Efficiently Answering Regular Simple Path Queries on Large Labeled Networks

Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, Srikanta Bedathur
IIT Delhi, Dept. of Computer Science and Engineering, New Delhi, India
[sarishtwadhwa.1996,anaghprasad95]@gmail.com,[sayanranu,bagchi,srikanta]@cse.iitd.ac.in

ABSTRACT

A fundamental query in labeled graphs is to determine if there exists a path between a given source and target vertices, such that the path satisfies a given label constraint. One of the powerful forms of specifying label constraints is through regular expressions, and the resulting problem of reachability queries under *regular simple paths* (RSP) form the core of many practical graph query languages such as SPARQL from W3C, Cypher of Neo4J, Oracle's PGQL and LDBC's G-CORE. Despite its importance, since it is known that answering RSP queries is NP-Hard, there are no scalable and practical solutions for answering reachability with full-range of regular expressions as constraints. In this paper, we circumvent this computational bottleneck by designing a random-walk based sampling algorithm called ARRIVAL, which is backed by theoretical guarantees on its expected quality. Extensive experiments on billion-sized real graph datasets with thousands of labels show that ARRIVAL to be 100 times faster than baseline strategies with an average accuracy of 95%.

KEYWORDS

reachability query, regular path query, random walks, knowledge graphs, regular expression

ACM Reference Format:
Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, Srikanta Bedathur. 2019. Efficiently Answering Regular Simple Path Queries on Large Labeled Networks. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319882

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319882

1 INTRODUCTION

A fundamental query in labeled graphs is to determine if there exists a path from a given source vertex to a specified target vertex, such that the labels encountered along the path form a string that is from a given *regular language*. Such queries are termed as Regular Path Queries (RPQs) and are often specified with a starting node, a destination node and a *regular expression* over the node and edge labels. For example, in a PPI, does there exist a pathway from protein P_1 to protein P_2 that proceeds only through either cleavage or covalent bonding interactions? Does there exist a cascade of interactions from user U on Twitter to user V such that all intermediate nodes are females of age between 20 and 30? Regular path queries have been studied under different possible semantics: *arbitrary*, *shortest* and *simple* [13]. In this paper, we will focus on the evaluation of RPQs under *simple* path semantics, or Regular Simple Path Queries (RSPQs), which require that in a valid path no node is visited more than once. RPQs form a proper superset of "label-constrained reachability" (LCR) queries (e.g., [11, 21, 23]) wherein the path from source to destination must use labels belonging to a specified subset of the graph's label set.

Limitations of existing techniques: Owing to their wide applications, RPQs have been extensively studied [2, 4, 6, 7, 12, 13, 16]. Nonetheless, several challenges, including scalability to large networks, remain open.

1. They handle a very small subclass of regular expressions. Existing techniques for RSP queries do not support most regular expressions as label constraints [10, 11, 21, 23]. Most existing techniques, including the most recent one by Valstar et al. [21], take a set of labels $L = \{a_1, \cdots, a_k\}$ as input, and every edge (or node) in the path defining the reachability must contain a label from this set L. This constraint reduces to the regular expression $(a_1|\cdots|a_k)^*$. Such a formulation significantly reduces the expressive power of the user in posing effective queries. An analysis of SPARQL query logs on Wikidata17 knowledge graph reveals that $\approx$ 35% of the RSP queries cannot be expressed through this restricted form [5]. Fan et al. [8] propose a technique that is slightly more generic and supports a subclass of regular expressions, which are computationally easier to solve. Fletcher et al.[10] replaces Kleene closure with paths of bounded length recursion. Hence, they operate on a restricted set.

Table 1: Limitations of existing techniques.

Algorithm	Regular Expressions	Non-exponential growth with label space	Query-time label definition	Dynamic Networks
Valstar et al. [21]	Only LCR	x	x	x
Jin et al. [11]	Only LCR	x	x	x
Zou et al. [23]	Only LCR	x	x	✓
Fan et al. [8]	✓ (partially)	x	x	x
Fletcher et al. [10]	✓ (partially)	✓	x	x
Koschmieder et al. [12]	✓	✓	x	x
ARRIVAL	✓	✓	✓	✓

2. Costs grow exponentially with number of labels. Existing techniques to answer LCR queries [8, 11, 21, 23] (which are a proper subset of RSP queries) employ index structures whose memory and computation costs grow exponentially with the number of labels. Consequently, they are unable to scale to networks containing more than a handful of labels. In the real world, this assumption is not true. For example, in a social network, if each node is tagged with the resident country of the corresponding user, the number of unique labels would run into hundreds.

3. Do not support query-time label definition. In some scenarios, the constraints are defined as a function over node or edge labels at query time. For example, recall the previous query where one wishes to check if there exists a cascade of interactions from user U on Twitter to user V such that all intermediate nodes are females of age between 20 and 30. Here, the constraint is not defined based on the presence or absence of an existing label, but as a function that is specified at query time, whose output is like a new label for that node. Existing techniques do not support query-time label definitions since they assume that the label set space is known.

4. Unable to handle dynamic networks. Many networks today are highly dynamic in nature. For example, new nodes are added to the Wikidata17 knowledge base every day and existing nodes get updated by creating new links among them. Several index-based techniques for LCR reachability [8, 11, 21] assume a static network. This limits their applicability to dynamic networks.

Proposed solution: Table 1 summarizes the limitations of existing techniques, most of which result due to relying on an index structure. This property forces us to ask the following question: *Is it possible to develop an index-free, near-optimal algorithm with linear storage and time complexity?* In this paper, we show that this is indeed possible. More specifically, we develop an algorithm called *ARRIVAL: Approximate Regular-simple-path Reachability In Vertex and Arc (directed edges) Labeled Graphs*, which addresses all of the above weaknesses while being scalable and accurate. Our key contributions are as follows:

- We formalize the problem of regular simple path queries (RSPQ) on labeled graphs. The proposed formulation supports a label set space that is unbounded by allowing the user to define functions over node or edge labels (Sec. 2).

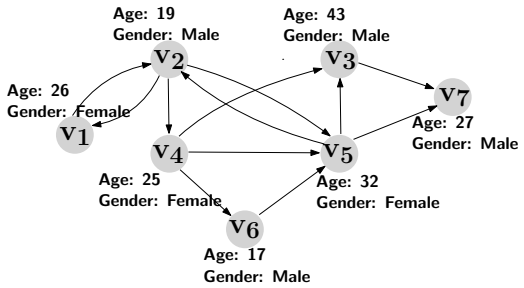


Figure 1: A sample graph dataset. For simplicity, we assume that labels are attached only with nodes.

- We develop an index-free, sampling-based algorithm called ARRIVAL, which is remarkably simple in its approach, and yet backed by theoretical analysis.
- We perform extensive experiments on several real graphs containing millions of nodes, billions of edges and thousands of unique labels. Our empirical evaluation establishes that ARRIVAL is up to 100 times faster than BBFS. Furthermore, ARRIVAL guarantees a precision of 1, while the recall is 95% on average. (Sec: 5).

2 PROBLEM FORMULATION

There are four pieces of input to the RSPQ problem: the graph, the source and destination nodes, and the label constraint.

Definition 1 (Graph). *A multi-labeled graph (directed or undirected) is a triple $G = (V, E, \mathcal{L})$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $\mathcal{L}$ is a finite non-empty set of labels over nodes and edges in the graph. A node or an edge may be labeled with zero or more labels from the set $\mathcal{L}$.*

We do not assume the graph to adhere to any structural property such as acyclicity or strong connectedness. However, certain quality guarantees may be provided only when the graph satisfies certain conditions. We explicitly mention those conditions in our proofs.

We use the notation $l(v)$ and $l(e)$ to denote the set of labels present in node v and edge e respectively.

Example 1. *A sample multi-labeled graph is shown in Fig. 1. In this graph, every node is characterized by two features: the age and the gender. The label of a node is formed by concatenating the feature name and feature value. For example, v_1's labels are "Age=26" and "Gender=Female". We use the notation $v.Age$ and $v.Gender$ to note the value of the "Age" and "Gender" features of v. For example, $v_1.Age = 26$.*

Definition 2 (Path). *A path P in graph G is a sequence of vertices $\langle v_1, v_2, \cdots, v_n \rangle$ such that $\forall i,\ 1 \leq i \leq n-1,\ (v_i, v_{i+1}) \in E$. Additionally if $\nexists v_i, v_j \in P$, such that $i \neq j$ and $v_i = v_j$, i.e., no vertex is repeated then the path is called a* simple *path. In this paper we consider only simple paths and so even when we use the term path it should be interpreted as simple path.*

We use the notation $P.v_i$ and $P.e_i$ to denote the i^{th} node and edge in path P respectively. Furthermore, $P \subseteq P'$ denotes

that path P is a sub-path (or sub-sequence) of another path P'. Given a path $P = \langle v_1, \cdots, v_k \rangle$, one can generate a sequence of labels $S = \langle a_1, \cdots, a_{2k-1} \rangle$ by arbitrarily picking a label from each node and edge in P. We use this idea to formally define the concept of *sequence containment* in a path P.

Definition 3 (Sequence Containment). *Given a path $P = \langle v_1, \cdots, v_k \rangle$, a sequence of labels $S = \langle a_1, \cdots, a_{2k-1} \rangle$ is contained in P if $\forall i,\ 1 \leq i \leq 2k-1,\ a_i \in l(P.v_{\lceil \frac{i}{2} \rceil})$ if i is an odd number, otherwise, $a_i \in l(P.e_{\frac{i}{2}})$. We use the notation $l(P)$ to denote the set of all label sequences contained in path P.*

More simply, if a sequence S can be generated from path P, then it is contained in P.

Definition 4 (Regular Expression (Regex)). *Let $\mathcal{L}$ be a set of labels disjoint from $\{\epsilon, \emptyset, (,)\}$, where ϵ and $\emptyset$ denotes the empty string and empty set respectively. A regular expression C over $\mathcal{L}$ is defined as follows:*

- *$\epsilon, \emptyset$, and each $l \in \mathcal{L}$ are regular expressions.*
- *If A and B are regular expressions, then $(A|B)$, (AB) and (A^*) are regular expressions.*
- *Nothing else is a regular expression.*

The expression $(A|B)$ is called *alternation* of A and B, (AB) is called *concatenation* and A^* is called the *closure* of A. Furthermore, the notation A^+ is equivalent to AA^* and is called the *positive closure* of A.

Any regex can be transformed into an equivalent *nondeterministic finite automaton* (NFA) $M = (Q, F, q_0, q_f)$ with states Q, labeled transition set F, an initial state q_0 and the final state q_f corresponding to the regular language defined by C. M is dependent only on the regex and not on the data graph. We construct M using the classical *Thompson's construction algorithm* [20]. In principle, we can support negation queries for all regular expressions but for practical reasons the negation operator is supported only on those queries where the NFA produced by Thompson's algorithm happens to be a deterministic finite automaton (DFA). For further details on the scope of the negation operator and the reasoning behind this restriction, please see Appendix A. Path specific constraints such as a path length within a given range are also supported.

Example 2. *The finite state automaton of the regex a^*ba^* is shown in Fig. 2(b).*

A label sequence is called a *pattern* under regex C if it is accepted by the finite state automaton of C. Given a regex label constraint C and a path P, we next define the concept of *path compatibility*.

