ARROW: Approximating Reachability using Random walks Over Web-scale Graphs

Neha Sengupta, Amitabha Bagchi, Maya Ramanath, Srikanta Bedathur
IIT Delhi, New Delhi
Email:{neha.sengupta, bagchi, ramanath, bedathur}@cse.iitd.ac.in

Abstract—**Efficiently answering reachability queries on a directed graph is a fundamental problem and many solutions – theoretical and practical – have been proposed. A common strategy to make reachability query processing efficient, accurate and scalable is to precompute indexes on the graph. However this often becomes impractical, particularly when dealing with large graphs that are highly dynamic or when queries have additional constraints known only at the time of querying. In the former case, indexes become stale very quickly and keeping them up-to-date at the same speed as changes to the graph is untenable. For the latter setting, currently proposed indexes are often quite bulky and are highly customized to handle only a small class of constraints.**

In this paper, we propose a first practical attempt to address these issues by abandoning the traditional indexing approach altogether and operating directly on the graph as it evolves. Our approach, called ARROW, uses random walks to efficiently approximate reachability between vertices, building on ideas that have been prevalent in the theory community but ignored by practitioners. Not only is ARROW well suited for highly dynamic settings – as it is index-free, but it can also be easily adapted to handle many different forms of ad-hoc constraints while being competitive with custom-made index structures. In this paper, we show that ARROW, despite its simplicity, is near-accurate and scales to graphs with tens of millions of vertices and hundreds of millions of edges. We present extensive empirical evidence to illustrate these advantages.

I. Introduction

Given a directed graph $G = (V, E)$ and a pair of vertices (u, v), $u, v \in V$, a *reachability query* asks if there is a path from u to v. Answering reachability queries efficiently has been a key area of research for several years because of its importance and applicability in a wide range of domains such as social networks, RDF and XML databases, bio-informatics, model verification, etc.

Most practical solutions to this problem propose graph indexes for fast reachability computation, see, e.g., [10], [13], [16], [50], [56], [57]. These methods construct an index on a preprocessed graph obtained by first collapsing the maximal strongly connected components to generate a *condensed graph* which is a DAG. While this works well for a graph that is not updated, or updated infrequently, for large-scale dynamic graphs –such as social networks– such an approach is not practical. The reachability problem in the dynamic setting has been studied extensively in the theoretical literature [31], [32], [44], [45], however it is not clear if the proposed methods scale to handle graphs with millions of vertices [34]. On the practical side, a few systems [9], [46], [58] have focused on efficiently handling updates to large-scale graphs, but as we show in our experiments, they still fall short of the required performance.

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

More important than performance shortfalls is the fact that none of these index-based methods can support additional constraints or specialized reachability semantics which crop up in many network analytics settings. For example, consider the increasingly important problem of supporting reachability queries over *temporal graphs*, i.e., graphs in which edges and vertices have start and end times corresponding to the evolution of the graph and queries may pertain to state of the graph in a time point or intervals in the past [33] [54] [55]. One can assign different reachability semantics in these settings (we will discuss them in section IV), and there is no single indexing technique that can support all the required semantics, support high dynamics and still be efficient in time and space usage.

A. Our Approach

In this paper, we propose ARROW (Approximating Reachability using Random-walks Over Web-scale graphs), an approach that addresses *all* the issues listed above. Our approach is seemingly counter-intuitive: we abandon the traditional methods of precomputing and storing an index, and instead compute reachability *on the fly* at query time, directly on the graph, by running a set of random walks that we will describe in detail later in this paper.

Our approach not only overcomes the limitations of index-based methods we have described above, it additionally offers the following advantages: First, we do not preprocess the graph in any way (such as condensing the graph). Since we do not have a separate index we have *no overhead* of index storage or maintenance. Second, our technique is very general and is able to handle several different kinds of reachability semantics. As an example, we show that ARROW is a unified and practically implementable strategy to solve multiple classes of reachability queries on *temporal graphs* (that is, graphs in which edges are "live" only in certain intervals).

And finally, ARROW is relatively easy to implement on top of modern graph storage and analytics platforms. It relies only on the availability of a random walk primitive which is usually already available in most graph processing systems or can be implemented easily. Thus ARROW can leverage all the attractive features of the underlying computing framework – graph compression, distributed computing, etc.

At first it might appear that this is a not a particularly novel approach since the utility of random walks in approximating the reachability has been explored in the past. However this has been done only in the theoretical literature, largely for *undirected* graphs [20], [23], [52], and dynamic graphs whose update model makes them similar to Erdös-Renyi random graphs as time passes [2] , which have been shown to not reflect properties typical of real world graphs [18]. In our work, on the other hand, we identify network properties that allow ARROW to obtain high quality results, and provide empirical evidence that many real-world networks possess such properties. The computational complexity community has also taken great interest in the s-t-connectivity problem, but most of the efforts in this area have been in solving the problem exactly with space that is comparable to logarithmic with high runtime complexity (see, e.g., [22]).

In practical settings where reachability must be efficiently computed on large and rapidly evolving graphs, ARROW is useful due to its low memory footprint and instantaneous updates. For example, it can be used to implement access control in an online social network, where the content uploaded by a user U shall be accessible to a requester R only if U can reach R via friendship or trust links [17]. ARROW is free of false positives, and thus the danger of R being able to access the contents of U while not having permission to do so is averted, the only cost being that a very small fraction of valid requesters may be wrongfully denied access. In other applications involving highly dynamic networks such as the Internet of Things (IoT) [3], [24] and real-time financial transactions [19], it is important to handle reachability queries between two vertices in order to identify if there is a communication path between devices to potentially trace a security breach or to track the flow of funds to known sensitive agents (e.g., terror funding suspect). Given the high incentive to identify reachability in these highly dynamic networks, high-accuracy efficient methods such as ARROW can be very attractive.

In summary, to the best of our knowledge, our work is the first practical attempt to use random walks for directed, dynamic graphs with no restrictions on the models of graph evolution, and a reasonable restriction on space used. Further we extensively evaluate the performance of our methods on large real world graphs.

B. Our contributions and paper organization

1) We present a novel approach called ARROW that uses random walks over *directed* graphs for approximating the s-t reachability at scale (Section II.)

2) ARROW runs a small number ($O(\sqrt[3]{n^2 \ln n})$, where n is the number of nodes in the graph), of random walks limited in length (proportional to the diameter of the graph). In Section II, a detailed theoretical analysis shows that ARROW achieves high accuracy results when the graph possesses certain structural properties and both query vertices can reach each other. Even though the theoretical analysis relies on these conditions, our empirical evaluation in Section V shows that in practice, they are not critical for ARROW to achieve high accuracy results.

3) We demonstrate that ARROW is general enough to handle different kinds of reachability semantics by considering reachability queries on temporal graphs. Edges (and vertices) in temporal graphs are typically associated with a time interval in which the edge is valid and reachability queries on these graphs come in a variety of flavors, that makes their computation more complex. We discuss how ARROW can be adapted to this setting in Section IV.

4) We present an extensive experimental evaluation using real-world graphs with up to millions of vertices and edges. These experiments show that the reachability estimates by ARROW are highly accurate – close to 100% under almost all query configurations, while requiring less than 1-10 milliseconds (Sections VI and VII.)

We discuss related work in more detail in Section VIII and conclude in Section IX.

II. ARROW

In this section, we introduce our algorithm ARROW, which is based on conducting multiple random walks of fixed length from both the source and destination vertex at query time. Instead of a large index, using random walk based techniques on rapidly evolving graphs is advantageous because of their locality and robustness to changes [4]. We begin with an analysis of the properties of ARROW and then show how it is used in the presence of a graph stream.

Algorithm 1 describes ARROW. Before delving into the details of the algorithm and the parameters on which it depends, we first give an intuitive explanation of how the algorithm works. Given a reachability query (u, v) on a graph snapshot $G_t(V_t, E_t)$ at time t, with $|V_t| = n$, ARROW constructs 2 sets of 'stops' – $F(u) = \{s : u \rightsquigarrow s\}$ of nodes that are reachable from u and $B(v) = \{t : t \rightsquigarrow v\}$ of nodes that can reach v. If $w \in F(u) \cap B(v)$, then u and v have a path via node w and the algorithm reports `true`. If $F(u) \cap B(v) = \emptyset$, ARROW returns `false`. This scheme is similar to existing 2-HOP labeling indexes, but with two important differences. First, $F(u)$ and $B(v)$ are constructed in a *non-deterministic* manner using *random walks*. Second, neither set of stops is part of a *stored index*, but are instead computed at query time. The immediate consequences of the latter is that ARROW has *no overhead* of index maintenance in the dynamic graphs setting.

