Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Web Engineering Research Group

Efficient processing of label-constrained reachability queries

L.D.J. (Lucien) Valstar

Supervisors:
G.H.L. Fletcher
Y. Yoshida

Committee members:
G.H.L. Fletcher
Y. Yoshida
M.A. Westenberg
A. Driemel

version 1.0

Eindhoven, July 2016

Abstract

Our world today is generating huge amounts of graph data such as social networks, biological networks, and the semantic web. Many of these real-world graphs are edge-labeled graphs, i.e. each edge has a label that denotes the relationship between the two vertices connected by the edge. A fundamental research problem on these is how to handle reachability in these kinds of graphs: *can u reach v using only edges with particular labels?*. There has not been very much research on this topic yet. Hence we have come up with our own solution. We dealt with the problem by looking for several algorithms that can deal with these queries. We ran several experiments to examine their performance relative to a baseline algorithm. The results show that there is a clear performance improvement by building an index.

Keywords: LCR, Label-constrained reachability, Labelled graph

Contents

Chapter 1

Introduction

In this chapter we introduce the problem studied in this thesis, we provide an overview of the state of the art, explain briefly our contributions and give a thesis outline.

1.1 Problem motivation

Our world today is generating huge amounts of graph data such as social networks, biological networks, and the semantic web. The sheer size of these graphs (e.g. Google Knowlede Graph has 570 million vertices/objects and 18 billion edges/facts) turns a simple query into a difficult one. One of these queries is "reachability", i.e. the question on whether we can from a point (or vertex) A reach another point B in the graph using the edges of the graph.

Many real-world graphs are edge-labelled graphs. This means that edges in the graph have a label from a pre-defined label set. This changes a reachability-query into a so-called "label-constrained reachability" query or "LCR"-query. The question for such a query is: "can we reach from point A another point B in the graph using only certain types of edges?".

Examples of graphs for which there are labels are numerous.

In **Social networks** each person is represented as a vertex. Any person can have an interaction with another person if they are related in some way. Examples of such relationships could be colleague, friend or family (sister-of, brother-of, son-of). It can be interesting to look at more complicated relationships between people. When we want to know whether person A is a remote relative of person B we might look to explore the graph using only edges brother-of, sister-of, son-of, etc.

In **Bioinformatics** there is a need to understand metabolic chain reactions in cellular systems. For this a metabolic network is used. In such a network, each vertex represents a chemical compound. An edge indicates that one compound can transform into another. The edge label records the enzymes that are needed to control the reaction. The basic question is to whether there is a pathway between two compounds where the enzymes are present under certain conditions.

RDF-graphs consist of triplets (or facts) that indicate that entity A is related to entity B through a certain relationship. Information from multiple sources (e.g. IMDb, Facebook or Wikipedia) may be combined. A given person A might create a FOAF-record describing personal information, interests and friendships. This may be linked to for instance a movie review this user has written on another website. A query could be "how is a respondent B to the movie review written by person A related to person A" or "in what ways are two particular movie reviews X and Y written by user A related".

1.2 State of the art

There has not been done much study to the topic of "LCR". Zou et al. [10] have built their solution to answering this type of query. The results in the paper are convincing. We were unable to obtain a version of their code and hence we made our own best-effort implementation of it.

Then there is Bonchi et al. [1]. They aimed at answering a query similar but more difficult than a LCR-query. Nevertheless it can be a good comparison of our two methods.

1.3 Contributions

This study contributes to a few novel things. The first is exploring the existing (Zou) and baseline (BFS) solutions and evaluating their performance. The next is trying to come up with our own solutions to the problem (PartialIndex , LI , ClusteredExact and DoubleBFS) and evaluating their performance as well. Finally we demonstrate that our major contribution (LI+OTH+EXTv2) is scalable and can be extended to answer richer queries as well.

1.4 Thesis outline

Chapter 2 defines the problem we study in our thesis. Chapter 3 gives a literature overview on "reachability" and "LCR". Also we discuss some of the ideas for actual algorithms. Chapter 4 discusses the experimental set-up used in the experiment and discusses the datasets. Chapter 5 explains the actual methods used in the experiments and the concepts behind them. These methods are based on the discussion in the literature review. Chapter 6 shows the results of the experiments. Chapter 7 concludes on all material and provides work that can be done in the future.

Chapter 2

Problem statement

This chapter defines two notions, i.e. "reachability" and "label-constrained reachability" which is abbreviated to "LCR".