Definition 5 (Path Compatibility). *Path P is compatible with regex C if $\exists S \in l(P)$, such that S is a pattern under regex C. We will refer to such a path as C-compatible or simply compatible if the regex C is understood.*

With the above definitions in place, we are now ready to formalize the concept of *reachability under regex constraint*.

Definition 6 (Reachability under regex constraint). *Node $v \in V$ is said to be reachable from $u \in V$ under regex constraint C if and only if there exists a C-compatible simple path P from u to v.*

Occasionally regular expressions may be defined using labels that are functions of the attributes of a node or an edge. Such labels may be too numerous to pre-compute and store, although it might be easy to compute them at query-time.

Definition 7 (Query-time labels). *Given a node or an edge with attributes $l(v)$ and $l(e)$ respectively, a query-time label is the output of an efficiently computable boolean-valued function of the form $f : l(v) \rightarrow \{1, 0\}$ or $g : l(e) \rightarrow \{1, 0\}$. Node v contains the label $f(l(v))$ if $f(l(v)) = 1$. Containment of $g(l(e))$ in edge e is defined analogously.*

Example 3. *In Figure 1, one can categorize a node as an adult female, if the age is at least* 18 *and the gender is female, i.e.,*

$$isAdultFemale(v) = \begin{cases} 1 & if\ v.Age \geq 18 \wedge v.Gender = Female \\ 0 & otherwise \end{cases} \tag{1}$$

Practical Constraints: Note that we use the term "efficiently computable" in the definition of query-time labels. This is a necessary condition, since otherwise, this computation itself may stall the computation of the reachability query. We leave the exact definition of "efficiently computable" open since it varies from application to application. Furthermore, it may also be necessary to check the consistency of the query-time label function and ensure it never crashes and returns a boolean value across any possible label set in a node or an edge. While these are necessary conditions that must be checked in a practical system, the goal of the proposed study is to only show that query-times labels can be supported in the proposed framework. Hence, we do not deliberate further on these aspects.

With the formalization of the above concepts, we are now ready to define the problem of *regular simple path queries (RSPQ)*.

Problem 1 (Regular Simple Path Query (RSPQ)). *Given a graph $G = (V, E, \mathcal{L})$, source and destination nodes $s, t \in V$ respectively, an optional set of query-time label definitions $\mathcal{Q}$, and a regex constraint C defined over $\{\mathcal{L} \cup \mathcal{Q}\}$, we need to determine if t is reachable from s under constraint C.*

Extension to Dynamic Graphs: When a multi-labeled graph is evolving with time, two kinds of changes are possible: structural change and information change. In a structural change, a node or an edge gets added or deleted. In an information change, a label information is updated, deleted, or

Algorithm 1 BFS exploration

Require: Graph $G(V, E, \mathcal{L})$, source and destination nodes s,t, query-time label definitions $\mathcal{Q}$, and regex $C \subseteq \{\mathcal{L} \cup \mathcal{Q}\}^*$.
Ensure: returns if v is reachable from u under constraint C

```
1:  Q ← empty queue
2:  P ← empty path
3:  Add tuple ⟨s, P⟩ to Q
4:  while Q is not empty do
5:      ⟨n, P⟩ ← Q.dequeue
6:      if n ∉ P then                                   ▷ Ensures simple paths
7:          P ← P ∪ {n}
8:          if P is potentially compatible with regex C then
9:              if n = v then                           ▷ If destination has been reached
10:                 if P is compatible with regex C then
11:                     return Reachable
12:             else
13:                 for each neighbor n′ of n do
14:                     Add ⟨n′, P⟩ to Q
15: return Not Reachable
```

added. We assume that each change is associated with a timestamp t. When a reachability query is posed on a dynamic network at time t_q, the query is answered with respect to the state of the graph at time t_q. As we will see in subsequent sections, no changes are required in the proposed algorithm for dynamic graphs, since we do not maintain any index. The only task is to maintain up-to-date snapshots of the graph.

2.1 Query Types

In this section, we discuss three key regular expression patterns that cover more than 96% of all property path queries in real-world SPARQL workloads [5]: (a) label-set restricted paths, (b) paths with repeated label-sequence, and (c) concatenated label-chains.

2.1.1 Query Type 1: Label-set Restricted Paths. $(l_0 \mid l_1 \mid .. \mid l_{n-1})^*$, where $L = \bigcup_{i=0}^{n-1}\{l_i\} \subseteq \mathcal{L}$. This query type restricts the exploration of reachability paths to only those vertices (and the subgraph induced) that have one or more labels in L. This is one of the most common RSP query types–generally referred to as *LCR queries*–that some of the previous works have also addressed [21].

2.1.2 Query Type 2: Repeated Label-Sequence Paths. $(l_0\, l_1\, \ldots\, l_{n-1})^+$, where, $L = \bigcup_{i=0}^{n-1}\{l_i\} \subseteq \mathcal{L}$. This query type imposes a strict, possibly repeating, order on the labels that can be explored while searching for a connecting path between query labels. Queries of this type are directly applicable whenever a local constraint on navigating searches through graphs are important. For example, when a user is interested in a specific pattern of connections between people on social network based on their category labels. It is important to note that this is a query class that makes regex reachability NP-Hard (see [16] for details).

2.1.3 Query Type 3: Paths with Concatenated Label-Chains. $(l_0)^+(l_1)^+..(l_{n-1})^+$, where $l_i \in \mathcal{L}$ and $\forall_{i=0}^{i=n-2} : l_i \neq l_{i+1}$.

This query, which is also NP-hard, expresses a constraint that allows only paths that follow a pre-determined order in relative positions for any pair of distinct labels. A path with the same label can be explored for some steps, and then it moves into a state where it explores the path induced by the successive label, before continuing with the same transition process. The exact number of nodes or edges (has to be at least 1) the path expands for through each label is left as a non-deterministic choice.

2.2 Baseline and Complexity Analysis

Given a source and a destination vertex, the simplest approach is to adopt a *breadth-first search (BFS)* strategy (Alg. 1). Specifically, we start exploring paths from the source vertex (line 3) and expand one-edge at a time in breadth-first manner (lines 4-14) while ensuring each expansion results in a path that is both simple (line 6) as well as *potentially compatible* to the given regex (line 8).

Definition 8 (Potential Compatibility). *A path P is potentially compatible with regex C if it is compatible with some prefix of C.*

Example 4. *Assume v_1 and v_5 in Fig. 2(a) as the source and destination nodes respectively. Furthermore, let a^*ba^* be the regex constraint. Here, path $P = \langle v_1, v_2\rangle$ is potentially compatible and the path $P' = \langle v_1, v_5\rangle$ is not potentially compatible.*

If at any stage of the search procedure, the destination node is reached (line 9), we check if the resultant path is compatible to C (line 10). If yes, then the search terminates by concluding that the nodes are reachable (line 11). Otherwise, BFS continues to explore other path possibilities (lines 12-14). Note that unlike traditional BFS where only the shortest path tree is explored, in our case, we explore all possible paths.

While the algorithm is simple, BFS needs to maintain information for *every* potentially compatible path in the network. In the worst case, the number of potentially compatible paths is exponential, which leads to the following theorem.

Theorem 1. *RSPQ is NP-hard.*

Proof. Mendelzon and Wood proved this result in [16] by reducing it from the fixed regular path problem. □

Intuition: To understand the intuition behind the proof, please see App. B.

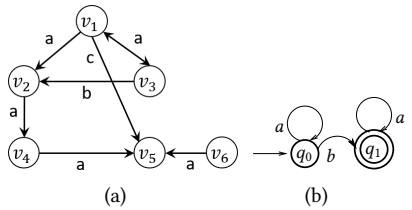


Figure 2: (a) A sample edge-labeled graph to understand the complexity of RSP queries. (b) The automaton that accepts a^*ba^*.

3 ARRIVAL

ARRIVAL is based on the simple idea of sampling a small number of paths and then answering the RSP query by processing only this sampled set. As in any sampling algorithm, several questions arise. How many paths should we sample? What should be the length of the sampled paths? What is the impact of sampling on the accuracy of our answers? How do we bias the sampling procedure towards paths that are compatible to the regex constraint? In the next few sections, we answer these questions.

3.1 Sampling Paths through Random Walks

In ARRIVAL, we perform a *bi-directional* random walk, where two random walks are simultaneously started from the source and destination nodes. Algorithm 2 outlines the pseudocode of our algorithm. The random walk starting from the source node is called the *forward walk* and the one starting from the destination node is called the *backward walk*. In the backward walk, the walk proceeds in reverse direction, i.e., the walker can jump from node u to v only if there is an incoming edge to u from v. The forward walk, on the other hand, proceeds through outgoing edges. In addition to the directionality, there is one more restriction in choosing the next neighbor; the random walker is allowed to jump to only those neighbors such that the path remains simple and potentially compatible (lines 20-21) (See Appendix C for details on the backward walk automaton and handling of automaton branches). Both the forward and backward walks continue, till one of the following terminating conditions is reached.

- **Case 1:** The walk reaches a dead end. This happens when either there are no neighbors of the current node, or all neighbors do not result in a simple and potentially compatible path (first condition in lines 22 and 27).
- **Case 2:** The length of the walk equals *walkLength*. *walkLength* is an input parameter to the ARRIVAL algorithm and denotes the maximum length of any random walk (second condition in lines 22 and 27).
- **Case 3:** One of the sampled forward paths *meets* a sampled backward path, and when these two paths are *joined*, it forms a path that is simple and compatible to the regex constraint (lines 14-17). Path P_1 *meets* path P_2 if (i) one of them is a backward path and the other is a forward path, and (ii) they contain a common node, i.e., $\exists n \in V$ such that $n \in P_1$, $n \in P_2$. Without loss of generality, let us assume that $P_1 = \langle s, \cdots, n, \cdots, v \rangle$ is the forward path and $P_2 = \langle v', \cdots, n, \cdots, t \rangle$ is the backward path. The *join* of P_1 and P_2 is the path $P = \langle s, \cdots, n, \cdots, t \rangle$, where the s to n portion is from P_1 and the remaining portion is from P_2. If P is simple and compatible to the regex constraint, which is a necessary condition for this case to be true, we can guarantee that the destination node t is reachable from source s.

