Landmark indexing for scalable evaluation of label-constrained reachability queries

Anonymized for Double-Blind Review

ABSTRACT

Consider a directed edge-labeled graph, such as a social network or a citation network. A fundamental query on such data is to determine if there is a path in the graph from a given source vertex to a given target vertex, using only edges with labels in a restricted subset of the edge labels in the graph. Such label-constrained reachability (LCR) queries play an important role in graph analytics, for example, as a core fragment of the so-called regular path queries which are supported in practical graph query languages such as the W3C's SPARQL 1.1, Neo4j's Cypher, and Oracle's PGQL. Current solutions for LCR evaluation, however, do not scale to large graphs which are increasingly common in a broad range of application domains. In this paper we present the first practical solution for efficient LCR evaluation, leveraging landmark-based indexes for large graphs. We show through extensive experiments that our indexes are significantly smaller than state-of-the-art LCR indexing techniques, while supporting orders of magnitude faster query evaluation times.

CCS Concepts

•Information systems → Graph database models;

Keywords

Label-constrained reachability, labeled graph, path queries

1. INTRODUCTION

Graph structured data sets are ubiquitous in contemporary application scenarios. Examples here include social networks, linked data, biological and chemical databases, and bibliographical databases. Typically, the edges of these graphs are labeled, denoting relationship semantics in the application domain. For example, in a social media network vertices represent people, webpages, and blog posts, and edges between people denoting social relationships might be labeled with "friendOf" or "relativeOf", whereas non-social relationships between vertices might be labeled "likes", "visits", or "bookmarked."

As the size of these data sets continues to grow, scalable solutions for graph query processing become increasingly important. A fundamental type of query on graphs is *reachability queries*, where we are interested to determine whether or not there is a path from a given source vertex to a given target vertex. Often, application scenarios require that the set of allowable paths be constrained. A basic path constraint is to limit our search to a subset of the edge-types present in the graph. We refer to these as *label-constrained reachability queries.*

> (LCR) Given vertices s and t of a graph G and a subset L of the set of all edge labels $\mathcal{L}$ of G, determine whether or not there exists a path from s to t in G using only edges with labels in L.

For example, in a social network we may only be interested in edges with labels denoting social relationships between people (and not, for example, edges labeled with non-social relationships such as people visiting particular webpages or liking posts); in a biological database we may only be interested in interaction pathways between proteins (and not, for example, edges labeled with citation relationships holding between protein vertices and vertices denoting scientific publications).

Such constrained reachability queries also appear as an important fragment of the language of *regular path queries* [4, 5, 30], which are essentially reachability queries constrained by regular expressions. Indeed, formulated in terms of regular path queries, LCR is equivalent to the problem of determining whether or not there is a path in G from s to t such that the concatenation of the edge labels along the path forms a string in the language denoted by the regular expression $(\ell_1 \cup \cdots \cup \ell_n)^*$, for $L = \{\ell_1, \ldots, \ell_n\}$, where $\cup$ is disjunction and $*$ is the Kleene star. LCR and, more generally, regular path queries are supported in practical graph query languages such as SPARQL 1.1[1], PGQL [27], and openCypher[2]. In the study of efficient processing of regular path queries, scalable solutions for reachability play an important role [13].

From these observations, it is clear that practical solutions for LCR query processing on large graphs are critical for contemporary graph analytics. The study of efficient domain-independent solutions for answering LCR queries was initiated in the work of Jin *et al.* [14] with several re-

SIGMOD 2017 Raleigh, North Carolina, USA

© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00

DOI: 10.475/1234

[1]http://www.w3.org/TR/sparql11-query/
[2]http://www.opencypher.org

cent follow-up studies [6, 8, 19, 23, 35]. As we detail in Section 2 and experimentally demonstrate in Section 6, current state-of-the-art solutions unfortunately do not scale well to larger graphs which are common in contemporary applications. Given the basic nature of LCR queries, it is an important problem to address this limitation.

Our contributions

In our work, we remedy this situation, presenting the first LCR solutions which scale to large graphs. Our approach leverages the conceptually simple idea of landmark-based indexing, which has been shown to be exceptionally successful for standard (unconstrained) reachability query processing [1, 2, 17, 31]. Roughly speaking, our method chooses a small number of landmark vertices and precomputes all the information needed to answer queries on LCR from a landmark. At query time, we basically conduct a label-respecting breadth-first search (BFS), but we exploit the stored information when possible to get a shortcut from a landmark to the target vertex.

To further improve query performance, we present two extensions: One is for finding landmarks more quickly during the BFS, and the other one is for efficiently figuring out irrelevant vertices even when there is no shortcut from a found landmark to the target vertex.

We demonstrate that our approach scales to orders of magnitude larger graphs and orders of magnitude faster query evaluation than current solutions. As an additional feature, our solutions are not overly complex, and hence have good potential for practical impact. A link will be made available to our complete C++ codebase, including testing framework and full implementations of all indexing strategies used in our experimental study, as open source for further study.

The rest of this paper is organized as follows. After a discussion of related work on this problem (Section 2) and preliminary definitions (Section 3), we explain our landmark-based method (Section 4) and the two extensions (Section 5). Then, we show our experimental results (Section 6). We give concluding remarks and indications for further research in Section 7.

2. RELATED WORK

Reachability queries (without label constraints) have recently attracted much research attention in the database community (see Xu and Cheng for an excellent survey [33]). For large graphs, the goal is to build an index that can answer queries faster than BFS, which takes $O(n+m)$ time, and has lower memory requirements than the full transitive closure of the graph, which requires $O(n^2)$ space. Here, n and m denote the number of vertices and edges of the graph, respectively. Many indexing methods have been proposed based on the idea of compressing the transitive closure [25, 28], online search guided by precomputed indices [3, 7, 16, 32, 34, 29], and labeling schemes [9, 10, 11, 17, 18, 24, 31].

We next review the relevant related work on LCR and the current best-known method for LCR query evaluation.

Jin et al. [15]. This work presents the first results on LCR queries. Here, two extremes for answering LCR queries are presented, namely, either BFS/DFS or building a full transitive closure on the data graph. They then suggest a *tree-based index framework*, which consists of a spanning tree T and a partial transitive closure NT of the graph. Taken together, T and NT contain enough information to recover the full generalized transitive closure. Based on T, all paths in the graph are partitioned into three sets P_s, P_e and P_n. P_s contains all pairwise paths of which the first edge is a part of T. P_e contains all pairwise paths of which the last edge is a part of T. P_n contains all pairwise paths of which neither the first or the last edge is in T. $NT(u,v)$ contains all path labels of paths in P_n between u and v.

A disadvantage of this method is that it will not work on dense graphs. In this case the size of the spanning tree T is relatively small compared to the graph G. Hence the size of the partial transitive closure NT will not be that much smaller than that of the full transitive closure.

Our method is similar as it also aims to provide a balance between BFS/DFS and building a full transitive closure. We do not build a full transitive closure, which then leads to a higher cost at query evaluation time. However, our approach differs in the sense that we build a full transitive closure for only a selected subset of the vertices. We also do not build or use a spanning tree.

Follow-up work has shown the very limited scalability of the approach of Jin *et al.* [19, 35], and hence we do not consider it further in our study.

Bonchi et al. [6]. There is recent progress on evaluating LCR queries where reasoning about distance is important [8, 12, 19, 23], represented by the state of the art methods of Bonchi *et al.* In particular, in this line of work, the focus of study is the label-constrained shortest path (LCSP) query, for which, given two vertices s, t and a label set L, we want to compute (the length of) the shortest path from s to t using only edges with labels in L. Note that LCSP is more general than LCR, and therefore constructing an index for such queries is a more challenging and significantly different task. Bonchi *et al.* propose two methods to LCSP queries. Both methods give an approximation of the actual shortest distance, and hence they are not helpful to answer LCR queries exactly. Consequently, in our study we do not consider these strategies further.

Zou et al. [35]. This approach is, to our knowledge, the current state of the art on LCR query evaluation. Hence, together with BFS, this represents our baseline for comparison in our experimental study of Section 6.

The Zou *et al.* method decomposes the input graph into strongly connected components (SCC's) $C_1, \ldots, C_k$. For each SCC C_i, a local transitive closure is computed, that is, we find all label sets connecting every pair of vertices in C_i. By using the in- and out-portals of an SCC, we can turn each SCC C_i into a bipartite graph B_i. The union of these bipartite graphs, D, is an acyclic graph. For each pair (C_i, C_j) of SCC's, the topological order of D is used to find all label sets connecting the in-portals of C_i and out-portals of C_j. Finally we find the label sets connecting the internal vertices of each SCC, that is, those that are not an in- or out-portal, to any other vertex and vice versa.

A significant limiting factor of this approach is that it is not effective on graphs with a relatively large SCC. For such graphs, we need to compute a local transitive closure for the SCC, which is computationally intensive. Moreover, if a large part of the vertices in an SCC is an in- and/or out-portal, we might get a bipartite graph consisting of more

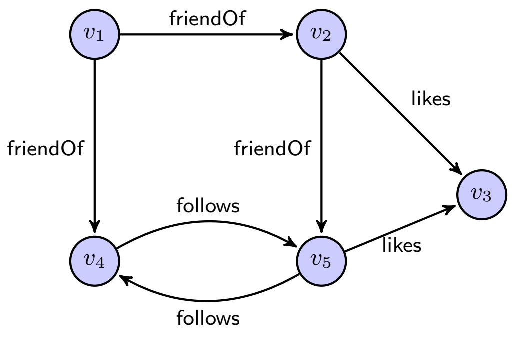


Figure 1: An example of a directed graph with $|V| = 5$ vertices, $|E| = 7$ edges, and edge labels $\mathcal{L} = \{\mathsf{likes}, \mathsf{follows}, \mathsf{friendOf}\}$.

edges than the SCC.

The method we propose does not suffer from these issues, as we do not divide the graph into several SCC's and then use a bottom-up approach to find the full transitive closure. Instead, our approach focuses on building a selective part of the transitive closure. As we show in our experimental study, the method of Zou *et al.* does not scale to the larger graphs which can be processed with our approach.

