater than s. We define a merging function to merge vertices to a representative vertex below.

Definition 4.2: A function $g : V \rightarrow V$ can merge vertices, if $g(g(u)) = g(u)$ for all $u \in V$. We say that u is merged to $g(u)$ by $g(\cdot)$ and u is a representative if $g(u) = u$, and we call $\mathbb{V}$ a vertices merging function for $\mathbb{V}(A) = \{g(u) \mid u \in A\}$. □

For a vertex $u \in G$ and a given s, we redefine *BFL* of a vertex u, denoted by $\mathcal{L}_{bf}(u)$, as a pair of sets that are subsets of $\{1, 2, \ldots, s\}$, $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$. $\mathcal{L}_{out}(u) = \mathcal{B}(\mathbb{V}(Out(u)))$ and $\mathcal{L}_{in}(u) = \mathcal{B}(\mathbb{V}(In(u)))$. It can be proved in a similar manner as the way of proving Theorem 4.2 such that if $u \rightsquigarrow v$ then $\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u)))$. Thus, we can test whether u can reach v by testing whether $\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u)))$ instead of testing $\mathcal{B}(Out(v)) \subseteq \mathcal{B}(Out(u))$. Given the fact that if $u \rightsquigarrow v$ then $\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u)))$, the merging will not change the recall rate which is 100%. However, there are two cases that can result in false positive when answering negative reachability queries $u \not\rightsquigarrow v$ (i.e., $Out(v) \not\subseteq Out(u)$).

- The first case is the false positive introduced by the vertices merging such as $\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u))$, where vertices are merged into the same representative vertex.
- The second case is the false positive introduced by *BFL* given the vertices merging is correct, such as $\mathbb{V}(Out(v)) \not\subseteq \mathbb{V}(Out(u))$ but $\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u)))$.

Considering the two cases, the false positive rate of $Pr(\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u))))$ is the sum of the probability of the first case plus the probability of the second case.

- For the first case, it is $\Pr(\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u)))$.
- For the second case, it is $\Pr(\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u))) \wedge \mathbb{V}(Out(v)) \not\subseteq \mathbb{V}(Out(u)))$.

Since the probability of the second case is not greater than $\Pr(\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u))) \mid \mathbb{V}(Out(v)) \not\subseteq \mathbb{V}(Out(u)))$, the false positive rate is bounded as follows.

$$\begin{aligned} & Pr(\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u)))) \\ \leq\ & \Pr(\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u))) + \\ & \Pr(\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u))) \mid \\ & \mathbb{V}(Out(v)) \not\subseteq \mathbb{V}(Out(u))) \end{aligned} \tag{9}$$

Here, the term of $\Pr(\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u))) \mid \mathbb{V}(Out(v)) \not\subseteq \mathbb{V}(Out(u)))$ in Eq. (9) is the false positive rate of testing $\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u))$ using Bloom filter, which is $\approx (1 - e^{-\bar{\alpha}/s})^{\bar{\beta}}$, where $\bar{\alpha}$ and $\bar{\beta}$ are given below.

- $\bar{\alpha} = |\mathbb{V}(Out(u))|$, and
- $\bar{\beta} = |\mathbb{V}(Out(v)) - \mathbb{V}(Out(u))|$.

Note that $\bar{\alpha}$ and $\bar{\beta}$ are given in a similar way like α and β. The only difference is that α and β are for *BFL* before vertices merging, and $\bar{\alpha}$ and $\bar{\beta}$ are for *BFL* after vertices merging.

To keep the overall false positive small, we discuss how to make the term of $\Pr(\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u)))$ in Eq. (9) as small as possible while keeping $|\mathbb{V}(Out(u))| \leq cs$. And to make $\Pr(\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u)))$ small is equivalent to make $\Pr(\mathbb{V}(Out(v)) \setminus \mathbb{V}(Out(u)) \subseteq \mathbb{V}(Out(u)))$.

First, we consider an ideal case that vertices merging does not introduce any false positive. Let V_ν be a set of vertices that are merged to a single representative vertex ν. That is $g(v) = \nu$ for all $v \in V_\nu$. For a vertex $v \notin V_\nu$, we assume that such v is merged to itself only, $g(v) = v$. The set of V_ν is ideal if, for every w in V, the following (a) and (b) are true for every pair of vertices, u and v, in V_ν.

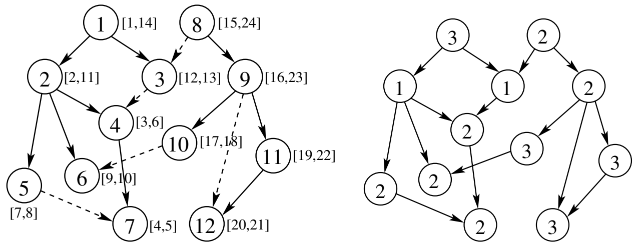
(a) if $u \in Out(w)$ then $v \in Out(w)$, and
(b) if $u \notin Out(w)$ then $v \notin Out(w)$.

In other words, either both of every pair of u and v in V_ν can be reached by w, or none of them can be reached by w, for all w in G. It is worth noting that w is a vertex in V such that w can be a vertex either in V_ν or out V_ν. By the ideal V_ν, we have $\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u))$ if and only if $Out(v) \subseteq Out(u)$. This implies that $\Pr(\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u))) = 0$ when $Out(v) \not\subseteq Out(u)$. However, there does not exist such V_ν when $|V_\nu| \geq 2$, since the graph G we are dealing with is a DAG, and $u \in Out(v)$ and $v \in Out(u)$ cannot co-exist at the same time in a DAG for $u \neq v$.

In this work, instead finding the ideal V_ν, we find an almost ideal set, V_ν, with probability guarantee. The set of V_ν is almost ideal if, for an arbitrary vertex w randomly selected from V, $\Pr(V_\nu \cap Out(w) = V_\nu \text{ or } V_\nu \cap Out(w) = \emptyset)$ is greater than a certain ρ. In general, in order to merge all vertices in V into d representative vertices, we partition V into disjoint sets, $V_1, V_2, \ldots, V_d$, such that $\Pr(V_i \cap Out(w) = V_i \text{ or } V_i \cap Out(w) = \emptyset) > \rho$, we get a vertices merging by merging all u in V_i to an arbitrary vertex as the representative in V_i. The false positive rate introduced by the almost ideal set is $1 - \rho$. Because $\Pr(\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u))) < (1 - \rho)^{|\mathbb{V}(Out(v) - Out(u))|} \leq 1 - \rho$, the probability of $\Pr(\mathbb{V}(Out(v)) \subseteq \mathbb{V}(Out(u)))$ will be small, given that $Out(v) \not\subseteq Out(u)$ and ρ is large. Note that ρ is bounded by $1 - |V_\nu|/|V|$, since w randomly selected to satisfy either $V_\nu \cap Out(w) = V_\nu$ or $V_\nu \cap Out(w) = \emptyset$ cannot be in V_ν.