A. Random Walks

The number and length of the random walks conducted for constructing both $F(u)$ and $B(v)$ are critical choices that affect both the efficiency and accuracy of the algorithm (see lines 2 and 4 in Algorithm 1). ARROW conducts $r = c_{\text{numWalks}} \times \sqrt[3]{n^2 \ln n}$ random walks, where n is the number of nodes in the graph and c_{numWalks} is a parameter to be set. Each walk is of length $l = c_{\text{walkLength}} \times diam$ from u on G_t, where $diam$ is the diameter of the graph and $c_{\text{walkLength}}$ is a parameter to be set. $F(u)$ is the union of the set of all nodes encountered in all of these walks. Similarly, the set $B(v)$ is constructed by conducting r random walks, each of length l from v on the transpose of G_t, i.e., by walking from v along the edges in the direction *opposite* to their orientation in G_t.

Algorithm 1: ARROW

```
Algorithm ARROW($G_t$, $u$, $v$)
    $wL \leftarrow c_{\text{walkLength}} \times diam$;
    $F(u) \leftarrow B(v) \leftarrow \varnothing$;
    for $w$ in $1 \ldots c_{numWalks} \times \sqrt[3]{n^2 \ln n}$ do
        currentNode $\leftarrow u$;
        for $s$ in $1 \ldots wL$ do
            neighbour $\leftarrow$ random neighbour of currentNode;
            $F(u) = F(u) \cup \{$neighbour$\}$;
            currentNode $\leftarrow$ neighbour;
        end
    end
    /* Repeat for $v$ in $G_t^T$ to construct $B(v)$ */
    return $F(u) \cap B(v) \neq \varnothing$;
```

Intuitively, we need to ensure that the random walks are "long enough" from both u and v so that $F(u) \cap B(v) \neq \varnothing$ if v is reachable from u. But when this condition is satisfied, a single set of random walks run from either side has low probability of intersecting, even when started from two nodes in the same strongly connected component. So we need to run a sufficiently high number of walks to boost the probability of success. Clearly, there is a tradeoff between efficiency and accuracy. Accuracy suffers if the random walks are too few or too short, but increasing the number and length of the random walks makes the algorithm less efficient.

1) Sources of false negatives: Note that ARROW has no false positives (that is, for a reachability query (u, v), ARROW never returns `true` if v is *not reachable* from u), but false negatives may occur (that is, if v is reachable from u, ARROW could return `false` instead of `true`). In order to minimize the false negative probability we need to construct $F(u)$ and $B(v)$ in such a way that the probability of them overlapping is high enough. One obvious way of doing this is by running a larger number of walks which reduces the false negative probability at the cost of greater query latency. However running a larger number of random walks is not much use if the distributions of the stops collected in $F(u)$ and $B(v)$ are very different, e.g., if the distance from u to v is k and we walk less than $k/2$ steps from either side then the probability of overlap is zero and so we get a false negative probability of 1. In order to ensure that we can get a handle on the false negative probability we require that the random walks are long enough to "mix" to close to their stationary distributions. If these distributions overlap sufficiently, as is likely to happen with most real-world graphs, we are in a position to bound the probability of false negative. Here too an issue of latency arises, since longer walks take more time. But an even more serious caveat applies here: when both query vertices belong to the same strongly connected component (as is likely in graphs with a dominant SCC), longer walks suffer from a higher probability of escaping the SCC, thereby *increasing* the false negative probability. This too needs to be managed.

B. Theoretical bound on number of walks

The theoretical underpinnings of the accuracy of ARROW are related to the well-known Birthday Paradox that often appears when studying collision in hashing. Stated in terms of balls and bins we ask the question: suppose we throw equal numbers of red and blue balls into n bins, what is the probability of overlap? And the related question of how many balls do we have to throw to ensure that there is a bin with a red and blue ball with high probability?

To relate this question to the query that asks if vertex v is reachable from vertex u: The red balls here are the stops reachable from u and the blue balls are those stops reached from v by random walks in the reverse direction. If a common stop is found we declare that v is reachable from u. A false negative occurs when v is, in fact, reachable from u but the two sets of walks do not choose a common stop. When we view the balls and bins problem in this light we see that we have to handle the situation where the blue balls and red balls are thrown into the bins according to different distributions since the backward and forward walk may have different distributions. Clearly in such a situation the two distributions must have reasonable overlap for us to be able to find a common stop without having to throw an unreasonably high number of balls (which would correspond to a large number of walks and make our query latency unacceptably high).

Keeping these considerations in mind, we state Proposition 1. This proposition follows as a corollary of a more general theorem that connects the balls-and-bins argument to random walks. We omit the general theorem here due to lack of space and discuss only the proposition (see [49] for details including proofs). Here all the nodes of the graph are considered to be bins of the balls-and-bins experiments. The query is "is v reachable from u?", where u and v are vertices of a strongly connected component of a directed graph. We run a random walk from u dropping blue balls in each vertex/bin the walk visits. We run a "reverse" random walk by traversing the edges in the opposite direction from v, dropping red balls in the bins/vertices visited. Clearly reachability is demonstrated if there is a bin with both a red and a blue ball. The proposition says that there will be such a bin with high probability as long as the walks are long enough, where the length is related to the mixing times of the walks.

Proposition 1. *Given a strongly connected graph G with vertex set V, such that $|V| = n$, let the random walk conducted on the directed edges in the forward direction have transition matrix $\overrightarrow{P}$ and stationary distribution $\overrightarrow{\pi}$, and the random walk on the directed edges traversed in the reverse direction have transition matrix $\overleftarrow{P}$ and stationary distribution $\overleftarrow{\pi}$. Given two integers $k, t^* > 0$, consider two collections of random walks of length t^*, $\{(X_t^i)_{t\geq 0}, X_0^i = x : 1 \leq i \leq k\}$ started from $x \in V$ conducted according to $\overrightarrow{P}$, and $\{(Y_t^i)_{t\geq 0}, Y_0^i = y : 1 \leq i \leq k\}$ started from $y \in V$ conducted according to $\overleftarrow{P}$ such that all the random walks are independent. Further, define*

$$S_x = \{X_{t^*}^i : 1 \leq i \leq k\}, S_y = \{Y_{t^*}^i : 1 \leq i \leq k\}.$$

If

$$t^* \geq \max\left\{ t_{mix}^{\overrightarrow{P}}\left(\frac{1}{\sqrt{2}n}\right), t_{mix}^{\overleftarrow{P}}\left(\frac{1}{\sqrt{2}n}\right)\right\}, \tag{1}$$

then $P\{S_x \cap S_y \neq \varnothing\} \geq 1 - 1/n$ if

$$k \geq k^* = \left\{\frac{16n^2 \ln n}{(\alpha(\overrightarrow{\pi}, \overleftarrow{\pi}))^2}\right\}^{\frac{1}{3}}, \tag{2}$$

whenever

$$k^* \cdot \left(\max\{\max\{\mu(i) : 1 \le i \le n\}, \max\{\nu(i) : 1 \le i \le n\}\} + \frac{1}{2n} \right) < 1, \tag{3}$$

where

$$\alpha(\overrightarrow{\pi}, \overleftarrow{\pi}) = n \left(\sum_{i=1}^{n} \max\left\{0, \overrightarrow{\pi}(i) - \frac{1}{2n}\right\} \cdot \max\left\{0, \overleftarrow{\pi}(i) - \frac{1}{2n}\right\} \right).$$

In particular, if $\alpha(\overrightarrow{\pi}, \overleftarrow{\pi})$ *is a constant then* $k^* = \theta\left(\sqrt[3]{n^2 \ln n}\right)$.

Proposition 1 establishes that if the random walks are long enough to mix to close to their stationary distribution then $r = c_{\text{numWalks}} \times \sqrt[3]{n^2 \ln n}$ is a pessimistic upper bound on the number of walks required to obtain the correct answer with high probability, where c_{numWalks} is a tunable parameter that helps us manage the trade-off between accuracy and query latency. Note that r grows faster than $\sqrt[3]{n^2}$ and so is a large number of walks. Hence although some walks may escape the strongly connected component containing u and v, a sufficient number of walks will still contribute to finding the common vertex/bin that demonstrates reachability.

