2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)

Reachability in Large Graphs using Bloom Filters

Arkaprava Saha
Indian Institute of Technology Delhi
saha0003@e.ntu.edu.sg

Neha Sengupta
Indian Institute of Technology Delhi
neha.sengupta@cse.iitd.ac.in

Maya Ramanath
Indian Institute of Technology Delhi
ramanath@cse.iitd.ac.in

***Abstract*—Reachability queries are a fundamental graph operation with applications in several domains. There has been extensive research over several decades on answering reachability queries efficiently using sophisticated index structures. However, most of these methods are built for static graphs. For graphs that are updated very frequently and are massive in size, maintaining such index structures is often infeasible due to a large memory footprint and extremely slow updates. In this paper, we introduce a technique to compute reachability queries for very large and highly dynamic graphs that minimizes the memory footprint and update time. In particular, we enable a previously proposed, index-free, approximate method for reachability called ARROW on a compact graph representation called Bloom graphs. Bloom graphs use collections of the well known summary data structure called the Bloom filter to store the edges of the graph. In our experimental evaluation with real world graph datasets with up to millions of nodes and edges, we show that using ARROW with a Bloom graph achieves memory savings of up to 50%, while having accuracy close to 100% for all graphs.**

***Index Terms*—Reachability, Dynamic graphs, Large graphs, Bloom filters, Low memory footprint**

I. Introduction

Reachability is a fundamental graph problem with wide applicability in several domains such as XML databases, GIS, web mining, analysis and access control on social networks, business intelligence, and bio-informatics among others [2], [8], [15]. A reachability query on a directed graph asks if there exists a path between a designated source node u and destination node v. To efficiently compute reachability, many solutions have been proposed that work by building and using sophisticated index structures on the graph [6], [7], [9], [13], [22], [26], [27], [29]. But, real-world applications such as social networks and call records often generate extremely large and dynamic graphs. For example, each time a new connection is formed or existing connection is discontinued in a social network, an edge is inserted or deleted in the corresponding graph. In such a scenario, an index is required to *maintain* the reachability information as the graph evolves, and therefore must be efficient to update.

Some techniques that efficiently maintain a reachability index as the graph evolves have been proposed in [4], [14], [17], [19], [28]. However, most of these methods do not scale for extremely large or highly dynamic graphs. For example, in [17], the amortized time taken to update the reachability index is $O(m + n \log n)$, where n and m are the number of nodes and edges in the graph, respectively. For web-scale graphs like social networks that have millions of nodes and edges, and also update several times per second, such methods are unsuitable. Moreover, the memory footprint of the graph as well as the associated reachability index is often extremely large for web-scale graphs. In this work, we address the challenge of answering reachability queries for very large and dynamic graphs by first, reducing the memory footprint of the graph itself by using a compact graph representation, and second, eliminating the memory footprint and update complexity of the index by employing an index-free technique for reachability.

We thank the Alumni-sponsored CS&E Research Acceleration Fund at IIT Delhi for their support.

Arkaprava Saha is currently with the Nanyang Technological University, Singapore

Bloom graphs, proposed in [10], are graphs that store the neighborhood of each node in summary data structures called Bloom filters [3], thereby leveraging the time and space efficiency of Bloom filters to facilitate compact storage of the graph and processing of graph updates. Operations such as checking the existence of an edge or insertion of a new edge in a Bloom graph correspond to the set membership and element insertion respectively, both of which are constant time operations on a Bloom filter. In [10], Bloom graphs are used as a framework for efficiently tracking the *conductance* of a large number of subsets of nodes as the graph evolves. The conductance metric, which in this case is useful to evaluate the popularity of topics on the Twitter network, is efficiently updated with the evolving graph by utilizing fast intersection operations on Bloom filters.

In this work, we explore the applicability of a Bloom graph for answering the reachability query. ARROW, introduced in [21], is an index-free method for computing reachability in directed, dynamic graphs. Given a reachability query from source node u to destination node v, ARROW is based on running random walks from both u and v directly at query time, without the use of any reachability index. Since ARROW does not build or employ an index, it enables instantaneous updates to the graph. In addition, it has a very small memory footprint, since it stores only the graph along with its transpose. Our technique for using ARROW on a Bloom graph, called BloomARROW, inherits the update efficiency of ARROW and further reduces its memory footprint by using the Bloom graph representation for compactly storing the graph.

A. Contributions

Our contributions are:

1) We introduce a variant of the Bloom graph, called the Hybrid Bloom graph, that reduces memory consumption of the graph dataset. The savings achieved in memory can be as high as 50% for large, real world graphs.
2) We propose BloomARROW, which uses the random walk based algorithm ARROW on a Hybrid Bloom graph to answer reachability queries on a large graph.
3) We conduct extensive empirical evaluation on real world graphs to illustrate the advantages of BloomARROW and Hybrid Bloom graphs.

B. Organization

The rest of this paper is organized as follows. In Section II, we review previous work on reachability for static and dynamic graphs. Section III describes Bloom filters and Bloom graphs in detail. We describe variants of the Bloom graph and our method BloomARROW in Section IV. We evaluate the Hybrid bloom graph and BloomARROW in Section V and conclude in Section VI.