Algorithm 1: $MergeVertices(G)$

```
1 let $v_i$ be assigned to a post-order by traversal of $G$
2 divide the interval $[1, |V|]$ into disjoint intervals $I_1, I_2, \ldots, I_d$
3 forall $I \in \{I_1, I_2, \ldots, I_d\}$ do
4     $r \leftarrow min\{I\}$
5     forall $i \in I$ do
6         $g(v_i) \leftarrow v_r$
7 return $g$
```



(a) a *DFS* over G (b) $h(g(u))$ for each vertices

Fig. 6. Vertices merging

Overall, the probability of $Pr(\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u))))$ (Eq. (9)) is bounded below.

$$Pr(\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u)))) \leq (1-\rho)+(1-e^{-\bar{\alpha}/s})^{\bar{\beta}} \tag{10}$$

We propose a simple but effective way to find a partition where every vertices merging is almost ideal. Here, for a graph G with $n = |V|$ vertices, we assign v_i to a post-order by *DFS* (Algorithm 3), and uniformly divide the interval $[1, n]$ for all vertices into d disjoint intervals, $I_1, I_2, \ldots, I_d$, where d is a user-given constant. Let $V_i = \{v_j \mid j \in I_j\}$. The algorithm to generate such a function $g(\cdot)$ and $\mathbb{V}$ is given in Algorithm 1, which is self-explained.

We observe that, in all real datasets and synthetic datasets tested, when $d = 1,600$, $\Pr(V_i \cap Out(w) = V_i \text{ or } V_i \cap Out(w) = \emptyset) > 99.5\%$ in most cases. And, if we set $s = 160$ and $d = 1,600$, then the false positive rate is bounded by $0.5\% + \Pr(\mathcal{B}(\mathbb{V}(Out(v))) \subseteq \mathcal{B}(\mathbb{V}(Out(u))) \mid \mathbb{V}(Out(v)) \not\subseteq \mathbb{V}(Out(u)))$. This suggests that the false positive rate can be very small if we choose a proper d. Recall that we also need to make $|\mathbb{V}(Out(u))| \leq cs$ for a small constant c. On one hand, since $|\mathbb{V}(Out(u))| \leq d$, a smaller d is better. On the other hand, d cannot be too small. If d is too small, since $\bar{\beta} = |\mathbb{V}(Out(v)) - \mathbb{V}(Out(u))| \leq d$, a small d makes $\bar{\beta}$ small makes false positive rate bounded by $(1 - e^{-\bar{\alpha}/s})^{\bar{\beta}}$ large. In addition, a small d will makes ρ small. By default, we set $d = 10s$, as also used in our experimental studies.

Example 4.2: A DAG G is shown in Fig. 1. The numbers in Fig. 1 identify the 12 vertices in $V = \{v_1, v_2, \ldots, v_{12}\}$. Let $s = 3$. Given a hash function $h(\cdot)$, assume $h(v_1) = 3$, $h(v_2) = 1$, $h(v_3) = 1$, $h(v_4) = 1$, $h(v_5) = 2$, $h(v_6) = 1$, $h(v_7) = 2$, $h(v_8) = 1$, $h(v_9) = 2$, $h(v_{10}) = 3$, $h(v_{11}) = 2$ and $h(v_{12}) = 3$. Fig. 6(a) shows a *DFS* over G, the pairs next to a vertex is their discovery time and finish time during the *DFS*, following the post-order traversal sequence of $(v_7, v_4, v_5, v_6, v_2, v_3, v_1, v_{10}, v_{12}, v_{11}, v_9, v_8)$. Let $d = 6$, then $g(\cdot)$ is obtained by Algorithm 1 as $g(v_7) = v_7$, $g(v_4) = v_7$, $g(v_5) = v_5$, $g(v_6) = v_5$, $g(v_2) = v_2$, $g(v_3) = v_2$, $g(v_1) = v_1$, $g(v_{10}) = v_1$, $g(v_{12}) = v_{12}$, $g(v_{11}) = v_{12}$, $g(v_9) = v_9$ and $g(v_8) = v_9$. Fig. 6(b) shows $h(g(v_i))$ for every $v_i \in V$. Table 3 shows $\mathcal{L}_{out}$ for G in Fig. 1(a).

Vertex	$\mathcal{L}_{out}(u)$	label(u)
v_1	$\{1,2,3\}$	$\{1\}$
v_2	$\{1,2\}$	$\{1\}$
v_3	$\{1,2\}$	$\{3\}$
v_4	$\{2\}$	$\{3\}$
v_5	$\{2\}$	$\{1\}$
v_6	$\{2\}$	$\{12\}$
v_7	$\{2\}$	$\{8\}$
v_8	$\{1,2,3\}$	$\{3\}$
v_9	$\{2,3\}$	$\{5\}$
v_{10}	$\{2,3\}$	$\{7\}$
v_{11}	$\{3\}$	$\{6\}$
v_{12}	$\{3\}$	$\{6\}$

TABLE 3
BFL and *IP* label of vertices in Fig. 1(a)

4.3 TC Zero-Cover

In this section, we discuss the effectiveness of *BFL* in comparison with the existing Label+G approaches, based on a notion of TC zero-cover. The notion of TC zero-cover is based on the fact that Label+G approaches are designed to effectively answer negative reachability queries such that a vertex v is not reachable by a vertex u.

Consider a matrix M representing the transitive closure (TC) for a graph G, where the value for $M[u, v]$ is 1 if $u \rightsquigarrow v$, otherwise 0. Given the small R-ratio, all Label+G approaches attempt to identify as many unreachable vertex pairs as possible, and make use of such pairs to answer a reachability query directly or prune during a *DFS*. The more unreachable vertex pairs an approach can identify, the higher efficiency the approach can achieve. In other words, by the matrix reorientation, the larger number of zeros in the matrix M an approach can produce, the higher efficiency the approach can achieve. All the state-of-art Label+G approaches can be considered in a way to cover as many zeros as possible in M. From a different viewpoint, all such Label+G labeling are an order-based labeling, by taking the TC zero cover into consideration. We explain it below. Intuitively, the order is the order to place vertices in the matrix M. That is, a node u that covers more zeros is placed in a position in M before a node v to be placed that covers less number of zeros.