Although Proposition 1 holds for graphs with certain structural properties such as high assortativity that results in $\alpha(\overrightarrow{\pi}, \overleftarrow{\pi})$ to be large enough, we show empirically in Section V that ARROW performs well in practice even for graphs that *a priori* appear to possibly not satisfy these constraints. In Section V, we specifically select such graphs and show the effectiveness of ARROW on them. Aside from structural properties, Proposition 1 also relies on both query vertices belonging to the same SCC, and therefore is not a bound on the accuracy of ARROW in the general case, even though a significant fraction of reachability queries on graph datasets with a giant SCC will satisfy the criterion of Proposition 1. Nevertheless, our analysis of ARROW in Section V does not rely on query vertices belonging in the same SCC, and is performed on queries randomly generated out of positive query pairs in real graph datasets.

C. *Determining walk length*

We estimate the number of steps we need to walk by observing that to mix to close to stationarity a random walk must walk at least as many steps as the weak diameter of the strongly connected component it is walking within.

Mixing in discrete time non-reversible ergodic Markov chains, such as random walks on general strongly connected directed graphs, was characterised by Fill [21] in terms of the spectral gap of the multiplicative "reversiblization" of the random walk. Without going into technical details, Fill defines a random walk that takes two steps at a time, one in the direction of the edges of the graph and one in the *reverse* direction defined in such a way that the stationary distribution of this two-step chain is the same as the stationary distribution of the original random walk on the directed graph, π, with the advantage that this new walk is now reversible. From Theorem 2.1 of [21] we get that if P is the transition matrix of a random walk on a graph and λ_1 is the second largest eigenvalue of its multiplicative reversiblization then the time taken for the walk to mix within distance $1/n$ from stationarity is given by

$$t_{\text{mix}}(\varepsilon) \le c\left(\frac{1}{\lambda_1} \ln\left\{\frac{n}{\pi_{\min}}\right\}\right),$$

where $\pi_{\min}$ is the minimum value of the stationary distribution over all the nodes of the graph and c is some constant, i.e. *the mixing time of the walk on the original directed graph is within a constant of the mixing time of a walk on a specific undirected graph, the reversiblization.*

How do we relate this to the diameter of the graph? For reversible random walks we know (c.f. Theorem 12.5 of [37]) that the time taken to mix to within distance $1/n$ of stationarity is lower bounded as

$$t_{\text{mix}}\left(\frac{1}{n}\right) \ge \left(\frac{1}{1-\lambda_1} - 1\right) \ln n.$$

Slightly modifying an argument of Chung [14], Spielman [51] shows that if λ_1 is the second largest eigenvalue of a lazy random walk on a d-regular graph then the diameter of the corresponding graph is upper bounded by $\ln n/(1-\lambda_1)$, which exactly corresponds, at least for the case of regular graphs to the lower bound on time taken to mix to within $1/n$ of stationarity. This suggests that *to mix to within* $1/n$ *of stationarity, we have to walk for at least the diameter of an (undirected) graph.*

Therefore, the length of the random walks required to mix adequately must be ≥ the so-called *weak diameter* of the graph – the longest shortest path in the graph measured while ignoring edge directions. We note one interesting fact: although the forward and backward walk on a strongly connected directed graph have different transition matrices and may have different stationary distributions, they have one thing in common: the graphs they both walk on have the same *strong diameter*, which is the length of the longest shortest path in the original directed graph we have queried. With this fact in hand we choose *diam*, the strong diameter of the directed graph as the walk length for both walks. This diameter is clearly an upper bound on the weak (undirected) diameter of the graph. In fact, if the graph is not strongly connected then this diameter could be much larger than the weak diameter of the SCC we are walking in. To ensure a conservative walk length, we select $wL = c_{\text{walkLength}} \times$ *diam*, where $c_{\text{walkLength}}$, like c_{numWalks} is a tunable parameter that helps balance accuracy and query latency.

1) *Estimating the diameter:*

Computing the value of *diam* exactly requires computing all-pairs shortest paths in the graph, which is prohibitively expensive for most real world graphs. We adopt a straightforward heuristic to quickly estimate the diameter of the graph – select the longest distance recorded across 10 rounds of BFS from randomly selected nodes on G_t. This approach is both fast and returns diameters that work well with $c_{\text{walkLength}} \le 1.5$, as illustrated in Sections V, VI, and VII. In [8], the diameter of a scale free network generated using the Barabási-Albert model [5] was shown to asymptotically be $\log n/\log(\log n)$. As we observe in Section VI-B, the diameter of real-world directed graphs is generally larger than this estimate.

2) Graph Constraints: The high probability result of Proposition 1 depends on two technical conditions. The first is (3) that constrains the probability of the vertices with the highest stationary probability in both the forward and backward random walks. For graphs that have maximum in-degree and maximum out-degree that are small compared to the size of the graph, we expect that this constraint is relatively mild.

Second, to be able to run $\theta\left(\sqrt[3]{n^2 \ln n}\right)$ walks, $\alpha(\overrightarrow{\pi}, \overleftarrow{\pi})$ must be relatively high, and not decrease as the size of the graph grows. In general we cannot always expect this to be the case, especially in networks where in-degree and out-degree are negatively correlated (i.e. where low in-degree implies high out-degree and vice versa). However, we see that most real-world networks do not have this kind of negative correlation, with the exception of certain special cases like Twitter.

3) The problem of escape from SCCs: Proposition 1 clearly specifies the number of random walks needed from both query nodes for an overlap to be ensured with high probability. However, the directed graph may not be strongly connected, and a random walk started from within an SCC will *never* return to it if it leaves the SCC and enters another part of the graph. This leads to the question: *how do we ensure that there are enough stops collected within the SCC so that our false negative rate for pairs of nodes within the SCC is low?*

For small SCCs which are more highly connected to the rest of the graph than they are within themselves we cannot argue that false negatives will be eliminated completely, but for larger SCCs with relatively narrow escape routes into the rest of the graph, a large fraction of the walks will stay within the SCC. To quantify this, let us say that a walk begun from a vertex v and run for t steps stays within its SCC with probability $p(t,v)$. Proposition 1 says that we need $\theta\left(\sqrt[3]{n^2 \ln n}\right)$ walks to stay within the SCC for false negatives to be eliminated with high probability. If we conduct $c\sqrt[3]{n^2 \ln n}$ independent walks for some appropriately chosen c, then we can argue by Chernoff bounds that $c'\sqrt[3]{n^2 \ln n}$ of them stay within the SCC with probability $1 - 1/n$ if

$$p(t,v) \geq \frac{\ln n}{\sqrt[3]{n^2 \ln n}}.$$

This bound decreases with n faster than $n^{-2/3}$ and, so, is a relatively modest requirement, especially from those vertices that are deep within the SCC. Clearly $p(t,v)$ drops as t increases, and so a balance has to be maintained between how many steps we walk and how many walks we conduct. For the same walk length, a larger value of c (or c_{numWalks}) minimizes false negative probability, at the cost of higher query latency. The choice of c_{numWalks} therefore depends on the requirements of the application using ARROW.

D. Complexity

The sizes of sets $F(u)$ and $B(v)$, and the time taken to construct and intersect them are all $r \cdot l$, where r is the number of random walks and l is the length of walks, as defined in Section II-A. Therefore, query answering time is $\mathcal{O}\left(\sqrt[3]{n^2 \ln n} \cdot diam\right)$.

III. ARROW on Dynamic Graphs

The analyses presented in Section II for a snapshot of an evolving directed graph carries over to the fully dynamic setting as well. We note that the length of the random walks conducted by ARROW depends on the estimated diameter of the graph. In the case of dynamic graphs, ARROW estimates the diameter for the initial graph, and the random walks for all future queries use the estimated diameter of the initial graph. Since most evolving graphs demonstrate shrinking behavior as updates arrive [36], the initial diameter is likely to be an upper bound on the diameter of the graph for all subsequent snapshots. Therefore, the performance of ARROW is not expected to decline as the graph evolves. We confirm this empirically in Section VI.

IV. Temporal Graphs

In this section, we show that ARROW is a *general* algorithm that can be easily adapted to graphs with different kinds of reachability semantics. In order to do so, we consider reachability queries on temporal graphs.

A temporal graph explicitly encodes the history of the evolving graph as intervals on the edges. Various definitions have been made to capture this idea [11], [25], [27], [43], [53], [54]. We consolidate these into the following general definition: A temporal graph $G(V,E)$ is a directed graph with temporal information associated with each node and edge. $\mathcal{T}$ is the lifespan of the graph – the set of time points that G spans. For each node $u \in V$, $T(u) \subseteq \mathcal{T}$ is the set of time points during which node u is active. Similarly, for all $e \in E$, $T(e) \subseteq \mathcal{T}$ is the set of time points during which edge e is active. Although we work in practice with a discrete notion of time the sets $T(u)$ for a vertex u and $T(e)$ for an edge can also be viewed as unions of disjoint contiguous time intervals.

