Landmark Indexing for Evaluation of Label-Constrained Reachability Queries

Lucien D.J. Valstar
Eindhoven University of Technology
The Netherlands
l.d.j.valstar@student.tue.nl

George H.L. Fletcher
Eindhoven University of Technology
The Netherlands
g.h.l.fletcher@tue.nl

Yuichi Yoshida
National Institute of Informatics, Japan *and*
Preferred Infrastructure, Inc.
yyoshida@nii.ac.jp

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 up to orders of magnitude faster query evaluation times. Our complete C++ codebase is available as open source for further research.

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. 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, citation relationships holding between protein vertices and vertices denoting scientific publications).

LCR queries also appear as an important fragment of the language of *regular path queries* [4, 5, 7, 31], 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 [28], and openCypher[2]. In the study of efficient processing of regular path queries, scalable solutions for reachability play an important role [16].

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 solutions for LCR queries was initiated in the work of Jin *et al.* [17] with several recent follow-up studies [9, 10, 20, 24, 37]. 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.

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA

 ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3035955

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

Our contributions

In our work, we remedy this situation, presenting the first LCR index solutions which scale to large graphs. Our approach leverages the conceptually simple idea of landmark-based indexing, which has been shown to be successful for standard (unconstrained) reachability query processing [1, 2, 19, 33]. Briefly, 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 results in up to orders of magnitude faster query evaluation than current solutions. Additionally, our solutions are not overly complex, and hence have good potential for practical impact. Our complete C++ codebase, including testing framework and implementations of all indexing strategies used in our experimental study, is available as open source for further study.[3]

We proceed 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 [35]). The goal is to build an index that can answer queries faster than BFS and its modern variants such as direction optimizing BFS (DBFS) [8], which take $O(n+m)$ time, and have lower memory requirements than the full transitive closure (TC) 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 TC [26, 29], online search guided by precomputed indices [3, 18, 30, 34, 36], and labeling schemes [11, 12, 13, 19, 25, 33].

With regard to strategies for efficient evaluation of the regular path queries introduced in Section 1, to the best of our knowledge current state-of-the-art systems (e.g., SPARQL engines) rely on variations of BFS which are either index-free (e.g., [32]) or use indexes which cannot be effectively applied towards LCR query evaluation, as they target different query types (e.g., [14, 16]). Furthermore, regular path query evaluation on current graph and RDF database systems is impractical on graphs with more than a few thousand edges [6]. Hence, existing systems would not be able to process queries on the graphs which we use in our study, which are multiple orders of magnitude larger. Indeed, we view our work here to be a step in the direction of practical implementation of graph query languages which feature LCR queries (such as the richer regular path queries).

[3] https://github.com/DeLaChance/LCR

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

Jin et al. [17]. 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 TC on the data graph. A *tree-based index framework* is presented, 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 TC. 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 NT will not be that much smaller than that of the full TC.

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

Follow-up work has shown the limited scalability of the approach of Jin *et al.* [20, 37], and hence we do not consider it further in our study.

Bonchi et al. [9]. There is recent progress on evaluating LCR queries where reasoning about distance is important [10, 15, 20, 24], represented by the state of the art methods of Bonchi *et al.* Here 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 are inexact, in the sense that false negatives are possible. Hence, they are not helpful to answer LCR queries exactly. In our study we do not consider these strategies further.

Zou et al. [37]. This approach is, to our knowledge, the current state of the art on LCR query evaluation. Hence, together with DBFS, 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 (i.e., those vertices in the SCC with edges from and to vertices outside of the SCC, respectively), 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. The resulting index holds full reachability information for every vertex of the graph (i.e., the complete TC of the graph with respect to label sets),

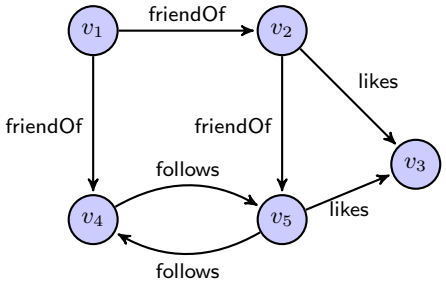


Figure 1: Example graph with $|V| = 5$ vertices, $|E| = 7$ edges, and edge labels $\mathcal{L} = \{\mathsf{likes}, \mathsf{follows}, \mathsf{friendOf}\}$.

and hence query answering amounts to a simple lookup in the index.