II. Related Work

Index-based reachability in static graphs may be broadly classified into transitive closure retrieval, interval labeling, and 2-HOP label matching [28], [30]. Methods that store compressed forms of the transitive closure, which can be retrieved at query time and used to answer reachability queries extremely fast, are infeasible for massive graphs due to their enormous memory and pre-processing overhead. The interval labeling technique assigns to each vertex a set of intervals such that a vertex u can reach v if any of the intervals of v is contained in those of u [5], [12], [15], [22]–[27]. While several of these methods are very efficient for large graphs, the difficulty in updating the index as the graph updates limits their use to static graphs only. In 2-HOP labeling methods, each vertex stores a set of nodes that it can reach (L_{out}), and a set of nodes that can reach it (L_{in}). In order to answer the reachability query (u, v), the sets $L_{out}(u)$ and $L_{in}(v)$ are intersected [6], [7], [18].

For dynamic graphs, an early work on reachability by Agrawal et al [1] observes that the challenges of maintaining transitive relationships are achieving small storage, efficient look-up and frequent updates, and low incremental cost. Some 2-Hop techniques have been adapted for the dynamic setting. In [30] for instance, a total order on the vertices is used to support addition and removal of node type updates in time $\mathcal{O}(n^2)$, where n is the number of nodes in the graph. In [4], a 2-Hop reachability index satisfying the *node separation property* is constructed, i.e. deletion of nodes in $L_{out}(u) \cap L_{in}(v)$ from G disconnects u and v for any pair of nodes (u, v) in G. This enables the index's update cost to be $\mathcal{O}(n)$ 2-HOP lookups. Roditty et al [17] propose another 2-HOP labeling based deterministic, reachability algorithm with an update time of $\mathcal{O}(m + n \log n)$. In [28], the authors present DAGGER, a reachability index for a dynamic graph that is based on relaxed interval labeling on the equivalent DAG of the graph constructed by condensing each of its strongly connected components. Edge deletion and node removal type updates are extremely expensive in this method, since a deleted edge can cause a strongly connected component to split, triggering re-computation for the DAG.

In the setting of large, highly dynamic graphs that we consider, all of the above methods of maintaining an index are unsuitable due to the update costs being at least $O(n)$. In contrast, we compute reachability in such graphs by using an index-free technique called ARROW on a compact representation of the graph called Bloom graphs. ARROW in [21] has a constant update time for all types of updates, and query computation is based on random walks conducted entirely at query time. With no index, we support extremely fast update time, and use Bloom graphs to minimize memory footprint of the graph.

III. Background

a) Bloom Filters: A Bloom filter [3] is a data structure used to compactly store a set of elements. The Bloom filter is composed of an array of m bits, and k independent hash functions, $h_1 \ldots h_k$. For a Bloom filter storing an empty set, each of the m bits is 0. In a Bloom filter storing a non-empty set, for each element x in the set, the k positions $h_1(x) \ldots h_k(x)$ in the bit array are set to 1.

The Bloom filter is designed to support fast membership queries. Given a Bloom fitler $\mathcal{B}(S)$ storing a set S, a membership query asks, "is $x \in S$" for a given element x. The result of the membership query "is $x \in S$" is positive only if all k locations $h_1(x) \ldots h_k(x)$ in the bit array are set to 1. However, note that the insertion of other elements could have contributed to these locations being set, and thus the probability of a false positive is non zero, and is equal to $\approx (1 - e^{(-kn/m)})^k$, where $n = |S|$. Thus, for a given size of the bit array, the false positive probability of the Bloom filter increases with the number of elements in S. The Bloom filter cannot have false negatives.

Other than the membership query, the union and intersection of a pair of Bloom filters can also be computed using bit-wise OR and AND operations respectively.

b) Sampling from a Bloom Filter: The authors of [20] introduce a technique to efficiently sample an element, almost uniformly at random, from a set (called the query set) stored as a Bloom filter (called the query Bloom filter). An auxiliary binary tree data structure called the BloomSampleTree uses a collection of Bloom filters to hierarchically organize the *name space*, which is the universe of elements from which a query set is drawn. A single BloomSampleTree suffices to support sampling from any number of query sets drawn from the same name space, as long as the Bloom filter configuration such as size and number of hash functions used is fixed across the query Bloom filters as well as those within the BloomSampleTree. Each node in the BloomSampleTree corresponds to a sub-range of the name space that is stored in its Bloom filter, and the sub-range associated with an internal node is partitioned into two roughly equal parts among its left and right child nodes. The entire namespace is associated with the root node, which is level 0. Roughly half of the namespace is associated with each node at level 1, and so on. The fraction of namespace associated with a tree node at level l is $M/2^l$, where M is the size of the entire namespace. Given the query Bloom filter b, the sampling algorithm begins at the root node. b is intersected with the two Bloom filters situated at the left and right child nodes of the root, and the estimated number of elements in each intersection, say e_l and e_r are computed. Note that e_l and e_r are estimates of the number of elements b contains within each half of the namespace. The algorithm moves to the left branch with probability $P_l = e_l/(e_l+e_r)$, and to the right branch with probability $P_r = 1 - P_l$. This process repeats and the search moves from the root down to a leaf node of the BloomSampleTree. At this point, each element in the significantly smaller sub-range of the name space associated with the current leaf is subjected to the membership test of the query Bloom filter. Finally, a sample is drawn uniformly at random from the set of values that pass the membership test.