Let label(u) be a labeling based on an order such that label(u) < label(v) if u covers more zeros than v does. We compare *BFL* with *IP*, since *IP* outperforms all other Label+G approaches. For simplicity, assume $k = 1$ in *IP* where k is the number of vertices in a label, and $s = 1$ in *BFL*. With *IP*, if label(u) > label(v), then $u \not\rightsquigarrow v$, that is to say, for vertex pairs of (u, v), *IP* can only cover a zero representing $u \not\rightsquigarrow v$, or a zero representing $v \not\rightsquigarrow u$, or none of them. Thus, *IP* covers at most $n(n-1)/2$ zeros in TC. With *BFL*, if $i \notin \mathcal{L}_{out}(u)$ and $i \in \mathcal{L}_{out}(v)$, then $u \not\rightsquigarrow v$. Thus, for all i, it covers $|\{u \mid i \in \mathcal{L}_{out}(u)\}| \times |\{u \mid i \notin \mathcal{L}_{out}(u)\}|$ zeros in TC. In other words, it covers at most $\lfloor n^2/4 \rfloor$ zeros in TC. It is worth mentioning that a single permutation number (integer) in *IP* only covers twice number of unreachable vertex pairs by a single bit in *BFL*. And an integer can be 32-bits or up to 64-bits long. When both k and s become larger, the zeros covered are not multiplied. In other words, when $k = 1$, it covers $n(n-1)/2$ zeros; but it does not necessarily mean that, when $k = 2$, it covers $2 \cdot n(n-1)/2$

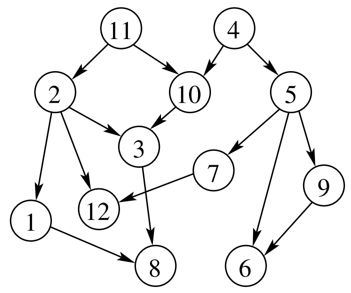


Fig. 7. A permutation π

(a) *BFL*

1	1	1	1	1	1	1	0	0	0	0	0
0	1	0	1	1	1	1	**0**	**0**	**0**	**0**	**0**
0	0	1	1	0	0	1	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	1	0	0	1	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	0	1	0	1	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	0	0	1	0	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	0	0	0	1	**0**	**0**	**0**	**0**	**0**
0	0	1	1	0	1	1	1	1	1	1	1
0	**0**	**0**	0	0	1	0	**0**	1	1	1	1
0	**0**	**0**	0	0	1	0	**0**	0	1	0	0
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	1
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	0	1

(b) *IP*

1	1	1	1	1	1	1	0	0	0	0	0
0	1	0	1	1	1	1	0	0	0	0	0
0	**0**	1	1	**0**	0	1	0	0	0	0	0
0	**0**	0	1	**0**	0	1	0	0	0	0	0
0	0	0	0	1	0	1	0	0	0	0	0
0	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	0	1	**0**	**0**	**0**	**0**	**0**
0	**0**	1	1	**0**	1	1	1	1	1	1	1
0	**0**	**0**	**0**	**0**	1	0	**0**	1	1	1	1
0	**0**	**0**	**0**	**0**	1	0	**0**	**0**	1	**0**	**0**
0	**0**	**0**	**0**	**0**	0	0	**0**	**0**	0	1	1
0	**0**	**0**	**0**	**0**	0	0	**0**	**0**	0	0	1

TABLE 4
Zero-Cover for the graph in Fig. 1(a)

zeros since there might be zeros covered multiple times. It is the same to s.

Example 4.3: Fig. 7 shows the permutation used to construct *IP* label for G in Fig. 1(a). The *BFL* and *IP* labels for G are shown in Table 3 with $s = 3, d = 6$ for *BFL* and $k = 1$ for *IP*. With *IP*, if $\mathsf{label}(u) > \mathsf{label}(v)$, then $u \not\rightsquigarrow v$. With *BFL*, if $i \notin \mathcal{L}_{out}(u)$ and $i \in \mathcal{L}_{out}(v)$, then $u \not\rightsquigarrow v$. We show the zeros covered by *BFL* and *IP* in Table 4 in bold. *IP* needs $12\log_2(12) \approx 43$ bits to store its index whereas *BFL* needs $12 \cdot 3 = 36$ bits. *IP* covers 59 zeros whereas *BFL* covers 72 zeros.

To confirm the effectiveness of *BFL*, we conduct testing over DAG datasets. The generator we used is based on the configuration model [18], which generates an undirected graph using two parameters, n and ρ, where n is the number of vertices and ρ is used to control the power-law distribution. We fix $n = 1,000$, and vary the parameter ρ. Given an undirected graph generated by random power-law graph generator, we convert it to a DAG by setting the direction of an edge (u, v) from u to v, if the identifier of v is larger than that of u. The values of the parameter ρ selected are 1.6, 1.7, 1.8, 1.9, 2.1, and 2.8, that correspond to outdegrees of the DAG 6 ($\approx$ 5.885), 5 ($\approx$ 4.554), 4 ($\approx$ 4.031), 3 ($\approx$ 2.891), 2 ($\approx$ 2.195), and 1 ($\approx$ 1.039), respectively. The outdegree distributions for the graphs generated using the selected

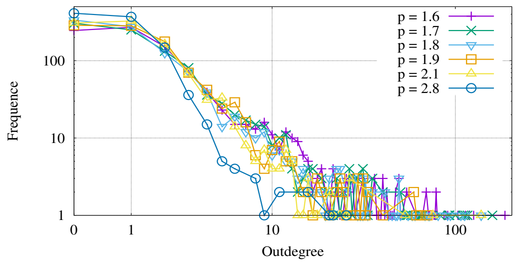


Fig. 8. The Outdegree Distribution

ρ are given in Fig. 8. We compare *BFL* with *IP*, where *IP* labels are generated using $k = 1, 2, 3, 4, 5$, and *BFL* labels are generated using $s = 32, 64, 96, 128$, and 160 accordingly. Note that k is the number of integers used in *IP*, whereas s is the number of bits used in *BFL* for $s = 32k$. The results of TC zero covers are shown in Table 5. In Table 5, the first column is for ρ values, the second column is the average outdegree of the DAG obtained, the last column is the total number of zeros in TC. The other columns show the percentage of the total number of TC zeros covered. As shown in Table 5, *BFL* outperforms *IP* in all cases. There two observations. First, *IP* and *BFL* cover more zeros when a graph becomes sparser. Second, for the same graph obtained, *IP* and *BFL* cover more zeros when k (or s) becomes larger.

5 Computing *BFL*

We discuss how to compute *BFL*. It is costly to compute $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$ directly based on Definition 4.1 and Definition 4.2, since it is costly to compute $Out(u)$ and $In(u)$ for every vertex u in G. We propose an efficient way to compute *BFL*, which is based on the property of $\mathcal{B}$ and $\mathbb{V}$, as given in the following theorem.

Theorem 5.6: $\mathcal{B}(\mathbb{V}(S \cup T)) = \mathcal{B}(\mathbb{V}(S)) \cup \mathcal{B}(\mathbb{V}(T))$.