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 TC 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 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 TC. Instead, our approach focuses on building a selective part of the TC. 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 $\langle v_0, e_1, v_1, \ldots, v_{p-1}, e_p, v_p \rangle$, for some $p > 0$, where $v_i \in V$ for every $0 \leq i \leq p$ and $e_i \in E$ is an edge from v_{i-1} to v_i for every $i \in [p]$. We say that P is a *path from* v_0 *to* v_p of *length* p. The length of P is also denoted by $|P|$. Furthermore, for $L \subseteq \mathcal{L}$, we say that P is an L-*path* if $\lambda(e_i) \in L$ for every $i \in [p]$. We denote the existence of such a path by $v_0 \overset{L}{\rightsquigarrow} v_p$. We say that a label set $L \subseteq \mathcal{L}$ is a *minimal label set connecting* v *to* w if (i) $v \overset{L}{\rightsquigarrow} w$ and (ii) $v \overset{L'}{\not\rightsquigarrow} w$ for any proper $L' \subsetneq L$.

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

Definition 3.1. *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* $s \overset{L}{\rightsquigarrow} t$, *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 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 the index for 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 $v \overset{L'}{\rightsquigarrow} w$ 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), we only construct indices 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 an index 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* $\{\mathsf{Ind}(v)\}_{v \in V}$, *where* $\mathsf{Ind}(v) \subseteq V \times 2^{\mathcal{L}}$. *We call each* $\mathsf{Ind}(v)$ *the* index *for* v. *For* $v, w \in V$, *we define* $\mathsf{Ind}(v, w) \subseteq 2^{\mathcal{L}}$ *as the set* $\{L \mid (w, L) \in \mathsf{Ind}(v)\}$.

We say that $\mathsf{Ind}(v)$ *is* complete *if, for any* L*-path from* v *to* w, *we have* $(w, L') \in \mathsf{Ind}(v)$ *for some* $L' \subseteq L$. *We say that* $\mathsf{Ind}(v)$ *is* sound *if, for any* $(w, L) \in \mathsf{Ind}(v)$, *there is an* L*-path from* v *to* w.

The goal of our indexing algorithm is, after choosing a small set of landmark, constructing an index such that the indices for landmarks are complete and sound and the indices for non-landmarks are empty. With such an index, we can immediately answer a query (s, t, L) for a landmark s by checking whether we have $(t, L') \in \mathsf{Ind}(s)$ for some $L' \subseteq L$. Although indices for non-landmarks are irrelevant at this point, we will construct sound but incomplete indices for them to make our method more efficient in Section 5.

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), following the strategy of

Algorithm 1

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

Algorithm 2

```
1: procedure LabeledBFSPerLM(s)
2:     q ← an empty priority queue.
3:     q.push(s, {}).
4:     while q is not empty do
5:         (v, L) ← 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) ∈ E do
12:            q.push(w, L ∪ {l}).
13:    indexed(s) ← true.

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

22: procedure ForwardProp(s, (v, L))
23:    for (w, L') ∈ Ind(v) do
24:        TryInsert(s, (w, L ∪ L')).
```