3. PRELIMINARIES

We study evaluation of LCR queries on directed graphs with edge labels. In this section we give formal definitions of such queries and data. For a positive integer n, we define $[n] = \{1, \ldots, n\}$.

An edge-labeled directed *graph* is a triple $G = (V, E, \mathcal{L})$, where V is a finite set of vertices, $\mathcal{L}$ is a finite non-empty set of labels, and $E \subseteq V \times V \times \mathcal{L}$ is a set of directed labeled edges, that is, $(v, w, l) \in E$ is an edge from v to w with label l. Let $\lambda : E \to \mathcal{L}$ be a mapping from edges to their corresponding labels, that is, $\lambda((v, w, l)) = l$, for $(v, w, l) \in E$.

See Figure 1 for an example of an edge-labeled directed graph. In the sequel, we will often refer to these as "labeled graphs" or just "graphs."

A *path* P in G is a sequence $\langle v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k\rangle$, for some $k > 0$, where $v_i \in V$ for every $0 \le i \le k$ and $e_i \in E$ is an edge from v_{i-1} to v_i for every $i \in [k]$. We say that P is a *path from* v_0 *to* v_k *of length* k. Furthermore, for $L \subseteq \mathcal{L}$, we say that P is an *L-path* if $\lambda(e_i) \in L$ for every $i \in [k]$. When denote the existence of such a path by $v_0 \overset{L}{\leadsto} v_k$. We say that a label set $L \subseteq \mathcal{L}$ is a *minimal label set connecting* v *to* w if (i) $v \overset{L}{\leadsto} w$ and (ii) $v \overset{L'}{\not\leadsto} w$ for any proper $L' \subsetneq L$.

The following gives the formal definition of an LCR-query.

Definition 3.1 (LCR-query). *An* LCR query *is a triple* $(s, t, L) \in V \times V \times 2^{\mathcal{L}}$, *where* $2^{\mathcal{L}}$ *denotes the powerset of* $\mathcal{L}$. *If* $s \overset{L}{\leadsto} t$, *then the query is said to be true (or a true-query). Otherwise, the query is said to be false (or a false-query).*

On the graph of Figure 1, the query $(v_1, v_5, \{\mathsf{friendOf}\})$ is true and the query $(v_1, v_3, \{\mathsf{friendOf}\})$ is false. Furthermore, although $(v_1, v_4, \{\mathsf{friendOf}, \mathsf{follows}\})$ is a true query, the label set $\{\mathsf{friendOf}, \mathsf{follows}\}$ is not a minimal label set connecting v_1 and v_4, as the query $(v_1, v_4, \{\mathsf{friendOf}\})$ is also true.

4. LANDMARK INDEX

In this section, we explain our indexing method called LandmarkIndex (LI for short) and its use in efficient LCR query evaluation.

We first sketch the overall idea in Section 4.1. Then, we explain our indexing algorithm and query algorithm in detail in Sections 4.2 and Section 4.3, respectively. A proof of correctness and the analysis of time and space complexities are presented in Sections 4.4 and 4.5, respectively.

4.1 Overall idea

We start with the basic intuitions behind LI. The most naive indexing method is the following: Given the input graph $G = (V, E, \mathcal{L})$, for each vertex $v \in V$, we store every pair $(w, L) \in V \times 2^{\mathcal{L}}$ to the index $\mathsf{Ind}(v)$ for v if there exists an L-path from v to w. Then, we can answer any LCR query (v, w, L) by checking whether or not the pair (w, L) is in $\mathsf{Ind}(v)$.

Of course, this naive indexing method does not scale to large graphs. An observation towards a more efficient indexing method is that we only have to store (w, L) such that L is a minimal label set connecting v to w. This is because $v \overset{L'}{\leadsto} w$ holds for any $L' \supseteq L$, and hence we can output "true" for the query (v, w, L').

This observation alone is not enough to make the indexing time and the index size sufficiently small. In order to further reduce the indexing time and the index size (at the cost of query time) is to construct $\mathsf{Ind}(v)$ for a small number of vertices, called *landmarks*. Given a query (s, t, L), we conduct a BFS (taking labels into consideration) from s. When we hit a landmark s for which $\mathsf{Ind}(s)$ is constructed, we use it to obtain the answer immediately.

4.2 Indexing algorithm

First, we formally define the notion of an index.

Definition 4.1 (Index). *An* index *on a labeled graph* $G = (V, E, \mathcal{L})$ *is a family* $\{\mathsf{Ind}(v)\}_{v \in V_L}$, *where* $V_L \subseteq V$ *and, for each* $v \in V_L$, *it holds that (i)* $\mathsf{Ind}(v) \subseteq V \times 2^{\mathcal{L}}$ *and (ii) for each* $(w, L) \in \mathsf{Ind}(v)$, *the LCR query* (v, w, L) *is true. For* $v, w \in V$, *we define* $\mathsf{Ind}(v, w) \subseteq 2^{\mathcal{L}}$ *as the set* $\{L \mid (w, L) \in \mathsf{Ind}(v)\}$.

Intuitively speaking, V_L is a set of landmarks and a pair $(w, L) \in \mathsf{Ind}(v)$ indicates that there is an L-path from v to w. The goal is to build a (minimal) index such that a query (s, t, L) for a landmark $s \in V_L$ is true if and only if there exists $(t, L') \in \mathsf{Ind}(s)$ with $L' \subseteq L$.

Now we explain our indexing algorithm (Algorithm 1). Given an integer parameter $k \in \mathbb{N}$ and a graph $G = (V, E, \mathcal{L})$, we start by choosing a set of k *landmark* vertices $V_L \subseteq V$. This is done by picking the k vertices with the highest total degree (in-degree plus out-degree). Let $v_1, \ldots, v_k$ be the chosen k landmarks (in descending order of total degrees). Then, we run LabeledBFSPerLM (in Algorithm 2) for each of $v_1, \ldots, v_k$ in this order.

The objective of LabeledBFSPerLM with a landmark $s \in V_L$ is constructing the index $\mathsf{Ind}(s)$ for s such that $(w, L) \in \mathsf{Ind}(s)$ if and only if L is a minimal label set connecting s to w.

Lines 2 to 3 initialize a min-priority-queue q. The entries for q are pairs $(u, L) \in V \times 2^{\mathcal{L}}$, sorted according to $|L|$, that is, the number of labels in L.

Algorithm 1

```
1: procedure LandmarkIndex($G, k$)
2:     Pick the $k$ vertices $v_1, \ldots, v_k$ with the highest total degree.
3:     for $v \in V$ do indexed($v$) $\leftarrow$ false.
4:     for $i \in [k]$ do
5:         Ind($v_i$) $\leftarrow$ an empty list.
6:         LabeledBFSPerVertex($v_i$).
```

Algorithm 2

```
1: procedure LabeledBFSPerLM($s$)
2:     $q \leftarrow$ an empty priority queue.
3:     $q$.push($s, \{\}$).
4:     while $q$ is not empty do
5:         $(v, L) \leftarrow q$.pop().
6:         if TryInsert($s, (v, L)$) = false then
7:             continue
8:         if indexed($v$) = true then
9:             ForwardProp($s, (v, L)$).
10:            continue
11:        for $(v, w, l) \in E$ do
12:            $q$.push($w, L \cup \{l\}$).
13:    indexed($s$) $\leftarrow$ true.

14: procedure TryInsert($s, (v, L)$)
15:     if $v = s$ then
16:         return true
17:     if $(v, L') \in$ Ind($s$) for some $L' \subseteq L$ then
18:         return false
19:     Remove every $(v, L')$ with $L \subsetneq L'$ from Ind($s$).
20:     Add $(v, L)$ to Ind($s$).
21:     return true

22: procedure ForwardProp($s, (v, L)$)
23:     for $(w, L') \in$ Ind($v$) do
24:         TryInsert($s, (w, L \cup L')$).
```

The loop of Lines 4 to 12 starts by taking the next entry (v, L) from q. The procedure TryInsert$(s, (v, L))$ tries to insert (v, L) into $\mathsf{Ind}(s)$. The variable $\mathsf{Ind}(s)$ has been implemented with a list. In this list we have pairs consisting of a vertex $uinV$ and a list of label sets $X = \{L \mid L \subseteq \mathcal{L}\}$. The pairs are sorted on the vertex id of u. On line 20 we do a binary search for u to find the position of u or the position where u could be inserted. In order to store only minimal label sets in $\mathsf{Ind}(s)$, we process as follows: When L is pairwise incomparable with any L' (with respect to inclusion) in $\mathsf{Ind}(s, v)$, we insert (v, L) to $\mathsf{Ind}(s)$. When L is a subset of some existing entries $(v, L') \in \mathsf{Ind}(s)$, we remove each of these entries and insert (v, L). When L is a superset of some existing entries, we do not insert (v, L) and return **false**. If TryInsert returns **false**, we skip Lines 8 to 12 and continue with processing q at Line 4.

We reach Line 9 if $\mathsf{indexed}(v) =$ **true**. This means that v is a landmark for which LabeledBFSPerLM(v) was already called and $\mathsf{Ind}(v)$ is available. We can expand the index $\mathsf{Ind}(s)$ by using the index $\mathsf{Ind}(v)$ as follows (in a call to ForwardProp): For each $(w, L') \in \mathsf{Ind}(v)$, we try to insert $(w, L' \cup L)$ to $\mathsf{Ind}(s)$. This is valid because $s \overset{L}{\rightsquigarrow} v$ and $v \overset{L'}{\rightsquigarrow} w$ imply $s \overset{L \cup L'}{\rightsquigarrow} w$. We then skip Lines 11 to 12 and continue

Algorithm 3

```
1: procedure Query($s, t, L$)
2:     if $s \in V_L$ then return QueryLandmark($s, t, L$).
3:     for $v \in V$ do marked($v$) $\leftarrow$ false
4:     $q \leftarrow$ an empty queue.
5:     $q$.push($s$).
6:     while $q$ is not empty do
7:         $v \leftarrow q$.pop().
8:         marked($v$) $\leftarrow$ true.
9:         if $v = t$ then
10:            return true
11:        if $v$ is a landmark then
12:            if QueryLandmark($v, t, L$) = true then
13:                return true
14:            continue
15:        for $(v, w, l) \in E \wedge$ marked($w$) = false do
16:            if $l \in L$ then
17:                $q$.push($w$).
18:    return false

19: procedure QueryLandmark($s, t, L$)
20:     // $s$ is a landmark
21:     for $L' \in$ Ind($s, t$) do
22:         if $L' \subseteq L$ then
23:             return true
24:     return false.
```

with processing q at Line 4.