Proof: The statement holds if $\mathcal{B}(S \cup T) = \mathcal{B}(S) \cup \mathcal{B}(T)$ and $\mathbb{V}(S \cup T) = \mathbb{V}(S) \cup \mathbb{V}(T)$. Let $y \in \mathcal{B}(S) \cup \mathcal{B}(T)$. There must exist a vertex u such that $y = h(u)$ and either $u \in S$ or $u \in T$, i.e, $u \in S \cup T$. Thus, we have $y = h(u) \in \mathcal{B}(S \cup T)$, and $\mathcal{B}(S \cup T) \supseteq \mathcal{B}(S) \cup \mathcal{B}(T)$. We can also prove that $\mathcal{B}(S \cup T) \subseteq \mathcal{B}(S) \cup \mathcal{B}(T)$ in a similar way. Thus, $\mathcal{B}(S \cup T) = \mathcal{B}(S) \cup \mathcal{B}(T)$. In a similar manner, we can prove that $\mathbb{V}(S \cup T) = \mathbb{V}(S) \cup \mathbb{V}(T)$. Thus, $\mathcal{B}(\mathbb{V}(S \cup T)) = \mathcal{B}(\mathbb{V}(S)) \cup \mathcal{B}(\mathbb{V}(T))$. □

Consider a vertex u, its $\mathcal{L}_{out}(u)$ can be computed by all $\mathcal{L}_{out}(w)$ in its direct successors ($w \in Suc(u)$), because $Out(u) = \{u\} \cup \bigcup_{w \in Suc(u)} Out(w)$. And, by Theorem 5.6, we have

$$\begin{aligned}\mathcal{L}_{out}(u) &= \mathcal{B}(\mathbb{V}(\{u\} \cup \bigcup_{w \in Suc(u)} Out(w))) \\ &= \mathcal{B}(\mathbb{V}(\{u\})) \cup \bigcup_{w \in Suc(u)} \mathcal{L}_{out}(w)\end{aligned}$$

Here, $\mathcal{L}_{out}(u)$ consists of $\bigcup_{w \in Suc(u)} \mathcal{L}_{out}(w)$ and $h(g(u))$. In a similar way, its $\mathcal{L}_{in}(u)$ can be computed by all $\mathcal{L}_{in}(w)$ in its direct predecessor ($w \in Pre(u)$), for the reason that $In(u) = \{u\} \cup \bigcup_{w \in Pre(u)} In(w)$. Also, by Theorem 5.6, we have

$$\begin{aligned}\mathcal{L}_{in}(u) &= \mathcal{B}(\mathbb{V}(\{u\} \cup \bigcup_{w \in Pre(u)} In(w))) \\ &= \mathcal{B}(\mathbb{V}(\{u\})) \cup \bigcup_{w \in Pre(u)} \mathcal{L}_{in}(w)\end{aligned}$$

Here, $\mathcal{L}_{in}(u)$ consists of $\bigcup_{w \in Pre(u)} \mathcal{L}_{in}(w)$ and $h(g(u))$.

Based on the ideas presented, we design an algorithm to compute *BFL* for vertices in G. We show the algorithm to compute $\mathcal{L}_{out}$ in Algorithm 2. $\mathcal{L}_{in}$ can be computed in a similar way. As shown in Algorithm 2, first, we generate function $g(\cdot)$ by Algorithm 1. Next, we enumerate all vertices u in V (line 2) and check if $\mathcal{L}_{out}(u)$ has been computed (line 3). If not, we call $Compute$ to compute $\mathcal{L}_{out}(u)$ (line 4). In the procedure $Compute$, we initialize $\mathcal{L}_{out}(u)$ as $\{h(g(u))\}$ (line 6). Then, we enumerate all direct successors w of u (line 7), and check if $\mathcal{L}_{out}(w)$ has been computed (line 8). If not, we call $Compute$

ρ	d_{avg}	$k=1$ $(s=32)$		$k=2$ $(s=64)$		$k=3$ $(s=96)$		$k=4$ $(s=128)$		$k=5$ $(s=160)$		**# of Zeros**
		IP	*BFL*	*IP*	*BFL*	*IP*	*BFL*	*IP*	*BFL*	*IP*	*BFL*	
1.6	5.89	65.01%	83.66%	75.81%	91.32%	77.99%	93.47%	79.02%	95.39%	81.14%	96.31%	840,456
1.7	4.55	71.54%	86.27%	76.44%	92.20%	74.83%	93.94%	79.39%	95.57%	81.83%	96.59%	855,030
1.8	4.03	68.22%	85.58%	76.85%	91.84%	77.84%	94.01%	79.86%	95.38%	80.70%	96.35%	854,352
1.9	2.89	72.48%	90.01%	76.59%	94.58%	80.03%	96.43%	81.45%	97.50%	81.96%	97.93%	900,218
2.1	2.20	70.81%	90.14%	76.97%	94.76%	75.42%	96.88%	79.26%	97.67%	80.62%	98.37%	920,335
2.8	1.04	77.44%	99.34%	90.80%	99.79%	94.33%	99.89%	96.02%	99.94%	96.07%	99.97%	994,046

TABLE 5
Zero-Covers for a graph with $n = 1,000$ vertices

Algorithm 2: $BuildBFLIndex(G, h)$

```
1  $g \leftarrow MergeVertices(G)$
2  forall $u \in V$ do
3      if $\mathcal{L}_{out}(u)$ has not been computed then
4          $Compute(G, g, h, u)$
5  Procedure $Compute(G, g, h, u)$:
6      $\mathcal{L}_{out}(u) \leftarrow \{h(g(u))\}$
7      forall $w \in Suc(u)$ do
8          if $\mathcal{L}_{out}(w)$ has not been computed then
9              $Compute(G, h, w)$
10         $\mathcal{L}_{out}(u) \leftarrow \mathcal{L}_{out}(u) \cup \mathcal{L}_{out}(w)$
```

Algorithm 3: $DFS(G)$

```
1  $current := 0$
2  forall $u \in V$ do
3      if $deg^{-}(u) = 0$ then
4          DFSVisit($u$)
5  Procedure $DFSVisit(u)$:
6      $current \leftarrow current + 1$
7      $\mathcal{L}_{dis}(u) \leftarrow current$ // discovery time
8      forall $w \in Suc(u)$ do
9          if $w$ has not been visited then
10             DFSVisit($w$)
11     $o \leftarrow o \cup \{u\}$ // post-order traversal
           sequence
12     $current \leftarrow current + 1$
13     $\mathcal{L}_{fin}(u) \leftarrow current$ // finish time
```

recursively to compute $\mathcal{L}_{out}(w)$ (line 9). Then, we set $\mathcal{L}_{out}(v)$ to $\mathcal{L}_{out}(v) \cup \mathcal{L}_{out}(w)$ (line 10). Here, we use a bit vector of $\lceil s/8 \rceil$ bytes to represent a subset of $\{1, 2, \ldots, s\}$, so every union operation can be efficiently done by doing bitwise-or on two bit vectors.