Algorithm 3

```
1: procedure Query(s, t, L)
2:     if s ∈ V_L then return QueryLandmark(s, t, L).
3:     for v ∈ V do marked(v) ← false
4:     q ← an empty queue.
5:     q.push(s).
6:     while q is not empty do
7:         v ← q.pop().
8:         marked(v) ← 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) ∈ E ∧ marked(w) = false do
16:            if l ∈ L then
17:                q.push(w).
18:    return false

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

Yano et al., which we also observed in our experiments to be a robust landmark selection strategy [33]. 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. In particular, $\mathsf{Ind}(s)$ will be complete.

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.

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 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, 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 vari-

able 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 $j \in \{0, 1, \ldots, |P|\}$*

- *$L_{\leq j} \stackrel{\text{def}}{=} \{\lambda(e_{j'}) \mid 1 \leq j' \leq j\}$ is a minimal label set connecting s to v_j.*
- *$L_{>j} \stackrel{\text{def}}{=} \{\lambda(e_{j'}) \mid j < j' \leq |P|\}$ 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 to 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. In particular, $\mathsf{Ind}(v_i)$ is complete and sound.*

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 $L_{\leq j}$ is minimal, TRYINSERT$(v_1, (u_j, L_{\leq j}))$ 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 $L_{>j^*}$ is a minimal label set connecting v_{j^*} to t, we have $(t, L_{>j^*}) \in \mathsf{Ind}(v_{j^*})$. Hence, by FORWARDPROP$(s, (v_{j^*}, L))$ at Line 9, we add $(t, L_{>j^*} \cup L_{\leq j^*}) = (t, L)$ 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.[4] 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 LI 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, resp. 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 sound but incomplete index for the non-landmarks as well. Suppose that we have

[4]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 [27].

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 for 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. For clarity, we write L-Ind to denote indices for landmarks and NL-Ind to denote indices for non-landmarks.

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 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 $s \overset{L}{\rightsquigarrow} v$ and $v \overset{L}{\not\rightsquigarrow} t$ for some landmark $v \in V_L$. We define $R_L(v)$ as $\{w \in V \mid v \overset{L}{\rightsquigarrow} w\}$. Then, we observe $w \overset{L}{\not\rightsquigarrow} t$ for any $w \in R_L(v)$; otherwise, we have $v \overset{L}{\rightsquigarrow} t$, 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 landmark $v \in V_L$, we introduce a set reachableBy(v) and only keep subsets of $R_L(v)$ for $L \in \mathcal{L}$. Each entry of reachableBy(v) is a pair (S, L), meaning that $S \subseteq R_L(v)$. We will guarantee that each entry (S, L) satisfies $|L| \leq |\mathcal{L}|/4 + 1$. There are three reasons for this. First, we do not want to store all the $2^{|\mathcal{L}|}$ combinations, as this uses a lot of memory. Second, if $|L|$ is small, then $|S|$ is likely to be large, and hence we can mark more vertices. The third reason is that, if $|L|$ is small, then we have a more chance of using the entry for pruning because we can use it whenever the query (s, t, L') satisfies $L \subseteq L'$.

Procedure LabeledBFSPerLM$^+$ in Algorithm 4 shows the details on how we select the entries (S, L) for reachableBy. The entries (S, L) are sorted descending with respect to $|S|$, that is, the number of vertices in S.

5.3 Indexing algorithm

We next explain the LI$^+$ indexing algorithm (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 reachableBy(s). When arriving at a vertex v, we either create a new entry $(L, \{v\})$ or add v to an existing entry in reachableBy(s) (Lines 18 to 21). Then, on Lines 27 to 29, we expand each entry in reachableBy(s) by copying all the vertices of an existing entry $(S, L) \in$ reachableBy(s) to another existing entry $(S', L') \in$ reachableBy(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 NL-Ind(s) reaches b (see Lines 35 and 48 in ForwardPropNonLM).
- We add $(v, L) \in V \times 2^{\mathcal{L}}$ to NL-Ind(s) only when v is a landmark (see Line 39).

Algorithm 4

```
1: procedure LandmarkIndex+(G, k, b)
2:     Let v_1, ..., v_n be the vertices sorted descending on their total degree.
3:     for v ∈ V do indexed(v) ← false.
4:     for i ∈ [k] do
5:         L-Ind(v_i) ← an empty list.
6:         reachableBy(v_i) ← an empty list.
7:         LabeledBFSPerLM+(v_i).
8:     for i ∈ {k+1, ..., n} do
9:         NL-Ind(v_i) ← an empty list.
10:        LabeledBFSPerNonLM(v_i, b).