Given the BloomSampleTree, and the query Bloom filter b, the above sampling algorithm returns an element selected almost uniformly at random from the set of elements that pass the membership query of b. Over the naive method of iterating over the entire name space and collecting the set of elements that pass the membership query of b, this algorithm achieves significant savings in sampling time by using the Bloom filter intersection operation to prune large parts of the name space.

c) Bloom Graph: A Bloom graph uses a collection of Bloom filters to space efficiently store a graph. As defined in [10], for a graph $G(V, E)$ and a positive integer m, the Bloom graph $\mathcal{B}(G, m)$ is the collection of Bloom filters $\{\mathbf{In}_G(u), \mathbf{Out}_G(u) : u \in V\}$, where $\mathbf{In}_G(u)$ and $\mathbf{Out}_G(u)$ are both Bloom filters of size m and are termed neighborhood filters of u.

$\mathbf{In}_G(u)$ stores the set $\{w \in V : (w, u) \in E\}$ or the incoming neighbors of u. Similarly, $\mathbf{Out}_G(u)$ stores the set $\{w \in V : (u, w) \in E\}$, or the outgoing neighbors of u. Therefore, the Bloom graph stores $2 \times |V|$ Bloom filters, each of size m, to store the directed graph.

IV. Method

To answer reachability queries on the Bloom graph, we employ the random walk based technique called ARROW, proposed in [21]. ARROW is an index-free method that answers reachability queries using random walks from both source and destination nodes conducted at query time. ARROW eliminates the need to store an index, and therefore has both very low memory footprint and highly efficient update time as opposed to existing index-based methods, making it significantly more suitable in scenarios where the underlying graph dataset is both large and highly dynamic. In this section, we describe our technique, called BloomARROW, to answer reachability queries when the graph is compactly stored as a Bloom graph, such that the memory footprint of BloomARROW is reduced even further from that of ARROW.

To answer a reachability query (u, v) on a Bloom graph with n vertices, BloomARROW must run, from both u and v, $c_1 \times \sqrt[3]{n^2 \ln n}$ random walks, each of length $c_2 \times diam$, where $diam$ is the graph diameter and c_1 and c_2 are user-defined constants that control the trade-off between BloomARROW accuracy and query latency, as in ARROW. While random walks from u are run along the out-edges of nodes in the graph, random walks from v are run along the in-edges , i.e. on the transpose of the graph where the direction of each edge is reversed. Nodes encountered during random walks from u and v are collected in sets $F(u)$ and $B(v)$ respectively, and reachability is reported to exist whenever $F(u) \cap B(v) \neq \emptyset$.

Conducting random walks on the Bloom graph requires the ability to sample from the Bloom filter. In the following we describe how the module for sampling from a given Bloom filter is used to implement random walks on a Bloom graph. Figure 1 illustrates the system for a random walk conducted on the Bloom graph. The Bloom graph is represented using a collection of Bloom filters, one for each node in the graph. Each Bloom filter uses the same set of k hash functions H, and the same size of Bloom filter m. In a pre-processing step, the BloomSampleTree is built. The name space of the BloomSampleTree is the set of all vertices in the graph, and each of the Bloom filters within the BloomSampleTree uses the same set of hash functions H and Bloom filter size m. Given the BloomSampleTree, and the Bloom graph, BloomARROW conducts a random walk with source as query node u. The walk begins at u, and the adjacency Bloom filter of u is passed to the sampling module, which uses its BloomSampleTree to sample an element from u's Bloom filter. The sampling module returns the node s, say, and that becomes the node that the walk transitions to. This process is repeated until the required number of steps are completed, or until the walk reaches a dead-end (i.e. when the sampling module receives an empty Bloom filter).

Clearly, accuracy of the sampling module has a direct effect on the degree to which random walks follow actual paths in the graph, and thus the reachability accuracy of BloomARROW. A low accuracy will create random walks along non-existent paths in the graph due a large number of false positives, and thereby introduce false positives in the result of BloomARROW. On the other hand, high accuracy comes at the cost of using larger Bloom filters, leading to an enormous memory footprint for the entire Bloom graph, which stores a collection of these Bloom filters. Together, accuracy of sampling, Bloom filter size and BloomSampleTree depth have a direct influence on the space, query latency, and accuracy of BloomARROW.

In the following section, we describe ways to reduce the memory footprint of the Bloom graph for a given desired accuracy of BloomARROW.