Algorithm 2 ARRIVAL

Require: Graph $G(V, E, \mathcal{L})$, source and destination nodes s,t, query-time label definitions Q, and regex $C \subseteq \{\mathcal{L} \cup Q\}^*$.
Require: Parameters *walkLength*, *numWalks*.
Ensure: returns if t is reachable from s under constraint C.
1: $fwalk, bwalk \leftarrow 0$ ▷ counters for number of forward and backwards paths sampled respectively
2: $Paths_f, Paths_b \leftarrow \emptyset$ ▷ stores the samples forward and backward paths respectively
3: $N_f \leftarrow \{s\}$ ▷ candidates nodes to proceed to in forward walk
4: $N_b \leftarrow \{t\}$ ▷ candidates nodes to proceed to in backward walk
5: $P_{fwalk}, P_{bwalk} \leftarrow \emptyset$ ▷ the current forward and backward walks respectively
6: $M_f, M_b \leftarrow$ empty hashmaps corresponding to forward and backward walks respectively.
7: **while** $fwalk + bwalk < numWalks$ **do**
8: $n_f \leftarrow$ choose a random node from N_f
9: $n_b \leftarrow$ choose a random node from N_b
10: $P_{fwalk} \leftarrow P_{fwalk} \cup \{n_f\}$
11: $P_{bwalk} \leftarrow P_{bwalk} \cup \{n_b\}$
12: $s_f \leftarrow$ State of P_{fwalk} in the finite state automaton of regex C
13: $s_b \leftarrow$ State of P_{bwalk} in the finite state automaton of regex C
14: **if** $\langle n_f, s_f \rangle \in M_b$ **then**
15: **if** $\exists i \in M_b.get(\langle n_f, s_f \rangle)$ such that $P_i \in Paths_b$ and $join(P_i, P_f)$ is a simple path **then return** Reachable
16: **if** $\langle n_b, s_b \rangle \in M_f$ **then**
17: **if** $\exists i \in M_f.get(\langle n_b, s_b \rangle)$ such that $P_i \in Paths_f$ and $join(P, P_b)$ is a simple path **then return** Reachable
18: $M_f.add(\langle n_f, s_f \rangle, fwalk)$
19: $M_b.add(\langle n_b, s_b \rangle, bwalk)$
20: $N_f \leftarrow \{n \mid n$ is an outgoing neighbor of n_f and $P_{fwalk} \cup \{n\}$ is potentially compatible, $n \notin P_{fwalk}\}$
21: $N_b \leftarrow \{n \mid n$ is an incoming neighbor of n_b and $P_{bwalk} \cup \{n\}$ is potentially backward compatible, $n \notin P_{bwalk}\}$
22: **if** $N_f = \emptyset$ **OR** $|P_{fwalk}| =$ *walkLength* **then**
23: $Paths_f \leftarrow Paths_f \cup \{P_{fwalk}\}$
24: $N_f \leftarrow \{s\}$
25: $fwalk \leftarrow fwalk + 1$
26: $P_{fwalk} \leftarrow \emptyset$
27: **if** $N_b = \emptyset$ **OR** $|P_{bwalk}| =$ *walkLength* **then**
28: $Paths_b \leftarrow Paths_b \cup \{P_{bwalk}\}$
29: $N_b \leftarrow \{t\}$
30: $bwalk \leftarrow bwalk + 1$
31: $P_{bwalk} \leftarrow \emptyset$
32: **return** Not Reachable

If the walk terminates through Case 1 or 2, then the resultant path is stored and a new walk is initiated (lines 22-31). Each time a new walk is initiated, we check if the total number of walks has equaled *numWalks*, which is the second input parameter to ARRIVAL to bound the number of samples to be taken (line 7). If this check returns true, then the algorithm exits by concluding that the two nodes are not reachable (line 32). On the other hand, if a walk terminates through Case 3, then we conclude that the source and destination nodes are reachable and the algorithm terminates (lines 14-17). Under some situations, it is possible that multiple terminating cases are true simultaneously. If Case 3 is one of them, it takes precedence. Otherwise, the walk simply ends and a new one is started.

Example 5. *Let us revisit the graph in Fig. 2(a), where the source and destination nodes are v_1 and v_5 respectively, and the regex constraint is a^*ba^*. Furthermore, let* walkLength = 3 *and* numWalks = 10. *ARRIVAL starts by initiating forward*

Forward Walks			Backward Walks			Case 1: Dead-end		Case 2: Equals *walkLength*		Case 3: Reachable path found
Time stamp	ID	Location	Time stamp	ID	Location	FW	BW	FW	BW	
0	0	v_1	0	0	v_5					
1	0	v_2	1	0	v_4					
2	0	v_4	2	0	v_2			✔	✔	
0	1	v_1	0	1	v_5					
1	1	v_3	1	1	v_6		✔			
2	1	v_2	0	2	v_5			✔		✔

(a)

Forward Walk Hashmap	Backward Walk Hashmap
$(\langle v_1, null\rangle: \mathbf{0})$	$(\langle v_5, null\rangle: \mathbf{0})$
$(\langle v_1, null\rangle: 0), (\langle v_2, q_0\rangle: \mathbf{0})$	$(\langle v_5, null\rangle: 0), (\langle v_4, q_1\rangle: \mathbf{0})$
$(\langle v_1, null\rangle: 0), (\langle v_2, q_0\rangle: 0), (\langle v_4, q_0\rangle: \mathbf{0})$	$(\langle v_5, null\rangle: 0), (\langle v_4, q_1\rangle: 0), (\langle v_2, q_1\rangle: 0)$
$(\langle v_1, null\rangle: 0, \mathbf{1}), (\langle v_2, q_0\rangle: 0), (\langle v_4, q_0\rangle: 0)$	$(\langle v_5, null\rangle: 0, \mathbf{1}), (\langle v_4, q_1\rangle: 0), (\langle v_2, q_1\rangle: 0)$
$(\langle v_1, null\rangle: 0,1), (\langle v_2, q_0\rangle: 0), (\langle v_4, q_0\rangle: 0), (\langle v_3, q_0\rangle: \mathbf{1})$	$(\langle v_5, null\rangle: 0,1), (\langle v_4, q_1\rangle: 0), (\langle v_2, q_1\rangle: 0), (\langle v_6, q_1\rangle: \mathbf{1})$
$(\langle v_1, null\rangle: 0,1), (\langle v_2, q_0\rangle: 0), (\langle v_4, q_0\rangle: 0), (\langle v_3, q_0\rangle: 1), (\langle v_2, q_1\rangle: \mathbf{1})$	$(\langle v_5, null\rangle: 0,1, \mathbf{2}), (\langle v_4, q_1\rangle: 0), (\langle v_2, q_1\rangle: 0), (\langle v_6, q_1\rangle: 1)$

(b)

Figure 3: (a) Illustration of Example 5. FW and BW denotes forward and backward walks respectively. (b) The key-value entries in the forward and backward hashmaps after each jump by the walker is shown in Fig. 3(a). The new key-value pairs inserted after each forward and backward jump are highlighted in bold, red font.

and backward random walks from v_1 and v_5 respectively. Thus, the location of the random walkers at time $t = 0$ is v_1 and v_5. We show these locations in Fig. 3(a). For the forward random walker, the candidate neighbors are v_2 and v_3; v_5 does not qualify since the path $\langle v_1, v_5\rangle$ is not potentially compatible. Similarly, the neighbor candidates for v_5 are v_4 and v_6. As shown in Fig. 3(a), at time $t = 1$, we randomly choose neighbors v_2 and v_4. At this stage none of the terminating conditions are true and thus both walks proceed to v_4 and v_2 respectively at time $t = 2$. At this stage, Case 2 returns true for both walks since they have reached their maximum lengths. Consequently, both walks are terminated, and at this point, we have sampled one walk each in both directions. Note that Case 3 fails at this stage although v_4 and v_2 are present in both forward and backward paths since when joined, the corresponding formed paths are not compatible to a^*ba^*. As shown in Fig. 3(a), the second forward and backward paths sampled are $\langle v_1, v_3, v_2\rangle$ and $\langle v_5, v_6\rangle$ respectively. The moment the second forward walk reaches v_2, Case 3 returns true, since v_2 has been traversed by the first backward path, and when these two paths are joined to form $\langle v_1, v_3, v_2, v_4, v_5\rangle$, it remains compatible to a^*ba^*. Hence, ARRIVAL concludes that v_5 is reachable from v_1.

3.1.1 Naïve Implementation of Case 3: ARRIVAL needs to check for Case 3 after each jump by the walker. Case 3 can be divided into three *sub-conditions*.

(1) **Meeting:** The first condition is to verify if the current forward (or backward) path meets any of the sampled backward (or forward) paths.
(2) **Compatibility:** If the above condition is true, then we proceed to verify if the meeting paths can be joined to form a compatible path.
(3) **Simplicity:** Finally, we need to also ensure the joined path is simple.

Lemma 1. *The time complexity of checking the meeting condition in Case 3 using the naïve implementation is* $O(\text{numWalks} \times \text{walkLength})$.

Proof: Let n_1 and n_2 be the current station of the random walker in the forward and backward paths respectively. Furthermore, let $\mathbb{P}_1$ and $\mathbb{P}_2$ be the set of sampled forward and backward walks respectively till now. We need to now check if n_1 has been traversed by any of the paths in $\mathbb{P}_2$ and vice versa with respect to n_2 and $\mathbb{P}_1$. Since any random walk is terminated after at most *walkLength* jumps, we perform *numWalks* random walk from each direction, the overall time complexity for this check is $O(walkLength \times numWalks)$. □

Lemma 2. *The time complexity of the compatibility check is* $O(L \times \text{walkLength})$ *for each meeting path, where L is the average number of labels per node (or edge). If* $L \ll \text{walkLength}$, *which is often the case,* $O(L \times \text{walkLength}) \approx O(\text{walkLength})$.

Proof: Let $p_1 \in \mathbb{P}_1$ and $p_2 \in \mathbb{P}_2$ be two paths that meet. Joining these two paths requires constant time since the node at which they meet is known. Checking compatibility of a path requires $O(L \times walkLength)$ time. Please see App. C for details. □

Lemma 3. *The time complexity of the simplicity check is* $O(\text{walkLength})$ *for each meeting path.*

Proof: To check if a path is simple, we need to iterate through each node in the path. Since the maximum length of a joined path is $2 \times walkLength$, the time complexity is bounded by $O(walkLength)$. □

Theorem 2. *The time complexity of checking Case 3 using the naïve implementation is* $O((\text{numWalks} + d_1 \times L + d_2) \times \text{walkLength})$.

Proof: Based on Lemma 1, we first consume $O(walkLength \times numWalks)$ time to identify all meeting paths. If d_1 paths satisfy the meeting conditionand d_2 paths satisfy both meeting and compatibility sub-conditionsit follows from Lemma 2 and Lemma 3 that the total cost of checking Case 3 is $O((numWalks + d_1 \times L + d_2) \times walkLength)$. □

3.1.2 Efficient Implementation to check Case 3. A cost of $O((numWalks + d_1 \times L + d_2) \times walkLength)$ is prohibitively large for an operation that needs to be performed after every jump by the random walker. We mitigate this issue, by exploiting the property that every sampled path is potentially

compatible and therefore resides in a particular state of the finite state automaton corresponding to the regex constraint. Thus, every time the walker takes a jump, we construct a tuple $\langle n, automatonState \rangle$, where n is the current location of the walk and $automatonState$ is the automaton state of the walk (lines 12-13).We maintain two hashmaps, one each for the forward and backward walks (lines 6), where we store the key-value pairs ($\langle n, automatonState \rangle$, i). Here, the key is the node-automatonState tuple created after every jump, and the value i points to the index of the current forward or backward path, which produced this key.

Example 6. *Fig. 3(b) shows the states of the forward and backward hashmaps after each jump in Fig. 3(a). The automaton corresponding to the regex constraint a^*ba^* is shown in Fig. 2(b), which contains the states q_0 and q_1. At the start of every walk (time $t = 0$), the state of the automaton is denoted as null.*

Owing to the utilization of hashmaps, in $O(1)$ time, we can evaluate the first two sub-conditions, i.e., if a forward path meets a backward path to form a compatible joined path, and vice-versa (lines 14,16). Specifically, the following properties can be established.

