Improved Graph Indexing Algorithms for Label-Constrained Reachability Queries*

Sankardeep Chakraborty[1], Mohammad Najafi[2], and Srinivasa Rao Satti[3]

[1] The University of Tokyo, Japan
sankardeep.chakraborty@gmail.com
[2] Seoul National University, South Korea
najafi@tcs.snu.ac.kr
[3] Norwegian University of Science and Technology, Norway
srinivasa.r.satti@ntnu.no

Abstract. Nowadays graph data have become absolutely ubiquitous in various applications starting from social/road networks to bio-medical data etc. Given such graph data, a *reachability* query asks if there exists a path from a source vertex to a target vertex in the graph. Due to its immense implications in both theory and applied domains, this query and many of its variants have been extensively studied in the literature. One such variant investigates the reachability between two vertices in an edge-labeled graph while constraining the label set simultaneously. This problem has recently been addressed by Valstar et al. [SIGMOD'17] who proposed an approach called the *landmark indexing* (LI) to support faster label-constrained reachability (LCR) queries. In this work, we introduce a simple, practical and space-efficient solution for answering LCR queries even faster. The experimental evaluation shows significant time and space efficiency benefits of our proposed solution over the LI approach for this problem in both real-world and synthetic graphs.

Keywords: Graph indexing · Reachability query · Landmark indexing

1 Introduction

As graph databases and real-world network graphs continue to increase in popularity and expand in size, the ability to efficiently query information out of these graphs will also increase in importance. One such family of queries which garnered notable amount of research is the *reachability* query [10, 12, 23, 25, 26]. Given any two nodes $v_1, v_2 \in V$ in a graph $G = (V, E)$, the reachability query answers if there exists a path between the queried nodes. As outlined in [28], these are used across a wide range of use cases including data mining [12], web site analysis [8], and biological network analysis [7] etc. For example in bioinformatics, vertices may be either molecules, reactions, or interactions in living

* The work of the first author was supported by MEXT Quantum Leap Flagship Program (MEXT Q-LEAP) Grant Number JPMXS0120319794.

cells and edges will represent how these various biological elements interact with each other. In this context, reachablity queries help biologists determine which genes are influenced, either directly or indirectly, by a given molecule.
Label-constrained reachability (LCR) [11, 13, 27] queries represent a subset of reachability queries that have been attracting much recent interest. This concerns with finding a path between two vertices in an edge-labeled graph, where the labels on the edges belonging to the path should come from a given query subset of the labels. For example, in a typical biological database, we might only be interested in the interaction pathways between different proteins and not so much in citation relationships holding between protein vertices and vertices denoting scientific publications. Another important application lies in the appearance of LCR queries as an important segment of the language of regular path queries [2, 4], which are actually reachability queries constrained by regular expressions. In practical graph query languages such as SPARQL 1.11 (see http://www.w3.org/TR/sparql11-query/) and PGQL [19], LCR (and other related) queries have been implemented. See [24] for more applications.

It is easy to see that there exists two very naive and straightforward approaches to answering reachability queries. The first approach is to create the transitive closure of the query graph, which can then be used to answer reachability queries in $O(1)$ time. The other approach is to perform a breadth-first search (BFS) over the graph at query time. Both these approaches on their own are not feasible on real-world graphs – the first approach uses too much space, while the second approach is too slow. Thus, any practical reachability query solution should aim to strike some sort of trade-off between these two extremes.

In this paper, we study a variant of the standard reachability query, called the labeled-constrained reachability which is formally defined as follows. Given a graph $G = (V, E, \mathcal{L})$ where each edge in E is labeled with a label from the label set $\mathcal{L}$, we want to preprocess the graph to build an index such that given an LCR query (s, t, L), we need to determine if there exists a path P from s to t such that all the edge labels on the path P belong to the set $L \subseteq \mathcal{L}$. As LCR query is fundamental to many practically motivated applications, obtaining an efficient solution for this problem is of paramount importance in literature of modern graph analytics. Starting from the work of Jin et al. [11], this problem has already been extensively studied by many researchers [6,30], and the state-of-the-art result is due to Valstar et al. [24] which is what we significantly improve in this work. More specifically, our main results are following.

Main contribution: We propose three different graph indexing schemes, each one successively improving the previous one and finally culminating with the ASL (details are deferred till Section 3) method which improves the current state-of-the-art landmark indexing (LI) method for LCR queries significantly; in terms of space consumption, our ASL method performs significantly better than the LI method while supporting the queries much faster (for both `true` and `false` queries). This makes our method more suitable for queries on extremely large graphs, where the memory demands of the LI method may make this approach infeasible. The rest of the paper is organized as follows. After a brief discussion

of related work on this problem in Section 2, we explain our three methods in Section 3. Then, we provide our experimental results in Section 4, before finally concluding in Section 5.