2.1 Reachability in graphs

We start by defining a directed graph $G = (V, E)$. Definition 2.1.1 defines one. Figure 2.1 shows an example of a graph. The circles are called nodes or vertices and the arrows are called edges. The circles have an number inside them and this is called the vertex identifier (or vertex id).

Definition 2.1.1. (Directed) graph: Let $G = (V, E)$ be a (directed) graph. The number of vertices is $|V| = n = N$. The number of edges is $|E| = m = M$. When a vertex v is in the graph, we say $v \in V$. When an edge from a vertex v to w is in the graph, we say $(v, w) \in E$. In an undirected graph, we have that $(v, w) \in E$ is equivalent to $(w, v) \in E$. For each pair of vertices (v, w) there is at most one edge.

The use of reachability queries in graphs has been studied extensively [9, Chapter 6] [8, 1]. A reachability query basically asks whether we can reach from a vertex $v \in V$ another vertex $w \in V$. By this we mean that we can move from v to w in G using any edge in the graph. The vertices that are traversed by going from v to w are said to be a path.

The notion of "reachability" (Definition 2.2.2) and a "path" (Definition 2.1.2) can formally be defined as:

Definition 2.1.2. Path: Let $G = (V, E)$ be a (directed) graph. Let $s, t \in V$ be two arbitrary nodes. A path P exists in G from s to t if and only if we have one of the following three cases.

1. $s = t$. In this case $P = \langle s \rangle$.
2. $(s, t) \in E$. In this case $P = \langle s, t \rangle$.
3. Then there exists a sequence of $k \geq 1$ vertices $w_1, \ldots w_k$ such that $(s, w_1) \in E$, that for all $1 \leq i \leq k - 1$ $(w_i, w_{i+1}) \in E$ and $(w_k, t) \in E$. In this case $P = \langle s, w_1, \ldots, w_k, s \rangle$.

Let $\#P \geq 1$ be the length of the path, which is the number of vertices in the path. Let $P[i]$ with $1 \leq i \leq \#P$ be the i'th element in the path.

Definition 2.1.3. Reachability: Let $G = (V, E)$ be a (directed). Let $s, t \in V$ be two arbitrary nodes. If there exists a path P from s to t, we say t is reachable from s or $s \rightsquigarrow t$.

Figure 2.1 shows an example of a graph and a path. Vertex 1 can reach vertex 5 over the orange colored vertices.

For large scale graphs answering this question is highly challenging. Certain trade-offs need to be made between index size, query answer time and index construction time. Methods that build up a transitive closure of the graph create an index of size $O(n^2)$ in $O(nm)$ time, but are able to answer a query in $O(1)$ time [9, Chapter 6]. One could think of a transitive closure as a binary matrix of size $n \times n$. Answering these queries on the fly by running a depth- or breadth-first search live which has a running time of $O(n + m)$ but has zero

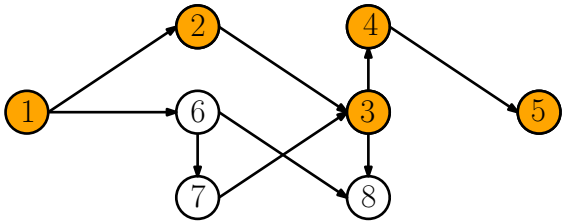


Figure 2.1: An example of a graph with $|V| = 8$ and $|E| = 9$. The orange colored vertices are a path P of the form $\langle 1, 2, 3, 4, 5 \rangle$. The length of the path, i.e. $\#P$, is 5.

construction time and index size. For large graphs both methods are unacceptable. For example, storing a transitive closure for $n = 1,000,000$ requires 125GB. We need methods that have lower storage requirements at the expense of a higher query time.

2.2 Label-constrained reachability

First we are going to extend the notion of a graph such that we include a pre-defined label set to it.

Definition 2.2.1. Labelled graph: Let $G = (V, E, \mathcal{L})$ be a labelled (directed) graph. $\mathcal{L}$ is a set of labels and $E \subseteq V \times V \times \mathcal{L}$. An edge from $v \in V$ to $w \in V$ can be written as $(v, w) \in E$ or $(v, w, l) \in E$ where $l \in \mathcal{L}$. Moreover let $\mathsf{Label}(e) : E \to \mathcal{L}$ be a mapping from edge $e \in E$ to its corresponding label. Any $l \in \mathcal{L}$ is called a "label" and any $L \subseteq \mathcal{L}$ is called a "label set".