For each time point $t \in \mathcal{T}$, we define the graph snapshot at time t as $G_t(V_t, E_t)$, where $V_t = \{v \in V, t \in T(v)\}$ and $E_t = \{e \in E, t \in T(e)\}$. The transpose of a temporal graph G is G^T, where the direction of each edge $e \in E$ is reversed, but $T(e)$ remains the same.

Reachability queries on temporal graphs may have different forms due to the different types of temporal constraints placed on the paths between nodes. For reachability queries on temporal graphs, ARROW builds on the algorithm described in Section II. The random walks are conducted under the specific temporal constraints of the query, and each 'stop' collected in $F(u)$ and $B(v)$, is a node along with temporal information of the random walk that reached that node. The intersection of two stops $s_1 \in F(u)$ and $s_2 \in B(v)$ checks if s_1 and s_2 have the same node, and the temporal information at both stops satisfy the temporal constraint imposed by the query.

A. Temporal Queries

Chained Queries: A chained query on a temporal graph between the nodes u, v, and on the query time interval q_t evaluates to `true` if there is a path from u to v such that each edge in the path starts after its predecessor has ended, and also lies within the interval q_t. In a transport network, for example, edge intervals typically indicate departure and arrival times, or departure time and duration. To plan a viable path in such a graph between nodes u and v within a time period q_t, one would perform a chained query. This type of path has been termed temporal paths, or journeys, in [11], [43], [54].

ARROW is modified to answer temporal queries by applying simple modifications to the way random walks are performed in order to cater to the temporal constraint. We allow each random walk to maintain a time interval, and at every step, a random walk moves along a sampled edge e if $T(e)$ and its current interval satisfy the temporal constraint imposed by the query.

For chained queries, the interval of each walk is initialized to only one time point, the start time of q_t for walks originating at u and end time of q_t for backward walks originating at v. At each step, a forward random walk with current interval w_t, samples and traverses along an edge e only if $T(e)$ starts after $w(t)$ ends and $T(e)$ ends before q_t ends. Since no such edge may exist, we restrict each random walk with a *budget* which is the maximum number of random draws it can make. Therefore, a walk ends whenever it has either traversed $l = c_{\text{walkLength}} \times diam$ edges or exhausted its budget. When a forward walk is at node x with current interval w_t, the stop (x, w_t) is included in $F(u)$. If the walk takes edge e, then $w(t)$ is updated to $[min(w_t), max(T(e))]$. An analogous procedure is employed by the backward walks to collect $B(v)$. The set,

$$I = \{(x, t_x) : (x, t_1) \in F(u), (x, t_2) \in B(v), (max(t_1) \leq t_x \leq min(t_2))\},$$

is the intersection of the sets $F(u)$ and $B(v)$. The chained query evaluates to `true` if $I \neq \emptyset$, `false` otherwise.

Snapshot Queries: A snapshot query asks if u can reach v in one or more snapshots of the temporal graph G. A c-Snapshot query asks if u can reach v in at least c snapshots of G lying within a given time interval. A c-Snapshot query is useful when analyzing social networks such as Facebook or Twitter, where users are connected via friendships or follower-followee links. Given a time period $q_t = [t_1, t_2]$, and an influential or controversial node u in the graph, a snapshot query can be used to retrieve the nodes connected to u in any non empty sub-interval of q_t.

We adapt ARROW to answer snapshot queries by allowing each random walk to maintain a time interval, like in chained queries. Each walk starts with the query interval q_t, and traverses a sampled edge e only if $T(e)$ and $w(t)$ have a non-empty intersection, in which case $w(t)$ is updated to $w(t) \cap T(e)$. The 'stops' in $F(u)$ and $B(v)$ are (vertex, interval) pairs, and a stop $(w, t_1) \in F(u)$ indicates that u can reach w in *all* graph snapshots lying within the interval t_1. Similarly, a stop $(x, t_2) \in B(v)$ implies that x can reach v in all graph snapshots lying within the interval t_2. The set,

$$I = \{(x, t_x) : (x, t_1) \in F(u), (x, t_2) \in B(v), (t_x = t_1 \cap t_2)\},$$

includes the stops where the forward walks from u and backward walks from v meet. If the set of snapshots included in I has cardinality $\geq c$, then the snapshot query evaluates to `true`, else `false`.

V. Experiments on Graph Snapshots

In Section II we indicated that the performance guarantees of ARROW are constrained by the structure of the graph. In this section we discuss this further and establish that our method works well for real-world graphs, even those that might be suspected to not satisfy the constraints.

The effect of discrepancy between the forward and the backward walks: ARROW works by running two kinds of random walks, one in the "forward" direction, i.e. along the orientation of the edge of the graph and the other in the "backward" direction. In general these two random walks may have different stationary distributions, we call them $\overrightarrow{\pi}$ and $\overleftarrow{\pi}$. ARROW cannot deliver on accuracy if these two distributions put most of their probability on different sets of vertices. For our method to work they must have sufficient "overlap" in the sense that $\sum_{v \in V} \overrightarrow{\pi}(v) \overleftarrow{\pi}(v)$ must be large enough. This overlap is large when $\overrightarrow{\pi}$ and $\overleftarrow{\pi}$ are the same. One way this can happen is if the backward walk assigns the same probability to an incoming edge that the forward walk assigns to its outgoing edges. Since the probability of choosing an edge is the inverse of the degree of the nodes, we expect that $\overrightarrow{\pi}$ and $\overleftarrow{\pi}$ are close to each other when the outdegree of neighbours are similar. In the study of complex networks the concept of *assortativity* measures the tendency of nodes to have neighbours with similar degree as themselves [41]. In this section we use assortativity as a measure of the *dissimilarity* of $\overrightarrow{\pi}$ and $\overleftarrow{\pi}$ on the assumption that these two distributions will be very different if the graph has a disassortative degree distribution, i.e., a distribution in which high degree nodes tend to have neighbours with low degree and vice versa. *In particular we will show empirically that ARROW performs well on real-world graphs with very low assortativity.*

The effect of overly concentrated stationary distributions: The other condition needed to prove Proposition 1 states that the maximum probability that both $\overrightarrow{\pi}$ and $\overleftarrow{\pi}$ assign to any vertex must not be too high. Since graph data sets typically report the power law exponent of the graph, we take this quantity as a proxy for the variance of the degree and hence for the variance of the probability distribution, i.e., the higher the power-law exponent the higher the maximum probability of the stationary distribution is likely to be. *In particular we show empirically that ARROW performs well on real-world graphs with high power law exponents.*

A. Setup

In Table I, we have listed networks for each of the 4 configurations of (high/low assortativity (ρ)) and (high/low power-law exponent (γ)). Facebook, Twitter, YouTube, and Google+ are social networks. LinuxReply is an email network, the Wikipedia graph is message communication on talk pages of English Wikipedia, while Libimseti is a dating site. All graphs have been obtained from the Konect repository [35].

For each graph, we generate 1000 positive queries by conducting BFS from high-degree nodes and selecting destination vertices farthest from the source node. Since ARROW is most likely to fail on queries (u,v) where v is reachable from u by a long path, we exclusively measure the false negative rate of

Dataset	$\|V\|$	$\|E\|$	ρ	γ
LinuxReply	63400	1096441	-0.179	1.841
Libimseti	220971	17359346	-0.139	1.911
Wikipedia	2987536	24981163	-0.090	1.811
YouTube	1138500	4942297	-0.036	2.141
Facebook	46953	876993	0.219	4.431
Twitter	465018	834797	-0.877	2.471
Google+	23629	39242	-0.381	2.621

TABLE I: Snapshot Datasets

ARROW on them to evaluate its worst-case performance on such graphs.

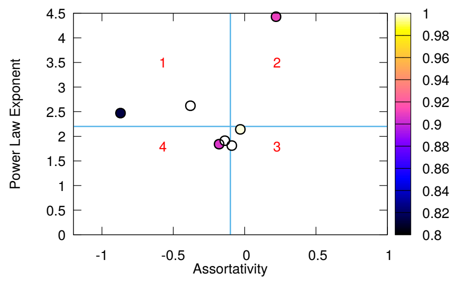


Fig. 1: Accuracy of ARROW with varying Assortativity and Power law exponent. $c_{\text{numWalks}} = 0.1$ and $c_{\text{walkLength}} = 1.5$.