A. Storing the Bloom graph

To address the issue of exploding memory usage of a Bloom graph built for increased accuracy, we explore alternative methods of storing it. The key observation is that real world

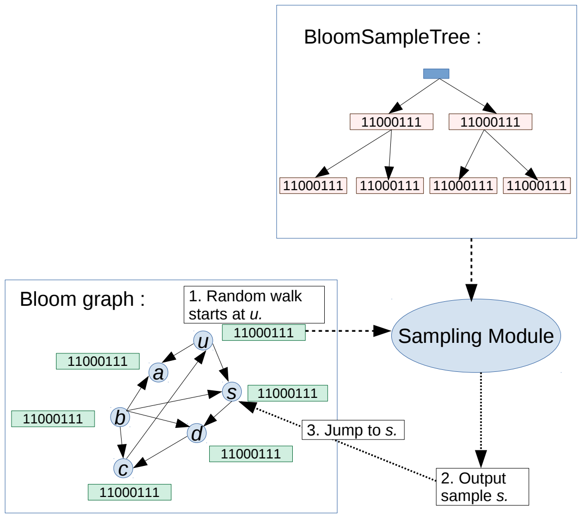


Fig. 1: Random Walk on Bloom graph

graphs typically have a power law degree distribution, i.e. a large fraction of nodes have very small degree while a small fraction have large degree. A Bloom filter size that can achieve the desired accuracy for the highest degree node is several orders of magnitude larger than that required for low degree nodes. The representation of Bloom graphs presented until now, that use fixed size Bloom filters for each node in the graph, are termed **Standard Bloom graphs**. The Standard Bloom graph faces the disadvantage of having to use a Bloom filter size suited to the highest degree node, irrespective of the degree distribution of the other nodes, leading to memory wastage.

In order to tailor the memory footprint due to any given node in the graph according to its actual degree, the Bloom filters used must possess the capability to grow with the sets they store. Dynamic Bloom filters, proposed by Guo et al [11], are composed of a list of small standard Bloom filters. Each unit Bloom filter in the list obeys a maximum false positive threshold of fp_0, and the last filter in the list is the *currently active* one. When an element must be inserted into the Dynamic Bloom filter, it is inserted into the active Bloom filter. If the number of elements in the active Bloom filter reaches a threshold n_0 (i.e. its false positive probability reaches fp_0), a new unit Bloom filter is appended to the end of the list, and becomes the currently active Bloom filter. n_0 is determined from fp_0 as

$$n_0 = \frac{\log(1 - fp_0^{1/k})}{k \log\left(1 - \frac{1}{m}\right)}$$

where k is the number of hash functions used and m is the size of each unit Bloom filter.

A Dynamic Bloom filter storing n_r elements has $\lfloor n_r/n_0 \rfloor$ unit Bloom filters, each with a false positive probability of fp_0. In addition, it has a presently active Bloom filter of size m, that stores $n_c = n_r \bmod n_0$ elements. The overall false positive probability of the Dynamic Bloom filter is

$$1 - (1 - fp_0)^{\lfloor n_r/n_0 \rfloor} \times \left(1 - \left(1 - \frac{1}{m}\right)^{n_c k}\right)^k$$

In general, the false positive rate of a Dynamic Bloom filter can be greater than fp_0.

We define a **Dynamic Bloom graph**, that uses Dynamic Bloom filters instead of standard Bloom filters for storing the neighborhood of each vertex in the graph. While Dynamic Bloom graphs help to reduce the memory footprint by a large extent, the choice of the unit size of the Dynamic Bloom filters used can have significant implications. A large unit defeats the purpose of the Dynamic Bloom graph affecting its ability to adapt to the degree distribution of the nodes, and a small unit size increases the number of units stored for high degree nodes, inflating the overall false positive probability of the Dynamic Bloom filter in spite of small fp_0 for individual units.

Clearly, the size of the individual Bloom filters must be large enough to retain performance, while still reducing memory usage. To this end, we propose the **Hybrid Bloom graph**, which is characterized by a *degree threshold* d. The in- or out-neighborhood of any node is stored in a Bloom filter of size m only if the neighborhood size exceeds d. For all other nodes, the set of neighbors is stored as a list. Therefore, given the adjacency list representation of the graph, i.e. where for each node in the graph the edges incident to it are simply stored as a list, the Hybrid Bloom graph is obtained by converting the largest edge lists into equivalent standard Bloom filters. Figure 2 illustrates the differences between the variants of Bloom graphs.

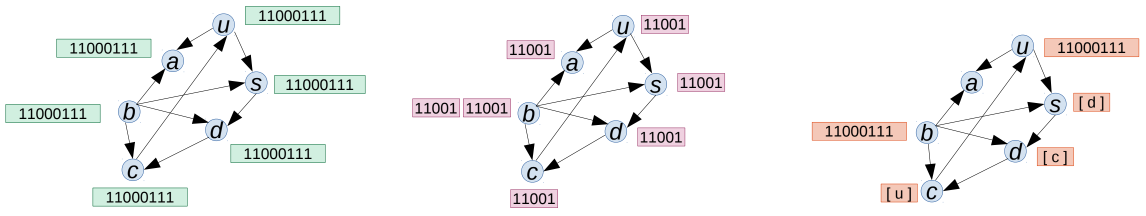
Dataset	Standard	Dynamic	Hybrid	Adjacency List
Epinions	525.5491	335.7801	**6.9107**	8.7891
Enron	156.9528	82.6306	**2.7615**	3.7107
Facebook	42.2873	32.0453	**7.1319**	8.6963
Flickr	46297.2135	37782.3005	**146.9525**	350.5596

TABLE I: Memory footprint (in MBs) for variants of the Bloom graph and the Adjacency list representation

Table I shows the memory consumed in MBs by each of the variants to obtain a maximum false positive rate (across all nodes) of 0.1 for 4 real world datasets (see Section V). Also, it shows the size of the graph when stored in adjacency lists. Clearly, among the Bloom graph variants, Hybrid Bloom graphs are capable of providing a given threshold on the error rate with the smallest memory footprint. For larger graphs, such as the Flickr graph, it achieves more than 50% savings in memory over the adjacency list representation. The remainder of this section, therefore, focuses on BloomARROW with Hybrid Bloom graphs.