In this thesis, we make a survey of the problem of label constrained reachability (LCR). Formally this can be defined as:

Definition 2.2.2. Label-constrained reachability (LCR): Let $G = (V, E, \mathcal{L})$ be a (directed) labelled graph. Let $s, t \in V$ be two arbitrary nodes. Let $L \subseteq \mathcal{L}$ be a label set. We say that there is a L-path from s to t if there exists a path P from s to t such that for each edge $(P[i], P[i+1], l) \in E$ with $1 \leq i \leq \#P$ we have that $l \in L$. When this is the case, we also write $s \overset{L}{\rightsquigarrow} t$.

An example of a directed labelled graph can be seen in Figure 2.2. There exists a $\{a, b\}$-path from 1 to 5: $\langle 1, 3, 5 \rangle$. From now on when we show a picture of a labelled graph we wish to stick to the following convention: red is label a, blue is label b, green is label c and orange is label d.

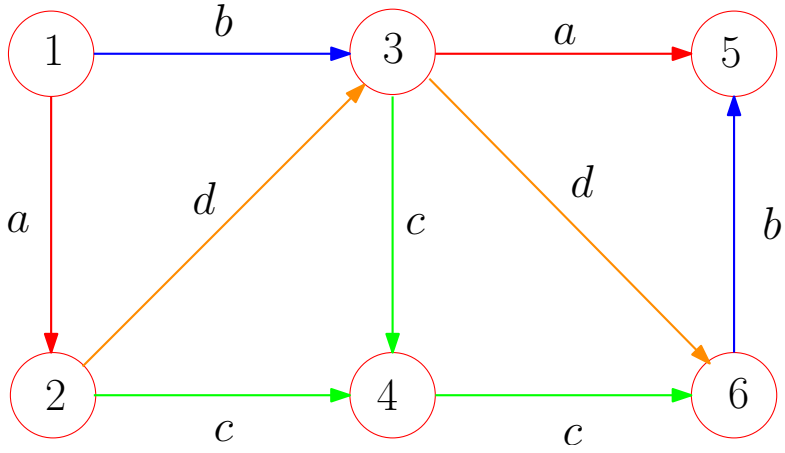


Figure 2.2: An example of a graph with labels. From node 1 to 6 we can see for instance an $\{a, c\}$-path: $(\langle 1, 2, 4, 6 \rangle)$. There is also an $\{a, d\}$-path, a $\{b, c\}$-path and a $\{b, d\}$-path. Hence query $q = (1, 6, \{a, c\})$ is answered by true whereas $q = (1, 6, \{c\})$ is answered by false.

LCR was introduced by Jin et al. [5] and further studied by Zou et al. [10] and Chen et al. [2]. The study of LCR was particularly motivated by the study of "regular path queries". These kinds of queries are prevalent in practical graph query languages such as SPARQL 1.1[1] and Cypher[2]. One can also think of the Facebook-graph (1.35 billion nodes[3]), where the edge labels indicate a type of relationship ('friend','colleague' or 'partner'), or Google Knowledge graph (570 million nodes, 18 billion edges[4]), where the edge labels could indicate that two entities are a synonym or entity A is a generalization of entity B, as examples of edge-labelled graphs.

A (LCR)-query q that determines the existence of a L-path from s to t could be defined as:

Definition 2.2.3. LCR-query: Let $G = (V, E, \mathcal{L})$ be a (directed) labelled graph. Let $s, t \in V$ and $L \subset \mathcal{L}$. Then, we can define a query $q \in V \times V \times \mathcal{L}$ as (s, t, L). If $s \overset{L}{\rightsquigarrow} t$, then q is said to be true (or a True-query). Otherwise, q is said to be false (or a False-query).

An example query on the graph in Figure 2.2 could be $(1, 6, \{a, c\})$ which would yield true. The query could also be written as: $(1, 6, (a + c)^*)$, where $(a + c)^*$ indicates that we are allowed to pick any number of a or c's in

[1] http://www.w3.org/TR/sparql11-query/, see Section 9: Property paths
[2] http://neo4j.com/docs/stable/cypher-query-lang.html
[3] https://nl.wikipedia.org/wiki/Facebook
[4] http://www.cnet.com/news/googles-knowledge-graph-tripled-in-size-in-seven-months/

our path from 1 to 6. One could easily think of more enhanced queries like: "Is there a path from 1 to 6 using 4 a's first and then at most 3 b's?" or "What is the distance in terms of the number of edges or unique labels?". However, this is not in the scope of our masters thesis. We mainly focus on the first type of queries, although we do elaborate on the possibilities to extend our solution to these types of queries.