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

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

46: procedure ForwardPropNonLM(s, (v, L))            ▷ s ∉ V_L
47:     for (w, L') ∈ Ind(v) and w ∈ V_L do ▷ v can be both a landmark and a non-landmark.
48:         if |NL-Ind(s)| < b then
49:             TryInsert(s, (w, L ∪ L'))
50:         else
51:             break
```

Algorithm 5

```
1: procedure QUERY+(s, t, L)
2:     if s ∈ V_L then return QUERYLANDMARK(s, t, L).
3:     for v ∈ V do marked(s) ← false.
4:     for (v, L') ∈ NL-Ind(s) do ▷ v is always a landmark
5:         if L' ⊆ L then
6:             if QUERYEXTENSIVE(v, t, L, marked) = true
   then
7:                 return true
8:         marked(v) ← true.
9:     Let q be a queue.
10:    q.push(s).
11:    while q is not empty do
12:        v ← q.pop().
13:        marked(v) ← true.
14:        if v = t then
15:            return true
16:        if v is a landmark then
17:            if QUERYEXTENSIVE(v, t, L, marked) = true
   then
18:                return true
19:            continue
20:        for (v, w, l) ∈ E and marked(w) = false do
21:            if l ∈ L then
22:                q.push(w).
23:    return false

24: procedure QUERYEXTENSIVE(s, t, L, marked) ▷ s ∈ V_L
25:    if QUERYLANDMARK(s, t, L) = true then
26:        return true
27:    for (S', L') ∈ reachableBy(s) do
28:        if L' ⊆ L then
29:            marked ← S' ∪ marked
30:            break
31:    return false
```

- We use a variable **marked** to ensure that we visit each vertex at most once (for the sake of efficiency).
- We do not continue after the call to FORWARDPROPNONLM (Line 42) because, when v is a non-landmark, NL-Ind(v) may not have all the information to other vertices because the size of NL-Ind(v) is bounded by b.

5.4 Query algorithm

Now we explain our query algorithm (Algorithm 5). We use the index NL-Ind and reachableBy to speed up the process of answering queries.

For a query $q = (s, t, L)$ with $s \in V_L$, QUERYEXTENSIVE$(s, t, L,$ **marked**$)$ returns **true** if q is a true-query. In case q is a false-query, it marks all vertices in S' for each $(S', L') \in$ reachableBy(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 reachableBy(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 reachableBy. 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 marked(v) = **false** for every $v \in V$. When 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 $(v, L') \in$ NL-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 marked(v) to be **true**. Note that QUERYEXTENSIVE(v, t, L) may mark more vertices using the information of reachableBy(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 let $k, b \in \mathbb{N}$ be non-negative integers. Suppose $\{$L-Ind(v_i)$\}_{i\in[k]}$ and $\{$NL-Ind(v_i)$\}_{i\in\{k+1,\dots,n\}}$ are the index constructed by calling* LANDMARKINDEX$^+(G, k, b)$ *with the ordering $v_1, \dots, v_n$. Then, the following hold:*

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

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 NL-Ind(v_i) for $i \in \{k+1, \dots, n\}$. □

LEMMA 5.2. *Let $G = (V, E, \mathcal{L})$ be a graph and $k, b \in \mathbb{N}$ be non-negative integers. Let $\{$reachableBy(v_i)$\}_{i\in[k]}$ be constructed by calling* LANDMARKINDEX$^+(G, k, b)$ *with the ordering $v_1, \dots, v_n$. Then, for every $i \in [k]$, $(S, L) \in$ reachableBy(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 $(t, L') \in$ NL-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 (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 reachableBy, 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.