2 Related Work

Landmark indexing (LI) algorithm [24]: The landmark indexing (LI) algorithm proposed by Valstar et al. [24] aims to provide a tradeoff for solving LCR queries. LI works by constructing indices for a small group of vertices, referred to as "landmarks". Given a graph $G = (V, E, \mathcal{L})$, for each landmark vertex v, an index is stored of $(w, L) \in V \times 2^{\mathcal{L}}$ if there exists an *L-path* (i.e., a path in which all the edge labels belong to the subset L) from v to w. Queries for which the source vertex is a landmark can be answered by simply checking the index. In general, queries are answered by performing a BFS from the source vertex, using the indices stored to speedup the lookup at landmark vertices, and ending the search as soon as we reach the target vertex. In this regard, the LI method can be seen as somewhat of a compromise between time efficient and memory efficient approaches.

Hotz et al. [9] have recently shown the efficiency of LI in real-world database Neo4j [17]. LI, being the most recent development for the LCR queries, serves as an excellent benchmark for our proposed algorithm. In fact, the focus of our work is to improve some of the issues of this algorithm. More specifically, one glaring issue with landmark indexing algorithm is its performance for non-landmark nodes. The authors note that especially for LCR queries on the non-landmark nodes where the answer is `false`, their algorithm performs no better than the standard BFS. Given the huge indexing space taken up by the landmark nodes, this algorithm represents a 'worst of both worlds' scenario for LCR queries which return `false`. Also, the result of [24] and the previous ones do not scale well to extremely large graphs required in most of today's applications, thus, we provide an improved time/space efficient, simple and practical solution for the LCR problem surpassing all the previous results.

Tree-based index framework [11]: This approach works by building a full transitive closure (TC) of the underlying graph to answer LCR queries. As storing the full TC is costly, authors devised a tree-based index framework which contains a spanning tree T and a partial transitive closure (PT) of the graph. PT and T have enough information to derive the full TC. One drawback of this method is that it is not practical for dense graphs since the size of PT increases with the density. Subsequent work has pointed out the limitation of this approach [30], thus, we do not use this in our experiments.

BFS Optimization [5]: In this work, the authors extend the standard top-down BFS as follows. In the standard BFS, starting from the source vertex, all the vertices at the same depth are visited before any vertex of the next depth. Experiments showed this approach to be between 2-5 times faster than the standard BFS on real-world social network graphs and other practical networks. BFS plays an instrumental role both in landmark indexing, and our proposed algorithms.

We use this optimized BFS in all of our implementations, including LI.
Standard reachability query [28]: We refer the readers to the survey of Yu and Chang [28] where they provided detailed history of the standard reachability query (and its many variants) along with sketching various solutions for this problem. This concludes our brief discussion on relevant related work.

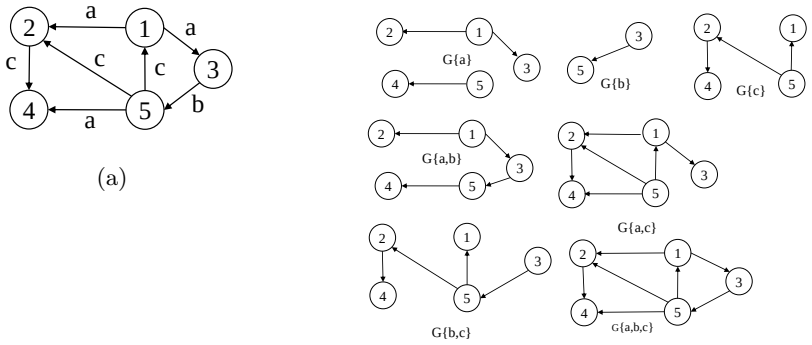
3 Proposed Methods

In this section, we propose three different indexing schemes for answering LCR queries and analyze their worst-case space, construction time and query time complexities. We first start with the All Label Combinations (ALC for short) method, and describe its query algorithm, and time/space complexities in Section 3.1. This is followed by our second method, called ALC+SCC in Section 3.2, which is an improvement over the ALC method, by incorporating strongly connected component decomposition (SCC) of directed graphs along with the ALC method to derive the ALC+SCC method. Finally, in Section 3.3, we explain how one can combine the ALC+SCC method with the landmark indexing (LI) scheme to get even better results. We call this as ASL (**A**LC+**S**CC+**LI**) algorithm.

3.1 All Label Combinations (ALC)

Given a labeled graph $G = (V, E, \mathcal{L})$, the ALC method simply constructs several unlabeled graphs, one for each non-empty label-subset. This leads to the construction of $2^\ell - 1$ unlabeled subgraphs (we use ℓ to denote the size of the label set $\mathcal{L}$ in this paper). For example, if the set of labels for a graph is $\mathcal{L} = \{a, b, c\}$ then the ALC algorithm first computes all possible label combinations $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$ and $\{a, b, c\}$, creating a (sub)graph corresponding to each subset L such that the labels of all the edges in the graph are only from L. In the rest of the paper, we denote, by G_L, the unlabeled subgraph of the given labeled graph G which contains exactly those edges in G whose edge labels are from L, and their incident vertices. Also, given a graph G, we use $G.V$ and $G.E$ to refer to the vertex and edge sets of the graph G. Now, given a query (s, t, L), we perform a BFS on the precomputed subgraph G_L of G to establish whether there exists a path from s to t. It is easy to see the correctness of this algorithm.