B. BloomARROW on the Hybrid Bloom Graph

Conducting a random walk on the Hybrid Bloom graph is a simple application of the Bloom filter sampling module. At



(a) Standard Bloom graph (b) Dynamic Bloom graph (c) Hybrid Bloom graph

Fig. 2: Bloom Graph Variants. *Figure 2a is the Standard Bloom graph that stores the neighborhood of each vertex in the same size Bloom filter, irrespective of their actual degree. Figure 2b is the Dynamic Bloom graph that uses Dynamic Bloom filters. High degree nodes such as node b store more number of small Bloom filters to accommodate for their large neighborhood list. Figure 2c is the Hybrid Bloom graph. Only nodes with degree* ≥ 2 *use Bloom filters of fixed size. Other nodes of smaller degree simply store their neighborhoods in a list. Note that sampling accuracy for all low degree nodes not storing Bloom filters is 1 in the Hybrid Bloom graph.*

each step of the walk, a random neighbor of the current node must be sampled. If the neighbors are stored in a Bloom filter, then it is passed to the sampling module, and if in a list, then a sample is directly drawn from it uniformly at random. Note that in a Hybrid Bloom graph, sampling accuracy for nodes that do not store Bloom filters is 1, thereby increasing the average sampling accuracy across all nodes.

In this section, we discuss implementation details for conducting a random walk on the Hybrid Bloom graph. A random walk on the Hybrid Bloom graph must sample from a Bloom filter each time it encounters a high degree node. Since sampling from a Bloom filter is far slower than sampling from a list, this can adversely impact the total time taken to conduct the random walk, and thus the query latency. Therefore, the significant savings in memory footprint offered by Hybrid Bloom graphs come at the cost of longer time to sample neighbors of nodes and thus slower random walks. The following optimization strategies were employed to reduce the time taken to conduct a random walk, thereby reducing query latency.

a) Sampling Multiple Items: Whenever the random walk encounters a node with a Bloom filter, it uses the sampling module to sample $k > 1$ items at once (see [20]). In our experiments, $k = 10$. The sampled elements are stored in a buffer of size k, and a distinct element is used each time a random walk returns to the node. After k visits, the buffer is exhausted, and is rebuilt on the $(k+1)^{th}$ visit. This procedure helps to amortize the walk time over a number of steps and random walks.

b) Multi-threaded Sampling: Sampling multiple items from the BloomSampleTree involves pursuing multiple paths along the tree concurrently. We implement a multi-threaded procedure based on the producer-consumer paradigm for sampling multiple items. A major cause for the traversal of *false positive paths*, or paths that do not lead to a valid sample in the BloomSampleTree is that the Bloom filters at levels closest to the root node are extremely dense, thereby generating more error (See Section III). As a result, the sampling procedure proceeds along both child nodes whenever it reaches such a node, even though samples may actually exist only along one branch. To prune out such false positive paths, the sampling procedure is updated as follows: At an internal node, the bit-wise intersection of the neighborhood Bloom filter (which is to be sampled from) is computed with the Bloom filters at the left and right child nodes. Only if the size of both intersections exceeds $T \times m$, where m is the Bloom filter size, and T is a user-selected threshold, (in our experiments, $T = 0.75$) the procedure is allowed to move along both branches. Otherwise, it randomly selects one branch with probability proportional to the corresponding intersection size.

c) Truncated BloomSampleTree Sampling: Note that the BloomSampleTree partitions the namespace into only two parts at level 1 (where root is level 0). This means that when a sample must be drawn from a given Bloom filter having elements in both halves of the namespace, it must be intersected with all 4 Bloom filters at level 2, rendering the two intersections at level 1 redundant. Moreover, the redundant computation is repeated each time samples are drawn from this Bloom filter. To avoid this, we pre-compute for each vertex v with a Bloom filter, the largest BloomSampleTree level l_v such that the intersection of v's neighborhood with each Bloom filter in the l_v^{th} level of the BloomSampleTree is non-empty. Each time sample neighbors for node v must be drawn, the sampling procedure starts multiple concurrent paths directly from the nodes at level l_v of the BloomSampleTree.