B. Results

Figure 1 shows the accuracy of ARROW. The worst case for ARROW is quadrant 1, with low assortativity and high power law exponent, while the best case is quadrant 3. ARROW's false negative rate is at most 0.2 across all quadrants, for an accuracy ≥ 0.8. This shows that even when not all assumptions on the structure are met by the underlying graph, ARROW still performs well in practice.

Figure 2 shows the accuracy of ARROW with varying c_{numWalks} and $c_{\text{walkLength}}$ for 4 datasets, one from each quadrant of Figure 1. For $c_{\text{walkLength}} \geq 1.5$, accuracy of ARROW is consistently ≥ 0.8. In general it is evident that ARROW is more sensitive to $c_{\text{walkLength}}$. The reason for this follows from the discussion in Section II-B that $\sqrt[3]{n^2 \ln n}$ is a pessimistic upper bound required to obtain correct answers with high probability. Smaller values of c_{numWalks} do not affect the false negative rate of ARROW because the number of walks actually conducted is still large enough.

VI. Experiments on Dynamic Graphs

Dynamic graphs are a sequence of graph snapshots, or alternatively, an initial graph together with a graph stream. At any time, only the *latest* state of the graph is maintained. Updates to the graph consist of edge events – insertion and deletion of edges and node events – addition and removal of nodes. Queries can arrive at any time in the stream, and must be computed on the latest state of the graph.

We evaluate the performance of ARROW, and compare it to an existing method for reachability on fully dynamic graphs – DAGGER [58]. While there have some alternative approaches proposed for reachability since DAGGER, it remains the most flexible and scalable algorithm that can process all types of updates, work on both DAGs and non-DAGs, while at the same time maintain a relatively small index and answer queries efficiently. In addition, we evaluate two baselines - regular BFS and bidirectional BFS (BBFS). In bidirectional BFS, BFS searches are conducted from both source and destination, and the searches are interleaved such that in each iteration both frontiers expand by one step. If the two frontiers overlap at the end of an iteration, the search ends. Interleaving achieves greater efficiency in the case of positive queries where reachability is achieved by a short path.

A. Setup

1) Datasets: We use 3 real world datasets from the Konect repository [35] consisting of online social networks **Epinions** and **Flickr**, and an email communication network from **Enron**. The **Facebook** dataset is obtained from the authors of [48]. Table II lists the datasets used in our experiments along with their relevant properties.

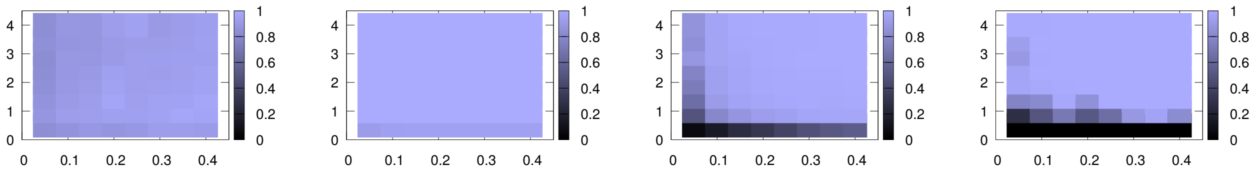
2) Updates and Queries: The update streams for each of the datasets contain edge events only. However, we note that ARROW can process node events with extreme ease (where an 'alive' flag is maintained for each node) and therefore, it is bound to be faster than DAGGER for the events not available in these datasets. For each dataset, we generate 500 reachability queries that include both positive and negative queries interspersed within the update stream. A query or an update is processed one at a time. This implies that a query is answered on the state of the graph which reflects all changes up to the latest update preceding the query. When a query arrives, the graph is locked and no more updates are processed until after the query computation is complete.

3) Metrics: Metrics measured include the average time taken to perform each update operation, time taken to compute a query, and preprocessing time. In the preprocessing step, ARROW estimates the diameter of the graph using multiple rounds of BFS while DAGGER builds its SCC based index. In addition, we measure the *total elapsed time*, which is the total time taken to process both queries and updates.

B. Results

a) Diameter Evolution: The heuristic for estimating the diameter of a directed graph, described in Section II-C1, runs multiple BFS on the initial graph. As the graph evolves, its diameter may or may not change. For each dataset, we apply the heuristic to estimate the diameter after every $p = 1, D/10, D/100$ snapshots, where D is the total number of *distinct* snapshots in the graph (Table IV). Figure 3 shows the evolution of estimated diameter on each of the dynamic graphs for varying values of the period p, and compares it to the asymptotic diameter of a scale free network [8] with the same number of nodes. With the exception of Enron, the estimated diameter remains fairly stable through the lifespan of the graph. Additionally, periodically re-estimating the diameter with larger values of p is sufficient to closely track the changes in each dataset. We conclude that an initial estimate of the diameter, or a periodic reestimation is sufficient to conserve both accuracy and efficiency of ARROW as the graph evolves.

b) ARROW Performance: Table III shows the comparison of ARROW, DAGGER, BFS, and BBFS on each of the datasets. DAGGER, inspite of having very fast queries, times out on 3 out of 4 datasets due to the extremely high

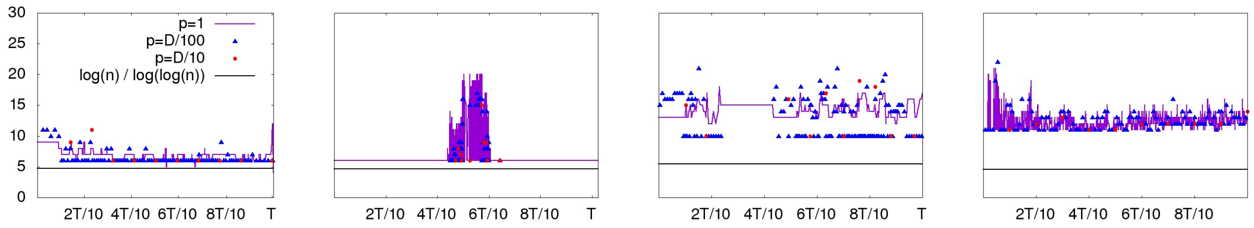


(a) Linux (b) Wikipedia (c) Facebook (d) Twitter

Fig. 2: Sensitivity of the accuracy to c_{numWalks} (x-axis) and $c_{\text{walkLength}}$ (y-axis)

Dataset	$\lvert V\rvert$	$\lvert E\rvert$(Initial)	# Inserts	# Deletes	# SCCs (Initial)	Size of Largest SCC (Initial)	*diam*
Epinions	131828	578996	262376	841372	98269	32053	12
Enron	87274	458853	688409	1147262	83054	4191	17
Flickr	2302936	17034806	16105211	33140017	322279	104426	19
Facebook [48]	63732	75	905490	90555	63721	3	14

TABLE II: Dynamic Graph Datasets



(a) Epinions (b) Enron (c) Flickr (d) Facebook

Fig. 3: Diameter evolution of dynamic networks (Timestep on x-axis; Estimated Diameter on y-axis)

update time. Notably, both insert and delete operations are about 3-4 orders of magnitude slower with DAGGER than with ARROW, while queries are at most 1 order of magnitude slower in ARROW than in DAGGER.

Note that DAGGER is an index based exact method, and in a setting with a large graph and high update rate, it is at a disadvantage for both update computation time and memory, as is reflected in Table III. On the other hand, while BFS and BBFS both have negligible update times, like ARROW, their query times are at least twice that of ARROW in most cases. While BBFS has slightly faster query times than BFS for positive queries, it is slower for negative queries, and thus fails to have a significant improvement.

VII. Experiments on Temporal Graphs

In this section we adapt ARROW for different temporal queries and compare simple adaptations of ARROW against specialized index-based methods built for each query type.

A. Setup

1) Datasets: The temporal datasets used in these experiments correspond to those used as dynamic graphs in Section VI. In Table IV, we additionally report the total lifespan of the dataset (range of timestamps on the edges), average duration of an edge, and the number of distinct snapshots. The value of T, or the lifespan, is also the total number of possible graph snapshots, where some consecutive snapshots may be identical due to the lack of any event. While it is possible to compress the timestamps such that they map to a contiguous range of numbers and alter queries accordingly, such compression loses information regarding duration of edges.