Construction time and Index size: To compute *BFL* by Algorithm 2, we need to generate $g(\cdot)$ in $O(n+m)$ time. Note that $\mathcal{L}_{out}(u)$ on every vertex u will be computed only once. To compute $\mathcal{L}_{out}(u)$ we need to initialize $\mathcal{L}_{out}(u)$ in $O(s)$ time and do union operation on sets for $deg^{+}(u)$ times. Each of union operations takes $O(s)$. Thus, the time used to compute $\mathcal{L}_{out}$ is $\sum_{u \in V}(1 + deg^{+}(u)) \cdot O(s) = O(s(n+m))$, for $deg^{+}(u) = |Suc(u)|$. Therefore, the overall time complexity is $O(s(n+m))$. In a similar way, $\mathcal{L}_{in}$ is computed in $O(s(n+m))$. With the bitwise operations, the constant factor is very small, which makes our approach faster than other Label+G approaches even if the time complexity is same. The *BFL* index size is $2n \cdot \lceil s/8 \rceil$ bytes, since every $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u)$ use $\lceil s/8 \rceil$ bytes and there are $2n$ of them. It is worth noting that the existing Label+G approaches use parameter k, to control the number of labels per vertex, as 32-bit or 64-bit integers, whereas *BFL* uses parameter s to denote the maximum size of *BFL* per vertex. Since *BFL* can be expressed as a bit vector, we can use k integers to represent a *BFL* of size $s = 32k$. By setting the parameters in this way, we use the similar index size with the existing Label+G approaches.

6 Answer Reachability Queries

BFL is efficient for reachability queries that end up $u \not\rightsquigarrow v$, which we call negative reachability queries. To handle queries that end up $u \rightsquigarrow v$, which we call positive reachability queries, the interval label is known to be effective [27], which helps to early terminate in *DFS* for answering Reach(u, v). We adopt the interval label in our work. In brief, an interval label for a vertex u is denoted as $\mathcal{L}_{int}(u)$, which is a pair of numbers representing an interval, $[\mathcal{L}_{dis}(u), \mathcal{L}_{fin}(u)]$, where $\mathcal{L}_{dis}(u)$ and $\mathcal{L}_{fin}(u)$ denote the discovery time and finishing time of u in a *DFS* of G. Algorithm 3 shows how to obtain interval labels. It takes $O(n+m)$ to run a *DFS* on G, and needs $O(n)$ space to save $\mathcal{L}_{int}$.

Algorithm 4 shows how to process a reachability query Reach(u, v). First, it checks the interval label, and answers $u \rightsquigarrow v$ if $\mathcal{L}_{dis}(u) \leq \mathcal{L}_{dis}(v)$ and $\mathcal{L}_{fin}(v) \leq \mathcal{L}_{fin}(u)$ (line 1-2). Second, it checks *BFL*, and answers $u \not\rightsquigarrow v$ if either $\mathcal{L}_{out}(v) \nsubseteq \mathcal{L}_{out}(u)$ is true or $\mathcal{L}_{in}(u) \nsubseteq \mathcal{L}_{in}(v)$ is true (line 3-4). We know that $\mathcal{L}_{out}(v) \subseteq \mathcal{L}_{out}(u)$ if only if $\mathcal{L}_{out}(v) \cap \mathcal{L}_{out}(u) = \mathcal{L}_{out}(v)$. Since we implement $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{out}(v)$ as bit vectors, the intersection can be computed by bitwise-and operations efficiently. Therefore, checking $\mathcal{L}_{out}(v) \subseteq \mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(u) \subseteq \mathcal{L}_{in}(v)$ can be done efficiently. Third, it recursively calls $Reach$ to check if it can reach v via a non-visited successor w.

Query Time: The time complexity of Algorithm 4 to answer Reach(u, v) is $O(s)$ when either *BFL* or the interval label over u and v can be used to answer reachability queries directly. For the cases that involve *DFS*, in the worst case, it will cost $O(sn+m)$ to finish the *DFS*. However, in most cases, as confirmed in our extensive experimental studies, it is far less than such complexity. This is because *BFL* can be effectively used to stop early in *DFS* with high probability. Therefore, a few vertices are visited in the *DFS*.

7 Experimental Studies

We report our experimental results in this section. We verify the efficiency of our algorithm in 19 large real graphs used in [26]. They have millions of nodes and edges but with a small

Algorithm 4: $Reach(G, \mathcal{L}, u, v)$

```
1 if $\mathcal{L}_{dis}(u) \le \mathcal{L}_{dis}(v)$ and $\mathcal{L}_{fin}(v) \le \mathcal{L}_{fin}(u)$ then
2     return $true$
3 else if $\mathcal{L}_{out}(v) \not\subseteq \mathcal{L}_{out}(u)$ or $\mathcal{L}_{in}(u) \not\subseteq \mathcal{L}_{in}(v)$ then
4     return $false$
5 else
6     forall $w \in Suc(u)$ do
7         if w has not been visited and $Reach(G, \mathcal{L}, w, v)$ then
8             return $true$
9     return $false$
```

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	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
mapped100K	2,658,702	2,660,628	1.000	1.56E-6
mapped1M	9,387,448	9,440,404	1.005	7.00E-7
twitter	18,121,168	18,359,487	1.013	7.39E-2
uniprot22m	1,595,444	1,595,442	0.999	1.45E-6
uniprot100m	16,087,295	16,087,293	0.999	1.60E-7
uniprot150m	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
web-uk	22,753,644	38,184,039	1.678	1.50E-1
yago	16,375,503	25,908,132	1.582	1.00E-6
citeseerx	6,540,399	15,011,259	2.295	4.07E-4
dbpedia	3,365,623	7,989,191	2.374	2.47E-2
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
HostLink	12,754,590	26,669,293	2.091	4.48E-2
patent	3,774,768	16,518,947	4.376	2.36E-3

TABLE 6
Real Datasets

reachability ratio. Table 6 gives the information of real datasets we have used including the number of vertices ($|V|$), the number of edges ($|E|$), the average degree (d_{avg}), and the R-ratio (r). As given in [26], we classify the graphs with average degree smaller than 2 as sparse graphs and the ones with average degree larger than or equal to 2 as dense graphs. Detailed description of the datasets can be found in [26].

Below, we denote *BFL* with interval labels as *BFL+*. *BFL+* algorithm is compared with the state-of-the-art reachability algorithms including *GRAIL* [28], *ScaGRAIL* [27], [12], *PWAH8* [23], *TF-Label* [7], *HL* [14], *DL* [14], *Ferrari* [20], *Feline* [24], *IP+* [26] and *TOL* [31]. The source codes of these existing algorithm are kindly provided by the authors. All algorithms, including *BFL+*, is implemented in C/C++ and compiled by GCC 4.8.3. All experiments are performed on machine with 3.60GHz Intel Core i7-4790 CPU, 32GB RAM and running Linux OS.

Most Label+G approaches have parameters to control their index size. As shown in [26], they get better performance when setting the parameters small for sparse graphs and larger for dense graphs. We follow the same parameter settings with [26]. That is, we set $k = 2$ for *GRAIL* and *Ferrari* in sparse graphs and $k = 5$ in dense graphs. And we compute $s = 32$ seeds for *Ferrari* as an additional index for seed based pruning, as suggested by the original paper. For *IP+*, we set $k = 2$, $h = 2$ for sparse graphs and $k = 5$, $h = 5$ for dense graphs. For fairness, we set $s = 32k$ for *BFL+*, that is $s = 64$ in sparse graphs and set $s = 160$ in dense graphs. We set $d = 10s$ in both sparse and dense graphs.