V. Experiments

We evaluate the performance of BloomARROW with experiments on real world datasets. We show that BloomARROW performs extremely accurately in practice, and has a significantly lower memory footprint than ARROW, although the query latency of BloomARROW is larger due to the requirement to frequently sample from the Bloom filter.

a) Setup: We use real datasets **Epinions**, **Enron**, and **Flickr** from the Konect repository [16], and **Facebook**, used in [21]. While Flickr and Facebook are social networks, Enron is an email network, and Epinions is a trust/distrust network on the product rating website. Table II shows properties of the graphs including number of nodes, edges, and average degree. In each of these datasets, we apply the graph stream to the initial graph and obtain the final graph snapshot. In addition, we use reachability queries generated corresponding to each dataset as in [21] to evaluate the performance of random walks and BloomARROW. To compute the result of a query, both ARROW and BloomARROW run $c_1 \times \sqrt[3]{n^2 \ln n}$ random walks, each of length $c_2 \times diam$, where $n = |V|$ is the number of nodes in the graph, $diam$ is its diameter (the longest shortest path in the directed graph), $c_1 = 0.01$ and $c_2 = 1$.

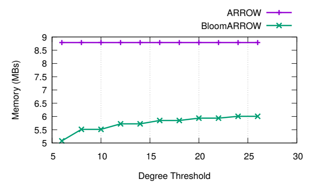
For each dataset, the Hybrid Bloom graph configuration is varied using the parameter d, or the degree threshold. Only nodes with degree $> d$ opt to store Bloom filters instead of the actual neighborhood list. Larger the value of d, smaller is the set of nodes that store Bloom filters. When d is equal to the maximum degree of the graph, the Hybrid Bloom graph reduces to the adjacency graph. In our experiments, d is varied from $m/16$ to $m/4$, where m is the Bloom filter size and is set to $m = 8 \times d_{avg}$, d_{avg} being the average degree of the graph (including in- and out- degrees).

Dataset	$\lvert V \rvert$	$\lvert E \rvert$	d_{avg}
Epinions	131828	841372	13
Enron	87274	1148072	8
Flickr	2302936	33140017	29
Facebook	63732	905565	29

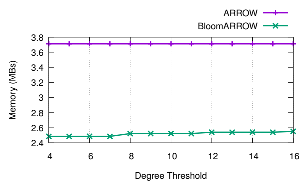
TABLE II: Datasets

We compare ARROW and BloomARROW with the Hybrid Bloom graph along the following metrics.

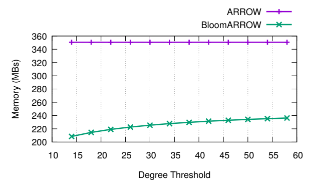
- Memory footprint: The in-memory size of the graph. In the case of the adjacency list representation, for each node in the graph its incident edges are stored as a list, and the memory consumption is the total memory occupied by all of the lists. For a Hybrid Bloom graph, the memory footprint includes the space used by Bloom filters employed for high degree nodes, and edge lists used for other nodes. For both representations, we include the memory occupied by node pointers in the total memory footprint. Since the Hybrid Bloom graph stores the largest neighborhood lists in the graph compactly in Bloom filters, its memory footprint is expected to be smaller than using adjacency lists.
- Random walk time: The average time taken by a random walk conducted by ARROW or BloomARROW. Since random walks rely on drawing random samples at each node, we expect that walks on a Hybrid Bloom graph will be slower than walks on the adjacency graph.
- Reachability accuracy: The random walk on a Hybrid Bloom graph may transition along edges that do not actually exist in the graph. Note that while sampling from a Bloom filter, it is not possible to distinguish between true and false positives. Therefore, if the Bloom filter at a node in the Hybrid Bloom graph generates a sample that is a false positive, the random walk transitions to the sampled node, and hence follows a *false positive* path. A false positive path may result in a random walk started at node s to reach a node t even when s cannot reach t in the underlying graph. Note, however, that this is not always true – s may reach t in the underlying graph via some other path, while the random walk reaches t via a false positive path. We report the reachability accuracy of random walks, which is equal to the fraction of random walks whose source nodes can reach their destination nodes in the underlying graph.
- ARROW Query time: The time taken to answer a reachability query with ARROW will include the time taken to conduct the random walks from both source and destination nodes and that to intersect the set of nodes visited in each direction. As with walk time, we expect that BloomARROW will be slower than ARROW
- ARROW Accuracy: Accuracy of ARROW is affected only due to false negatives, i.e., it may happen that no pair of random walks in either direction visit a common node, in spite of there being a path between the source and destination. In BloomARROW on the Hybrid Bloom graph, the accuracy may further be affected due to false positives. Since random walks can follow false positive paths, random walks in either or both directions can visit nodes that are not reachable from the source, or cannot reach the destination. As a result, intersection of the two sets may be non-empty and the query can result in `true` even though reachability does not actually exist.