* [37], and DBFS). In the first part we use all of the real datasets (see Table 1) to compare all of 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 and vary the degree D for two types of synthetic graphs (ER and PA) in order to compare all of 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 and we vary the degree per node (D) and the label set size ($|\mathcal{L}|$) in order to study the effect of these parameters. In the fourth and final part we 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 DBFS. We do this to study the effect of certain parameters (e.g., degree, label set size). For the real datasets we report all of 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.

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. In general, it is always better (for query performance) to have more landmarks. Hence, in practice, choosing the number of landmarks boils down to this trade off between space and time, which is based on the constraints of the environment in which the index is deployed.

Datasets. We used both synthetic graphs and real datasets. We generated the synthetic graphs using SNAP [22, 23] following the 'preferential attachment' (PA) and 'Erdős-Rényi' (ER) models; we did not get directed graphs initially, hence we set the direction of the edges randomly. 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. The labels incident on the edges are exponentially distributed with $\lambda=\frac{|\mathcal{L}|}{\alpha}$, were $\alpha=1.7$.

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 [22] or KONECT [21]. 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 **StringsHS** 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.

Query generation. For each dataset, we generated two query sets, using $|\mathcal{L}|/4$ and $|\mathcal{L}|-2$ labels. Each query set consists 1,000 true-queries and 1,000 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

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

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	$\|V\|$	$\|E\|$	$\|\mathcal{L}\|$	Synthetic labels
robots[5]	1.4k	2.9k	4	
advogato [21]	5.4k	51k	4	
epinions [21]	131k	840k	8	✓
StringsHS[6]	16k	1.2M	7	
NotreDame [21]	325k	1.4M	8	✓
BioGrid[7]	64k	1.5M	7	
StringsFC[8]	15k	2.0M	7	
webStanford [22]	281k	2.3M	8	✓
webGoogle [22]	875k	5.1M	8	✓
webBerkstan [22]	685k	7.6M	8	✓
Youtube[9]	15k	10.7M	5	
zhishihudong [21]	2.4M	18.8M	8	✓
socPokec [22]	1.6M	30.6M	8	✓
wikiLinks(fr) [21]	3.0M	102.3M	8	✓

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 and DBFS as for each $v \in V_L$ we can use QueryLandmark to get a relatively quick answer to the query.

Speed-up calculation. The goal of an index is to speed up the query answering process relative to the index-free baseline, i.e., DBFS. Hence, we define a total speed-up over DBFS. 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 two query sets for each dataset and each query set consists of two query conditions, we have four 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 Q_c is defined as: $\frac{\sum_{i=1}^{1,000} T_{DBFS}(Q_c(i))}{\sum_{i=1}^{1,000} T_M(Q_c(i))}$. The *individual speed-up* for the i'th query and a method M under a query condition condition Q_c is defined as: $\frac{T_{DBFS}(Q_c(i))}{T_M(Q_c(i))}$. And, the *average speed-up* for a method M is the average of the total speed-ups belonging to the four query conditions.

6.1 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. The major performance criteria here are indexing time and size and query speed-up. We expect to see a trade off between indexing costs and query performance.

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 DBFS. First, except for extremely small graphs, LI$^+$ almost always shows better performance than LI. For true queries, we have that LI$^+$ is always the fastest method, with up to two orders of magnitude speed up over DBFS.