Time and Space Analysis. For constructing $2^\ell - 1$ unlabeled subgraphs, we spend overall $O(2^\ell(n+m))$ time during preprocessing. The query time is at most linear in the size of the subgraph G_L (hence $O(n+m)$). The overall space usage of the algorithm is the sum of the sizes of all graphs G_L, for every $L \subseteq \mathcal{L}$, which is $O(2^\ell(n+m)\log n)$ bits. As it will be clear later (in Section 4) that for all data sets, the ALC method performs worse that the LI method for answering queries, but its main novelty lies in the fact that it reduces the LCR problem from a labeled graph reachability problem to an unlabeled graph reachability problem.



(a)

(b)

Fig. 1: (a) An example graph with 5 vertices, 7 edges and labels $\mathcal{L} = \{a, b, c\}$. (b) All the subgraphs based on all possible non-empty label combinations of $\mathcal{L}$.

3.2 ALC+SCC Method

This method tries to address one of the drawbacks of ALC method, which is it's high query time. As mentioned earlier, given an LCR query (s, t, L), ALC right away jumps to the graph G_L, and by doing this, we prune many unnecessary edges (whose labels are not from L) to look at from the point of view of answering this query. But this advantage soon gets waned as we are essentially performing a BFS to scan the whole of G_L. Thus, if we can perform this second step faster, this will result in an overall faster algorithm. And, the way we achieve this is by incorporating the strongly connected component decomposition of each of the individual graphs produced by the ALC method. We refer to the resulting algorithm as ALC+SCC method. Also, if the graph has several vertices that belong to these SCCs, then the overall space usage improves as a result. We now describe the algorithm and analyze its time, space and query complexities.

Strongly connected components: A directed graph G is said to be strongly connected if for every pair of vertices u and v in V, both u and v are reachable from each other. A subgraph G' is a strongly connected component (SCC) of G if G' is maximal and is strongly connected. If G is not strongly connected, it is possible to decompose G into its SCCs. It is also possible to test the strong connectivity of G, and to find its SCCs, in $O(|V| + |E|)$ time [20]. For undirected graphs, we use connected component decomposition instead of the SCC decomposition, but for simplicity, we refer to this also as SCC decomposition.

Decomposition Algorithm: Given an edge-labeled graph, we first build the collection $SCCALC_{out}$ of $2^\ell - 1$ pairs $\langle L, G'_L \rangle$, where L is a non-empty subset of $\mathcal{L}$, and G'_L is obtained as follows. Given a label subset L, let G_L denote the graph which contains all (and only those) edges whose label belongs to L. We first perform an SCC decomposition of G_L [20]. This is followed by merging of each of the SCCs of G_L into a super node (i.e., contracting all the vertices inside an SCC into a single node). This results in a directed acyclic graph (DAG), which

we refer to as G'_L. Note that, by doing this we are not losing any reachability information, since all the vertices inside an SCC are mutually reachable. This completes the description of our ALC+SCC algorithm.
Query Algorithm: Given an LCR query (s, t, L), the algorithm first locates the corresponding unlabeled DAG G'_L in $SCCALC_{out}$, and then compares whether the vertices s and t belong to the same SCC. If yes, the algorithm returns `true`. Otherwise, we answer the query by performing a BFS over G'_L starting at s.
Time and Space Analysis: As the SCC algorithm is called once for each of the $2^\ell - 1$ (unlabeled) graphs, the construction time of our algorithm is $O(2^\ell(n+m))$ (but in practice, the running time is much smaller as many of these graphs may contain much smaller than m edges). SCC algorithm stores the SCCs of $2^\ell - 1$ graphs, each with size at most $O(n + m)$, thus, the total space consumption is $O(2^\ell(n+m)\log n)$ bits. The query algorithm performs BFS on the relevant graph in $O(n + m)$ time to answer LCR queries. Note that for undirected graphs, the query algorithm simply needs to check whether the two query vertices belong to the same SCC or not, and hence queries can be supported in $O(1)$ time. This completes the description of our ALC+SCC method.

3.3 ASL Method

As mentioned above, ALC+SCC method improves upon the query time over the naive ALC method. In what follows, we show that we can improve the query time even further (at the cost of negligible space increase) by incorporating the landmark indexing (LI) method on top of the ALC+SCC method, resulting in an algorithm which we call ASL (**A**LC+**S**CC+**L**I). As we will see later in Section 4, this combined method performs much better than any single algorithm individually (ALC, ALC+SCC or LI) as it inherits the good traits from each one.