7.1 Performance on Large Real Graphs

We report the index construction time, index size, and query time for real datasets in Table 7, Table 8 and Table 9. The best results are highlighted in **bold** font. We terminate the programs that exceeds the memory limit (32GB) of the machine we use and mark them with "—" in the tables. We mainly focus on the comparisons between our *BFL+* algorithm and the state-of-art *IP+*. For the analysis of other reachability algorithms, it is similar as discussed in [26] and we omit it for the limited space.

Table 7 reports the index construction time. We can see that *BFL+* is the fastest algorithm in all datasets. *Feline* is the second fastest algorithm and its time consumption is about 3 times longer than *BFL+* in the sparse graphs and 2 times longer than *BFL+* in dense graphs. For the largest dataset web-uk with 22 million nodes and 38 million edges, *BFL+* can finish its indexing using about 1 second while the time consumption of *Feline* is 4.6 times longer. Although both *BFL+* and *IP+* perform twice graph traversals, *BFL+* uses efficient bitwise operations in indexing which makes it about 4 times faster than *IP+*.

Table 8 shows the index size. *2-Hop* approaches build up relatively small index in sparse graphs because there are a few reachable pairs that need to be covered. In dense graphs, *Feline* has the smallest index size because it only has two topological orders and one level filter label with a total index size of $3n$. Its small index size cannot provide good query performance as shown in Table 9. The index size of *BFL+* is comparable to *IP+*'s index size.

We randomly generate 1 million queries for each dataset such that all possible vertex pairs will be chosen with the equal probability. Table 9 shows the total query time taken to answer all the reachability queries generated. *BFL+* performs the best in 16 out of all 19 datasets, except citeseer, email, and patent. Its performance is also comparable to the best one for citeseer and email. In patent, *BFL+* needs to perform more *DFS* for answering the query but it is still the best among the Label+G approaches. The *BFL+* query time is less than a half of the *IP+* query time in most datasets because *BFL+* has higher probability than *IP+* to answer the query directly. To verify our approach, we show the estimated pruning power and the actual pruning power computed in Table 10. Here, the pruning power is one minus the false positive rate. The estimated pruning power is computed by $(1-\rho) + (1 - e^{-\frac{\bar{\alpha}}{s}})^{\bar{\beta}}$ where the average $\bar{\alpha}$ and $\bar{\beta}$ are computed based on the 1 million queries, and $\rho = 0.5\%$ as discussed. The actual pruning power is computed based on the 1 million queries generated.

7.2 Scalability Study on Synthetic Graphs

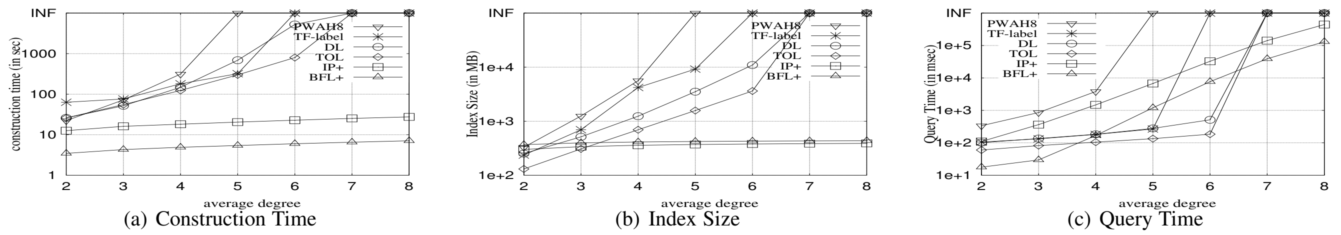
To study the influence of graph density, we generate large synthetic DAGs with 10 million nodes with average degree from 2 to 8 using the same graph generation algorithm used in [27], [26]. Fig. 9 shows the experimental results of different approaches on the synthetic graphs. For simplicity, we only compare *BFL+* with *PWAH8*, *TF-Label*, *DL*, *IP+*, and *TOL*, because they represent the state-of-the-art transitive closure compression, *2-Hop*, and Label+G approaches, respectively. Some results of Label-Only approaches in dense graphs are marked as "INF" in the figures

Dataset	*GRAIL*	*ScaGRAIL*	*PWAH8*	*TF-Label*	*HL*	*DL*	*TOL*	*Ferrari*	*Feline*	*IP+*	*BFL+*
citeseer	0.413	0.823	0.330	0.437	0.645	0.383	0.29	0.338	0.184	0.20	**0.051**
email	0.084	0.381	0.132	0.064	0.144	0.078	0.05	0.098	0.044	0.04	**0.007**
LJ	0.380	1.960	0.588	0.333	0.622	0.350	0.26	0.455	0.199	0.26	**0.038**
mapped100K	1.112	2.328	0.279	1.161	2.324	1.179	4.48	1.505	0.532	0.82	**0.116**
mapped1M	4.032	8.882	1.475	7.478	9.626	4.310	6.07	5.562	1.933	2.61	**0.435**
twitter	7.196	707.943	5.782	5.897	13.241	8.008	3.85	9.394	4.117	5.06	**0.764**
uniprot22m	0.595	2.659	0.872	1.067	1.140	0.635	0.25	0.481	0.347	0.40	**0.061**
uniprot100m	7.472	30.662	10.061	18.395	14.207	9.398	4.47	7.321	4.844	5.24	**0.857**
uniprot150m	12.083	51.374	16.772	27.741	25.012	14.256	8.62	13.587	8.341	8.56	**1.538**
web	0.229	0.682	0.409	0.265	0.399	0.278	0.24	0.265	0.115	0.14	**0.031**
wiki	0.689	1.984	0.231	0.476	1.305	0.868	0.46	0.947	0.443	0.52	**0.068**
web-uk	9.014	—	116.437	650.641	552.714	13.623	17.52	12.848	5.617	7.12	**0.958**
yago	13.704	28.431	6.649	7.400	25.098	14.209	12.24	20.550	7.929	7.53	**1.815**
citeseerx	16.050	25.578	12.118	46.859	72.733	7.608	13.50	11.689	2.905	3.81	**1.022**
dbpedia	9.775	18.989	4.277	6.307	5.827	3.141	3.63	3.715	1.495	2.01	**0.513**
go-uniprot	25.008	28.865	20.935	29.808	12.541	12.429	7.71	19.729	2.832	5.82	**0.955**
govwild	18.855	22.053	15.439	63.548	14.048	9.704	10.50	10.909	3.300	4.43	**0.980**
HostLink	36.476	64.017	19.863	142.057	16.862	9.338	18.01	12.126	4.592	6.00	**1.464**
patent	17.654	25.858	633.426	128.329	—	92.373	51.46	22.626	3.321	4.75	**1.375**

TABLE 7
Index Construction Time on Real Datasets (in second)