2) Competitors: No single algorithm or system works for all the query types we consider in this paper so we establish a baseline based on BFS for each of these queries. Rather than track the set of vertices already visited, our BFS-based baselines keep track of the (vertex, timestamp) pairs which have already been 'discovered', i.e. temporal paths from source to the vertex have been discovered for the timestamp. In each step, the temporal BFS examines the edges of a discovered (vertex, timestamp) pair and discovers further temporal paths according to the specific temporal constraint. BBFS simply conducts temporal BFSs from both the source and destination nodes. If the searches meet at a given vertex, i.e. a pair (v, t_1) is discovered from the source, and a pair (v, t_2) is discovered from the destination node, t_1 and t_2 are compared to check if they satisfy the temporal constraint imposed by the query.

3) Query Generation: We generate both positive and negative queries for each query type and dataset using the BFS-based algorithms. Random selection of node pairs sufficed to generate negative queries for all datasets and query types. For positive Snapshot queries, the following methodology was used. Select the most persistent edge (longest temporal length) in the graph, say (u, v) with interval $[t_s, t_e]$. Conduct a BFS on the union of all graph snapshots in the interval $[t_s, t_e]$, and collect the nodes that u can reach in any non-empty sub-interval of $[t_s, t_e]$. From this list we generate at most 100 queries of length $(T, T/5, T/10, T/15, T/20)$. Note that if a node w is reachable from u in an interval i via a snapshot path, then it is also reachable from u via snapshot paths in all sub-intervals of i.

To generate positive chained queries, define the start time of a node u as $start(u) = \min_{e=(u,\cdot)} T(e)$. Select the node

Dataset	Method	Insert(mS)	Delete(mS)	Query(mS)	Init(mS)	Mem (MB)	Total Time(mS)	Accuracy
Facebook	DAGGER	0.22	90.405	0.1655	446.091	4007.7	8244220	
	ARROW	0.00025	0.00049	0.7868	3.437	12.69	**674.99**	0.956
	BBFS	0.000192	0.000321	6.8775	0.00016	13.172	4324.954	
	BFS	0.000143	0.00031	18.8432	0.000138	6.809	10286.772	
Enron	DAGGER	864.27	1.08188	0.0048	604.135	44.04	TIMEOUT	
	ARROW	0.000229	0.000514	0.0468	6.45	18.53	**2818.158**	1
	BBFS	0.000263	0.000627	1.0011	0.00014	18.537	3091.943	
	BFS	0.00021	0.00051	2.1536	0.00027	8.631	3470.022	
Epinions	DAGGER	55.7711	71.0257	2.577	1786.1	1453.75	TIMEOUT	
	ARROW	0.000135	0.000191	6.599943	95.166	17.42	**808.029**	0.998
	BBFS	0.00017	0.00024	14.345296	0.00024	18.086	8198.45	
	BFS	0.00011	0.00013	14.966	0.000086	8.86	8561.26	
Flickr	DAGGER	NA	NA	NA	TIMEOUT	NA	TIMEOUT	
	ARROW	0.000223	0.000532	285.872	4354.84	415.011	**45334.908**	0.986
	BBFS	0.000307	0.000703	463.7533	0.00022	470.299	290706.677	
	BFS	0.000249	0.001004	665.575	0.000149	208.584	405832.625	

TABLE III: Dynamic Graph Results (Timeout = 2 hours, c_{numWalks} = 0.01, $c_{\text{walkLength}}$ = 1)

with the smallest start time and conduct a *Chained BFS* from it. In a Chained BFS from source s, each node u has an *arrival time* $arr(u)$. $arr(s)$ is initialized to 0, while all other nodes' initial arrival time is ∞. Chained BFS explores an edge (x, y) only if its start time is greater than the $arr(x)$ and its end time is less than the $arr(y)$. If so, $arr(y)$ is updated to the end time of edge (x, y). Note that if a node w is reachable from u in an interval i via a chained path, then it is also reachable from u via chained paths in all super-intervals of i. We use this to generate positive queries of length $(T, T/5, T/10, T/15, T/20)$ from u. Repeat Chained BFS from the node with next smallest start time until enough positive queries have been generated. While this procedure is sufficient to generate the overall required number of positive queries, it is unable to generate positive queries for each possible query duration. We note that in our datasets, positive queries may or may not exist for certain query durations. Since a Chained BFS exploration from each node in the graph and all potential query intervals is a prohibitively expensive procedure, we report our results only on the positive queries generated using the above process.

4) Parameters: The length of the query interval directly affects the number of edges considered in the query answering algorithms, and therefore the query answering time. We vary the query interval in the set $(T, T/5, T/10, T/15, T/20)$, where T is the total number of graph snapshots in the dataset. Unless indicated otherwise, the value of c_{numWalks} = 0.01, $c_{\text{walkLength}}$ = 1, and c_{budget} = 3.

5) Metrics: We report the time taken to answer queries (query time), time taken for preprocessing as well as graph and index creation in memory (construction time), peak memory usage of the program, and accuracy of ARROW.

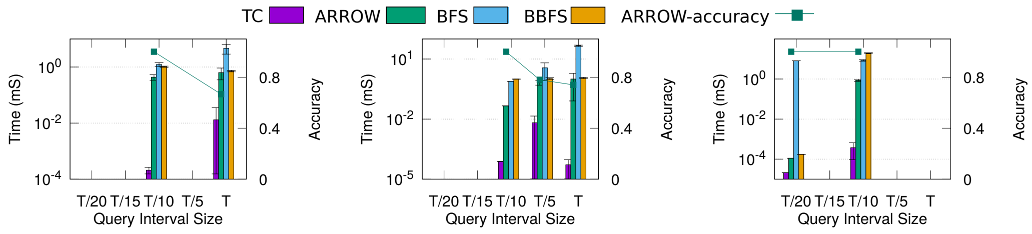
B. TopChain

TopChain, in [54], is used to answer chained reachability queries. We have obtained the C++ code for TopChain from the authors, which does not include support for updates to the temporal graph. TopChain unrolls the graph into a directed acyclic graph such that each vertex has as many copies as the number of distinct timestamps on its incident edges.

Figures 4a and 4b compare the query latencies of BFS, BBFS, TopChain (**TC**), and ARROW for chained queries. Since TopChain builds a temporally unrolled graph, and answers queries based on 2-hop index labels on it, queries are answered much faster (about 2-3 orders of magnitude) than ARROW. In an extension of TopChain [55], a method to update the index that requires $O(k * (|V| + |E|)$ time per update is proposed, where $|V|$ and $|E|$ are the number of vertices and edges in the temporal graph respectively, and k corresponds to the number of labels that is stored for each node. In contrast, ARROW requires constant time to effect updates in the graph. As a result, while query latency is higher for ARROW, in scenarios where updates to the graph must also be supported, the overall cost of ARROW will be much smaller.

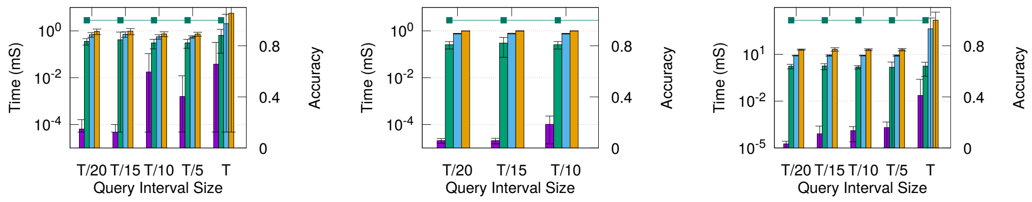
C. TimeReach

The TimeReach index [48] computes conjunctive and disjunctive queries, where the disjunctive queries are identical to the c-Snapshot query for c = 1. It is based on maintaining the strongly connected components for each snapshot of the graph, indicating an explosive memory usage for datasets spanning more than a few thousand graph snapshots. We have obtained Java code from the authors of this paper. To answer a query, TimeReach examines each individual timestep in the query interval to establish reachability between the source and destination nodes, and returns `true` whenever the first such timestep is found. To enable TimeReach to answer c-Snapshot queries for $c > 1$, we update the code to continue examining successive time points in the query interval until the specific constraint is satisfied, or the query interval is exhausted. The code assumes that if the dataset spans the time interval $[t_s, t_e]$, then for each time instant $t_i \in [t_s, t_e]$, there is at least one edge that starts at t_i. Since it is a non-trivial task to transform an existing dataset to conform to the above assumption, we compare TimeReach with our methods on only the Facebook dataset, also obtained from the authors of [48]. Finally, there are two other variants of TimeReach - TimeReach Condensed (TRC) which consumes less memory, and TimeReach Condensed 2-Hop (TRCH), which is a condensed variant using 2-hop index. We were unable to compare with either because of a potential bug in the code of TRC (which we communicated to the authors) and out-of-memory errors on all our datasets for TRCH.