Finally, Lines 11 to 12 push entries of the form $(w, L \cup \{l\})$ to the queue q for every edge $(v, w, l) \in E$ incident to v. We reach this part if and only if (v, L) was inserted into $\mathsf{Ind}(s)$ and v is not a landmark for which LabeledBFSPerLM(v) was already called.

We conclude by noting that landmark s is now indexed.

4.3 Query algorithm

In this section we explain the query algorithm (Algorithm 3) for the landmark index. The query algorithm is very similar to BFS. In fact, if we were to omit Lines 2 to 3 and 11 to 14, Algorithm 3 coincides with BFS.

Query starts by verifying whether or not s is a landmark. If this is the case, we can just return the result of QueryLandmark(s, t, L), which returns true if and only if there exists $L' \in \mathsf{Ind}(s, t)$ with $L' \subseteq L$.

If s is not a landmark, then we start to explore the graph in a similar manner to a BFS. In case v is a landmark on Line 11, we call QueryLandmark(v, t, L). If this call succeeds, we return **true**. If this call fails, we take a new entry from the queue. In case v is not a landmark, we push a vertex w to the queue on Lines 15 to 17 when there exists an edge (v, w, l) with $l \in L$ and $\mathsf{marked}(w) =$ **false**. The variable marked indicates whether a vertex is allowed to push any of it's eligible neighbors to the queue.

4.4 Correctness

In this section, we prove the correctness of LI.

The following lemma is useful for our analysis.

Lemma 4.2. *Let $G = (V, E, \mathcal{L})$ be a labeled graph. Let $L \subseteq \mathcal{L}$ be a minimal label set connecting $s \in V$ to $t \in V$. Then, there exists an L-path $P = \langle v_0, e_1, v_1, \ldots, v_{|P|-1}, e_{|P|}, v_{|P|} \rangle$ from s to t such that for each $j \in \{0, 1, \ldots, |P|\}$*

- $L_{\leq j} := \{\lambda(e_{j'}) \mid 1 \leq j' \leq j\}$ *is a minimal label set connecting* s *to* v_j.
- $L_{>j} := \{\lambda(e_{j'}) \mid j < j' \leq |P|\}$ *is a minimal label set connecting* v_j *to* t.

PROOF. Consider the following auxiliary weighted unlabeled graph $G' = (V', E', d)$. Here, $V' = V \times 2^{\mathcal{L}}$, and we have an edge from (v, L) to (v', L') if and only if there exists an edge (v, v', l) with $L \cup \{l\} = L'$. The weight of an edge $((v, L), (v', L'))$ is set to be $|L'| - |L|$. Note that the weight is always zero or one. It is clear that there exists an L-path from $s \in V$ to $t \in V$ in G if and only if there is a path from $(s, \emptyset)$ to (t, L') for some $L' \subseteq L$ in G'.

Now, consider a shortest path tree rooted at $(s, \emptyset)$ in G' (with respect to the weights on edges of G'). Since L is a minimal label set connecting s from t, there exists a path P' from $(s, \emptyset)$ to (t, L) in the shortest path tree. Let $(v_0 = s, L_0 = \emptyset), (v_1, L_1), \ldots, (v_{|P'|} = t, L_{|P'|} = L)$ be the vertices in P'. From the property of the shortest path tree, for every $j \in \{0, 1, \ldots, |P'|\}$, $L_{\leq j}$ is a minimal label set connecting s to v_j; otherwise, there exists an L'-path from s to v_j for some $L' \subsetneq L_{\leq j}$, which is a contradiction. Similarly, for every $j \in \{0, 1, \ldots, |P'|\}$, $L_{>j}$ is a minimal label set connecting v_j to t; otherwise, there exists an L'-path from v_j to t for some $L' \subsetneq L_{\leq j}$, which is a contradiction.

Then, the path P in G associated with P' has the desired property. □

LEMMA 4.3. *Let* $G = (V, E, \mathcal{L})$ *be a graph and* $k \in \mathbb{N}$ *be an integer. Let* $\{\mathsf{Ind}(v_i)\}_{i \in [k]}$ *be the index constructed by* LANDMARKINDEX(G, k)*, where* $v_1, \ldots, v_k$ *are landmarks. Then, for every* $i \in [k]$*, we have* $(t, L) \in \mathsf{Ind}(v_i)$ *if and only if* L *is a minimal label set connecting* v_i *to* t.

PROOF. ($\Leftarrow$). We prove by induction on i.
Base case $(i = 1)$**:** Let $P = \langle u_0, e_1, u_1, \ldots, u_{|P|-1}, e_{|P|}, u_{|P|}\rangle$ be the L-path from $u_0 = v_1$ to $u_{|P|} = t$ with the property guaranteed by Lemma 4.2. We define $L_{\leq j}$ and $L_{>j}$ for each $j \in 0, 1, \ldots, |P|$ as in Lemma 4.2. For every $j \in \{0, 1, \ldots, |P|\}$, since L_j is minimal, the TRYINSERT$(v_1, (u_j, L_j))$ at Line 6 never fails. Hence, (t, L) is added to $\mathsf{Ind}(v_1)$.
Induction step $(i \geq 2)$**:** Let $P = \langle u_0, e_1, u_1, \ldots, u_{|P|-1}, e_{|P|}, u_{|P|}\rangle$ as the L-path from $u_0 = v_i$ to $u_{|P|} = t$ with the property guaranteed by Lemma 4.2. We define $L_{\leq j}$ and $L_{>j}$ for each $j \in 0, 1, \ldots, |P|$ as in Lemma 4.2. Let $j^* \in [k]$ be the first index such that $u_{j^*} = v_{i^*}$ for some $i^* < i$.

If there exists no such j^*, the claim holds using the same argument as the base case.

If such j^* exists, by induction hypothesis and the fact that $L_{\geq j^*}$ is a minimal label set connecting v_{j^*} to t, we have $(t, L_{\geq j^*}) \in \mathsf{Ind}(v_{j^*})$. Hence, by FORWARDPROP$(s, (v_{j^*}, L))$ at Line 9, we add $(t, L_{\geq j^*} \cup L_{<j^*}) = (t, L)$ to $\mathsf{Ind}(v_i)$.

($\Rightarrow$) Because all the minimal label set L connecting v_i to t are stored in $\mathsf{Ind}(v_i)$ and we only keep minimal sets in TRYINSERT, the claim holds. □

THEOREM 4.4. *Let* $G = (V, E, \mathcal{L})$ *be a graph and* $k \in \mathbb{N}$ *be an integer. Suppose we have constructed an index by* LANDMARKINDEX(G, k)*. Then,* $q = (s, t, L)$ *is a true-query if and only if* QUERY(s, t, L) *with the constructed index returns* ***true***.

PROOF. QUERY basically conducts a BFS only on edges with labels in L. The only difference is that, when we hit a landmark $v \in V_L$ with $(t, L') \in \mathsf{Ind}(v)$ for some $L' \subseteq L$, then we immediately return **true**. This is valid because L' is a minimal label set connecting v to t from Lemma 4.3. □

4.5 Space and time complexity

In this section, we analyze the space and time complexities of our method. For simplicity, we assume that $O(\log n)$ bits and $O(|\mathcal{L}|)$ bits fit in a register and we can perform operations on registers in constant time.

We first analyze the index size of our method. Each landmark needs to store at most $n - 1$ vertices and at most $2^{|\mathcal{L}|}$ label sets per vertex.[3] Since the numbers of landmark is k, the total number of entries in the index is $O(nk2^{|\mathcal{L}|})$. Since each entry requires $O(\log n + |\mathcal{L}|)$ bits, the total index size is $O(nk2^{|\mathcal{L}|}(\log n + |\mathcal{L}|))$ bits.

Next, we analyze the time complexity for index construction. For each landmark $v \in V_L$, we may push $n2^{|\mathcal{L}|}$ entries to the priority queue. Each push takes $O(\log n)$ time. For each label set L, we may traverse each edge. Hence, except for calls to TRYINSERT and FORWARDPROP, we need $O((n \log n + m)2^{|\mathcal{L}|})$ time. Note that each call to TRYINSERT requires $O(2^{|\mathcal{L}|})$ time. Each call to TRYINSERT made by FORWARDPROP would have been made without FORWARDPROP as well. Hence FORWARDPROP does not increase the total time complexity. To summarize, the total time complexity is $O(k((n \log n + m)2^{|\mathcal{L}|} + n2^{|\mathcal{L}|} \cdot 2^{|\mathcal{L}|})) = O((n(\log n + 2^{|\mathcal{L}|}) + m)k2^{|\mathcal{L}|})$.

Finally, we analyze the time complexity for processing queries. In the worst case, we do a full BFS over the graph $O(n + m)$ and run QUERYLANDMARK possibly for each of k landmarks. Each call to QUERYLANDMARK compares L to at most $2^{|\mathcal{L}|}$ label sets. Hence, the time complexity for processing a query is $O(n + m + k2^{|\mathcal{L}|})$.

5. EXTENDED LANDMARK INDEX

In this section, we propose two extensions that make the landmark method more efficient. We first explain the idea of these extensions in Sections 5.1 and 5.2. Then, we show our indexing algorithm and query algorithm in Sections 5.3 and 5.4, respectively. We call the new method with the two extensions LI$^+$.

5.1 Indexing non-landmarks

The first issue of the LI method is that, given a query (s, t, L), it may take a long time before finding a landmark.

We remedy this issue by building a small index for the non-landmarks as well. Suppose that we have constructed an index for all the landmarks. Then, for each non-landmark vertex $v \in V \setminus V_L$, we insert (at most) b entries (v', L) with $v' \in V_L$ to the index $\mathsf{Ind}(v)$, where $b \in \mathbb{N}$ is a parameter. For each of the added entries (v', L) we guarantee $v \overset{L}{\rightsquigarrow} v'$. The entries are found by exploring the graph from $v \in V$ and we stop after having inserted b of them. Procedure LABELEDBFSPERNONLM in Algorithm 4 shows the details.