LI does not show any advantage over DBFS for false-queries. While LI$^+$ is an order of magnitude faster than DBFS for false-queries in 6 out of 28 cases and within an order of magnitude in 18 cases, in the remaining 4 cases it is slower by an order of magnitude or more. In general, false-queries take much more time than the true-queries, given the BFS-like forward-searching strategy of LI$^+$. 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, that is, 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**. Indeed, digging deeper, we found that the nature of these data sets and the false-queries were such that failure was determined much closer to the query targets than to their sources, i.e., BFS starting from s often stopped very close to t in false queries. Hence, the subgraphs explored from the source nodes were significantly larger than those explored from the target nodes. In these cases DBFS was able to take advantage of search direction optimization to more quickly resolve the queries.

The number of labels in a query can also affect the speed-ups of LI$^+$. A true-query with a high number of labels should have less difficulty in 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 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.

In general, the numbers in Table 3 meet our expectations for LI$^+$, with up to two orders of magnitude improvement achieved for query processing on true queries and within the same order of magnitude or better in 85% of the cases

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	11,867	369
epinions	272	2,903	205	2,091	-	-	-	-
StringsHS	15	556	13	542	93	5,993	-	-
NotreDame	242	2,424	193	1,895	-	-	-	-
BioGrid	58	1,410	50	1,302	36	3,207	-	-
StringsFC	21	513	19	503	91	5,055	-	-
webStanford	935	7806	906	7,719	-	-	-	-
webGoogle	5,887	33,931	5,665	33,497	-	-	-	-
webBerkstan	2,463	15,690	2,364	14,874	-	-	-	-
Youtube	3,121	336	2,841	300	-	-	-	-
zhishihudong	4,068	9,488	3,574	8,351	-	-	-	-
socPokec	9,762	77,155	9,461	75,698	-	-	-	-
wikiLinks(fr)	24,873	98,125	25,641	103,414	-	-	-	-

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

Dataset	$\lvert\mathcal{L}\rvert/4$						$\lvert\mathcal{L}\rvert - 2$					
	true			**false**			**true**			**false**		
	LI	LI$^+$	DBFS (μs)	LI	LI$^+$	DBFS (μs)	LI	LI$^+$	DBFS (μs)	LI	LI$^+$	DBFS (μs)
robots	17.63	17.63	1.77	6.95	6.95	0.70	24.43	24.43	2.44	6.08	6.08	0.61
advogato	84.25	93.08	20.6	3.33	1.89	0.67	124.92	86.15	20	3.74	2.11	0.82
epinions	69.18	52.10	106	0.00	0.58	1.91	63.30	82.22	96.7	0.00	0.66	2.06
StringsHS	161.27	64.83	247	0.00	0.83	5.1	199.58	600.42	235	0.01	4.79	44.8
NotreDame	22.27	7.27	159	5.17	49.44	555	21.76	19.12	58.1	0.86	58.37	260
BioGrid	16.98	20.79	848	0.18	37.95	709	91.41	90.28	1180	0.31	46.22	2,610
StringsFC	55.60	84.20	565	0.01	0.83	17.9	73.90	86.52	529	0.02	2.69	39.8
webStanford	407.17	448.00	780	0.01	2.16	40.6	681.71	964.86	1370	0.02	0.07	84.4
webGoogle	338.26	184.52	1,340	0.00	0.57	9.65	199.76	177.00	1050	0.00	0.00	10
webBerkstan	126.72	157.59	995	0.01	0.78	97.6	95.25	236.50	886	0.02	0.23	260
Youtube	16.10	21.10	2,000	0.53	12.19	3,880	91.69	51.81	3,080	1.56	50.00	6,730
zhishihudong	3.45	4.18	53.6	0.01	6.21	260	29.50	34.17	437	0.00	0.05	115
socPokec	9.20	10.81	1,290	0.00	0.25	39.8	7.52	9.09	972	0.00	0.09	21.5
wikiLinks(fr)	54.53	44.54	3,120	0.00	0.42	38.8	43.75	43.74	2,790	0.00	0.18	34.5

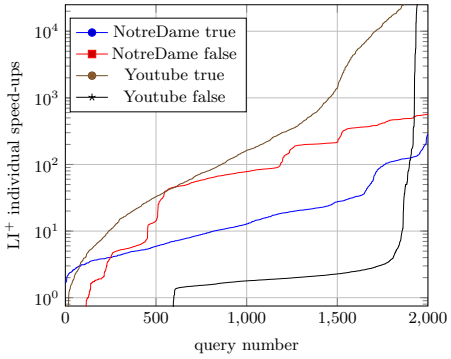


Figure 2: Individual speed-ups for LI$^+$ on **NotreDame** and **Youtube**, sorted in ascending order.

involving false queries. Towards improving LI$^+$ performance on false queries, an interesting direction for future work is to study alternative bi-directional DBFS-like search strategies for query evaluation.