Index construction: We use ASL_{out} to refer to the output of the ASL algorithm, which consists of a set of $2^\ell - 1$ pairs $\langle L, G''_L\rangle$, for every non-empty label subset L, and G''_L is constructed by first selecting the corresponding graph G'_L from $SCCALC_{out}$, and applying landmark indexing on it. Note that, in our case, we apply landmark indexing on unlabeled graphs (DAGs). To do so, for each landmark v in each of these unlabeled DAGs, we store the set of all vertices reachable from v, referred to as $Ind(v)$, as part of the index for v. For non-landmark vertices, our algorithm does not store any data.
Query Algorithm: Given an LCR query (s, t, L), the algorithm first checks whether s is a landmark. If so, it answers by looking at $Ind(s)$. Otherwise, it starts a BFS from s to either reach t or reach a landmark v for which $t \in Ind(v)$. Then answer `true`; otherwise, return `false`.
Time and Space Analysis: The construction of LI for each unlabeled graph G'_L, with n' vertices, m' edges and z landmarks, takes $O((n' \log n' + m')z)$ time [24] (by substituting $\ell = 1$ in the construction time for LI). The algorithm constructs indices for each landmark in each of the $2^\ell - 1$ (unlabeled) graphs, (in addition to the construction of the ALC and SCC decomposition from the previous sections. Consequently, the overall construction time is $O(2^l z(n \log n + m))$.

Table 1: Worst-case space, query time, and construction time complexities of all three algorithms. z denotes the number of landmarks chosen, and $\ell = |\mathcal{L}|$.

	Space (bits)	Construction Time	Query Time
`LI`	$O(nz2^{\ell}(\log n + \ell))$	$O((n(\log n + 2^{\ell}) + m)z2^{\ell})$	$O(n + m + z2^{\ell})$
`ALC`	$O(2^{\ell}(n + m)\log n)$	$O(2^{\ell}(n + m))$	$O(n + m)$
`ALC+SCC for Directed`	$O(2^{\ell}(n + m)\log n)$	$O(2^{\ell}(n + m))$	$O(n + m)$
`ALC+SCC for Undirected`	$O(2^{\ell}n \log n)$	$O(2^{\ell}(n + m))$	$O(1)$
`ASL for Directed`	$O(2^{\ell}zn \log n)$	$O(2^{\ell}z(n \log n + m))$	$O(n + m)$

The space complexity of ASL for storing G''_L is $O(n'z \log n')$ bits, So, the overall space usage is bounded by $O(2^{\ell}nz \log n)$ bits. The query time depends on the size of the query graph G''_L. The query algorithm performs a BFS over the graph G'; as a result, it takes $O(n' + m')$ time. For an undirected graph, we do not use LI in but simply use the ALC+SCC to answer the queries, and hence the amount of space and query time for an undirected graph are the same as that for the ALC+SCC method. We summarize the worst case space, query time and construction time complexities of all our algorithms (along with LI) in Table 1.

Applying Compression: In all our methods, we convert the given labeled graph into several unlabeled graphs, and at query time, perform a BFS on one of those unlabeled graphs. Hence, to save the space further, we apply a standard compression scheme (we use `gzip` in our experiments) on these unlabeled graphs. As we will see later, the query time does not get effected much since we only need to decompress one of these graphs at query time. In all our experiments, ALC, ALC+SCC, and ASL always refer to the corresponding methods described earlier with the `gzip` compression applied on the resulting data structure.

4 Experiments

In this section, we evaluate our approaches against the LI method. We use the following three standard criteria to judge the competing algorithms:

- *Response time*: time in milliseconds (ms).
- *Memory*: index space taken in Kilobyte (Kb).
- *Construction time*: preprocessing time to create the index in seconds (s).

All of our approaches i.e., ALC, ALC+SCC, and ASL are implemented in C++. Test cases and all implementations are accessible as open-source for future studies[1]. Although ASL uses LI as subroutine, it is worth mentioning that the input to ASL is an unlabeled graph whereas LI takes a labeled graph as input, and hence the indices constructed by these two methods are totally different. More specifically, as part of the index, ASL stores vertices that are modified by the SCC decomposition without any associated labels, while LI stores initial vertices of the given graph along with some label subsets.

[1] https://github.com/MSNTCS/ALC

Table 2: The table below lists all the data sets that we use in our work.

DataSets	$\|V\|$	$\|E\|$	$\|\mathcal{L}\|$	Label?	DataSets	$\|V\|$	$\|E\|$	$\|\mathcal{L}\|$	Label?
Synthetic 1	100	242	4	Yes	**robots**	1400	2900	4	No
Synthetic 2	500	2485	4	Yes	**Advogato**[14]	14010	70472	4	No
Synthetic 3	5000	12492	4	Yes	**webGoogle** [16]	875K	5.1M	8	Yes
Synthetic 4	100	485	8	Yes	**webStanford** [16]	281K	2.3M	8	Yes
Synthetic 5	1000	2994	8	Yes	**WebBerkstan** [16]	658K	7.6M	8	Yes
Synthetic 6	2000	5994	8	Yes	**Youtube**[29]	15K	10.7M	5	No
Synthetic 7	5000	24985	8	Yes	**StringFC**[22]	15K	2M	7	No
					BioGrid[18]	64k	1.5M	7	No
					StringHS[21]	16K	1.2M	7	No

We use the following setup for our work. The experiments were performed on a windows machine with intel i7-6700 processor, 32 GB memory, and using single-thread. The number of landmarks in both (ASL and LI) approaches was set to be 10 percent of the number of vertices of the corresponding input graph.