Theorem 3. *If the key $\langle n, automatonState \rangle$ is present in both the forward and backward hashmaps, then there exists a compatible path from the source to the destination.*

Proof. Since the node n has been visited by both a forward and backward walk, it is clear that there exists a path from s to t. Furthermore, since the automaton states of the forward and backward walks are identical, it follows that the joined path is compatible to the regex. □

Corollary 1. *Verifying the meeting and compatibility sub-conditions in Case 3 requires $O(1)$ time with the efficient hashmap-based implementation.*

Proof: From Theorem 3, checking the meeting and compatible conditions require performing an $O(1)$ look-up each in the forward and backward hashmaps. □

Theorem 4. *The time complexity of checking Case 3 using the hashmap-based implementation is $O(d_2 \times \text{walkLength})$ where d_2 is the number of compatible and simple meeting paths.*

Proof: As established in Cor. 1, the first two sub-conditions can be verified in $O(1)$ time. To verify the third sub-condition of simplicity, we fetch all path IDs corresponding to the key $\langle n, automatonState \rangle$ and check if at least one of these paths can be joined with the current forward or backward path to form a simple path. From Lemma 3, checking simplicity of a compatible meeting path requires $O(walkLength)$ time. Thus, if there are d_2 such paths, the time complexity of checking Case 3 is $O(d_2 \times walkLength)$. □

When compared to the naïve implementation (Thm. 2), we obtain a significant increase in efficiency.

3.2 Properties of the ARRIVAL Algorithm

3.2.1 Storage Cost. ARRIVAL does not store any information other than the hashsets and the current forward and backward walks. In the worst case, each jump adds a new entry to the hashset and thus the storage is bounded by $O(walkLength \times numWalks)$.

3.2.2 Complexity Analysis. In the worst case, ARRIVAL needs to make $O(walkLength \times numWalks)$ random walk jumps. For each jump, we need to scan through all neighbors and identify those that are compatible to the given regex constraint. This requires $O(dL)$ time where d is the average degree of a node in the data graph and L is the average number of labels per node. Every other step in Alg. 2 consumes $O(1)$ time. Thus, the computation complexity of ARRIVAL is bounded by $O(walkLength \times numWalks \times dL)$.

3.2.3 Qualitative Aspects. ARRIVAL will never have a false positive in its answer since any reachable path that it finds actually exists in the graph.

4 THEORETICAL GUARANTEES

Random walk-based reachability comes with strong theoretical guarantees in the cases of *unlabeled* reachability as long as we set the parameters *walkLength* and *numWalks* correctly. In Section 4.2 we explain how we use the guidance from the unlabeled case to achieve highly accurate results in the labeled RSP setting.

4.1 Random Walk-based Reachability on Unlabeled Directed Graphs

Random walk-based reachability in unlabeled directed graphs is based on a simple idea: if v is reachable from u in G then with non-zero probability a random walk beginning from u and following the edges according to their orientation will intersect a random walk begun from v that traverses the edges opposite to their orientation. But we need to ensure two things (a) if v is reachable from u then this method returns a YES answer in reasonable time with good probability and (b) if v is not reachable from u then the method times out and gives a NO answer in reasonable time.

The way to think about the two main parameters of ARRIVAL is as follows: (1) *walkLength*: if this is too small we may not have enough probability of giving a YES answer correctly, and (2) *numWalks*: increasing this helps boost the probability of giving a positive answer correctly. So increasing both these parameters helps us get better accuracy at the cost of running time, i.e., we need to choose the smallest possible values that allow us to get good accuracy.

Let us consider the nodes of the graph to be bins. Each random walk ends in a particular bin according to its stationary distribution. Now, to set *numWalks* we ask ourselves if we throw k red balls into n bins according to one distribution

(the forward walks) and k blue balls into n bins according to another distribution (the backward walks) how large should k be such that there is a bin with both a red and a blue ball with high probability? That value of k is our *numWalks*. Here, n is the total number of nodes in the graph.

4.1.1 Formalizing the theoretical guarantee. Clearly if the overlap between the forward and backward stationary distributions is not large enough then we will have trouble boosting the probability in the positive query case. So, we define the *robust undirectedness* of a strongly connected directed graph as a measure of the probability that the forward and backward random walks overlap. If $\overrightarrow{\pi}$ is the stationary distribution of the forward random walk and $\overleftarrow{\pi}$ is the stationary distribution of the backward random walk, then the robust undirectedness of the graph is

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

We now state the result for the unlabeled case

Proposition 1. *Given a strongly connected directed graph G with vertex set V, such that $|V| = n$, if we set*

$$\text{walkLength} = \textit{Diameter of } G \quad (3)$$

$$\text{numWalks} = \left\{\frac{16n^2 \ln n}{(\alpha(\overrightarrow{\pi}, \overleftarrow{\pi}))^2}\right\}^{\frac{1}{3}} \quad (4)$$

then the forward and backward walks overlap with probability at least $1 - 1/n$.

Proof. See Appendix D.

4.1.2 Implications when the graph is not strongly connected. Any non-strongly connected, directed graph can be partitioned into a set of strongly connected components, each of which will have less than n nodes. For source-destination pairs that lie within a single strongly connected component the guarantees of the proposition definitely hold. Similarly for non-connected pairs we have no problem since ARRIVAL has no false positives. The only issue arises when u is reachable from v and the two nodes lie in different connected components. In this case there is no direct theoretical guarantee to be obtained from the proposition. However, as we will see in our empirical evaluation in Section 5, even on real-world graphs, the false negative rate is less than 6% on average.

4.2 From Unlabeled to Labeled Reachability

The parameter *numWalks* is unaffected when we move to the labeled reachability setting since it essentially helps us boost the success of the random path sampling procedure by repetition, but carrying *walkLength* over to the labeled setting requires taking some care. Unlabeled reachability is, in general, an easier problem than regex reachability. However there is clearly a one-sided relationship between these two problems. Specifically:

(1) For any regex C, the existence of a C-compatible (u, v) path in $G = (V, E, \mathcal{L})$ implies the existence of an unlabeled (u, v) path in $G = (V, E)$.
(2) For any regex C, the length of the shortest C-compatible (u, v) path in $G = (V, E, \mathcal{L})$ is at least the length of the shortest unlabeled (u, v) path in $G = (V, E)$.

The implication of property (2) above is that setting *walkLength* to the unlabeled diameter of the graph is not enough to ensure that there will be sufficient probability of the forward and backward paths intersecting. To mitigate this problem, we take *self-avoiding* random walks, i.e., we never revisit an edge that has already been taken. This boosts the probability of success by ensuring that there are more vertices covered in the forward and backward walks, thereby giving more opportunities for the walks to intersect.

4.3 Complexity of Computing *walkLength* and *numWalks*

Both *walkLength* and *numWalks* are computed once on the input graph and therefore their computation costs do not affect the querying time complexity.

- *walkLength*: As specified in Proposition 1, *walkLength* needs to be set to the *Diameter* of the graph in the unlabeled case which can take $O(|V|^3)$ time. Here, we note that as long as *walkLength* $\geq$ *Diameter*, all accuracy guarantees of Proposition 1 continue to hold. Thus, in practice, we only need an efficient method to compute an upper bound on the diameter of the graph. Towards that, we select a random sample of s nodes, and compute the shortest path tree using Dijkstra's algorithm at $O(s(|E| + |V| \log |V|))$ cost, where $s \ll |V|$. From the search tree, we set *walkLength* to the longest path in the shortest path tree across all s nodes. In the case of labeled graphs, we replicate the same procedure. Specifically, we select s random regular expressions from the real-world SPARQL query log (See Sec. 2.1) and compute the shortest path tree through only paths that are compatible to the query regular expressions. In our experiments, $s = 1000$.
- *numWalks*: As noted in Eq. 4, *numWalks* $= \left\{\frac{16n^2 \ln n}{(\alpha(\overrightarrow{\pi}, \overleftarrow{\pi}))^2}\right\}^{\frac{1}{3}}$. The denominator, which is the robust undirectedness (Eq. 2), is a function of the stationary distribution of the graph. This can be estimated by running a number of random walks close to mixing (i.e. up to *walkLength* steps) and examining the final vertex: which amounts to sampling from the stationary distribution. Since ARRIVAL conducts such walks to answer queries, we can continuously refine our estimate of robust undirectedness, i.e., we can amortise the cost of improving our choice of *numWalks* against the cost of answering the query. With this in view, we choose an initial value of *numWalks* =

Table 2: Datasets.

Dataset	$\|V\|$	$\|E\|$	$\|\mathcal{L}\|$	Directed	Node Labels	Edge Labels	Dynamic
GPlus	107K	13.6M	17073	✓	✓		
DBLP	1.75M	7.4M	679		✓		
Freebase	3.6M	57.7M	7513	✓	✓	✓	
Stack Overflow	2.6M	67.5M	3	✓		✓	✓
Twitter	47M	1.965B	1000	✓	✓		

$\sqrt[3]{n^2 \ln n}$ and proceed, refining as we go. The initial estimate is just a numerical value and thus it requires $O(1)$ time.

5 EXPERIMENTS

In this section, we benchmark ARRIVAL and establish that:

- **Quality:** ARRIVAL produces near-optimal results with an average recall of 95%.
- **Efficiency:** ARRIVAL is scalable to billion-sized networks.

Our implementation is available at https://github.com/idea-iitd/ARRIVAL.

5.1 Datasets

We use real-world graphs spanning the domains of social networks, collaboration/authorship networks, and knowledge graphs. Table 2 summarizes the various characteristics of the datasets. Of particular note is the label set size $|\mathcal{L}|$. None of the existing algorithms for regex reachability queries [8, 11, 21, 23] has been evaluated on datasets containing hundreds of labels due to non-scalability. Fig. 9 in Appendix presents the frequency distribution of labels in each dataset. The semantics of each dataset are as follows.

GPlus: The GPlus[1] graph has been constructed from the Google Plus social network. Each person represents a node and annotated with labels representing gender, place, institution and occupation. A directed edge from node u to v implies that person u follows person v.

DBLP: DBLP[2] is a co-authorship network in which two authors (nodes) have an edge between them if they have collaborated on at least one paper together. Each node has 5 features: (i) number of papers published by the author, (ii) the number of years for which the author has been active (calculated as the timegap between the latest and the first paper), (iii) the set of venues in which the author has published, (iv) the set of subject area(s) of the venues in which the author has published, and (v) the median rank of the venue (computed as the median of the ranks of all the venues where the author has published). The first three features are obtained from the DBLP database while the last two are added from the CORE rankings portal [1].

Freebase: Freebase[3] [19] is based on the open-source knowledge graph "Freebase". Each entity (node) is labeled with multiple labels according to the real world categories it can be placed in such as 'person', 'location', 'athlete', 'artist', etc. Each edge represents a semantic link between the entities with labels such as, 'created', 'lives in', etc.

StackOverflow: StackOverflow[4] is a *dynamic* network containing interactions on the stack exchange web site Stack Overflow. Each directed edge (u, v, t), denotes an interaction between user (node) u and v at time t. There are three types of edges: (1) user u answered user v's question at time t (2) user u commented on user v's question at time t, and (3) user u commented on user v's answer at time t. The label of an edge corresponds to its type. An RSPQ on this dynamic dataset is solved as discussed as in Sec. 2.

Twitter[5]: This is the largest dataset with close to 2 billion edges. To construct the labels, we identify the 1000 nodes with the highest number of followers. These nodes are typically highly popular entities corresponding to celebrities, sport clubs, etc. We call these nodes "community" nodes. If a node v follows a community node c, then v is tagged with the handle of c.