Towards a deeper understanding of the distribution of query speed-ups, Figure 2 shows the individual speed-ups of **NotreDame** and **Youtube** for their 2,000 true- and 2,000 false-queries and method LI$^+$, sorted in ascending order. Only a small minority (6%) of the **NotreDame** false-queries and less than one third of the **Youtube** false-queries have an individual slow-down, i.e., a speed-up lower than 1.0. We also note that over 99% of the **NotreDame** true-queries and 97% of the **Youtube** true-queries have at least 2x speed-ups, and furthermore, the majority of the true-queries on both data sets have at least 10x speed-up.

6.2 Results II: synthetic graph performance

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 (i.e., number of edges from 10,000 to 25,000), thereby increasing the density of the graph. Our aim here is to understand, using two significantly different synthetic graph models, the impact of graph density on performance, as density is a basic property of graphs. We expect that building indexes on denser graphs will be more difficult for all methods, as the number of possible paths to explore and index between nodes, and the number of minimal label sets, increase with density. Table 4 summarizes the results.

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	3,528.3	270.0	146.3	90.2	217.5
	3	5.3	143.1	4.7	141.1	10.8	471.8	12,676.0	471.8	116.0	77.4	196.4
	4	9.7	196.0	8.6	194.1	18.7	333.0	-	-	95.1	66.5	208.5
	5	16.7	247.0	16.2	244.4	33.5	786.3	-	-	88.2	58.6	186.0
ER	2	4.1	115.6	3.5	115.1	8.5	357.9	-	-	171.0	96.7	371.7
	3	3.1	129.3	2.4	128.5	7.6	462.7	-	-	57.6	103.4	281.4
	4	11.4	182.7	10.7	181.7	23.4	637.4	-	-	53.6	73.9	242.0
	5	33.8	267.4	37.6	266.3	64.3	909.0	-	-	39.9	77.3	257.6

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 with 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. We can note that as D increases, speed-up on average decreases. This is due to the fact that the average size of connected components in the graphs increase as D increases, and hence our methods must perform relatively more work on false queries (and only on the false queries), as discussed in the previous section.

6.3 Results III: impact of degree and label set size

We next 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). Our aim here is to better understand the impact of both the number of labels in a graph and graph density on LI$^+$ performance. We expect that, as graphs grow in either of these dimensions, indexing costs will increase, since the number of possible paths to explore and index between nodes will only increase.

Figures 3 and 4 show the indexing time, the index size and the query times for the PA-datasets and the ER-datasets. 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 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 is explained by the number of minimal label sets connecting a (typical) pair of vertices, which heavily affects the performance of LI$^+$. The number is exponential 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 query performance there is not a clear difference between ER- and PA-datasets. Overall, we observe stable query times as the number of labels grows.

6.4 Results IV: impact of graph structure

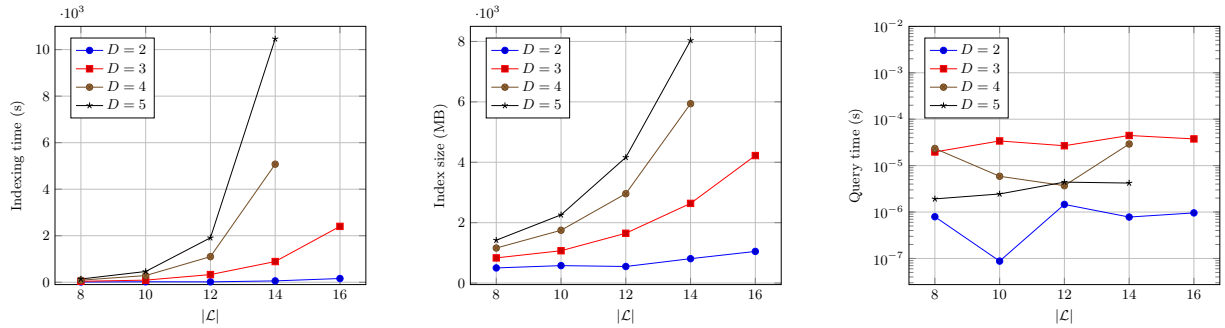
Finally, we analyze the performance of LI$^+$ on PA- and ER-datasets as we vary the number of nodes $n \in \{5,000, 25,000, 125,000, 250,000\}$, fixing the degree as $D = 5$. Our goal here is to understand the scalability of LI$^+$.

Figure 5 shows the indexing time, the index size and the average query times. 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, we see that average query times increase for both ER- and PA-datasets, as expected on larger graphs. The PA-datasets exhibit a stronger increase 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 that search on PA-datasets has to explore larger parts of the graph and hence take more time to evaluate a query.