5.2 Pruning for accelerating false-queries

The second issue is that there can be a strong asymmetry in performance between true- and false-queries. This has

[3]Note that we can tighten this upper bound on the number of labels sets by observing that we can form a set of at most $\binom{|\mathcal{L}|}{\lfloor |\mathcal{L}|/2 \rfloor} = O(2^{|\mathcal{L}|}/\sqrt{|\mathcal{L}|})$ pairwise incomparable label sets over $\mathcal{L}$, a result due to Sperner [26].

to do with the fact that a true-query can immediately stop after finding a landmark, whereas a false-query often needs to explore larger parts of the graph before returning **false**.

We first give an example to explain the idea behind our extension. Let $q = (s, t, L)$ be a query. Suppose that $s \overset{L}{\leadsto} v$ and $v \overset{L}{\not\leadsto} t$ for some landmark $v \in V_L$. We define $R_L(v) := \{w \in V \mid v \overset{L}{\leadsto} w\}$. Then, we observe $w \overset{L}{\not\leadsto} t$ for any $w \in R_L(v)$; otherwise, we have $v \overset{L}{\leadsto} t$, which is a contradiction. Hence we can mark every vertex $w \in R_L(v)$ so that we do not visit w in the future.

It is not practical to compute and store $R_L(v)$ for all $L \subseteq \mathcal{L}$. Hence, for each $v \in V$, we introduce a set $\mathsf{canReach}(v)$ and only keep subsets of $R_L(v)$ for $L \in \mathcal{L}$. Each entry of $\mathsf{canReach}(v)$ is a pair (S, L), meaning that $S \subseteq R_L(v)$. We will guarantee that each entry (S, L) satisfies $|L| \le |\mathcal{L}|/4 + 1$. There are two reasons for this. The first is that we do not want to store all the $2^{|\mathcal{L}|}$ combinations, as this uses a lot of memory. The second reason is that the label sets L with only a few labels can be used by a majority of the queries (s, t, L'), as for smaller L the probability that $L \subseteq L'$ is larger. Moreover the entries (S, L) are sorted descendingly with respect to $|S|$, that is, the number of vertices in S.

Procedure LabeledBFSPerLM$^+$ in Algorithm 4 shows the details on how we select the entries (S, L) for $\mathsf{canReach}$.

5.3 Indexing algorithm

In this section, we explain the indexing algorithm for LI$^+$ (Algorithm 4).

Algorithm 4 starts by creating an ordering $v_1, \ldots, v_n$ for all vertices in V according to their total degrees. The first k vertices are said to be *landmarks* and we let $V_L = \{v_1, \ldots, v_k\}$. Other vertices are said to be *non-landmarks*. Then, we call LabeledBFSPerLM$^+(v_i)$ for each $i \in [k]$ and call LabeledBFSPerNonLM(v_i) for each $i \in \{k + 1, \ldots, n\}$.

LabeledBFSPerLM$^+(s)$ is a slightly adapted version of LabeledBFSPerLM(s). The only difference is that it also constructs $\mathsf{canReach}(s)$. When arriving at a vertex v, we either create a new entry $(L, \{v\})$ or add v to an existing entry in $\mathsf{canReach}(s)$ (Lines 18 to 21). Then, on Lines 27 to 29, we expand each entry in $\mathsf{canReach}(s)$ by copying all the vertices of an existing entry $(S, L) \in \mathsf{canReach}(s)$ to another existing entry $(S', L') \in \mathsf{canReach}(s)$ with $S \subseteq S'$.

LabeledBFSPerNonLM(s, b) is also a slightly adapted version of LabeledBFSPerLM(s). The differences can be summarized as follows:

- The BFS immediately stops when the size of $\mathsf{Ind}(s)$ reaches b (see Line 35 and Line 48 in ForwardPropNonLM).
- We add $(v, L) \in V \times 2^{\mathcal{L}}$ to $\mathsf{Ind}(s)$ only when v is a landmark (see Line 39).
- We explicitly use a variable **marked** to ensure that we visit each vertex at most once.
- We do not continue after the call to ForwardPropNonLM (Line 42) because, unlike the landmark case, $\mathsf{Ind}(v)$ may not have all the information to other vertices because the size of $\mathsf{Ind}(v)$ is bounded by b.

5.4 Query algorithm

Now we explain our query algorithm (Algorithm 5). We use the index Ind and $\mathsf{canReach}$ to speed up the process of answering queries.

For a query $q = (s, t, L)$ with $s \in V_L$, QueryExtensive$(s, t, L, \textbf{marked})$ returns **true** if q is a true-query. In case q is a false-query, it marks all vertices in S' for each

Algorithm 4

```
 1: procedure LandmarkIndex$^+(G, k, b)$
 2:     Let $v_1, \ldots, v_n$ be the vertices sorted descendingly on their total degree.
 3:     for $v \in V$ do indexed$(v) \leftarrow$ false.
 4:     for $i \in [k]$ do
 5:         Ind$(v_i) \leftarrow$ an empty list.
 6:         canReach$(v_i) \leftarrow$ an empty list.
 7:         LabeledBFSPerLM$^+(v_i)$.
 8:     for $i \in \{k+1, \ldots, n\}$ do
 9:         Ind$(v_i) \leftarrow$ an empty list.
10:         LabeledBFSPerNonLM$(v_i, b)$.

11: procedure LabeledBFSPerLM$^+(s)$                 ▷ $s \in V_L$
12:     $q \leftarrow$ an empty priority queue.
13:     $q$.push$(s, \{\})$.
14:     while $q$ is not empty do
15:         $(v, L) \leftarrow q$.pop$()$.
16:         if TryInsert$(s, (v, L))$ = false then
17:             continue
18:         if $(S, L) \in$ canReach$(s)$ for some $S \subseteq V$ then
19:             Replace $(S, L)$ with $(S \cup \{v\}, L)$.
20:         else if $|L| \le |\mathcal{L}|/4 + 1$ then
21:             Add $(\{v\}, L)$ to canReach$(s)$.
22:         if indexed$(v)$ = true then
23:             ForwardProp$(s, (v, L))$.
24:             continue
25:         for $(v, w, l) \in E$ do
26:             $q$.push$(w, L \cup \{l\})$.
27:     for $(S, L) \in$ canReach$(s)$ do
28:         for $(S', L') \in$ canReach$(s)$ and $S \subsetneq S'$ do
29:             Replace $(S', L')$ with $(S \cup S', L')$.
30:     indexed$(s) \leftarrow$ true.

31: procedure LabeledBFSPerNonLM$(s, b)$           ▷ $s \notin V_L$
32:     $q \leftarrow$ an empty priority queue.
33:     $q$.push$(s, \{\})$.
34:     for $v \in V$ do marked$(v) \leftarrow$ false
35:     while $q$ is not empty and $|$Ind$(s)| < b$ do
36:         $(v, L) \leftarrow q$.pop$()$.
37:         if marked$(v)$ = true then continue
38:         marked$(v) \leftarrow$ true.
39:         if $v \in V_L \wedge$ TryInsert$(s, (v, L))$ = false then
40:             continue
41:         if indexed$(v)$ = true and $|$Ind$(s)| < b$ then
42:             ForwardPropNonLM$(s, (v, L))$.
43:         for $(v, w, l) \in E$ do
44:             $q$.push$(w, L \cup \{l\})$.
45:     indexed$(s) \leftarrow$ true.

46: procedure ForwardPropNonLM$(s, (v, L))$ ▷ $s \notin V_L$
47:     for $(w, L') \in$ Ind$(v)$ and $w \in V_L$ do
48:         if $|$Ind$(s)| < b$ then
49:             TryInsert$(s, (w, L \cup L'))$
50:         else
51:             break
```

Algorithm 5

```
 1: procedure Query$^+(s,t,L)$
 2:     if $s \in V_L$ then return QueryLandmark$(s,t,L)$.
 3:     for $v \in V$ do $\mathsf{marked}(s) \leftarrow$ false.
 4:     for $(v, L') \in \mathsf{Ind}(s)$ do        ▷ $v$ is always a landmark
 5:         if $L' \subseteq L$ then
 6:             if QueryExtensive$(v,t,L,\mathsf{marked})$ = true then
 7:                 return true
 8:         $\mathsf{marked}(v) \leftarrow$ true.
 9:     Let $q$ be a queue.
10:     $q$.push$(s)$.
11:     while $q$ is not empty do
12:         $v \leftarrow q$.pop().
13:         $\mathsf{marked}(v) \leftarrow$ true.
14:         if $v = t$ then
15:             return true
16:         if $v$ is a landmark then
17:             if QueryExtensive$(v,t,L,\mathsf{marked})$ = true then
18:                 return true
19:             continue
20:         for $(v,w,l) \in E$ and $\mathsf{marked}(w)$ = false do
21:             if $l \in L$ then
22:                 $q$.push$(w)$.
23:     return false

24: procedure QueryExtensive$(s,t,L,\mathsf{marked})$        ▷ $s \in V_L$
25:     if QueryLandmark$(s,t,L)$ = true then
26:         return true
27:     for $(S', L') \in \mathsf{canReach}(s)$ do
28:         if $L' \subseteq L$ then
29:             $\mathsf{marked} \leftarrow S' \cup \mathsf{marked}$
30:             break
31:     return false
```

$(S', L') \in \mathsf{canReach}(s)$ with $L' \subseteq L$ because we know that $v \overset{L}{\not\rightsquigarrow} t$ holds for $v \in S'$. This is done at most once, because in practice the computation at Line 28 is relatively costly. Moreover by doing it once, we ensure the following. If we have two entries $(S_1, L_1), (S_2, L_2)$ in $\mathsf{canReach}(s)$ with $L_1 \subsetneq L_2$, we get that $S_2 \subseteq S_1$ and that entry (S_2, L_2) is placed before (S_1, L_1) in $\mathsf{canReach}$. By doing the operation at most once, we ensure that we do not redo the computation at Line 28 for (S_1, L_1) after having done it for (S_2, L_2).