Dataset	*GRAIL*	*ScaGRAIL*	*PWAH8*	*TF-Label*	*HL*	*DL*	*TOL*	*Ferrari*	*Feline*	*IP+*	*BFL+*
citeseer	18.53	11.94	6.57	1.79	7.72	7.15	**1.39**	13.61	7.94	9.44	9.24
email	6.17	4.36	2.35	0.85	2.70	2.59	**0.83**	7.55	2.64	3.66	3.75
LJ	25.94	18.79	9.65	3.96	11.68	11.23	**3.81**	25.51	11.11	19.11	15.89
mapped100K	71.00	51.11	10.72	17.19	39.28	34.43	**9.06**	51.05	30.42	53.90	39.07
mapped1M	250.67	180.65	**38.08**	73.94	137.35	116.09	42.53	181.17	107.43	171.25	136.03
twitter	483.89	348.83	95.57	70.11	211.14	202.48	**64.22**	375.36	207.38	384.89	284.43
uniprot22m	42.60	30.67	18.64	6.30	18.62	18.47	**6.09**	45.65	18.25	24.49	26.13
uniprot100m	429.58	310.67	208.63	76.72	204.75	197.18	**62.29**	460.26	184.10	251.15	270.75
uniprot150m	668.58	486.19	349.25	131.92	337.00	318.51	**98.94**	716.33	286.53	394.97	428.54
web	9.93	7.62	4.33	2.56	5.90	4.85	**1.94**	11.11	4.25	6.95	6.48
wiki	60.93	43.94	8.99	8.86	26.33	26.21	**8.79**	44.03	26.11	51.99	37.06
web-uk	607.59	—	260.10	2719.85	4520.44	356.02	**197.17**	631.90	260.39	442.89	378.09
yago	437.27	315.02	134.65	**98.83**	290.39	223.77	**98.83**	437.64	187.40	250.57	265.48
citeseerx	399.19	303.02	148.78	1523.49	1531.19	117.55	111.08	236.55	**74.84**	151.02	185.10
dbpedia	205.42	159.64	59.90	52.19	85.04	53.30	**25.32**	131.88	38.51	94.28	109.16
go-uniprot	425.29	399.49	242.66	414.41	291.75	247.86	90.76	431.94	**79.74**	184.68	193.10
govwild	489.68	409.90	304.06	3122.55	319.49	191.04	118.21	464.02	**91.81**	193.96	249.63
HostLink	778.48	580.03	115.44	5670.27	269.33	201.45	**57.25**	326.70	145.96	369.60	363.80
patent	230.39	250.21	5334.12	4731.99	—	625.28	347.31	234.00	**43.19**	137.87	132.91

TABLE 8
Index Size of Real Datasets (in MB)



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

Fig. 9. Performance on synthetic graphs with average degree 2 to 8

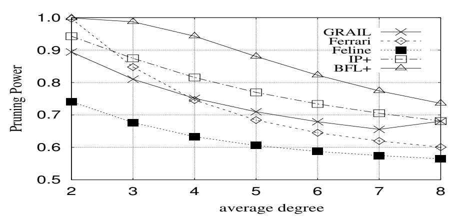


Fig. 10. Pruning Power

because their memory usage exceed the memory limit of the machine (32GB). It shows that Label-Only approaches cannot deal with dense graphs efficiently.

Fig. 9(a) and Fig. 9(b) show the index construction time and index size on graphs with different density. Unlike the Label-Only approaches, *BFL+* and *IP+* have good scalability because they can efficiently build up index for the dense graphs. *BFL+* has better indexing time and is about 2 times faster than *IP+* no matter how dense a graph is. The index size of *BFL+* is close to that of *IP+*, and as the graph density increases, the difference between their index size becomes marginal. We also randomly generated 1 million reachability queries over the synthetic graphs. The total query time to answer all reachability queries generated is shown

Dataset	*GRAIL*	*ScaGRAIL*	*PWAH8*	*TF-Label*	*HL*	*DL*	*TOL*	*Ferrari*	*Feline*	*IP+*	*BFL+*
citeseer	67.786	71.754	108.196	**6.450**	48.313	46.589	28.216	40.163	42.707	16.399	14.548
email	83.086	112.665	148.805	**14.440**	22.728	18.100	17.367	64.903	841.754	26.692	16.528
LJ	168.408	609.324	126.589	36.159	48.265	46.830	32.343	59.233	126916	56.573	**24.532**
mapped100K	117.733	40.229	42.126	48.181	64.336	57.058	38.757	15.363	40.667	19.377	**14.684**
mapped1M	133.635	42.924	44.045	50.163	61.784	55.148	39.582	15.899	42.129	21.005	**14.567**
twitter	128.708	176918	69.607	41.522	57.418	53.955	36.356	35.474	1385020	35.532	**14.920**
uniprot22m	44.194	162.321	233.163	21.972	60.612	53.394	31.828	25.511	24.932	15.801	**14.740**
uniprot100m	78.332	225.355	277.731	41.132	69.700	63.549	37.819	50.421	28.741	21.432	**14.354**
uniprot150m	92.465	246.180	286.685	44.976	72.778	65.673	36.520	55.248	28.266	21.243	**14.700**
web	200.872	1011.790	209.311	36.588	42.262	40.586	28.376	74.536	13622	65.110	**24.571**
wiki	47.099	124.099	44.344	29.713	53.761	52.084	33.225	16.988	21948	18.534	**14.481**
web-uk	282.671	—	163.994	72.956	132.298	61.178	42.140	113.661	1557320	123.393	**27.891**
yago	108.700	91.888	91.352	56.624	67.451	60.279	38.216	68.186	30.672	44.619	**14.347**
citeseerx	272.058	360.483	157.486	107.950	101.821	65.407	47.876	61.579	1512.890	57.636	**29.543**
dbpedia	199.270	246.283	257.634	60.315	65.230	74.545	40.243	112.043	2892.010	58.683	**24.046**
go-uniprot	37.496	79.916	412.901	32.852	158.068	137.656	34.979	179.868	27.956	18.383	**17.458**
govwild	221.994	106.852	258.227	136.765	95.859	74.851	48.471	155.857	118.195	60.620	**18.591**
HostLink	272.415	3293.990	121.774	112.655	62.611	68.287	42.442	89.790	210708	79.148	**41.510**
patent	4445.194	876.148	12608	335.014	—	184.601	**95.207**	3312.690	4637.990	1620.470	257.671

TABLE 9
Query Time on Real Datasets (in millisecond)

Dataset	avg $\bar{\alpha}$	avg β	EPP	APP
citeseer	1.60	1.57	99.20%	100.00%
email	32.44	13.21	99.50%	99.91%
LJ	75.45	111.75	99.50%	99.89%
mapped100K	1.03	1.07	98.32%	100.00%
mapped1M	1.08	1.04	98.10%	100.00%
twitter	10.39	44.25	99.50%	99.99%
uniprot22m	2.00	1.00	96.44%	100.00%
uniprot100m	2.06	1.06	96.92%	100.00%
uniprot150m	2.10	1.11	97.26%	100.00%
web	72.74	65.82	99.50%	99.59%
wiki	1.13	5.96	99.50%	100.00%
web-uk	1.61	1.59	99.22%	99.96%
yago	6.55	6.43	99.50%	99.89%
citeseerx	41.97	15.91	99.50%	99.86%
dbpedia	2.59	1.53	99.32%	99.98%
go-uniprot	5.01	4.75	99.50%	99.89%
govwild	21.04	26.51	99.50%	99.64%
HostLink	96.56	78.76	99.50%	96.72%
patent	59.07	77.04	99.50%	99.77%

TABLE 10
Estimated Pruning Power (EPP) and Actual Pruning Power (APP)

in Fig. 9(c). The *2-Hop* approaches have the good performance in the datasets if they can construct the index given the time and space limited. However, they cannot be applied in dense graphs. Compared with the scalable approach *IP+*, *BFL+* is about 2 times faster due to the higher pruning power as mentioned below.