2.3 The problem

The problem is to find a way to answer reachability queries in labelled graphs efficiently and accurately. Trade-offs between index size, query time, index construction time and accuracy have to be made in this regard. Also, we need to take into account that an index might update from time to time due to updates of the graph (node insertion, node removal, edge insertion, edge removal or label change). The frequency of the updates need to be taken into account.

On the one hand, we can use a slightly adapted version of BFS (or DFS) to answer queries. This approach has the maximal query time but no minimal index construction time. BFS also has minimal update costs.

On the other hand, we can build an index for the full graph that answers all possible queries instantly. This approach has the minimal query time but the maximal index construction time. We assume index construction time and size are correlated. For large graphs building a full index is too cumbersome. Also, updates might require a lot of re-work.

Other approaches will mostly lie into the middle of this spectrum. At the expense of a higher query time, the index time will be reduced.

2.4 Auxiliary definitions

Below we give some auxiliary definitions that will be used along the thesis.

The first definitions (Definitions 2.4.1, 2.4.2, 2.4.3 and 2.4.4) can be used both for labelled and unlabelled graphs. Definition 2.4.1 defines for each vertex $v \in V$ two subsets, the set of vertices that can be reached from v and the set of vertices that can reach v. Definition 2.4.2 defines a strongly connected component (SCC). This is used by Zou and to define the connectedness of datasets. Definition 2.4.3 defines the properties 'simple' and 'shortest' for a path which is used by Defintion 2.4.4. Definition 2.4.4 defines the set of paths between two points, which is used by Definition 2.4.6.

Definition 2.4.1. Ancestors and descendants: In a directed graph for a given node $v \in V$ a set of ancestors and descendants can be defined. Formally, we can define the set of ancestors as $ANCS(v) = \{w \in V \mid w \text{ can reach } v\}$ and the set of descendants $DESC(v) = \{w \in V \mid v \text{ can reach } w\}$. Note that $v \in ANCS(v) \wedge v \in DESC(v)$.

Definition 2.4.2. Strongly connected component: In a directed graph, a strongly connected component (SCC) is a subset of the nodes $A \subseteq V$ s.t. $\forall[v, v' \in A \mid v \rightsquigarrow v']$. A weakly connected component (WCC) is a subset A s.t. $\forall[v, v' \in A \mid v \rightsquigarrow v' \vee v' \rightsquigarrow v]$.

Definition 2.4.3. Simple and shortest path: Let G be a (directed) graph. Let $s, t \in V$. Let P be a path from s to t. P is said to be 'simple' if and only if each vertex $P[i]$ with $1 \leq i \leq \#P$ occurs exactly once in P. P is said to be 'shortest' if there does not exist a subsequence P' of P that is also a path from s to t.

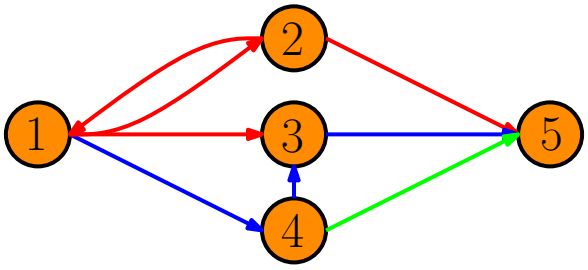
Definition 2.4.4. Set of paths: In a (directed) graph, the set of simple paths between s and t can be expressed as $P(s, t)$. For $s = t$, we have that $P(s, t) = \emptyset$.

Definitions 2.4.5 defines the notion of a "path label", i.e. the union of all labels of edges that are between vertices of the path. Note that there can never be multiple edges between two vertices and hence L_P is unique for any path P. Definition 2.4.6 claims that a path P is minimal if its path label is pairwise incomparable to that of any other path $P' \in P(s, t)$. In this way the path label of P is guaranteed to be minimal. Definition 2.4.7 defines the set of all minimal paths between s and t. This definition is used by our analysis of algorithm LI . Figure 2.3 shows an example graph G with $P(s, t)$ and $P_{\min}(s, t)$ defined for two particular vertices.