Query$^+$ is our procedure for answering queries. Line 1 determines whether v is a landmark and call QueryLandmark(v) when it is the case. Otherwise, we set $\mathsf{marked}(v) =$ **false** for every $v \in V$. When $\mathsf{marked}(v)$ is set to **true**, the vertex v is no longer relevant for our query, that is, either we have visited it or we pruned it.

As s is a non-landmark vertex, we loop over entries $(v, L') \in \mathsf{Ind}(s)$ (Line 4). If $L' \subseteq L$, that is, there is an L-path from s to v, then we can run QueryExtensive(v,t,L). If it returns **true**, we can return **true**. Otherwise, we set $\mathsf{marked}(v)$ to be **true**. Note that QueryExtensive(v,t,L) may mark more vertices using the information of $\mathsf{canReach}(v)$.

In case no landmark v was able to resolve the query on the loop from Line 4, we proceed to the part that essentially conducts a BFS (from Line 9). When we hit a landmark $v \in V_L$ along the way, we run queryExtensive(v).

5.5 Correctness

In this section we prove the correctness of LI$^+$.

Lemma 5.1. *Let $G = (V, E, \mathcal{L})$ be a graph and $k, b \in \mathbb{N}$ be non-negative integers. Let $\{\mathsf{Ind}(v_i)\}_{i \in [n]}$ be the index constructed by calling* LandmarkIndex$^+$*(G, k, b) with the ordering $v_1, \ldots, v_n$. Then, the following hold:*

- *For every $i \in [k]$, we have $(t, L) \in \mathsf{Ind}(v_i)$ if and only if L is a minimal label set connecting v_i to t.*
- *For every $i \in \{k+1, \ldots, n\}$, we have $(t, L) \in \mathsf{Ind}(v_i)$ only if L is a minimal label set connecting v_i to t.*

Proof. The proof of the first claim is completely the same as that of Lemma 4.3. The second claim holds analogously because the only change is imposing an upper bound on the size of $\mathsf{Ind}(v_i)$ for $i \in \{k+1, \ldots, n\}$. □

Lemma 5.2. *Let $G = (V, E, \mathcal{L})$ be a graph and $k, b \in \mathbb{N}$ be non-negative integers. Let $\{\mathsf{canReach}(v_i)\}_{i \in [k]}$ be constructed by calling* LandmarkIndex$^+$*(G, k, b) with the ordering $v_1, \ldots, v_n$. Then, for every $i \in [k]$, $(S, L) \in \mathsf{canReach}(v_i)$, and $v \in S$, we have $v_i \overset{L}{\rightsquigarrow} v$.*

Proof. Vertex v is added when BFS from v_i using only edges with labels in L hits v. Hence, the claim holds. □

Theorem 5.3. *Let $G = (V, E, \mathcal{L})$ be a graph and $k, b \in \mathbb{N}$ be non-negative integers. Suppose we have constructed an index by* LandmarkIndex$^+$*(G, k, b). Then, $q = (s, t, L)$ is a true-query if and only if* Query$^+$*(s, t, L) with the constructed index returns* ***true****.*

Proof. When s is a landmark, we answer correctly at Line 2.

From the second claim in Lemma 5.1, we reach Line 6 only when there is an L-path from s to v. When the call QueryExtensive at Line 6 returns **true**, then q is a true query because $s \overset{L'}{\rightsquigarrow} v \overset{L}{\rightsquigarrow} t$ holds. When it returns **false**, from Lemma 5.2 and the argument in Section 5.2, we only mark vertices from which there is no L-path to t.

From Line 9, we basically conduct a BFS only with edges with labels in L. The only difference is that, when we hit a landmark $v \in V_L$ with $(t, L') \in \mathsf{Ind}(v)$ for some $L' \subseteq L$, then we immediately return **true**. This is valid because L' is a minimal label set connecting v to t from Lemma 4.3. □

5.6 Space and time complexity

We analyze the space and time complexities of LI$^+$. We only mention differences from LI.

Each non-landmark vertex may store $O(b)$ entries and the the number of non-landmark vertices is $n - k$. Hence the total index size is $O((n(k2^{|\mathcal{L}|} + b)(\log n + |\mathcal{L}|))$ bits.

Next, we analyze the time complexity of index construction. For each non-landmark, each call to TryInsert requires only $O(b)$ time. Hence, the time complexity for index construction is $O(n(\log n + 2^{|\mathcal{L}|}) + m)k2^{|\mathcal{L}|}) + O(n(\log n + b) + m)(n-k))2^{|\mathcal{L}|}) = O((n \log n + m)n2^{|\mathcal{L}|} + (2^{|\mathcal{L}|}k + b(n-k))n2^{|\mathcal{L}|}) = O((n \log n + m + 2^{|\mathcal{L}|}k + b(n-k))n2^{|\mathcal{L}|})$.

Finally, we analyze the query complexity. In the worst case, we call QueryExtensive for each of k landmarks. Each call to QueryExtensive compares L to at most $2^{|\mathcal{L}|}$ label sets and sets at most n vertices in marked. Hence, the query time complexity is $O(n + m + k \cdot (2^{|\mathcal{L}|} + n)) = O(m + k(2^{|\mathcal{L}|} + n))$.

5.7 Implementation details

Let $w = 64$ be the number of bits per word. For the variables marked and $\mathsf{canReach}$, we use a dynamic bitset of length $\lceil n/w \rceil$ in favor of a list or array. The advantage of using a bitset is the fact that setting a bit (adding a vertex to a set) or reading a bit (verifying the presence/absence of a vertex in a set) can be done in constant time. An array of length n would need $O(\log n)$ time to find a specific element. Yet another advantage is the fact that we can efficiently join two bitsets in $\frac{n}{w}$ operations on a w-bit machine. Joining two bitsets is done by Procedure QueryExtensive on line **29**.

6. EXPERIMENTAL RESULTS

In this section, we evaluate our proposed methods over a variety of both synthetic and real datasets. We divided the experiments in four parts. In the first two parts we use all methods (LI, LI$^+$, Zou *et al.* [35] and BFS). In the first part we use all the real datasets (see Table 1) to compare all our methods against each other and to study how well our methods perform on real data. In the second part we fix the number of vertices ($n = 5,000$) and vary the degree D for two types of synthetic graphs (ER and PA) in order to compare all our methods against each other and to study the effect of increasing the degree. In the last two parts we study in depth our extended method, i.e. LI$^+$. In the third part we fix the number of vertices ($n = 25,000$) and we vary the degree per node (D) and the label set size ($|\mathcal{L}|$) for two types of synthetic graphs (ER and PA) in order to study the effect of these parameters. In the fourth and final part we compare two types of synthetic graphs (ER and PA) in order to study the effect of increasing the graph size.

For the synthetic datasets we only report the index construction time (s), index size (MB) and average speed-ups over the baseline BFS. We do this to study the effect of certain parameters (e.g., degree, label set size). For the real datasets we report all the total speed-ups as well.

All methods have been implemented using C++. Details on Zou *et al.* (Zou) can be found in Appendix A. Full-LI is the same as LI with the number of landmarks k equal to the number of vertices n. Full-LI uses the naive landmark approach; both extensions of LI$^+$ are irrelevant for Full-LI, as we have that $k = n$ which implies that we can use QueryLandmark(v) for all $v \in V$.

On a given graph, Zou and Full-LI build the same index, that is, there is no difference between the final constructed indexes. The same data structure $\{\mathsf{Ind}(v)\}_{v \in V}$ was used by both, and the same procedure TryInsert was used to insert a label set L into $\mathsf{Ind}(v, w)$ for $v, w \in V$. Both can answer any query (v, w, L) correctly and use only a minimum number of entries in their index. Hence they have the same index size and the same speed-ups. The difference between the two is in the index construction time.

6.1 Settings

We used a Linux server with 258GB of memory and a 2.9GHz 32-core processor. We did not let any of our experiments exceed a 128GB memory limit or a 6 hour (21,600 s) time limit. The experiments were all single-threaded.

In all experiments we set the number of landmarks k to $1250 + \sqrt{n}$ and the budget b per non-landmark vertex to 20, where n is the number of vertices in the input network. Preliminary experiments have demonstrated that these values for k and b bring a good balance between space and time.

Table 1: Overview of all real datasets sorted on the number of edges. The last column indicates whether the edge labels are synthetic, i.e. the original graph did not have edge labels.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathcal{L}\rvert$	Synthetic labels
robots[4]	1.4k	2.9k	4	
advogato [20]	5.4k	51k	4	
epinions [20]	131k	840k	8	✓
StringsHS[5]	16k	1.2M	7	
NotreDame [20]	325k	1.4M	8	✓
BioGrid[6]	64k	1.5M	7	
StringsFC[7]	15k	2.0M	7	
webStanford [21]	281k	2.3M	8	✓
webGoogle [21]	875k	5.1M	8	✓
webBerkstan [21]	685k	7.6M	8	✓
Youtube[8]	15k	10.7M	5	
zhishihudong [20]	2.4M	18.8M	8	✓
socPokec [21]	1.6M	30.6M	8	✓

6.2 Datasets

For the experiments, we used both synthetic graphs and real datasets. We generated the synthetic graphs using SNAP [21, 22]. The synthetic graphs can be distinguished according to a number of parameters. First of all, we have the number of vertices $n \in \{5{,}000,\ 25{,}000,\ 125{,}000,\ 250{,}000\}$, Next we have the label set size $|\mathcal{L}| \in \{8,\ 10,\ 12,\ 14,\ 16\}$. Then comes the model which determines the graph structure, which is either 'preferential attachment' (PA) or 'Erdős-Rényi' (ER). A synthetic dataset generated using the PA- or ER-model is also referred to as a PA-dataset or an ER-dataset respectively. The main difference between both models is that PA has a skew in its out-degree distribution whereas ER has a close to uniform out-degree distribution. Next, we have the parameter $\alpha = 1.7$ that determines the shape of the label set distribution. The labels incident on the edges are exponentially distributed with $\lambda = \frac{|\mathcal{L}|}{\alpha}$. Finally, we can have a parameter $D \in \{2, 3, 4, 5\}$, which is the average degree per node for both ER-datasets and PA-datasets. For the PA- and ER-datasets, we did not get directed graphs initially. We set the direction of the edges randomly.