5.2 Experimental Setup

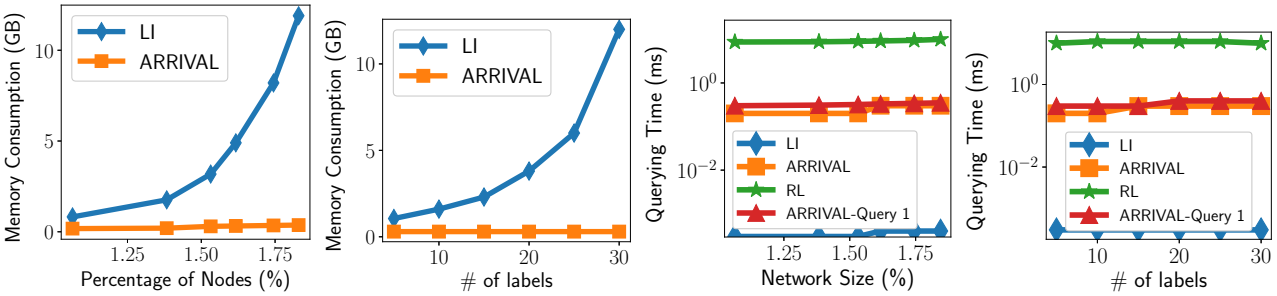
All experiments are implemented in C++ and performed on a machine with Intel(R) Xeon(R) 2.5GHz CPU E5-2609 with 64 GB RAM running Ubuntu 14.04. We call a query *positive* if the destination is actually reachable from the source given the query constraints; and vice versa for a *negative* query.

5.2.1 Baseline. We consider the following baselines.

- **Landmark Indexing (LI)[21]:** LI is an index-based technique that only supports query type 1 as discussed in Section 2.1. We use the code shared by the authors.
- **Rare Labels (RL)[12]:** RL is index-free and supports all regular expressions that can be expressed through an NFA. However, RL does not guarantee simplicity of paths, whereas in ARRIVAL we only explore simple paths. Though enforcing simplicity makes the problem harder, its practical advantages as well as technical challenges are discussed in [16]. We use the code shared by the authors. This code is multi-threaded and uses six parallel threads.
- **Bidirectional BFS (BBFS):** While ARRIVAL samples a subset of the potentially compatible paths, BBFS explores all possible potentially compatible paths. BBFS follows the same state maintenance as in the ARRIVAL; that is, it explores only those edges at every step that do not violate the current state of path. Among the three baselines, BBFS is the only technique that enforces simplicity and supports all regular expressions handled by ARRIVAL. BBFS is error-free and provides the ground truth.

5.2.2 Query Workload. All our performance measurements are reported over workloads consisting of randomly

[1] https://snap.stanford.edu/data/ego-Gplus.html
[2] https://dblp.org/search/
[3] http://resources.mpi-inf.mpg.de/espresso/entity.csv.bz2
[4] https://snap.stanford.edu/data/sx-stackoverflow.html
[5] http://twitter.mpi-sws.org/links-anon.txt.gz



(a) Twitter (b) Twitter (c) Twitter (d) Twitter

Figure 4: Comparison of ARRIVAL with Landmark Indexing (LI) and Rare Labels(RL): Growth of (a-b) memory consumption and (c-d) querying time against network size and number of labels.

generated 10,000 queries. For each query, the source and destination nodes are chosen in a uniformly random manner from the graph. Unless specifically mentioned, we uniformly sample one of the three query types outlined in Sec. 2.1. These three query types cover 96% of all regular expression patterns in the Wikidata17 query log [5]. The number of labels in a given query is chosen randomly from the range 2 to 8. To choose a label, we use frequency proportional sampling; i.e., a frequent label has a higher chance of being used in the query. This choice is made based on the assumption that a popular label in graph is also popular in the query. Nonetheless, we also benchmark with other sampling procedures.

For StackOverflow, which is a dynamic network, we also need to generate a timestamp for each RSPQ. This timestamp is generated by picking a random time between the first temporal edge in the network and the last temporal edge.

5.2.3 Parameter Selection. ARRIVAL employs two parameters: *numWalks*, the maximum number of random walks to be conducted, and *walkLength*, the maximum length (number of nodes) of each walk. As discussed in Sec. 4.3, we set $numWalks = \sqrt[3]{n^2 * \ln n}$ where n is the number of nodes in the graph. For *walkLength*, we compute a fast upper-bound on the actual diameter of the graph using Dijkstra's algorithm. To further amplify the quality, we set $walkLength = 2 \times Diameter\_upper\_bound$.

5.2.4 Metrics. Since ARRIVAL produces *no false positives* (it never falsely reports two vertices to be reachable when they are not), all its errors are due to *false negatives* where it answers them to be unreachable when they are indeed reachable. Therefore, we use *recall*, which is the fraction of positive queries correctly answered by ARRIVAL. Recall also reveals the false negative rate, since by definition, *recall* = (1 − false negative rate).

For efficiency we report average value of speedup = $\frac{t_{ARRIVAL}}{t_{BBFS}}$ where $t_{ARRIVAL}$ is the time taken by ARRIVAL and t_{BBFS} is the time taken by BBFS.

5.3 Comparison with LI [21] and RL [12]

Recall that LI only supports Query type 1, and hence, when we compare the performance of ARRIVAL, we present the running time of ARRIVAL only on Query type 1. On the other hand, when ARRIVAL is compared with RL, we measure the querying time across all query types.

Table 3 lists the querying times of ARRIVAL and RL (as well as BBFS, which we discuss in next section) across all four static graphs listed in Table 2. RL successfully completes on GPlus and Dblp; in Twitter and Freebase it crashes due to running out of memory. Compared to RL, ARRIVAL is at least 40 times faster. This result is particularly significant given the fact that the code of RL made available by the authors is multi-threaded.

LI crashes on all of our datasets due to running out of memory. This non-scalability of LI stems due to relying on an index structure whose memory cost grows exponentially with label set size. Since LI fails to run on any of our datasets, we extract small subgraphs from the Twitter network to compare its performance against ARRIVAL. To extract a subgraph containing X% of the nodes, a random node is selected and a breadth first search (BFS) tree is generated till the tree spans X% of nodes. Next, we add all edges that have both their endpoints among nodes included in the tree. Finally, we grow this BFS tree further for a larger value of X and add edges in the same manner. Owing to our subgraph extraction algorithm, an extracted graph containing X% of the nodes is always a subgraph of the one containing Y% of nodes if $X < Y$.

LI crashes out of memory even when we run it on a subgraph of Twitter containing only 1% of the nodes. Therefore, to further reduce the memory burden, we retained only the top-30 labels in Twitter instead of the top-1000. Fig. 4(a) presents the memory consumption of LI and ARRIVAL against the network size.

[6]In Twitter, we abandoned BBFS searches that exceeded 1 minute. The reported time is the average only for walks that finished within this time limit. The true querying time is expected to be larger.

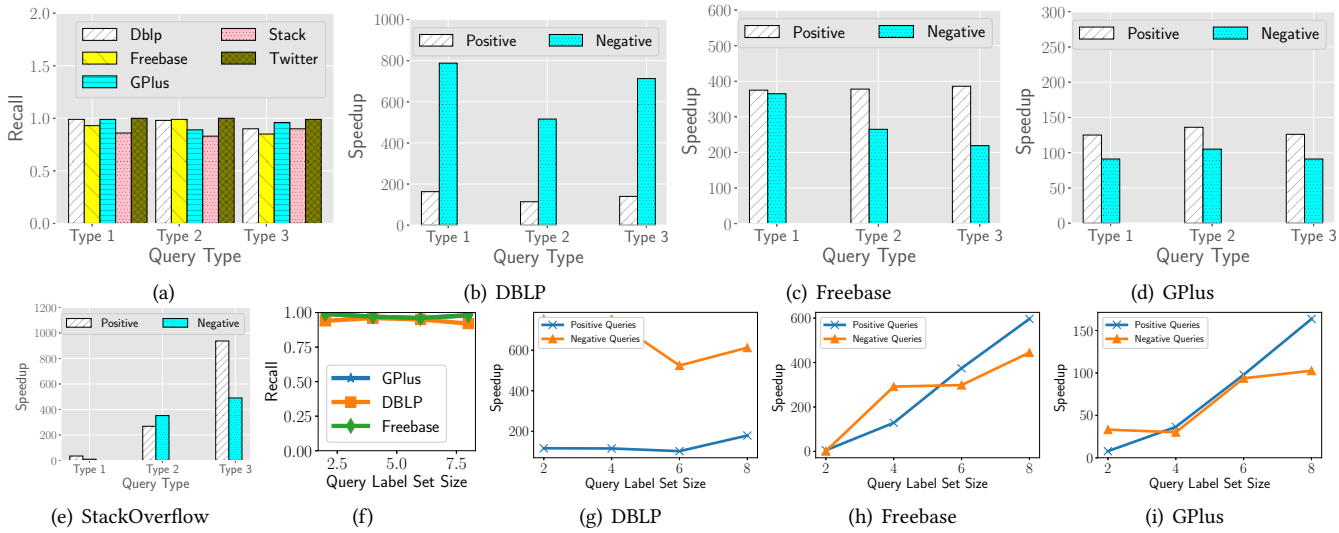


Figure 5: (a-e) Impact of query-type on (a) the recall and (b-d) speedup over BBFS.(f-i) Impact of the number of labels in the query regex on (f) the recall and (g-i) speedup over BBFS.

As we see in Fig. 4(a), the memory consumption of LI increases exponentially with network size since, as more nodes are added, the number of unique labels in the subgraph increases as well. Beyond 1.8% of the Twitter network, even with only 30 labels, LI crashes. In contrast, ARRIVAL consumes 30 times less memory with a linear growth rate.

To establish the exponential growth of memory consumption of LI more directly, in Fig. 4(b), we vary the number of labels in the subgraph containing 1.8% of the Twitter network and measure the impact on memory consumption. While the memory consumption remains almost constant in ARRIVAL, it grows exponentially in LI.

Although LI is more memory intensive, it is close to 1000 times faster than ARRIVAL (Fig. 4(c) and Fig. 4(d)). On the other hand, ARRIVAL is at least 30 times faster than RL. The fast querying time of LI is a direct consequence of two key factors. First, LI is supported by a pre-computed index. Second, and more importantly, LI is optimized for Query type 1 where it knows that no edge in a graph needs to be processed more than once. In contrast, both ARRIVAL and RL need to handle any possible regular expression and thus cannot exploit query-type-specific optimization. This advantage of LI in querying time however comes at the cost of being limited to networks with low number of labels and lack of generalizability to handle a wider variety of regular expressions.

Overall, the empirical evaluation reveals that when the number of labels in a network that is small, LI provides faster querying time. However, for networks with more than 32 labels, which is often the case on real world networks, ARRIVAL is more appropriate. The other competitor, RL, works for large networks but is 30 times slower than ARRIVAL.