Datasets. There are two types of data sets used in our experiments, synthetic and real-world data sets. We use 9 different real-world data sets in our experiments, which are chosen mainly from SNAP [16], and also from a variety of other sources. These are 9 of the 14 real-world data sets that were used in the experimental evaluation of the LI algorithm [24]. (The remaining 5 data sets were accessible at the time of our experiments.) For the unlabeled data sets, labels were augmented randomly, with label set size $\ell = 4$ or 8. **BioGrid**, **StringsFC**, and **StringsHS** are originally undirected due to the structure of the graph representing protein relations. In addition, we also use synthetic graphs that are generated by SNAP [15, 16] using Scale-Free Model (SF). The direction and labels are augmented to the synthetic graphs randomly. For all the synthetic graphs, the size of label set is either 4 or 8. The reason why we chose preferential attachment over other graph generation models is that the scale-free model follows the power-law connectivity distribution [3], thus, making them more likely to be close to real-world graphs. The power-law connectivity distribution parameter α is set to 2.5 to simulate real-world networks [1]. Table 2 shows all information regarding data sets used in our experiments.

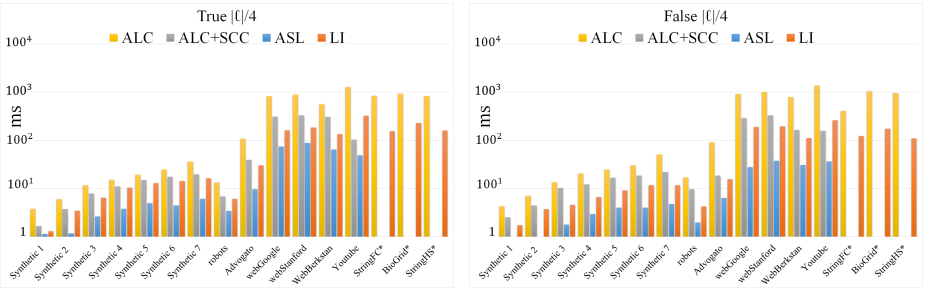
Queries. Four types of query sets were used in these experiments, two `true` and two `false` query sets. Each query set contained either t (= 1000) `true` queries or t `false` queries. For both `true` and `false`-query sets, we considered two different label sizes $\ell/4$ and $\ell-2$, where ℓ is the total number of distinct labels in the input graph. For query generation, we followed the same methodology as the landmark indexing algorithm [24]. For completeness, we describe the details here. Given a labeled graph $G = (V, E, \mathcal{L})$, the query generation starts by selecting a random vertex $v \in V$, and then generates a random number r between $50 + \log n$ and $50 + n/50$, where $n = |V|$. Then, the procedure begins a loop for $t/100$ times. In

each round, the procedure chooses another random $u \in V$ and generates 10 label sets $L_1, ..., L_{10}$. For each label set L_i, for $1 \leq i \leq 10$, the procedure runs a BFS to check whether the LCR query (v, u, L_i) returns `true` or `false`. In the BFS traversal, the procedure counts the number of vertices visited, say r'. If $r' < r$, the procedure disregards the query and runs the loop from the beginning. The procedure also disregards duplicates and runs until we find all t `true`-queries and t `false`-queries.

4.1 Performance Analysis

In this section, we compare the performance of ALC, ALC+SCC, and ASL methods with LI on both real-world and synthetic data sets mentioned in Table 2. In particular, we will compare the query time, space usage and construction time for all these methods. From this comparison, in what follows, we conclude that ASL method takes much less space and query time compared to LI, while taking larger construction time.

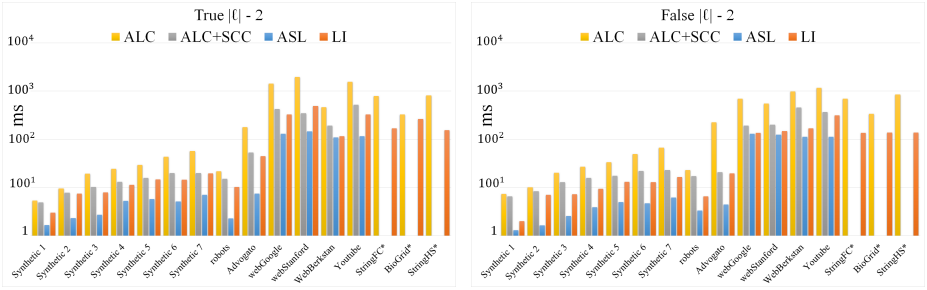
Query Time. The first experiment is designed to measure the query time of each approach. Figures 2 and 3 depict query times of `true` and `false` queries, with label set sizes $\ell/4$ and $\ell - 2$.