Table 1 provides an overview of the real datasets used in the experiments. These datasets have been taken from various sources, but most of these have been taken from either SNAP [21] or KONECT [20]. Six of the graphs, e.g., **advogato**, already have natural edge labels. The last column indicates whether or not we synthetically generated labels. In case we appended the labels synthetically we followed the same approach as with the synthetic data sets, setting the number of labels $|\mathcal{L}|$ to 8 and the parameter α to 1.7.

BioGrid, **StringsFC**, and **StringsHS** are undirected graphs, where the vertices represent proteins and the edge labels represent an interaction between two proteins. For both Strings-datasets we used the 'is-acting' field to determine the direction of any edge. **StringsFC** and **String-**

[4]http://tinyurl.com/gnexfoy
[5]http://string-db.org
[6]http://thebiogrid.org
[7]http://string-db.org
[8]http://socialcomputing.asu.edu/datasets/Youtube

Table 2: Indexing time (IT) in seconds, and index size (IS) in megabytes for real datasets. In this table, "-" indicates that the method timed out on this dataset.

Dataset	LI		LI$^+$		Full-LI		Zou	
	IT	IS	IT	IS	IT	IS	IT	IS
robots	0.1	5	0.1	5	0.1	5	9	5
advogato	4	135	3	131	7	369	11867	369
epinions	272	2903	205	2091	-	-	-	-
StringsHS	15	556	13	542	93	5,993	-	-
NotreDame	242	2424	193	1895	-	-	-	-
BioGrid	58	1410	50	1302	36	3207	-	-
StringsFC	21	513	19	503	91	5055	-	-
webStanford	935	7806	906	7719	-	-	-	-
webGoogle	5887	33931	5665	33497	-	-	-	-
webBerkstan	2463	15690	2364	14874	-	-	-	-
Youtube	3121	336	2841	300	-	-	-	-
zhishihudong	4068	9488	3574	8351	-	-	-	-
socPokec	9762	77155	9461	75698	-	-	-	-

sHS represent the protein networks of the organism 'felis catus' and 'homo sapiens' respectively. In **BioGrid**, an undirected edge $(u, v, l) \in E$ was replaced by two directed edges $(u, v, l), (v, u, l) \in E$ to create a directed graph.

6.3 Query generation

For each dataset, we generated 3 query sets. The queries in the first, second, and third query set use $|\mathcal{L}|/4$, $|\mathcal{L}|/2$, and $|\mathcal{L}| - 2$ labels, resp. Each query set consists of $2 \times 1,000$ queries. The first half are true-queries and the second half are false-queries.

We generated the queries for each query set in the following way. Let $t = 1,000$ be the number of queries we need to generate for any query condition. First, we chose a random vertex $v \in V$. We also generate a random number d between $50 + \log n$ and $50 + n/50$. Then we enter a loop. We leave this loop after $\frac{t}{100}$ iterations. We pick another random vertex $w \in V$. We generate 10 label sets $L_1, \ldots, L_{10}$ and run BFS for each L_i to find out whether $v \overset{L_i}{\rightsquigarrow} w$. Our implementation of BFS did not only return the answer of the query, but also the number $d' \in \mathbb{N}$ of vertices visited by BFS. In case we have that $d' < d$, we discard the query. If $v \overset{L_i}{\rightsquigarrow} w$ and the number of true-queries is below t, we add (v, w, L_i) to the true-query set. If $v \overset{L_i}{\not\rightsquigarrow} w$ and the number of false-queries is below t, we add (v, w, L_i) to the false-query set. For both true- and false-queries we do not allow duplicate queries, i.e. each query appears only once in a query set. Finally, after this loop ends we choose a new $v \in V$ and d until we have generated t true- and false-queries.

There were two reasons for setting up the procedure in this fashion. First of all processing a set of queries should require some effort. For instance, if for all queries we have that BFS can find the answer in a few steps, then we have that there is little to no advantage of building an index. Secondly, we did not want a large part of the queries to have the same start vertex $v \in V$. If this were to be, then our landmark methods would have an advantage over BFS as for each $v \in V_L$ we can use QueryLandmark to get a relatively quick answer to the query.

6.4 Speed-up calculation

The goal of an index is to speed up the query answering process relative to BFS. Hence, we define a total speed-up over BFS. First, recall that a query set is determined by the number of labels and whether it consists of true- or false-queries. We say that $Q_c \in \mathbb{N} \times \{\textbf{true}, \textbf{false}\}$ is a *query condition*. For example, the true-queries with $|\mathcal{L}|/4$ labels belong to the query condition $(|\mathcal{L}|/4, \textbf{true})$. As we generated 3 query sets for each dataset and each query set consists of 2 query conditions, we have 6 query conditions in total. Recall that, for each query condition, the number of queries in the corresponding query set (if exists) is 1,000. Let $Q_c(j)$ for $1 \leq j \leq 1,000$ be the j'th query belonging to Q_c. Let $T_M(Q_c(j))$ be the time taken by method M to answer query $Q_c(j)$. Then, the *total speed-up* for a method M under a query condition condition Q_c is defined as:

$$\frac{\sum_{i=1}^{1,000} T_{\text{BFS}}(Q_c(i))}{\sum_{i=1}^{1,000} T_M(Q_c(i))}$$

The *individual speed-up* for the i'th query and a method M under a query condition condition Q_c is defined as:

$$\frac{T_{\text{BFS}}(Q_c(i))}{T_M(Q_c(i))}$$

And, the *average speed-up* for a method M is the average of the total speed-ups belonging to the 6 query conditions.

6.5 Results I: performance on real graphs

In our first experiment, we compare the performance of our landmark-based methods LI, LI$^+$, and Full-LI with the state of the art Zou on the real datasets of Table 1.

Indexing times and sizes are summarized in Table 2. The results clearly show that both Full-LI and Zou do not scale well on large graphs. In particular, Zou timed-out on all but two of the datasets, while Full-LI was able to process only five of the graphs. As Zou and Full-LI build the same index, they have the same index size; however, the indexing time of Full-LI is orders of magnitude faster than that of Zou. Also, we can observe the indexing time as well as the index size of LI$^+$ is only slightly higher than those of LI.

Table 3 summarizes the total speed-ups of LI and LI$^+$ over BFS. First, both methods achieve significant improvement over BFS. Except for extremely small graphs, LI$^+$ almost always shows better performance than LI. Although LI does not show a large advantage over BFS for false-queries, LI$^+$ is orders of magnitude faster than BFS for false-queries.

In general, false-queries take much more time than the

Table 3: Speed-ups of LI$^+$ and LI for each of the real datasets. For BFS the average query time is given in the last column.

Dataset	$\lvert\mathcal{L}\rvert/4$				$\lvert\mathcal{L}\rvert/2$				$\lvert\mathcal{L}\rvert - 2$				BFS
	true		**false**		**true**		**false**		**true**		**false**		(ms)
	LI	LI$^+$	LI	LI$^+$	LI	LI$^+$	LI	LI$^+$	LI	LI$^+$	LI	LI$^+$	
robots	67	67	66	66	140	148	172	157	149	149	231	231	0.1
advogato	1011	1117	1506	853	1542	885	1791	942	1624	1120	1725	974	0.3
epinions	3390	2553	3.0	3017	4370	5659	2.2	4144	4368	5673	2.2	4462	9.2
StringsHS	4838	1945	2.8	853	3091	1455	2.7	638	6586	19814	2.3	871	7.0
NotreDame	49	16	9.3	89	59	16	4.6	170	74	65	3.7	251	0.8
BioGrid	107	131	1.6	334	111	147	1.5	271	649	641	1.4	208	8.0
StringsFC	1112	1684	4.0	316	2974	10582	4.2	198	1552	1817	4.4	533	7.0
webStanford	2443	2688	5.4	836	1929	481	5.5	19	2386	3377	5.5	17	10.3
webGoogle	7780	4244	1.1	5263	3660	2644	1.7	19	4994	4425	1.4	19	52.0
webBerkstan	735	914	3.3	383	163	185	2.8	45	381	946	3.0	43	23.7
Youtube	161	211	1.7	39	51	168	1.5	63	761	430	1.4	45	16.2
zhishihudong	760	920	0.9	875	991	1170	1.0	83	885	1025	0.9	16	24.5
socPokec	2080	2442	0.9	2833	2582	2808	0.9	3478	2721	3290	0.9	2900	463.9
Average	1887	1456	124	1212	1666	2027	153	787	2087	3290	153	813	48

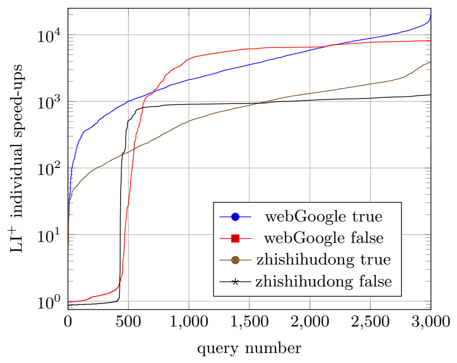


Figure 2: Individual speed-ups for LI$^+$ on **webGoogle** and **zhishihudong**. The individual speed-ups are sorted in ascending order.

true-queries. This is reflected in both the average query time and the total speed-up. A true-query can stop after hitting its target, whereas a false-query has to explore larger parts of the graph to get to an answer: When the target $t \in V$ of a query (s, t, L) has been found in an index $\mathsf{Ind}(v)$ of a landmark $v \in V_L$, we can immediately return **true**.

The number of labels in a query could affect the speed-ups of LI$^+$ over BFS. A true-query with a high number of labels should have less difficulty finding a landmark and resolving the query from that landmark. A false-query with a relatively high number of labels should be able to visit most of the graph. Using LI$^+$ we can use pruning for these kinds of false-queries to speed up the query evaluation. However it may take a long time before the query algorithm of LI$^+$ (Algorithm 5) is not able to push a new vertex on the queue. The numbers in Table 3 generally meet our expectations. The average speed-ups of both methods (the last row of the table), for both true and false queries, show that multiple orders of magnitude improvement are achieved for query processing.