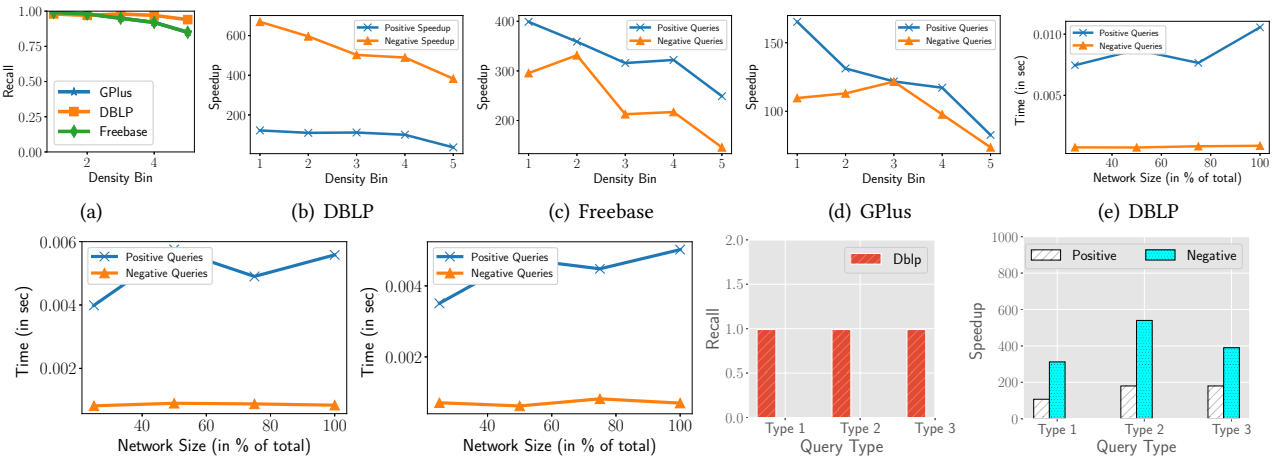
Table 3: Average recall of ARRIVAL and running times of ARRIVAL, RL and BBFS.

Dataset	Recall	ARRIVAL time(ms)	RL time (ms)	BBFS time (ms)
GPlus	0.98	12	490	1404
DBLP	0.93	3.8	1230	2139
Freebase	0.98	3.7	Crash	1273
StackOverflow	0.86	3.97	NA	1322
Twitter[6]	1	3.96	Crash	530

5.4 Performance Evaluation against BBFS

Before we dive deeper, in Table 3 we present the average recall, running times and speedup obtained by ARRIVAL. ARRIVAL achieves an average recall of at least 0.94 across all datasets, while being at least two orders of magnitude faster than BBFS. It is interesting to note that the running time in GPlus is higher than in DBLP and Freebase, although GPlus is a much smaller dataset. This behavior stems from the fact that the running time of ARRIVAL is smaller in negative queries than in positive queries. In negative queries, the explored paths are often small in length where any possible extension violates the regex constraint and therefore the walk hits a dead-end. In positive queries, on the other hand, there often exists a long path that ultimately leads to the destination. In GPlus, the proportion of positive queries is much higher and hence we observe a larger running time.

5.4.1 Performance in each Query-type. In this experiment, we analyze the performance with respect to each query-type. Fig. 5(a) presents the recall rates and Figs. 5(b)-5(e) present the speedups in a range of datasets including StackOverflow, which is a dynamic network. Overall, ARRIVAL achieves a recall of higher than 0.85 across all scenarios, while providing substantial speedups. We also observe that the speedups across query-types are not consistent with dataset. This inconsistency is a manifestation of the different label distribution in each network.



(a) (b) DBLP (c) Freebase (d) GPlus (e) DBLP

(f) Freebase (g) GPlus (h) DBLP (i) DBLP

Figure 6: (a-d) Impact of label frequency on (a) recall and (b-d) speedup. (e-g) Growth of running time with network size. (h-i) Recall and speedup in queries with query-time labels.

5.4.2 Performance against Query-Set Size. We next analyze the recall and speedup against the number of labels in the query regex. Fig. 5(f) presents the recall obtained across a range of datasets. As can be seen, the recall is above 0.94 under all circumstances. In Figs.5(g)-5(i), we present the speedup gained while answering these queries. As visible, ARRIVAL is orders of magnitude faster across all datasets. This result clearly brings out the efficacy of *numWalks* and *walkLength* in keeping the sample size small enough without compromising on the accuracy.

The second trend visible in the above plots is that the speedups generally improve with larger number of labels in the query set. On the other hand, the quality remains almost constant. This is expected since in Query type 1 and 3, the probability of a path satisfying the regex increases with increase in number of labels. Thus, a path is abandoned by BBFS due to non-compatibility with the regex much later after a large number of hops have already been taken. In ARRIVAL, the length is bounded by *walkLength*, and thus, we see an increase in speedup with larger label sets.

5.4.3 Performance against the Label-density Distribution. The default sampling strategy to choose labels in the query regex is frequency-proportional sampling. In this experiment, we investigate how the performance of ARRIVAL varies against the frequency of labels. Towards that end, we divide all labels in a dataset into five buckets based on their density (frequency) in the graph as follows:

(1) Bucket 1 with top-10 labels.
(2) Buckets 2, 3, and 4 with the next 10 most frequent labels each.
(3) Bucket 5 with bottom 20% of labels.

We next construct query regular expressions by randomly sampling labels from a given bucket and measure the performance. This process is then repeated for each of the five buckets. The accuracy obtained in this experiment is shown in Fig. 6(a) and the speedups are shown in Figs. 6(b)-6(d). In these figures, the x-axis indicates the bucket from where the labels are sampled, and the y-axis shows the accuracy and speedup of positive and negative queries.

The results indicate that the problem gets more difficult with decrease in frequency of labels. This behavior is expected. Being a sampling based strategy, ARRIVAL is most effective when the labels are spread across the graph in considerable density. When there are multiple compatible paths between the source and the destination nodes, ARRIVAL has a higher chance of identifying one of them in the random walk. When the label density is low, the number of compatible paths goes down and as a result, the accuracy reduces. Nonetheless, this reduction is minimal and above 0.85 even in bucket 5 across all datasets.

Similar to recall, the speedups also reduce at lower density labels. When the regex is defined over infrequent labels, very few compatible paths exist in the graph. Consequently, BBFS is fast since it does not explore non-compatible paths. As a result, the speedup of ARRIVAL reduces. Nonetheless, the speedup is always above 1.

5.4.4 Scalability against Network Size. We next verify the growth rate of its running time against the size of the network. To measure this property on a given dataset, we extract subgraphs of progressively larger sizes and quantify the change in speedup. The subgraphs are extracted in the same manner as discussed in Sec. 5.3. Figs. 6(e)-6(g) present the results. In general, the running times of both positive and negative queries increase with increase in network size. We notice that ARRIVAL is significantly faster in negative queries than in positive queries. Typically, negative queries correspond to regex constraints that are hard to satisfy. As a result, the paths sampled through random walks are smaller in length than

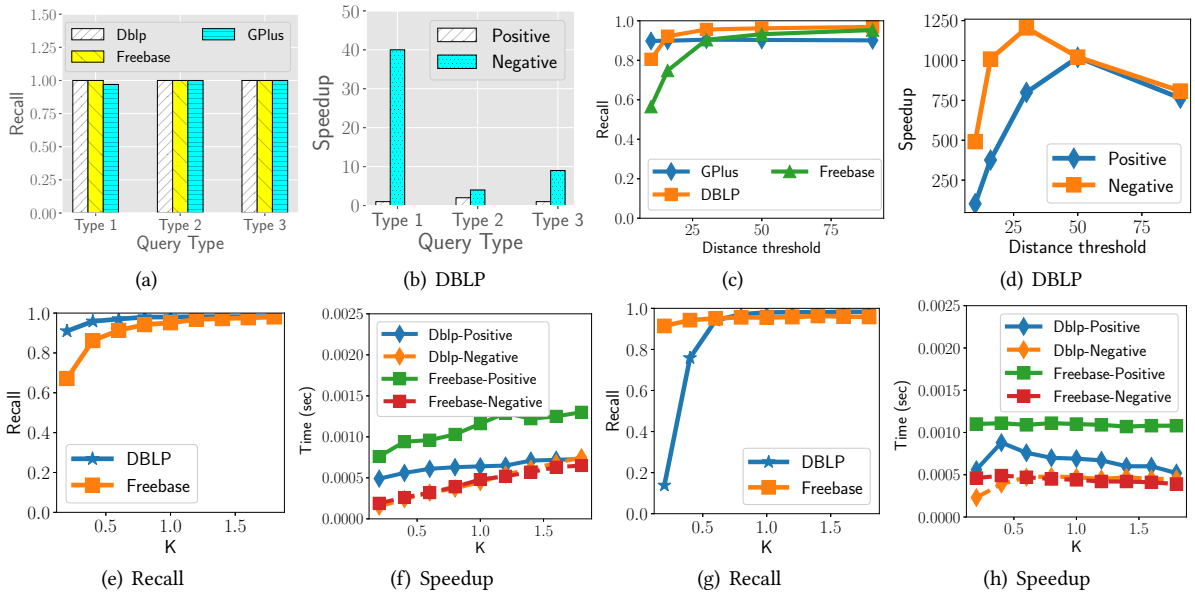


Figure 7: Recall and speedup in queries with (a-b) negation and (c-d) distance bounds. Variation in recall and running time against (e-f) number of walks and (g-h) walk length.

those in positive queries. Hence, we observe the significant disparity in the querying times.

5.4.5 Incorporating Query-time labels. We study the impact of query-time labels on the recall and speedup. This experiment is conducted on the DBLP dataset, where each node is tagged with a feature vector (See Sec. 5.1). We consider the following four query-time label functions.

- **High-quality publisher:** Each node (author) is tagged with the median rank of all papers published by him/her. The rank of a paper ranges from 1 to 5. We use the function $medianRank > \theta$, where θ is randomly chosen from the range $[1, 5]$.
- **Prolific publisher:** Each node is also tagged with the number of papers published. We use the function $numPapers > \theta$, where θ is randomly chosen from the range $[3 - 9]$.
- **Diverse and experienced author:** Each node is tagged with the number of venue-types in which the author has published as well as the number of active years, which is the gap from the year of the first publication to the latest. On these features, we use the function $(yearsActive > \theta_0 \wedge numSubjects > \theta_1)$, where θ_0 and θ_1 are chosen from the range $[3 - 9]$.
- **Diverse or experienced author:** We change the above function by replacing the "and" with an "or". Specifically, $(yearsActive > \theta_0 \vee numSubjects > \theta_1)$.

The queries are generated in the same manner as earlier with the only difference being the use of query-time labels instead of static labels. We do not expect any significant change in the performance of ARRIVAL since supporting query-time labels does not cause any change in the underlying algorithm of ARRIVAL. This effect is visible in Fig. 6(h) and Fig. 6(i), where, as in previous experiments, we observe recall values of 0.99 and speedups in the range of 100 to 500 times across all three query-types.

5.5 Performance on Other Query Classes

In this section, we evaluate the performance of ARRIVAL in other query classes.

5.5.1 Negation queries. In this experiment, we generate queries in the same manner as described in Sec. 5.2.2; however, we apply a negation operation on the generated query.

Fig. 7(a) presents the recalls in the three datasets of GPlus, DBLP and Freebase. In general, a negation operation enlarges the set of compatible and reachable paths between the source and destination since we are specifying only what is not allowed. Since more compatible and reachable paths exist, it is easier for ARRIVAL to sample one of them and consequently we see a recall of nearly 1 across all scenarios. Owing to this same reason, we do not see a big bap in the querying time of BBFS and ARRIVAL on positive queries (Fig. 7(b)) as a compatible reachable path is found in the initial stages of the search procedure. However, on negative queries, ARRIVAL is significantly faster since, unlike BBFS, it only samples *numWalks* paths. Combining both positive and negative queries, ARRIVAL is 27.4, 3.9, and 8.9 times faster than BBFS on query types 1, 2, and 3 respectively.