From the testings over synthetic graphs, we show that only the Label+G approaches are scalable to handle large dense graphs. To further study the query time of the Label+G approaches, we compare their pruning power, which is the ratio of the queries that can be answered directly without performing *DFS*. *BFL+* always has the best pruning power compared with the other Label+G approaches. As the average degree increases, all Label+G algorithms' pruning power decrease correspondingly but *BFL+* can keep about 75% the pruning power even in the densest graph with an average degree 8. High pruning power of *BFL+* leads to good query performance as shown in Table 9 and Fig. 9(c) because *BFL+* needs to visit only a few nodes, if any, for answering queries.

The pruning power for the representative approaches are shown in Fig. 10. Among them, *BFL+* shows the highest pruning power.

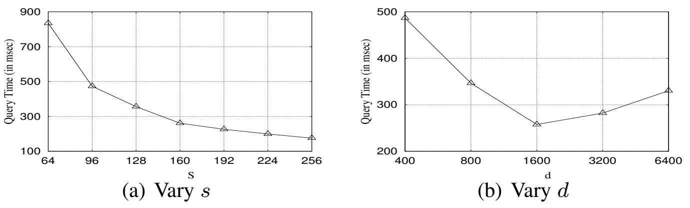


Fig. 11. Parameter Study

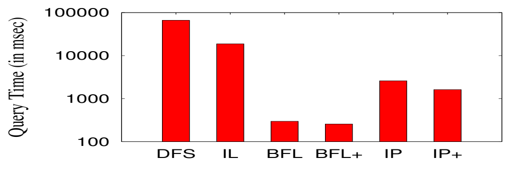


Fig. 12. *BFL+* label

7.3 More about *BFL+*

We study how the 2 parameters used in *BFL*, namely, the size s and the number of intervals, d, when merging the vertices, affect the query performance of *BFL+*. We generate 1 million random queries on dataset patent and set $s = 160, d = 1,600$ as default.

Fig. 11(a) shows that a larger s can improve the query performance because we can have a better pruning power or equivalent a smaller false postive rate, according to Theorem 4.3. However, as s becomes sufficiently large, the improvement of the query time is marginal because too large s cannot reduce the false positive rate effectively but incurs high index cost. So $s = 160$ is a reasonable choice for dense graph patent. As we explain before, too small or too large d will lead to worse query performance and we can see that setting d = 1600 is suitable in Fig. 11(b).

In addition, *BFL+* is *BFL* label plus an interval label (*IL*). Fig. 12 shows the query performance of different label combinations for the patent dataset. Here, *BFL* label itself without any additional label can answer queries efficiently, compared with the naive *DFS* traversal or interval labels. Its query performance can be further improved with the combination of interval label. On the other hand, we can see that *IP* label without any additional label is significantly slower than *BFL*.

8 Conclusions

In this paper, we propose a new *BFL* approach based on Bloom filter to answer reachability queries. *BFL* significantly improves the state-of-art *IP* approach, which is based on k-min-wise independent permutations to process reachability queries. Comparing with *IP* using up to k smallest numbers by a permutation where a number is represented as an integer, we show that *BFL* has high pruning power or small false positive rate to answer reachability queries directly, since *BFL* fully utilizes every bit to reduce the false positive rate. We also discuss how to balance the index space and the reduction of the false positive rate based on vertices merging. With the same 19 large datasets tested, *IP* with two additional labels performs best in 14 out of 19 for the index construction time and performs best in 10 out of 19 for query time [26]. *BFL* with a simple interval label is the best in index construction time in all 19 cases, and is the best in query time for 16 out of 19 cases. *BFL* improves the performance significantly.

In general, the DAGs of real graphs tend to have a small R-ratio. *BFL* outperforms the others when a DAG has a small R-ratio. *BFL* has limitation to deal with high R-ratio DAGs. If a DAG constructed has high R-ratio, which may be possibly caused by high-average-degree of the DAG, the performance by *BFL* will be affected. Regarding the performance over high-average-degree DAGs, *BFL* outperforms the other Label+G approaches; the condition for Label-Only approaches to outperform *BFL* is that they can construct an index for high-average-degree DAGs which is not always feasible. *BFL* can be used to answer reachability queries for any graphs.

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] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

[15] 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.

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

[17] A. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. *Random Structures & Algorithms*, 7(1):59–80, 1995.

[18] J. Leskovec and R. Sosič. SNAP: A general purpose network analysis and graph mining library in C++. http://snap.stanford.edu/snap, June 2014.

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

[20] 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.

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

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

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

[24] R. R. Veloso, L. Cerf, W. M. Jr., and M. J. Zaki. Reachability queries in very large graphs: A fast refined online search approach. In *Proc. of EDBT*, 2014.

[25] 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.

[26] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: An independent permutation labeling approach. *PVLDB*, 2014.

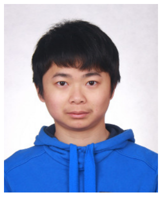
[27] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1), 2010.

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

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

[30] Z. Zhang, J. X. Yu, L. Qin, Q. Zhu, and X. Zhou. I/o cost minimization: reachability queries processing over massive graphs. In *Proc. of EDBT'12*, 2012.

[31] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: A total order approach. SIGMOD '14, 2014.



Jiao Su received his BE degree in School of Information, Renmin University of China in 2015. He is working towards his PhD degree in Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong. His research interests include graph algorithms and keyword search.

Qing Zhu received the PhD degree in computer science from the Renmin University of China in 2005, and received the MS degree in computer science from the Beijing University of Technology in 1991. She is the associated professor and supervisor at department of Computer Science in School of Information, Renmin University of China. She is a senior member of the China Computer Federation (CCF). Her research interests include big graph search, service recommendation, security, privacy, trust and data management issues in big data systems, cloud computing and distributed computing systems.

Hao Wei is pursuing his PhD degree in Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong. His research interests include graph data management and graph algorithms.

Jeffrey Xu Yu has held teaching positions at the Institute of Information Sciences and Electronics, University of Tsukuba, and at the Department of Computer Science, Australian National University, Australia. Currently, he is a Professor in the Department of Systems Engineering and Engineering Management, the Chinese University of Hong Kong, Hong Kong.