Towards a deeper understanding of the distribution of query speed-ups, Figure 2 shows the individual speed-ups of **webGoogle** and **zhishihudong** for their 3,000 true- and 3,000 false-queries and method LI$^+$. The individual speed-ups are sorted in ascending order. Only a small minority of the false-queries have an individual speed-up lower than 1.0 in both occasions. We have that for both datasets at least 2,500 false-queries and nearly all true-queries have an individual speed-up of at least 10.

6.6 Results II: performance on synthetic graphs

In our second experiment, we compare the performance of our methods (LI, FULL-LI, and LI$^+$) against the state of the art ZOU on synthetic datasets. We chose $n = 5,000$ and $L = 8$, and we vary the node degree from 2 up to 5. Table 4 summarizes the results.

Again, as ZOU and FULL-LI build the same index, we have that ZOU and FULL-LI have the same index size and the same average speed-up. Hence the average speed-up of FULL-LI is that of ZOU as well (in the cases where the ZOU method managed to build an index). ZOU did not build an index within the time limit when $D \geq 4$ for the PA-datasets. It did not build an index in any case for the ER-datasets.

From the results, we can observe that ZOU has some major difficulties with high-degree graphs. It is not competitive to FULL-LI. In Appendix A we give a more detailed explanation of ZOU and the results here.

In all cases, our proposed methods exhibited significant query processing time improvements.

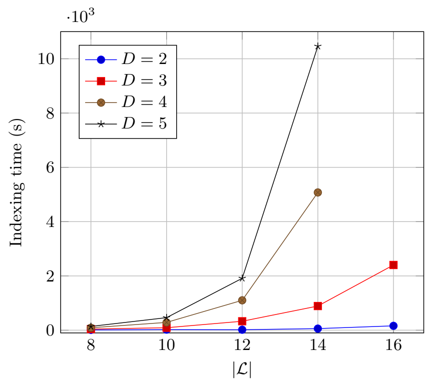
6.7 Results III: impact of degree and label set size

In our third experiment, we analyze the performance of LI$^+$ while varying the number of edges and labels using synthetic graphs. We used PA- and ER-datasets with $n = 25,000$ varying the node degree D (either 2, 3, 4, or 5) or the number of labels $|\mathcal{L}|$ (either 8, 10, 12, 14, or 16).

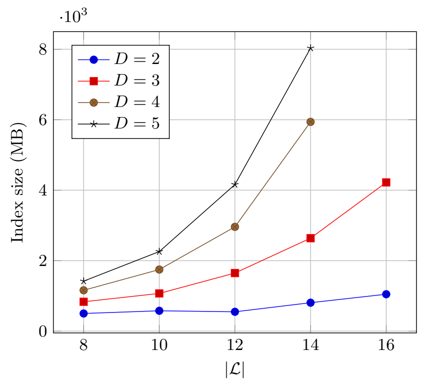
Figures 3 and 4 show the indexing time, the index size and the average speed-ups for the PA-datasets and the ER-datasets, respectively. In case a data point is missing in any of the figures, we were not able to build an index within the 21,600-second (6 hour) time limit. When D is large, we can observe that, both the indexing time and the index size rapidly grows as $|\mathcal{L}|$ increases. On the other hand, when D is small, the growth of the indexing time and the index size is slow. This can be understood by the number of minimal label sets connecting a (typical) pair of vertices, which heavily affects the performance of LI$^+$. The number is exponential

Table 4: Indexing time (IT) in seconds, index size (IS) in megabytes, and average speed-ups for PA- and ER-datasets with $n = 5,000$ and $L = 8$ for which we vary the node degree (from 2 up to 5). In this table, "-" indicates that the method timed out on this dataset.

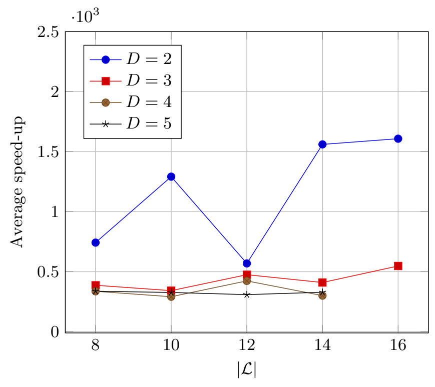
	D	LI		LI$^+$		Full-LI		Zou		Average speed-up		
		IT	IS	IT	IS	IT	IS	IT	IS	LI	LI$^+$	Full-LI
PA	2	1.7	87.5	1.2	86.0	3.6	270.1	3528.3	270.0	679.8	397.0	998.9
	3	5.3	143.1	4.7	141.1	10.8	471.8	12676.0	471.8	1115.2	605.3	1856.6
	4	9.7	196.0	8.6	194.1	18.7	333.0	-	-	532.7	576.6	2254.1
	5	16.7	247.0	16.2	244.4	33.5	786.3	-	-	404.4	710.4	2422.4
ER	2	4.1	115.6	3.5	115.1	8.5	357.9	-	-	641.7	286.6	1068.8
	3	3.1	129.3	2.4	128.5	7.6	462.7	-	-	900.8	798.5	1849.5
	4	11.4	182.7	10.7	181.7	23.4	637.4	-	-	1093.4	793.9	2298.4
	5	33.8	267.4	37.6	266.3	64.3	909.0	-	-	1067.4	685.6	2287.6



(a) Indexing time (s)

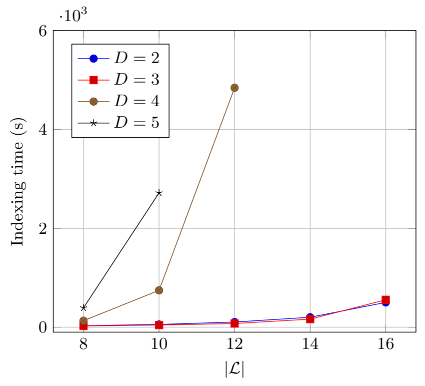


(b) Index size (MB)

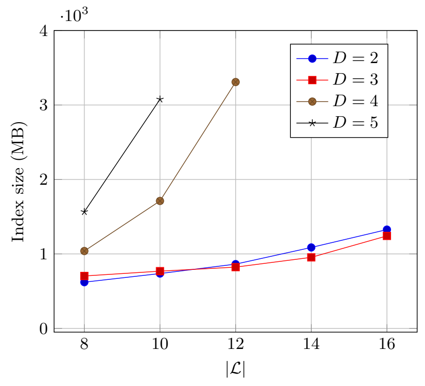


(c) Average speed-ups

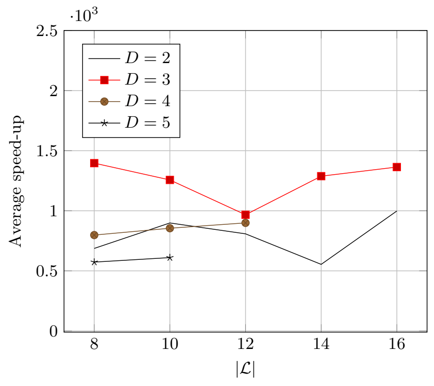
Figure 3: Indexing time, index size and average speed-up for PA-datasets with n =25,000, as a function of the label set size $|\mathcal{L}|$. The different lines indicate the node degree (either 2, 3, 4, or 5) of the datasets.



(a) Indexing time (s)



(b) Index size (MB)



(c) Average speed-ups

Figure 4: Indexing time, index size and average speed-up for ER-datasets with n =25,000, as a function of the label set size $|\mathcal{L}|$. The different lines indicate the node degree (either 2, 3, 4, or 5) of the datasets.

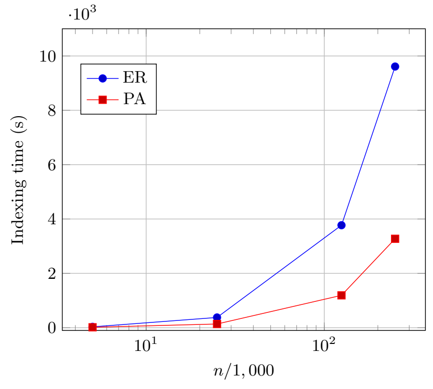
in $|\mathcal{L}|$ when D is large and remains small no matter what $|\mathcal{L}|$ is when D is small.

The growths of the indexing time and index size are stronger for ER-datasets than for PA-datasets. This can be explained by the fact that ER-datasets have a close to uniform out-degree distribution. On average there are more paths connecting any two vertices, which increases the number of minimal label sets connecting them.

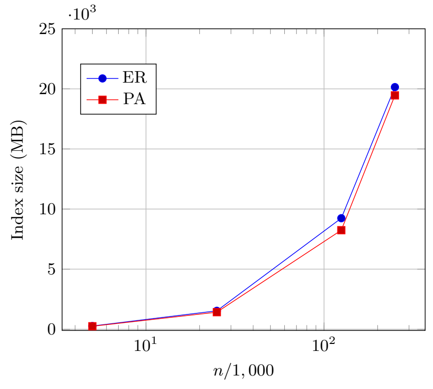
Concerning the speed-ups there is not a clear difference between ER- and PA-datasets. The dip in average speed-up for $D = 2$ and $|\mathcal{L}| = 12$ in Figure 3 (c) is a random outlier. Overall, we observe significant speed-ups across all indexed datasets.

6.8 Results IV: impact of structure of the graph

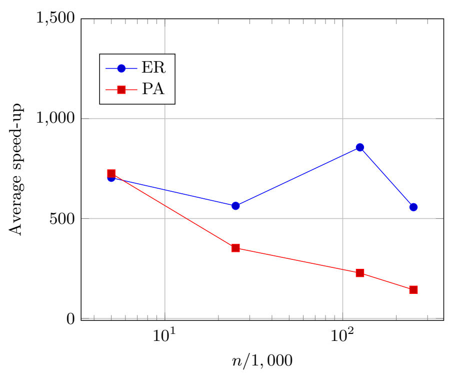
Finally, we analyze the performance of LI$^+$ on PA- and



(a) Indexing time (s)



(b) Index size (MB)



(c) Average speed-ups

Figure 5: Indexing time, index size and average speed-ups for PA- and ER-datasets as a function of the number of vertices.

ER-datasets as we vary the number of nodes $n \in \{5,000, 25,000, 125,000, 250,000\}$. For all datasets we set the degree $D = 5$.