(a) (b)

Fig. 2: Query times for ALC, ALC+SCC, ASL and LI algorithms, measured in μs (microseconds). Here $\ell/4$ refers to the number of labels in the query. (a) `true`-queries (b) *False*-queries.

For ALC, response time was consistently slower than LI for both `true` and `false` queries. As can be seen in Figures 2 and 3, the proposed ALC method is between 3-75 times slower than LI for `true` queries and 1-29 times slower for `false` queries. This makes sense, because in LI as soon as any landmark is reached, the algorithm can immediately return `true` if the target node t is reachable from the landmark. Hence the speed up of LI over ALC for the `true` queries is better than for `false` queries. It is worth mentioning that the query time for ALC does not have an adverse effect with an increase in the query label set size,



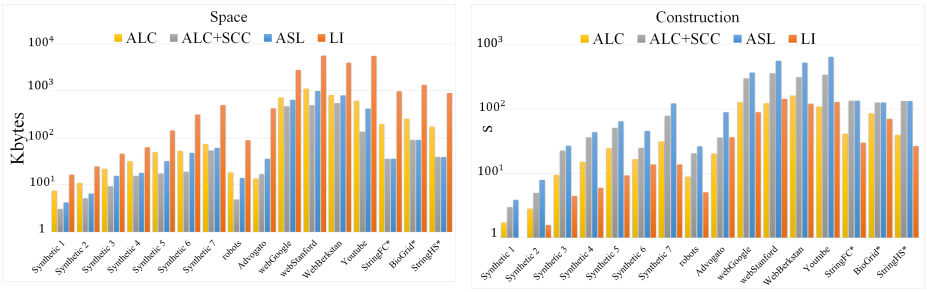
(a) (b)

Fig. 3: Query times for ALC, ALC+SCC, ASL and LI algorithms, measured in μs (microseconds). Here $\ell - 2$ refers to the number of labels in the query. (a) `true`-queries (b) *False*-queries.

$|L|$, while this parameter has a more adverse impact on LI. The y-axis in the figures is log normalized. To improve the response time of the basic ALC method, the ALC+SCC procedure was designed. This algorithm sacrifices construction time to improve query time significantly. This trait can be particularly useful for static graphs where construction time cost is paid only once. The ALC+SCC approach performs significantly better on undirected graphs since this approach can answer the queries in constant time (for both `true` and `false` queries) due to the fact that after employing the connected component decomposition for undirected graphs, the outcome would be disjoint subgraphs (connected components). Consequently, ALC+SCC is significantly faster than LI in the case of undirected graphs. For directed graphs, the query time largely depends on the number of SCCs in the input graph. Therefore, the response time is relatively slow compared to LI. In the worse case, ALC+SCC is 5 times slower than LI for `true` queries and 11 times slower for `false` queries. This happens in the case of relatively sparse graphs such as **WebStanford**, which give rise to many small sized SCCs, and hence results in a larger DAG. The difference between the `false` and `true` queries which were different in the case of ALC, has improved in ALC+SCC by storing extra information about out-portal vertices for each SCC. Again, as in the case of ALC, an increase in the query label set size does not have an adverse effect on the query time for ALC+SCC.

Finally, the ASL method does not have original drawbacks that were present in the LI as it neither depends on the query label set size, nor it stores any labels in the indices. As a result, iterating over the indices is much quicker, resulting in overall faster query time. In the best case, ASL performs 44 times faster than LI for `true` queries over **Synthetic 7** which is a dense graph, hence, has large SCCs; and 51 times faster for `false` queries in **WebBerkstan** which is a sparse graph, but the algorithm managed to store a good amount of out-portal information to answer `false` queries much faster than LI.

Memory Usage. In the second experiment, we compare the memory usage of each method. Figure 4(a) shows the individual space usage of the 4 methods. The space usage in both ASL and LI relies on the number of landmarks. The details of our findings are described as follows.



(a) (b)

Fig. 4: (a) Memory usage for ALC, ALC+SCC, ASL and LI algorithms, measured in Kb and (b) construction time for ALC, ALC+SCC, ASL and LI algorithms, measured in s (seconds).

The ALC algorithm required less space than LI. As can be seen in Figure 4(a), the ALC method used between 1.3 (on **Synthetic 4**) and 81 (on **Youtube**) times less space when compared to the LI algorithm. The difference in space usage depends on the number of landmarks k used by the algorithm, the size of the graph, and the total number of labels. The ALC algorithm had the least notable memory usage benefits over LI for small graphs, **Synthetic 1** and **Synthetic 4**. The ALC algorithm had the most notable space savings over LI for **WebBerkstan**.