5.5.2 Distance bound queries. Next, we evaluate the performance on queries with distance bounds, i.e., in addition to regex compatibility, the reachable path must have less than θ edges, where θ is a user-provided threshold. The accuracy achieved by ARRIVAL is shown in Fig. 7(c). Two trends

emerge from Fig. 7(c). The recall improves with increase in distance bound threshold. When the threshold is small, very few paths exist that satisfy all criteria. Consequently, their chances of being sampled by ARRIVAL is low. The second trend visible in this plot is that the recall is lowest in Freebase. This results from the fact that Freebase has the longest diameter of 92 among all datasets, and in general, the paths are longer. Consequently, the chance of a path being pruned out due to the bound is larger in Freebase. In Fig. 7(d), we plot the speedup in DBLP. Consistent with previous results, ARRIVAL is orders of magnitude faster. Similar results are observed in Freebase and GPlus.

5.6 Impact of Parameters

5.6.1 Number of Walks. To control the number of walks to be performed, we set it to $K \times numWalks$, where K is varied in the range $[0.2, 2.0]$ and *numWalks* is the default value of this parameter. For example, when $K = 0.5$, *numWalks* is set to half its recommended value. Similarly, at $K = 1.5$, *numWalks* is 50% higher than its recommended value. With higher number of walks, we draw more samples and thus it is expected that both recall and running time would increase. These effects are visible in Fig. 7(e) and Fig. 7(f)

5.6.2 Length of Walks. Next, we investigate the impact of *walkLength*. To increase it systematically, as in the case of *numWalks*, we set the maximum length of each walk to $K \times walkLength$ and vary K in the range $[0.2, 2.0]$. Fig. 7(g) presents the impact on quality, which as expected, improves marginally. Interestingly, we see a slight decrease in the running times of positive queries with increase in *walkLength* (Fig. 7(h)). Although it may appear counter-intuitive, in reality it is not. When *walkLength* is high, the chance that a walk reaches the destination through a compatible path is higher. Since positive queries terminate as soon as a compatible reachable path is found, the total number of walks performed in a positive query is likely to go down with increase in *walkLength*. On the other hand, for a smaller *walkLength*, we are likely to perform more walks till a reachable path is found. Hence, the above effect is observed.

6 RELATED WORK

Regular Path Queries on Graphs: In their highly cited work, Mendelzon and Wood [16] addressed the difficulty of answering reachability on labeled graphs using simple paths whose label sequence conform to a user-specified regular language. They showed that, in general, this is NP-Hard but there are certain combinations of regular expressions and graphs for which the problem can be solved optimally in polynomial time. Recent works have further classified the regular languages for which regular simple path reachability can be answered in polynomial time and which can not be [4]; also explored alternatives to simple path semantics (i.e., shortest and arbitrary paths satisfying regular expressions) [14]. Despite the activity, none of these methods were implemented and evaluated on large-scale graphs such as those we consider in our work.

As a result of these formidable complexity results for general regular path reachability queries, researchers have focused on building indexing and query processing systems for a much simpler label constraints on paths – particularly for the variant where edge labels in the reachability path can only come from a specified whitelist (or blacklist) of allowed (or disallowed) labels [11, 21, 23]. However, not supporting the full spectrum of regular expressions can be a serious limitation for advanced SPARQL querying over knowledge graphs [5] and social graphs. Some algorithms for RPQs enumerate all C-compatible paths between the source and destination for a given regex C[16, 22]. We do not compare against these works since the answer sets are different.

Random Walks for Reachability on Graphs: Use of random walks for answering *s-t* reachability on graphs has been considered in theoretical computer science in the past. For instance, Feige [9] used random walks over static undirected graphs to develop a randomized algorithm for the reachability problem. More recently, Anagnostopoulos et al. [3] developed random walks based graph algorithms, including for reachability, over rapidly changing dynamic undirected graphs. Unfortunately, all these works only offer a theoretical treatment of the approach, and more importantly, limit themselves to unlabeled graphs.

Nazi et. al. [17] propose a strategy to randomly sample *nodes* from a graph according to the stationary distribution. ARRIVAL works by sampling paths from a given source node to a destination node and finding compatible points of intersection between these paths. The length of the paths and number of samples to be drawn are therefore optimizing two different distributions, which in turn, leads to incompatible design choices.

7 CONCLUSION

In this work, we developed an index-free algorithm called *ARRIVAL: Approximate Regular-simple-path Reachability In Vertex and Arc (directed edges) Labeled Graphs* to answer label-constrained reachability queries. ARRIVAL extends the state of the art through several dimensions. First, it unleashes the power of regular expressions in defining label constraints. Second, ARRIVAL allows the user to define *query-time labels*, which are functions over existing labels in the graph. The index-free approach also allows ARRIVAL to easily adapt to dynamic graphs since all search operations are directly performed on the graph without relying on any other precomputed information. Finally, ARRIVAL achieves a recall of 0.95 on average while being orders of magnitude faster than the baseline algorithm of BBFS.

8 APPENDIX

A Negation Operator

The negation of a regex is also a regex. The standard procedure to compute the negation of a regex is as follows.

(1) First, the non-deterministic finite automaton (NFA) of the regex is constructed using Thompson's algorithm.
(2) Next, the NFA is converted to a deterministic finite automaton (DFA).
(3) Finally, in the DFA, all the accepting states are made non-accepting and vice-versa.

Step 2, where the NFA is converted to a DFA, may take exponential time in the worst case. Hence, we support negation only on those regex types where the NFA constructed by Thompson's algorithm is already a DFA. The procedure is as follows.

(1) First, the NFA of the un-negated regex is constructed using Thompson's algorithm.
(2) Next, we check if the automaton is non-deterministic by checking for the presence of multiple edges outgoing from a single state on the same symbol.
(3) If there is no such state, then it is a DFA and we flip the accepting and non-accepting states. Otherwise, we reject the query as an unsupported one.

Note that the DFA constructed has outgoing edges for *every* label in $\mathcal{L}$ associated with each state. Those labels that the DFA cannot accept in a given state are directed to a non-accepting state that the automaton cannot exit from (it has a self-loop). When the flipping process takes place, that non-accepting state becomes accepting.

B Intuition behind NP-hardness

[16] studies the *fixed regular path* problem, where given a graph and a regex R, the goal is to identify all pairs of nodes that are connected by a simple path compatible to R. In our problem, we are interested only on a specific pair of nodes identified as the source and the destination nodes. Theorem 2.1 in [16] show that even for a specific source-destination pair the problem is NP-hard. This result is not surprising; if there exists a polynomial-time algorithm for a given pair, then there also exists a polynomial-time algorithm for all pairs of nodes as there are only $O(n^2)$ pairs in a graph. We next illustrate why the search space is exponential.

Let us consider the graph in Fig. 2(a) where v_1 and v_5 are the source and destination nodes respectively and a^*ba^* is the regex constraint. Let us consider the paths $P_1 = \langle v_1, v_2 \rangle$ and $P_2 = \langle v_1, v_3, v_2 \rangle$. Both paths have originated from the source node v_1 and have reached node v_2, and both are potentially compatible with a^*ba^* at this stage. However there is a difference between these two paths: P_1 cannot be expanded to achieve a feasible solution for reachability, whereas P_2 can. The reason is because P_2 has already visited b and P_1 has not.

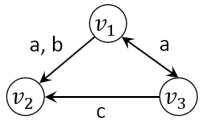


Figure 8: A sample multi-labeled graph

To put it in terms of the automaton that accepts a^*ba^* (Figure 2(b)), P_1 is still in state q_0 while P_2 has already moved to q_1. The key point here is this: *the continuation from v_2 can give us a feasible solution from one of the states of the automaton and not from the other.* Generalizing this argument, we realize that simply *reaching* an intermediate vertex between the source and the destination along a potentially compatible path is not enough; we need to know which states of the automaton can be reached along a path to that vertex. This implies that at any intermediate vertex, BFS needs to maintain information for *every* path from the source to that intermediate vertex. This is potentially an exponential number.

Note that there exist some regular expressions for which BFS returns the optimal solution in polynomial time. For example, if we approach the graph in Fig. 2(a) with the regex $(a|b)^*$ then this amounts to simple reachability in the graph with the edges labeled c eliminated. In fact, this is the only class of regex considered in majority of the existing RSP querying techniques [11, 21, 23], and we have been referring to this class as LCR (Label Constrained Reachability).

Memory Consumption: Computation cost is not the only bottleneck. The memory consumption of BFS is also exorbitantly high since we need to store all paths corresponding to each possible extension (See line 5 in Alg. 1). Without explicitly storing the paths, there is no alternative to determining whether an extension of the current path is simple or cyclic.

To address these challenges, we develop an algorithm called *ARRIVAL: Approximate Regular-simple-path Reachability In Vertex and Arc (directed edges) Labeled Graphs.*

C Managing automaton branches, path compatibility and backward walks

C.1 Handling branches in the NFA. In a multi-labeled graph, while performing a random walk, we may encounter branches. Consider the automata in Fig. 2(b) for regex $C = a^*ba^*$ and the path $P = \{v_1, v_2\}$ in Fig. 8. Two different label sequences are contained in this path: $S_1 = \{a\}$ and $S_2 = \{b\}$. While S_2 is C-compatible S_1 is not. In the random walk, we randomly sample one of the labels from a multi-labeled edge (or node) and decide path compatibility based on the sampled sequence. Thus, when we abandon a walk in the random walk due to not being potentially compatible, it only tells us that the sampled sequence is not potentially compatible. The path may contain some other sequence that is potentially

Algorithm 3 Compatibility and Potential Compatibility Checks

Require: Path $P = \{e_1, \cdots, e_m\}$ and automata $M = (Q, F, q_o, q_f)$ of regex C, where Q is the set of states, F is the set of state transitions, q_o and q_f are the initial and final states.
Ensure: returns 1 if P is C-compatible, 0 is potentially compatible, -1 otherwise.
1: $curr \leftarrow q_o$
2: $i \leftarrow 1$
3: **while** $i \leq m$ **do**
4: $S \leftarrow l(P.e_i) \cap F(curr)$ ▷ Set of labels that allow a transition. $F(curr)$ denotes the set of all transition in state $curr$
5: **if** $S = \emptyset$ **then**
6: **return** -1
7: $l \leftarrow$ random label from S
8: $curr \leftarrow Q(curr, l)$ ▷ $Q(curr, l)$ denotes the next state following transition through label l from state q
9: $i \leftarrow i + 1$
10: **if** $curr = q_f$ **then return** 1
11: **else return** 0

compatible. The next section presents the exact algorithm of this process.

C.2 Checking path compatibility. Alg. 3 presents the pseudocode to check if a path is C-compatible (returns 1) or potentially compatible (return 0) to a given regex C. Starting from the first edge $P.e_1$ in path P and initial automata state q_o, we first check if q_o allows some transition through a label that is contained in $P.v_1$ (lines 4-6). If yes, then one such label l is picked randomly (line 7) and we move to the next state in the automata that follows from q_o through label l (line 8). In addition, we also move to the next edge $P.e_2$ in path P (line 9). This process continues iteratively till one of the following events happen:

- We reach a node $P.e_i$ that does not allow any transition from the current automata state (line 5). This means that the generated label sequence from P is not potentially compatible (Recall App. C.1).
- After traversing the last edge in P, the current automata state is the final state q_f. This ensures that P is C-compatible.
- We traverse the last edge in P where the current automata state is not the final state q_f. This ensures that P is potentially compatible.

In this pseudocode, we assume labels are present only in edges. Incorporating node labels can be done analogously.