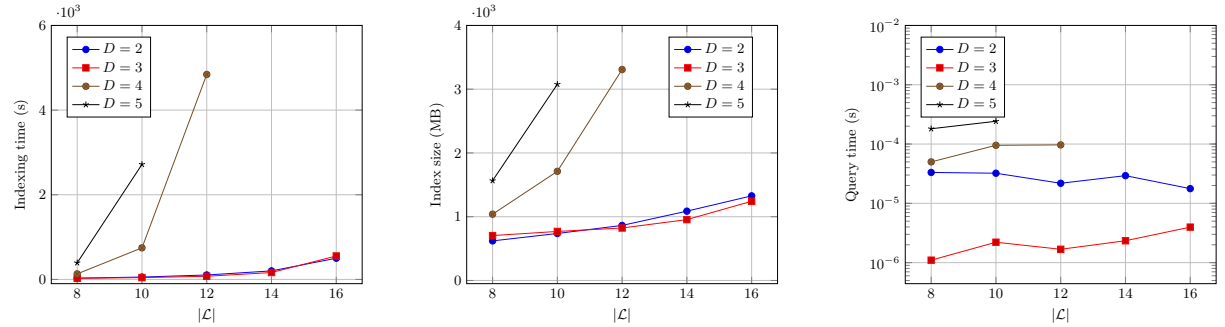
7. CONCLUDING REMARKS

We have presented and analyzed novel index-based methods for efficient scalable evaluation of LCR queries. We demonstrated through an in-depth experimental study that our methods scale to graphs which are orders of magnitude larger than those supported by previous index-based solutions, with significant speed-ups in query evaluation times.



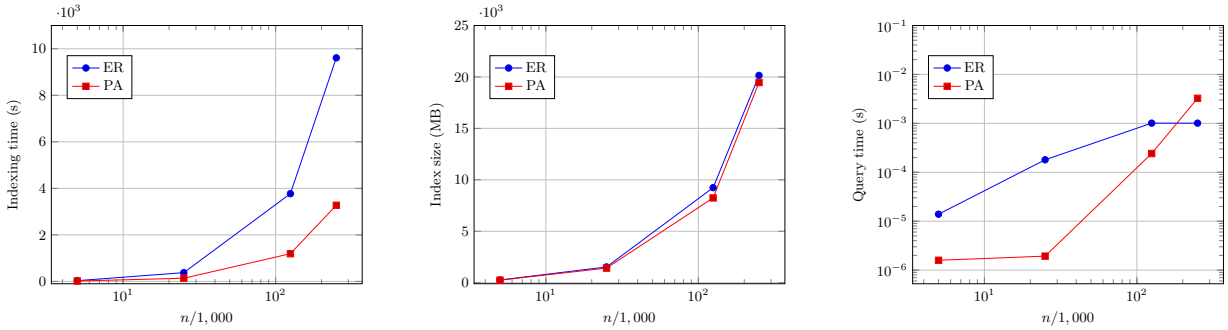
(a) Indexing time (s) (b) Index size (MB) (c) Query time

Figure 3: Indexing time, index size and average query times 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) Query time

Figure 4: Indexing time, index size and average query times 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.



(a) Indexing time (s) (b) Index size (MB) (c) Query time

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

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. Another important area for further study is alternative search strategies for improving LI$^+$ performance on false queries, as discussed in Section 6.1. Finally, applications of our indexes for evaluation of practical query languages such as SPARQL1.1 and openCypher is an important direction for future research.

Acknowledgments. We thank Yongming Luo for his collaboration on the LCR problem and the reviewers for their careful feedback.

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] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern graph query languages. *CoRR*, abs/1610.06264, 2016.

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

[6] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 2017 (to appear). Preprint at http://arxiv.org/abs/1511.08386.

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

[8] S. Beamer, K. Asanovic, and D. A. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

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

[10] M. Chen, Y. Gu, Y. Bao, and G. Yu. Label and distance-constraint reachability queries in uncertain graphs. In *DASFAA*, pages 188–202, 2014.

[11] 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.

[12] 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.

[13] 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.

[14] G. H. L. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In *EDBT*, pages 636–639, 2016.

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

[16] A. Gubichev et al. Sparqling kleene: fast property paths in RDF-3X. In *GRADES*, 2013.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[32] N. Yakovets, P. Godfrey, and J. Gryz. Query planning for evaluating SPARQL property paths. In *SIGMOD*, pages 1875–1889, 2016.

[33] 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.

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

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

[36] 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.

[37] 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.* [37] introduced in Section 2. We can break this method down into nine steps. Steps 1 and 2 are

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

	D	Step 1	2	3	4	5	6	7	8	9	# SCC
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

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 [37] 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 $\mathsf{lnd}(v, w)$, where $L' \in \mathsf{lnd}(w)$. 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.

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 [5, 31].

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

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 $(w, L') \in \mathsf{lnd}(v)$ 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.