The ALC+SCC method improves query time over ALC, while at the same time reducing the space usage. The ALC+SCC approach reduces space to the extent that in the **Youtube** dataset, it uses 961 times less amount of space, compared to the LI method. For the **StringFC** data set, it can store all the necessary information to answer the queries with 532 times less amount of space than the LI method. In some cases, like **WebBerkstan**, due to the fact that the data set is not dense and does not have a lot of SCCs, space usage is 71 times better than LI. Therefore, label set size, ℓ and the number of SCCs mainly affect the space usage of ALC+SCC.

The space usage of ASL is in general more than that of the ALC+SCC, and it depends upon various factors i.e., the density of the graph, the label set size, number of SCCs, and the number of landmarks chosen. Also, the space usage of ASL and ALC+SCC methods are the same in the case of undirected graphs (since LI is not employed in ASL for undirected graphs). The amount of used space in comparison to LI is up to 177 times less, and this is a significant saving when we are especially dealing with large graphs. The denser the graph is, the

lesser the space usage becomes for ASL.

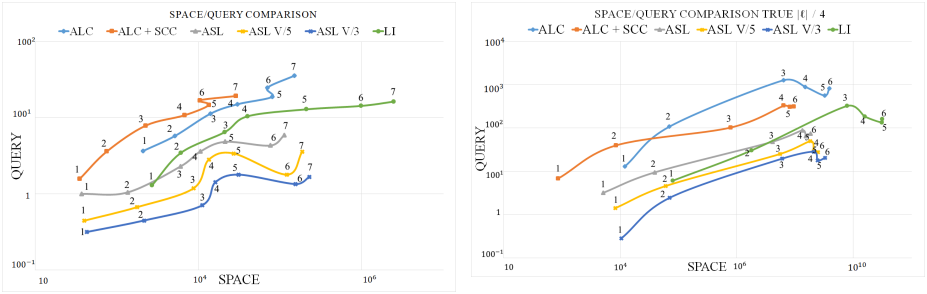
Construction Time. Finally, we compare the construction time for all four approaches in Figure 4(b) (here the y-axis is log normalized) for all the data sets. Note that the times mentioned for ASL and LI are based on the number of landmarks mentioned in the previous sections.

The ALC method takes 3-7 times longer construction time when compared to the LI algorithm (see Figure 4(b)). ALC benchmarked best in comparison to LI on **Youtube**, while it benchmarked the worst on **Synthetic 7**. The reason why **Youtube** construction works well is that the label size of the graph is small in comparison to other real-world graphs. Also the reason that **Synthetic 7** behaves worst is the graph becomes dense and LI can perform well on dense graphs. Applying the SCC decomposition adds more overhead in construction time, which is a tradeoff to gain both space savings and query time performance benefits. In some cases, ALC+SCC construction time is relatively comparable to LI while in other cases due to the size of the networks and also the size of the label set, the construction time is up to 37 times larger (on **StringFC** which has the less number of SCCs with $\ell = 7$). One can observe that the size of the label set, the size and density of the graph, and the number of SCCs have a huge impact on the construction. As expected, the ASL method consumes even more construction time than the ALC+SCC (except for undirected graphs). The construction time is up to 78 times slower than LI (on **Synthetic 6**). In all cases, the construction time is proportionate to the label set size of the graph.

ASL Speedup. To accelerate the query performance in ASL even further, we can increase the number of landmarks in each subgraph. In this set of experiments, we increase the number of landmarks to $|V|/5$ and $|V|/3$, which we refer to as ASLV/5 and ASLV/3, respectively. This improves the query performance slightly without adding any substantial space overhead. Figure 5 shows the space and the query time comparison of all methods. Figure 5(a) is for synthetic graphs; numbers from 1 to 7 are assigned respectively to Synthetic 1 to Synthetic 7. Figure 5(b) is for real-world graphs, in which numbers from 1 to 6 are assigned respectively to robots, Advogato, Youtube, webGoogle, WebBerkstan and webStanford. Increasing the number of landmarks in the ASL approach improves query performance to some extent even though its space usage can become higher. Even after increasing the number of landmarks in the ASL methods, their space usage is still much smaller than that of LI (with fewer of landmarks).

5 Conclusions

In this paper, we introduce three novel algorithms to answer LCR queries. Comprehensive experiments on both real-world and synthetic data sets confirm that our ASL algorithm supports the LCR queries faster with much less amount of space, in comparison with the current state-of-the-art solution, landmark indexing (LI) algorithm, at the cost of increased construction time. As is the case



(a) (b)

Fig. 5: Space and query comparison for all methods including ASL with different number of landmarks on (a) synthetic data sets and (b) real-world data sets. Query times are for `true` queries with label size $\ell/4$.

with Landmark Indexing, the space usage of our index structures is also exponential in terms of the alphabet size. It would be an interesting open problem to design an indexing structure that is scalable with the alphabet size. Another challenging problem is to support LCR queries for graphs changing dynamically, where either the edges are inserted/deleted dynamically or the labels are modified dynamically.