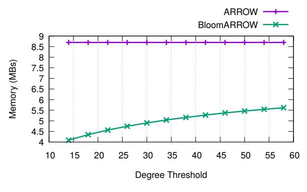
(a) Epinions



(b) Enron



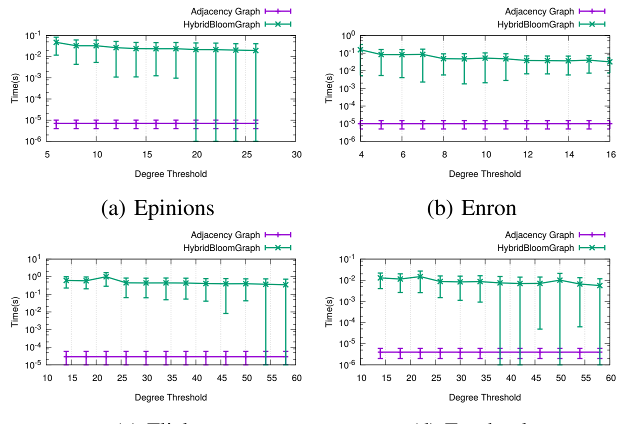
(c) Flickr



(d) Facebook

Fig. 3: Memory footprint of ARROW vs BloomARROW

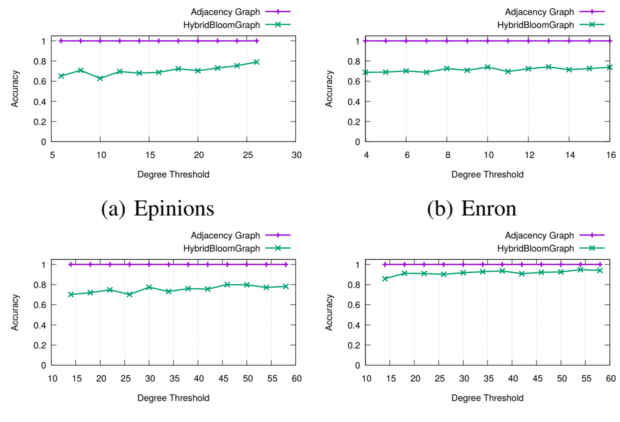
b) Results: In Figure 3, the memory footprint of BloomARROW using the Hybrid Bloom graph, is compared to that of ARROW which uses the adjacency graph. The Hybrid Bloom graph achieves space saving of up to 50 %, which can be highly significant when storing a massive graph that cannot fit in memory. With smaller Bloom filter sizes, i.e. where a higher false positive rate can be tolerated, the memory advantage will be even more pronounced.



(a) Epinions (b) Enron

(c) Flickr (d) Facebook

Fig. 4: Time taken by a Random Walk on the Hybrid Bloom graph vs the adjacency graph

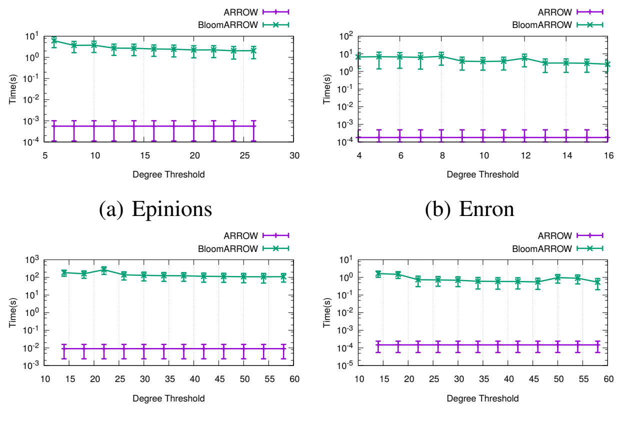


(a) Epinions (b) Enron

(c) Flickr (d) Facebook

Fig. 5: Reachability Accuracy of the Random Walk on the Hybrid Bloom graph vs the adjacency graph

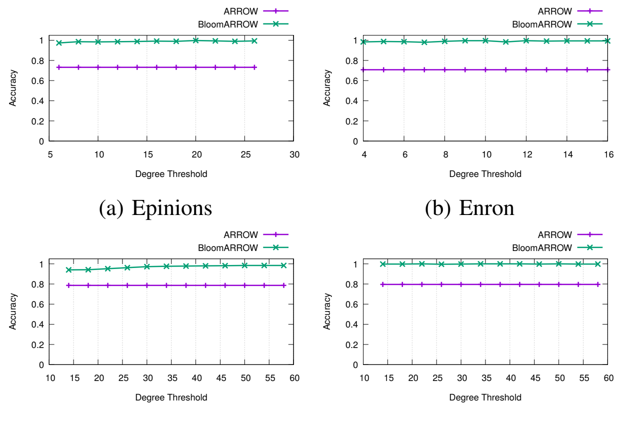
Figure 4 compares the average time taken by random walks (across all queries) for the Hybrid Bloom graph and the adjacency graph. As predicted, the walk time on the Hybrid Bloom graph is significantly larger than that on the adjacency graph. As the degree threshold increases, fewer nodes store Bloom filters, and therefore, sampling at those nodes becomes faster, reducing the average walk time. Similarly, figure 5 evaluates the reachability accuracy of the random walks across all queries on the Hybrid Bloom graph and the adjacency graph. The reachability accuracy of random walks on the adjacency graph is trivially 1, since the random walks are conducted directly on the edges of the underlying graph, and there is no source of false positives. The reachability accuracy of walks on the Hybrid Bloom graph range from $60-90\%$. This is due to the fact that a false positive path can result from a false positive sample on *any* Bloom filter storing node on the path of the random walk, increasing the probability of visiting a node that is not reachable in the underlying graph.