Figure 5 shows the indexing time, the index size and the average speed-ups. We can observe that the indexing time of ER-datasets grows much faster than that of PA-datasets whereas the index size of ER-datasets are comparable to that of PA-datasets. This can be understood as follows. In a graph with a more uniform out-degree distribution, the average number of (direct) paths between any two vertices $s, t \in V$ is higher than in a graph with more skewed out-degree distribution. If the number of paths increases between two vertices $s, t \in V$, so does the number of label sets connecting s and t. Due to this effect both the index size and construction time exhibit a stronger growth for ER- than for PA-datasets. The growth for the indexing time is stronger because Algorithm 4 might need to explore all simple paths, whereas it might need to store only some of the label sets belonging to this path due to the fact that given two vertices $v, w \in V$, $L, L' \subseteq \mathcal{L}$, $v \overset{L}{\rightsquigarrow} w$ and $v \overset{L'}{\rightsquigarrow} w$ we only need to include (w, L) to the index of v $\mathsf{Ind}(v)$. We suspect that the differences between the index size of ER- and PA-datasets are very small because of the same reason.

As the number of vertices increases, the average speed-ups decrease for both ER- and PA-datasets. This is due to the fact that for larger graphs the ratio $\frac{k}{n}$ decreases, where k is the number of landmarks. Still the numbers are substantial even for $n = 250,000$. The PA-datasets exhibit a stronger decline than the ER-datasets. This can be understood as follows. As the ER-datasets have a more uniform out-degree distribution and each vertex has more neighbors, we might have BFS has to explore larger parts of the graph and hence take more time to evaluate a query. This higher query evaluation time leads to a better speed-up.

7. CONCLUDING REMARKS

In this paper we have studied the problem of label constrained reachability in edge-labeled graphs. We have presented and analyzed in detail two novel landmark-based methods for efficient scalable evaluation of such queries. We demonstrated through an in-depth experimental study on a variety of both synthetic and realworld graphs that our proposed methods scale to graphs which are orders of magnitude larger than those supported by previous solutions. Furthermore, our query processing approach leads to orders of magnitude speed-ups in query evaluation times.

There are several interesting directions for further study, building on our results. One natural direction for future research is a finer analysis of the impact of graph topology and complexity on the performance and further optimization of our solutions, e.g., graphs of bounded treewidth. Another rich avenue for investigation is the study of landmark-based evaluation methods for extensions to the class of LCR queries; in Appendix B we sketch one such application of our methods. An additional interesting direction is the study of landmark indexing in multi-core environments. Finally, applications of our indexes for optimization and execution of practical query languages such as SPARQL1.1 and openCypher is an important direction for future research.

8. REFERENCES

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360, 2013.

[2] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*, pages 237–248, 2014.

[3] A. Anand, S. Seufert, S. Bedathur, and G. Weikum. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*, pages 1009–1020, 2013.

[4] P. Baeza. Querying graph databases. In *PODS*, pages 175–188, 2013.

[5] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[6] F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen. Distance oracles in edge-labeled graphs. In *EDBT*, pages 547–548, 2014.

[7] L. Chen, A. Gupta, and M. Kurul. Stack-based algorithms for pattern matching on DAGs. In *VLDB*, pages 493–504, 2005.

[8] M. Chen, Y. Gu, Y. Bao, and G. Yu. Label and distance-constraint reachability queries in uncertain

graphs. In *DASFAA*, pages 188–202, 2014.

[9] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. TF-label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*, pages 193–204, 2013.

[10] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[11] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[12] R. Geisberger, M. N. Rice, P. Sanders, and V. J. Tsotras. Route planning with flexible edge restrictions. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.

[13] A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling kleene: fast property paths in RDF-3X. In *GRADES*, 2013.

[14] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD*, pages 123–134, Indianapolis, 2010.

[15] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD*, pages 123–134, 2010.

[16] R. Jin, N. Ruan, S. Dey, and J. Y. Xu. SCARAB: scaling reachability computation on large graphs. In *SIGMOD*, pages 169–180, 2012.

[17] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *PVLDB*, 6(14):1978–1989, 2013.

[18] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.

[19] P. Klodt. Indexing strategies for constrained shortest paths over large social networks. BSc thesis, Universität des Saarlandes, 2011.

[20] J. Kunegis. KONECT - the koblenz network collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350, Koblenz, 2013.

[21] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

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

[23] A. Likhyani and S. Bedathur. Label constrained shortest path estimation. In *CIKM*, pages 1177–1180, 2013.

[24] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An efficient connection index for complex XML document collections. In *EDBT*, pages 237–255, 2004.

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

[26] E. Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift*, 27(1):544–548, 1928.

[27] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *GRADES*, 2016.

[28] S. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, pages 913–924, 2011.

[29] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: an independent permutation labeling approach. *PVLDB*, 7(12):1191–1202, 2014.

[30] P. T. Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60, 2012.

[31] 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*, pages 1601–1606, 2013.

[32] H. Yildirim, V. Chaoji, and M. Zaki. GRAIL: Scalable reachability index for large graphs. *PVLDB*, 3(1-2):276–284, 2010.

[33] J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*. Springer, 2010.

[34] Z. Zhang, J. Yu, L. Qin, Q. Zhu, and X. Zhou. I/O cost minimization: reachability queries processing over massive graphs. In *EDBT*, pages 468–479, 2012.

[35] L. Zou, K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems*, 40:47–66, 2014.

APPENDIX

A. ZOU ET AL. PERFORMANCE

We provide here a more detailed analysis of Zou, the method of Zou *et al.* [35] introduced in Section 2. We can break this method down into nine steps. Steps 1 and 2 are about finding the SCC's in the graph $\langle C_1, \ldots, C_k \rangle$ and building a local index for each SCC C_i. A local index can answer any query (u, v, L) correctly if $u, v \in C_i$. Steps 3 to 7 create a DAG out of the SCC's by looking at the incoming and outgoing vertices of each C_i and the label sets connecting any of these. At the end of step 7 we have that any query (u, v, L) is answered correctly if u and v are incoming or outgoing ports of any SCC. Steps 8 and 9 are about finding the label sets connecting the inner vertices to the incoming and outgoing ports and vice versa.

In the rightmost column of Table 5 we can see that for both the PA- (upper section) and ER-datasets (bottom section) the number of SCC's decreases as we increase the average node degree. As we see in this Table, the time necessary to complete each step increases exponentially as well. The first two steps may require from a few minutes ($D = 2$) up to an hour ($D = 5$). This is already much slower than building the same index using Full-LI (see Table 4). A further optimization discussed in [35] only affects this first part. However this does not affect the most costly part of the method, i.e. steps 3 to 9. Indeed, the last steps involve many operations where, given $v, w \in V$, $L \subseteq \mathcal{L}$ and $v \overset{L}{\rightsquigarrow} w$, we try to insert $L' \cup L$ into $\mathsf{Ind}(v, w)$, where $L' \in \mathsf{Ind}(w)$. This can become extremely costly, especially if there are many different L that connect v and w. In general, Zou is at least a factor of 90 times slower than Full-LI in all cases where we could obtain results without time-out.

Table 5: The time cost (s) for each of the 9 steps in the Zou *et al.* indexing algorithm.

	D	Step									# SCC
		1	2	3	4	5	6	7	8	9	
PA	2	0	284	284	284.02	284.02	289.37	2,184	3,507	3,528	1,821
	3	0	261.46	261.46	261.47	261.47	264.3	6,970	12,610	12,675	572
	4	0	1,022.77	1,022.77	1,022.77	1,022.77	1,025.72	-	-	-	203
	5	0	3,756.99	3,756.99	3,756.99	3,756.99	-	-	-	-	81
ER	2	0	48.56	48.56	48.57	48.57	51.26	-	-	-	2,182
	3	0	370.36	370.36	370.37	370.38	379.03	-	-	-	859
	4	0	1,431.45	1,431.45	1,431.46	1,431.46	1,440.33	-	-	-	349
	5	0	2,349.39	2,349.39	2,349.4	2,349.4	2,357.6	-	-	-	151

Table 6: Average speed-ups in time to run 100 for-all-queries with either $|L| = \mathcal{L}/2$ or $|L| = \mathcal{L} - 2$.

	D	avg $\lvert\mathcal{L}\rvert/2$	avg $\lvert\mathcal{L}\rvert - 2$
ER	2	3.1	4.3
	3	8.0	9.5
	4	7.1	6.1
	5	5.2	5.0
PA	2	5.3	7.0
	3	5.3	5.7
	4	4.8	5.4
	5	5.8	6.5

B. FOR-ALL QUERIES

A natural extension to LCR is to retrieve *all* nodes reachable from a given start node.

> (For-all-LCR) Given a vertex s of graph G and a subset L of the set of all edge labels $\mathcal{L}$ of G, compute the set $\{t \mid s \overset{L}{\rightsquigarrow} t\}$.

Such queries are a natural generalization of LCR, with applications, e.g., in efficient evaluation of conjunctions of regular path queries, which are a core feature of practical graph query languages [4, 30].

A straightforward way to answer a For-all-LCR query would be to run an LCR-query (s, t, L) for each $t \in V$; another straightforward solution is a single BFS that stops either after it has hit all the vertices or it gets an empty queue.

A minor tweak to LI$^+$, however, leads to much faster evaluation than either of these two strategies. Given a For-all-LCRquery (s, L), we can run a similar query procedure as in Algorithm 5. However, now when we visit a landmark $v \in V_L$ we look for all entries $(w, L') \in \mathsf{Ind}(v)$ such that $L' \subseteq L$ and set $\mathsf{marked}(w)$ to **true**. In the end we return marked.

Table 6 shows the average speed-ups of running 100 random for-all-queries (v, L) with $|L| = |\mathcal{L}|/2$ or $|L| = |\mathcal{L}| - 2$. We did this for ER- and PA-datasets with $n = 25,000$ while varying the degree D. We used LI$^+$ with $k = 500$ and $b = 20$. We required each query to hit at least 10% of the vertices, i.e. $|\{w | w \in V \wedge v \overset{L}{\rightsquigarrow} w\}| \geq \frac{n}{10}$. In case a query does not hit this many vertices we think it is better to always use BFS.

The results indicate the value of even this lightweight extension to our methods.