References

1. Albert, R., Barabási, A.: Statistical mechanics of complex networks. CoRR **cond-mat/0106096** (2001)
2. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern graph query languages. CoRR **abs/1610.06264** (2016)
3. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. science **286**(5439), 509–512 (1999)
4. Barrett, C.L., Jacob, R., Marathe, M.V.: Formal-language-constrained path problems. SIAM J. Comput. **30**(3), 809–837 (2000)
5. Beamer, S., Asanović, K., Patterson, D.: Direction-optimizing breadth-first search. Scientific Programming **21**(3-4), 137–148 (2013)
6. Chen, M., Gu, Y., Bao, Y., Yu, G.: Label and distance-constraint reachability queries in uncertain graphs. In: DASFAA. Lecture Notes in Computer Science, vol. 8421, pp. 188–202. Springer (2014)
7. El-Samad, H., Prajna, S., Papachristodoulou, A., Doyle, J., Khammash, M.: Advanced methods and algorithms for biological networks analysis. Proceedings of the IEEE **94**(4), 832–853 (2006)
8. Fernandez, M., Florescu, D., Levy, A., Suciu, D.: A query language for a web-site management system. SIGMOD record **26**(3), 4–11 (1997)
9. Hotz, M., Chondrogiannis, T., Wörteler, L., Grossniklaus, M.: Experiences with implementing landmark embedding in neo4j. In: Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). p. 7. ACM (2019)

10. Jagadish, H.: A compression technique to materialize transitive closure. ACM Transactions on Database Systems (TODS) **15**(4), 558–598 (1990)
11. Jin, R., Hong, H., Wang, H., Ruan, N., Xiang, Y.: Computing label-constraint reachability in graph databases. In: SIGMOD. pp. 123–134 (2010)
12. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: SIGMOD. pp. 595–608. ACM (2008)
13. Koschmieder, A., Leser, U.: Regular path queries on large graphs. In: SSDBM. pp. 177–194. Springer (2012)
14. Kunegis, J.: Konect: the koblenz network collection. In: Proceedings of the 22nd International Conference on World Wide Web. pp. 1343–1350. ACM (2013)
15. Leskovec, J., Sosič, R.: A general purpose network analysis and graph mining library (2014)
16. Leskovec, J., Sosič, R.: Snap: A general-purpose network analysis and graph-mining library. ACM Transactions on Intelligent Systems and Technology **8**(1), 1 (2016)
17. Miller, J.J.: Graph database applications and concepts with neo4j. In: Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA. vol. 2324 (2013)
18. Oughtred, R., Chatr-aryamontri, A., Breitkreutz, B.J., Chang, C.S., Rust, J.M., Theesfeld, C.L., Heinicke, S., Breitkreutz, A., Chen, D., Hirschman, J., et al.: Biogrid: a resource for studying biological interactions in yeast. Cold Spring Harbor Protocols **2016**(1), pdb–top080754 (2016)
19. v. Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: a property graph query language. In: Boncz, P.A., Larriba-Pey, J. (eds.) GRADES. p. 7. ACM (2016)
20. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications **7**(1), 67–72 (1981)
21. Szklarczyk, D., Franceschini, A., Wyder, S., Forslund, K., Heller, D., Huerta-Cepas, J., Simonovic, M., Roth, A., Santos, A., Tsafou, K.P., et al.: String v10: protein–protein interaction networks, integrated over the tree of life. Nucleic acids research **43**(D1), D447–D452 (2014)
22. Szklarczyk, D., Morris, J.H., Cook, H., Kuhn, M., Wyder, S., Simonovic, M., Santos, A., Doncheva, N.T., Roth, A., Bork, P., et al.: The string database in 2017: quality-controlled protein–protein association networks, made broadly accessible. Nucleic acids research p. gkw937 (2016)
23. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD. pp. 845–856. ACM (2007)
24. Valstar, L.D.J., Fletcher, G.H.L., Yoshida, Y.: Landmark indexing for evaluation of label-constrained reachability queries. In: SIGMOD. pp. 345–358 (2017)
25. Wadhwa, S., Prasad, A., Ranu, S., Bagchi, A., Bedathur, S.: Efficiently answering regular simple path queries on large labeled networks. Age **3**, v7 (2019)
26. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: 22nd ICDE. pp. 75–75. IEEE (2006)
27. Xu, K., Zou, L., Yu, J.X., Chen, L., Xiao, Y., Zhao, D.: Answering label-constraint reachability in large graphs. In: CIKM. pp. 1595–1600. ACM (2011)
28. Yu, J.X., Cheng, J.: Graph Reachability Queries: A Survey, pp. 181–215. Springer US, Boston, MA (2010)
29. Zafarani, R., Liu, H.: Social computing data repository at ASU (2009)
30. Zou, L., Xu, K., Yu, J.X., Chen, L., Xiao, Y., Zhao, D.: Efficient processing of label-constraint reachability queries in large graphs. Inf. Syst. **40**, 47–66 (2014)