(a) Epinions (b) Enron

(c) Flickr (d) Facebook

Fig. 6: Query latency of ARROW vs BloomARROW



(a) Epinions (b) Enron

(c) Flickr (d) Facebook

Fig. 7: Accuracy of ARROW vs BloomARROW

Figures 6 and 7 evaluate the query latency and reachability query accuracy of ARROW and BloomARROW. Due to slower random walks, the time taken by BloomARROW is larger than ARROW. Contrary to intuition however, BloomARROW is consistently more accurate than ARROW, achieving accuracy close to 100% in each case. This seems unexpected because the usage of Bloom filters introduced false positives, in addition to the already existing false negatives of ARROW, and so should have decreased average accuracy. But, the ability of random walks to follow false positive paths actually helps to mitigate false negatives in BloomARROW. For a reachability query in which both nodes belong to the same strongly connected component (SCC), random walks on the adjacency graph may escape the SCC and never return, thereby constituting the dominant cause of false negatives of ARROW. In a Bloom graph, on the other hand, such random walks have a non-zero probability of returning to the SCC, due to false positives, and therefore are likely to eliminate the false negative. While the reduced probability of a false negative does not readily justify theoretically, it is very often observed in practice, as is evident from Figure 7.

VI. Conclusion

In this paper, we have enabled an index-free, approximate technique for reachability called ARROW on Bloom graphs, which are a form of graph representation that allow storing the edges of massive graphs in compact summary structures called the Bloom filter. Other than the previously proposed Standard Bloom graphs, we have introduced two further variants called Dynamic Bloom graphs and Hybrid Bloom graphs, and have shown using experiments on real world graph datasets that Hybrid Bloom graphs are capable of achieving significant memory advantages over other representations. In summary, our technique, called BloomARROW using Hybrid Bloom graphs provides a method for computing reachability in highly dynamic, massive graphs with minimized memory usage and high accuracy.

References

[1] Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. *Efficient management of transitive relationships in large data and knowledge bases*, volume 18. ACM, 1989.

[2] Kemafor Anyanwu and Amit Sheth. ρ-queries: enabling querying for semantic associations on the semantic web. In *Proceedings of the 12th international conference on World Wide Web, WWW*, pages 690–699. ACM, 2003.

[3] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. Incremental maintenance of 2-hop labeling of large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 22(5):682–698, 2010.

[5] Li Chen, Amarnath Gupta, and M Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st international conference on Very large data bases, VLDB*, pages 493–504. VLDB Endowment, 2005.

[6] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S Yu. Fast computing reachability labelings for large graphs with high compression rate. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology, EDBT*, pages 193–204. ACM, 2008.

[7] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[8] Imen Ben Dhia. Access control in social networks: a reachability-based approach. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 227–232. ACM, 2012.

[9] Uriel Feige. A fast randomized logspace algorithm for graph connectivity. *Theoretical Computer Science*, 169(2):147–160, 1996.

[10] Sainyam Galhotra, Amitabha Bagchi, Srikanta Bedathur, Maya Ramanath, and Vidit Jain. Tracking the conductance of rapidly evolving topic-subgraphs. *Proceedings of the VLDB Endowment*, 8(13):2170–2181, 2015.

[11] Deke Guo, Jie Wu, Honghui Chen, and Xueshan Luo. Theory and network applications of dynamic bloom filters. In *Proceedings of the 25th International Conference on Computer Communications, INFOCOM*, pages 1–12. IEEE, 2006.

[12] HV Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems, TODS*, 15(4):558–598, 1990.

[13] Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Yu Xu. Scarab: scaling reachability computation on large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 169–180. ACM, 2012.

[14] Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Transactions on Database Systems, TODS*, 36(1):7, 2011.

[15] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 595–608. ACM, 2008.

[16] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web, WWW*, pages 1343–1350. ACM, 2013.

[17] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733, 2016.

[18] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Hopi: An efficient connection index for complex xml document collections. In *Proceedings of the 7th International Conference on Extending Database Technology, EDBT*, pages 237–255. Springer, 2004.

[19] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *Proceedings of the 21st International Conference on Data Engineering, ICDE*, pages 360–371. IEEE, 2005.

[20] Neha Sengupta, Amitabha Bagchi, Srikanta Bedathur, and Maya Ramanath. Sampling and reconstruction using bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, 2017.

[21] Neha Sengupta, Amitabha Bagchi, Maya Ramanath, and Srikanta Bedathur. Arrow: Approximating reachability using random-walks over web-scale graphs. In *Proceedings of the 35th International Conference on Data Engineering, ICDE*, 2019.

[22] Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *Proceedings of the 29th International Conference on Data Engineering, ICDE*, pages 1009–1020. IEEE, 2013.

[23] Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1-3):325–346, 1988.

[24] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 845–856. ACM, 2007.

[25] Renê Rodrigues Veloso, Loïc Cerf, Wagner Meira Jr, and Mohammed J Zaki. Reachability queries in very large graphs: A fast refined online search approach. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT*, pages 511–522, 2014.

[26] Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE*, pages 75–75. IEEE, 2006.

[27] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010.

[28] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. *CoRR*, abs/1301.0977, 2013.

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

[30] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1323–1334. ACM, 2014.