Enron Epinions Flickr

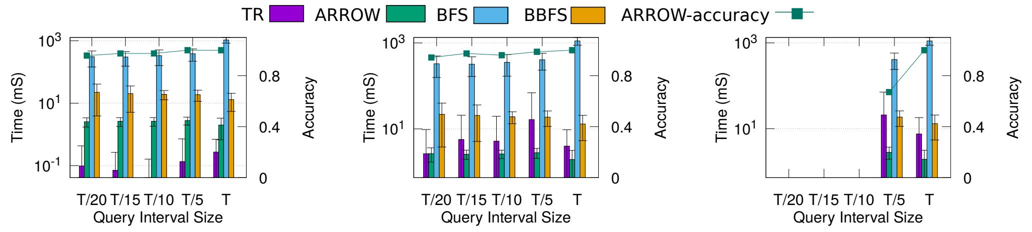
(a) Positive Queries



Enron Epinions Flickr

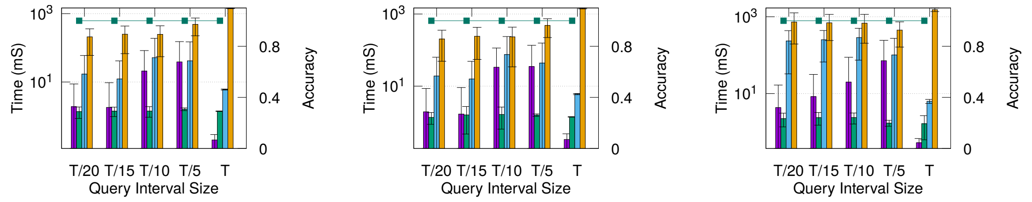
(b) Negative Queries

Fig. 4: Chained Queries



$c = 1$ $c = 10$ $c = 100$

(a) Positive Queries



$c = 1$ $c = 10$ $c = 100$

(b) Negative Queries

Fig. 5: c-Snapshot Queries on Facebook

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	T	Avg. Edge Dur.	# distinct snapshots
Epinions	131828	841372	89109001	7139603.6	1877
Enron	87274	1148072	1401187798	132093607.7	440727
Flickr	2302936	33140017	18632521	1657309.2	267
Facebook [48]	63732	905565	870	259.3	870

TABLE IV: Temporal Datasets

Figures 5a and 5b pertain to true and false c-Snapshot queries on the Facebook dataset. We split the results according to the query result since the performance of TimeReach (**TR**) varies for true and false queries. Note that negative queries constitute the worst case for TimeReach, since each timestamp in the query interval must necessarily be examined. Inspite of this, TimeReach is slower than ARROW for both true and false queries, indicating that even on small datasets with a pre-built index for TimeReach, ARROW can still be faster and highly accurate. As the value of c increases, BFS experiences considerable slow down as it tracks a larger number of intervals. The query latency for TimeReach differs by almost 2 orders of magnitude for $c = 1$ and $c = 10$ with positive Snapshot queries (Figure 5a), since it must continue examining timestamps until c timestamps with reachability have been discovered, and therefore query latency is directly dependent on c. Further, in Figures 5a and 5b, the time taken by TimeReach drops when interval size increases from $T/5$ to T. While this seems counter-intuitive, the query latency for TimeReach depends on a number of factors, including the vertices in the query and the distance between their respective SCCs in the condensed graph. In contrast, ARROW is slightly slower, but mostly consistent for all cases.

D. *Construction Time and Memory*

We compare the construction time (which is the time taken to read in the initial graph, the graph stream, and complete whatever precomputation is necessary) and the total memory consumed by each of the methods. (Note that BBFS has identical memory usage as ARROW and construction time as BFS). In Figure 6a, TimeReach consumes much more memory and construction time than ARROW. In case updates must

		ARROW	TimeReach [48]	TopChain [54]
Query	Snapshot	✓	✓	✗
	Chained	✓	✗	✓
Algorithmic Property	Incremental	✓	✗	✗
	Exact Method	✗	✓	✓
	IndexSize	$\mathcal{O}(1)$	$\mathcal{O}\left(\lvert V\rvert \times T + \sum_{t\in\mathcal{T}} (\lvert V_t^c\rvert + \lvert E_t^c\rvert)\right)$	$\mathcal{O}(\lvert E\rvert)$
	Query Time	$\mathcal{O}\left(\sqrt[3]{\lvert V\rvert^2 \ln \lvert V\rvert} \cdot diam\right)$	$\mathcal{O}\left(\sum_{t\in q} (\lvert V_t^c\rvert + \lvert E_t^c\rvert)\right)$	$\mathcal{O}(\lvert E\rvert)$

TABLE V: Methods for Temporal Reachability.

be supported, it is unclear how well variants of TimeReach will perform with regard to construction time. In Figures 6b, 6c, and 6d, the construction time for TopChain and ARROW are comparable. While TopChain must construct a DAG and its associated index, ARROW must conduct multiple BFS in order to build an estimate for the diameter. Since TopChain stores and uses the unrolled graph and its index, the memory consumed is far higher than ARROW or BFS.

In Table V, to put the strengths of our method in context, we compare the basic characteristics of ARROW to both TopChain [54] and TimeReach [47]. We find that ARROW is the only method that can handle both query classes under consideration. Although ARROW is the only approximate method in this grouping, it takes time sublinear in the size of the graph, requires only a constant amount of additional storage as opposed to the other methods and can handle changes in the graph easily.

To illustrate its simplicity, we have implemented ARROW methods on the Titan graph database and compared it against the BFS baseline for c-Snapshot queries on the Epinions dataset. The results of this experiment are included in the full version at [49] due to space constraints.

VIII. Related Work

a) Reachability on Static and Dynamic Graphs: A number of indexing and query processing strategies have been proposed for reachability [1], [12], [13], [16], [46], [50], [56], [57]. However, since all these strategies focus on static graphs, we do not discuss them in detail, instead directing the reader to [59]. Instead we consider *dynamic graphs*–i.e., the graph representations which maintain only the *latest* state of connections between vertices which are changing with time. Only a few of the reachability indexes proposed for static graphs have been extended to handle dynamic graphs [9], [46], [58]. However, most of them do not scale well for large graphs. A few proposals exist that specifically focus on dynamic graphs [60].

Random walks were used for reachability by Feige [20] who focussed on static undirected graphs and by Anagnostopoulos et. al. [2] whose methods covered a very specific class of dynamic graphs that resemble Erdös-Renyi random graphs in the limit. Our work differs from these because we consider a complex and practical representation of directed temporal graphs and look to answer *time-respecting* reachability queries over them; and, importantly, we focus on the practical aspects of implementing our algorithms. Seemingly related but different is the work on computing the *sizes* of reachability sets by Cohen [15] which does not focus on computing $s-t$ reachability in a highly dynamic setting, as we do.

b) Temporal Graph Models and Temporal Reachability: Kempe et. al., in their seminal work [27], proposed notions of temporal graphs and time-respecting paths over them and showed the problem was NP-complete. Subsequently temporal graphs were largely used to study the dynamics of influence/information spread over networks and for computing time-aware centrality measures [6], [7]. Recently there has been a renewed interest in temporal graphs, with an exhaustive study by Holme and Saramäki [25] being followed by other similar studies [11], [42]. Michail [38] provides a survey of complexity results of various graph algorithms over temporal graphs. Within the data management domain recent work on temporal graphs has addressed three different aspects: first, compact storage of temporal graphs and reconstruction of a graph snapshot at a given time instant in the past [28]–[30], [33]; secondly, on developing a formal algebraic framework for graph queries on the same lines as temporal algebra extensions for SQL [39], [40]; and, finally, on proposing new indexing and query processing strategies for reachability and path problems on temporal graphs [26], [43], [47], [48], [54]. The first two aspects are not directly relevant to the work presented in this paper, since our focus is on indexing and query processing strategies over temporal graphs.

ARROW addresses various types of temporal reachability queries defined in [47], [48], [54] and uses random walk based methods to solve them. We discuss these methods in further detail in Table V and Section VII.