Computation Complexity: $F(curr)$, which provides the transitions from an automata state $curr$, is maintained as a hashset. Thus, in $O(1)$ time, we can identify if some label l allows a transition from $curr$. Similarly, we maintain $Q(curr)$ as a hashmap that takes a label l as key and outputs the state that follows from $curr$ through label l in $O(1)$ time. In each iteration, we hash each label present in the i^{th} node of P and identify those labels that allow a transition from current automata state $curr$ (line 4). This operation consumes $O(|l(P.v_i)|)$ time. We pick one of the valid transitions randomly and move to the next state in the automata (line 8), which consumes $O(1)$ time. Since $m \leq walkLength$, the complexity of checking compatibility or potential compatibility is $O(walkLength \times L)$, where L is the average number of labels per node in the graph. Since L is typically a small value and $L \ll walkLength$, $O(walkLength \times L) \approx O(walkLength)$.

In the above complexity analysis, we assume that the cost of query-time label functions is $O(1)$. If that is not the case, then the time complexity changes to $O(walkLength \times L \times T)$, where T is the complexity of computing query-time labels. As discussed in Sec. 2, in this work, we offer the ability to use query-time labels, but do not deal with analyzing its efficiency and correctness.

C.3 Backward walks: In the backward walk, we need to ensure potential *backward* compatibility. A path is potentially backward compatible if it is a pattern to some suffix of the regex constraint. For the backward walk, the automaton of the regex needs to be reversed. To reverse an automata, a transition from state s_1 to s_2 is changed to s_2 to s_1. In addition, the initial and final states are interchanged.

D Random Walk Guarantees for Unlabeled Reachability

Definition 9. *Given probability distributions μ and ν defined on a finite set X, the* total variation distance *between μ and ν is defined as*

$$\|\mu - \nu\|_{TV} := \max_{A \subseteq X} |\mu(A) - \nu(A)|,$$

and has the property that

$$\|\mu - \nu\|_{TV} = \frac{1}{2} \sum_{x \in X} |\mu(x) - \nu(x)|.$$

Definition 10. *A Markov chain defined on a state space $\mathcal{X}$, having transition matrix P and stationary distribution π, has t-*step distance from stationarity *$d(t)$, where*

$$d(t) := \max_{x \in \mathcal{X}} \|P^t(x, \cdot) - \pi\|_{TV}.$$

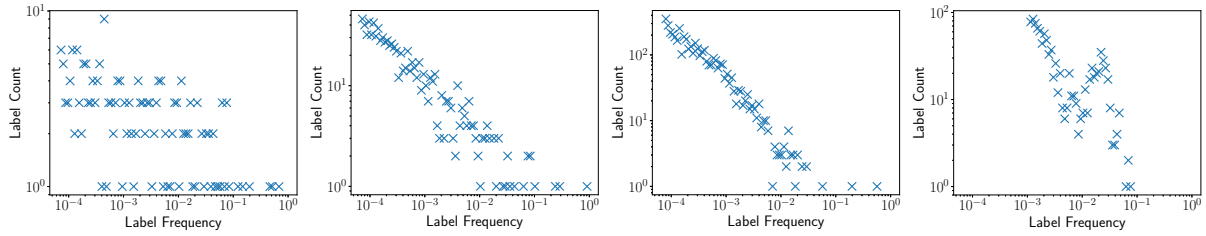
*Further, given $\varepsilon > 0$, we say the ε-*mixing time *of the chain is defined as*

$$t_{mix}(\varepsilon) := \min\{t : d(t) \leq \varepsilon\}.$$

Definition 11. *Given two probability distributions $\overline{\mu}, \overline{\nu}$ on the set $\{1, 2, \ldots, n\}$, the* fuzzy overlap *between $\overline{\mu}, \overline{\nu}$ is defined as*

$$\alpha(\overline{\mu}, \overline{\nu}) = n \left(\sum_{i=1}^{n} \max\left\{0, \overline{\mu}(i) - \frac{1}{2n}\right\} \cdot \max\left\{0, \overline{\nu}(i) - \frac{1}{2n}\right\} \right)$$

Theorem 5 (Sengupta et. al. [18]). *Suppose we have n bins, k red balls and k blue balls. Each ball is thrown in a bin independently of all other balls. The red balls are thrown according to the probability distribution μ and the blue balls according to ν.*



(a) DBLP (b) Freebase (c) GPlus (d) Twitter

Figure 9: Distribution of label frequencies. The frequency of a label ranges from 0 to 1 and denotes the proportion of nodes (or edges) that contain the particular label. We do not plot the distribution of StackOverflow since it contains only three labels with frequencies 0.28, 0.32 and 0.4.

Further, given probability distributions $\overline{\mu}$ and $\overline{\nu}$ such that

$$\|\mu - \overline{\mu}\|_{TV} \leq \frac{1}{2n}, \|\nu - \overline{\nu}\|_{TV} \leq \frac{1}{2n}, \tag{5}$$

the probability that there is a bin which has both a red and a blue ball is at least $1 - \frac{1}{n}$ if

$$k \geq k^* = \left\{ \frac{16n^2 \ln n}{(\alpha(\overline{\mu}, \overline{\nu}))^2} \right\}^{\frac{1}{3}},$$

whenever $k^* \cdot \max\{\max\{\mu(i) : 1 \leq i \leq n\}, \max\{\nu(i) : 1 \leq i \leq n\}\} < 1$.

Proof. Let $X_i = 1$ if the i^{th} bin has a red ball and a blue ball. Since all throws are independent

$$p_i(k) = \mathrm{E}[X_i] = \mathrm{P}\{X_i = 1\} = \left(1 - \{1 - \mu(i)\}^k\right) \cdot \left(1 - \{1 - \nu(i)\}^k\right).$$

Define $X = \sum_{i=1}^{n} X_i$ to be the number of bins containing both a red and blue ball. We are interested in the event $\{X > 0\}$ whose complement is $\{X \leq 0\}$ which can be rewritten as $\{\mathrm{E}[X] - X > \mathrm{E}[X]\}$.

To bound the probability of this event we use the following concentration result:

Lemma 4 (McDiarmid [15]). *Given a set of m independent random variables $Y_i, 1 \leq i \leq m$ and a function f such that $|f(x_1, \ldots, x_m) - f(y_1, \ldots, y_m)| < c_i$ whenever $x_j = y_j, 1 \leq j \leq m, j \neq i$, and $x_i \neq y_i$, then*

$$P\{E[f(Y_1, \ldots, Y_m)] - f(Y_1, \ldots, Y_m) > \varepsilon\} \leq \exp\left(-\frac{2\varepsilon^2}{\sum_{i=1}^{m} c_i^2}\right).$$

If we take the m independent random variables to be the $2k$ choices of bins made by the balls and the function f is the number of bins that contain balls of both colours, we can see that changing the location of a single ball can make a difference of absolute value at most 1 to this function. So we have that

$$\mathrm{P}\{\mathrm{E}[X] - X > \mathrm{E}[X]\} \leq \exp\left(-\frac{(\sum_{i=1}^{n} p_i(k))^2}{k}\right).$$

So, to achieve the desired probability we need

$$\left(\sum_{i=1}^{n} p_i(k)\right)^2 \geq k \ln n. \tag{6}$$

Note that $(1 - (1 - x)^k) \geq 1 - e^{-xk} \geq xk - (xk)^2/2 + \sum_{i \geq 3} (xk)^i / i!$. If $xk < 1$ then we can say that $(1 - (1 - x)^k) \geq xk/2$. So, whenever $\mu(i)k < 1$ and $\nu(i)k < 1$ for $1 \leq i \leq n$, we get that

$$p_i(k) \geq \frac{1}{4}\mu(i)\nu(i)k^2.$$

In view of (5) this implies that (6) is satisfied if

$$k \geq \left\{ \frac{16n^2 \ln n}{(\alpha(\overline{\mu}, \overline{\nu}))^2} \right\}^{\frac{1}{3}}.$$

□

REFERENCES

[1] [n. d.]. CORE Rankings Portal. http://portal.core.edu.au/conf-ranks/. ([n. d.]).

[2] Serge Abiteboul and Victor Vianu. 1999. Regular path queries with constraints. *J. Comput. System Sci.* 58, 3 (1999), 428–452.

[3] A. Anagnostopoulos, R. Kumar, M. Mahdian, E. Upfal, and F. Vandin. 2012. Algorithms on Evolving Graphs. In *Proc. Innovations in Theoretical Computer Science (ITCS '12)*. ACM, 149–160.

[4] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2013. A trichotomy for regular simple path queries on graphs. In *Proc. PODS '13*. 261–272.

[5] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 149–161. https://doi.org/10.14778/3149193.3149196

[6] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. 1999. Rewriting of regular expressions and regular path queries. In *PODS*. 194–204.

[7] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A graphical query language supporting recursion. In *SIGMOD*. 323–330.

[8] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. 2011. Adding regular expressions to graph reachability and pattern queries. In *Proc. ICDE '11*. 39–50.

[9] U. Feige. 1996. A Fast Randomized LOGSPACE Algorithm for Graph Connectivity. *Theor. Comput. Sci.* 169, 2 (1996), 147–160.

[10] George HL Fletcher, Jeroen Peters, and Alexandra Poulovassilis. 2016. Efficient regular path query evaluation using path indexes. In *EDBT*. OpenProceedings. org, 636–639.

[11] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. 2010. Computing Label-constraint Reachability in Graph Databases. In *Proc. SIGMOD '10*. ACM, New York, NY, USA, 123–134.

[12] André Koschmieder and Ulf Leser. 2012. Regular Path Queries on Large Graphs. In *Proc. SSDBM 2012*. 177–194.

[13] Wim Martens and Tina Trautner. 2018. Evaluation and Enumeration Problems for Regular Path Queries. In *ICDT*. 19:1–19:21.

[14] Wim Martens and Tina Trautner. 2018. Evaluation and Enumeration Problems for Regular Path Queries. In *Proc ICDT '18*. Article 19, 21 pages.

[15] C. McDiarmid. 1989. On the method of bounded differences.. In *Surv. Comb.* Cambridge University Press, 148–188.

[16] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (Dec. 1995), 1235–1258.

[17] Azade Nazi, Zhuojie Zhou, Saravanan Thirumuruganathan, Nan Zhang, and Gautam Das. 2015. Walk, not wait: Faster sampling over online social networks. *Proceedings of the VLDB Endowment* 8, 6 (2015), 678–689.

[18] Neha Sengupta, Amitabha Bagchi, Maya Ramanath, and Srikanta Bedathur. 2019. ARROW: Approximating Reachability using Random-walks Over Web-scale Graphs. (2019). To appear in Proc. ICDE '19.

[19] Stephan Seufert, Klaus Berberich, Srikanta Bedathur, Sarath Kumar Kondreddi, Patrick Ernst, and Gerhard Weikum. 2016. ESPRESSO: Explaining Relationships between Entity Sets. In *CIKM*. 1311–1320.

[20] Ken Thompson. 1968. Programming Techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. https://doi.org/10.1145/363347.363387

[21] Lucien D.J. Valstar, George H.L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *Proc. SIGMOD '17*. ACM, New York, NY, USA, 345–358. https://doi.org/10.1145/3035918.3035955

[22] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query planning for evaluating SPARQL property paths. In *SIGMOD*. 1875–1889.

[23] Lei Zou, Kun Xu, Jeffrey Xu Yu, Lei Chen, Yanghua Xiao, and Dongyan Zhao. 2014. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems* 40 (2014), 47 – 66. https://doi.org/10.1016/j.is.2013.10.003