IX. Conclusion and Future Work

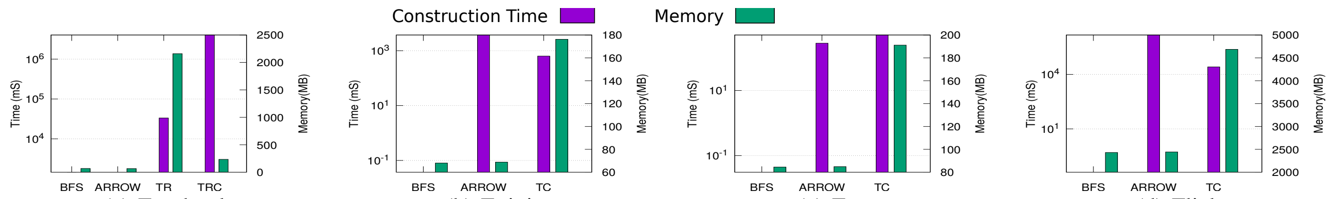
Reachability queries on highly dynamic graphs present a level of complexity that tends to defeat traditional methods that rely on building indexes. In this paper, we propose a unified on-the-fly method of answering them at query time with minimal preprocessing. Our method, ARROW, is based on running multiple random walks from the queried vertices to find witnesses for reachability, and works in $\mathcal{O}\left(\sqrt[3]{n^2 \ln n} \cdot diam\right)$ time. Since the diameter of most real-world networks tends to be very small compared to size of the network, this method can be considered to be sublinear in practical settings.

Future directions of research include extending ARROW for additional types of reachability queries. ARROW can be readily adapted to answer reachability queries for other temporal constraints, such as *Persistent* queries - where reachability must exist during all time points in the query interval. In addition, it can be generalized to answer label-constrained reachability queries for graphs with labels on edges. Label constraints, which may be specified as regular expressions, restrict the set of paths that can be considered for $s-t$ reachability to only those in which the sequence of labels on the edges obey the given constraint. Similar to that for temporal graphs, random walks in ARROW for edge-labeled graphs discard randomly drawn edges that cannot advance their current state with respect to the query constraint, with an overall upper bound on the total number of random draws.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *SIGMOD Record*, 18(2), 1989.

[2] A. Anagnostopoulos, R. Kumar, M. Mahdian, E. Upfal, and F. Vandin. Algorithms on evolving graphs. In *ITCS*, 2012.



(a) Facebook (b) Epinions (c) Enron (d) Flickr

Fig. 6: Construction Time and Memory

[3] L. Atzori, A. Iera, and G. Morabito. Siot: Giving a social structure to the internet of things. *IEEE communications letters*, 15(11):1193–1195, 2011.

[4] C. Avin, M. Koucký, and Z. Lotker. How to explore a fast-changing world (cover time of a simple random walk on evolving graphs). In *ICALP*, 2008.

[5] A. Barabási and R. Albert. Emergence of scaling in random networks. *science*, 286(5439), 1999.

[6] K. Berberich, S. J. Bedathur, M. Vazirgiannis, and G. Weikum. Buzzrank ... and the trend is your friend. In *WWW*, 2006.

[7] K. Berberich, M. Vazirgiannis, and G. Weikum. Time-aware authority ranking. *Internet Mathematics*, 2(3):301–332, 2005.

[8] B. Bollobás and O. Riordan. The diameter of a scale-free random graph. *Combinatorica*, 24(1), 2004.

[9] R. Bramandia, B. Choi, and W. K. Ng. Incremental maintenance of 2-hop labeling of large graphs. *TKDE*, 22(5), 2010.

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

[11] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *IJPEDS*, 27(5), 2012.

[12] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, 2008.

[13] J. Cheng, S. Huang, H. Wu, and A. W. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*, 2013.

[14] F. R. K. Chung. Diameters and eigenvalues. *J. Amer. Math. Soc.*, 2(2), 1989.

[15] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, 1997.

[16] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5), 2003.

[17] I. B. Dhia. Access control in social networks: a reachability-based approach. In *EDBT/ICDT Workshops*, 2012.

[18] S. Dorogovtsev and J. F. F. Mendes. Scaling properties of scale-free evolving networks: Continuous approach. *Phys. Rev. E*, 63, 2001.

[19] W. Eberle, J. Graves, and L. Holder. Insider threat detection using a graph-based approach. *Journal of Applied Security Research*, 6(1):32–81, 2010.

[20] U. Feige. A fast randomized LOGSPACE algorithm for graph connectivity. *Theor. Comput. Sci.*, 169(2), 1996.

[21] J. A. Fill. Eigenvalue bounds on convergence to stationarity for non-reversible markov chains, with an application to the exclusion process. *Ann. Appl. Probab.*, 1(1), 1991.

[22] P. Gopalan, R. J. Lipton, and A. Mehta. Randomized time-space tradeoffs for directed graph connectivity. In *FSTTCS*, 2003.

[23] I. Gorodezky and I. Pak. Generalized loop-erased random walks and approximate reachability. *RSA*, 44(2), 2014.

[24] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[25] P. Holme and J. Saramäki. Temporal networks. *Phys. rep.*, 519(3), 2012.

[26] W. Huo and V. J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, 2014.

[27] D. Kempe, J. Kleinberg, and A. Kumar. Connectivity and inference problems for temporal networks. In *STOC*, 2000.

[28] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, 2013.

[29] U. Khurana and A. Deshpande. Hinge: enabling temporal network analytics at scale. In *SIGMOD*, 2013.

[30] U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. In *EDBT*, 2016.

[31] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *FOCS*, 1999.

[32] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *J. Comput. Syst. Sci.*, 65(1):150–167, 2002.

[33] G. Koloniari, D. Souravlias, and E. Pitoura. On graph deltas for historical queries. *arXiv:1302.5549*, 2013.

[34] I. Krommidas and C. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *J. Exp. Algorithmics*, 12, 2008.

[35] J. Kunegis. Konect: The koblenz network collection. In *WWW*, 2013.

[36] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, 2005.

[37] D. A. Levin and Y. Peres. *Markov Chains and Mixing Times, 2nd Edition*. American Mathematical Society (first edition, 2008), 2017.

[38] O. Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Math.*, 12(4), 2016.

[39] V. Z. Moffitt and J. Stoyanovich. Portal: A query language for evolving graphs. *CoRR*, abs/1602.00773, 2016.

[40] V. Z. Moffitt and J. Stoyanovich. Towards sequenced semantics for evolving graphs. In *EDBT*, 2017.

[41] M. Newman. *Networks: an introduction*. Oxford university press, 2010.

[42] V. Nicosia, J. Tang, C. Mascolo, M. Musolesi, G. Russo, and V. Latora. Graph metrics for temporal networks. In *Temporal networks*, pages 15–40. Springer, 2013.

[43] R. K. Pan and J. Saramäki. Path lengths, correlations, and centrality in temporal networks. *Phys. Rev. E*, 84(1), 2011.

[44] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008.

[45] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM J. Comput.*, 45(3):712–733, 2016.

[46] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, 2005.

[47] K. Semertzidis and E. Pitoura. Time traveling in graphs using a graph database. In *EDBT/ICDT Workshops*, 2016.

[48] K. Semertzidis, E. Pitoura, and K. Lillis. Timereach: Historical reachability queries on evolving graphs. In *EDBT*, 2015.

[49] N. Sengupta, A. Bagchi, M. Ramanath, and S. Bedathur. Arrow: Approximating reachability using random-walks over web-scale graphs. http://www.cse.iitd.ernet.in/~neha/Reachability.pdf.

[50] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*, 2013.

[51] D. Spielman. Lecture 8: Diameter, doubling and applications. Online lecture notes for "Spectral Graph Theory and Applications", Fall 2004, Yale University. URL:http://www.cs.yale.edu/homes/spielman/eigs/.

[52] M. Starnini, A. Baronchelli, A. Barrat, and R. Pastor-Satorras. Random walks on temporal networks. *Phys Rev E*, 85(5), 2012.

[53] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Temporal distance metrics for social network analysis. In *WOSN*, 2009.

[54] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *Proc. VLDB Endow.*, 7(9), 2014.

[55] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke. Reachability and time-based path queries in temporal graphs. In *ICDE*, 2016.

[56] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, 2013.

[57] H. Yıldırım, V. Chaoji, and M. J. Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *VLDB J.*, 21(4):509–534, 2012.

[58] H. Yıldırım, V. Chaoji, and M. J. Zaki. DAGGER: A scalable index for reachability queries in large dynamic graphs. *CoRR*, abs/1301.0977, 2013.

[59] J. X. Yu and J. Cheng. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, chapter 6: Graph Reachability Queries: A Survey. Springer, 2010.

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