Definition 2.4.5. Path label: In a labelled (directed) graph, let $\mathsf{Labels}(P)$ be defined as: $\bigcup_{i=1}^{\#P-1} \mathsf{Label}(P[i], P[i+1])$. This is also called "the label of a path" or L_P.

Definition 2.4.6. A minimal path: Let G be a (directed) labelled graph. Let $s, t \in V$. Let $P \in P(s,t)$. P is said to be a minimal path if and only if for all other $P' \in P(s,t)$ we have that $\mathsf{Labels}(P) \not\subseteq \mathsf{Labels}(P') \wedge \mathsf{Labels}(P') \not\subseteq \mathsf{Labels}(P)$.

Definition 2.4.7. Set of minimal paths: Let G be a (directed) labelled graph. Let $s, t \in V$. Let $P_{\min}(s,t)$ be the largest subset of $P(s,t)$ such that for all $P \in P_{\min}(s,t)$ we have that P is minimal.



$$P(1,5) = \{\langle 1,2,5\rangle, \langle 1,3,5\rangle, \langle 1,4,5\rangle, \langle 1,4,3,5\rangle\}$$

$$P_{\min}(1,5) = \{\langle 1,2,5\rangle, \langle 1,4,3,5\rangle\}$$

Figure 2.3: An example graph G illustrating the definitions. Path $\langle 1,2,1,2,5\rangle$ has not been included in $P(1,5)$ because it is not simple. Path $P_1 = \langle 1,3,5\rangle$ is not included to $P_{\min}(1,5)$ because of path $P_2 = \langle 1,4,3,5\rangle$. We have that $L_{P_2} = \{b\}$, whereas $L_{P_1} = \{a,b\}$. Hence path P_1 is not minimal.

Chapter 3

Literature analysis

In this chapter we go over a number of papers and chapters that have studied the topics of reachability and LCR. First we go over techniques to deal with reachability. Then we go over techniques that can deal LCR-queries or some extension of LCR.

3.1 Reachability

Some of these techniques, e.g. 2-hop cover, are designed for directed acyclic graphs (DAGs). A DAG can be created out of a directed graph by finding a set of strongly connected components and representing each node as such a SCC (see Definition 2.4.2). Tarjan's algorithm is an algorithm that can do this. We wish to not restrict ourselves only to directed or DAG-graphs but rather would like to investigate the methods used by some of these algorithms.

However we do note that LCR adds a whole new dimension to the data. Reachability is like LCR but then with $|\mathcal{L}| = 1$. Setting $|\mathcal{L}| > 1$ makes it impossible to reduce a group of nodes to a single node like with a SCC without losing information.

Many of the approaches stated here are to find a balance between building a full transitive closure (TC) and having to answer all queries using BFS (or DFS). A transitive closure can be seen as a binary matrix of size $O(n^2)$. Using 1 bit per entry, we can store a TC using $n^2/8$ bytes.

3.1.1 2-hop cover

In a 2-hop cover [9, Chapter 6], a node v in a DAG G is assigned a 2-hop code which consists of two lists $\mathsf{In}(v)$ and $\mathsf{Out}(v)$ for each $v \in V$. These lists contain (a subset of) the set of ancestors of v, i.e. $ANCS(v)$, and the set of descendants of v, , i.e. $DESC(v)$. A valid 2-hop cover is achieved if and only if for every $u, v \in V$ with $u \rightsquigarrow v$ we have that $\mathsf{In}(v) \cap \mathsf{Out}(u) \neq \emptyset$.

A trivial 2-hop cover is the full 2-hop cover. This means that we include for each $v \in V$ all ancestors of v to $\mathsf{In}(v)$ and all descendants of v to $\mathsf{Out}(v)$. This clearly gives no advantage over storing a TC.

The goal is to find a minimum 2-hop cover. However finding such a cover is NP-hard [9, Chapter 6].

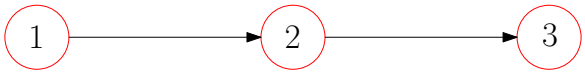


Figure 3.1: A simple graph. In the table below, i.e. Table 3.1, one can see a full hop-cover versus a minimum one.

In Figure 3.1, we can see a simple graph. Its full-hop cover and a minimum cover can be seen in the Table 3.1 below.

Cohen's 2-hop cover approximation

Cohen et al. [3] gives an approximation algorithm which gives a cover at most $O(\log(n))$ larger than the minimum 2-hop cover. The worst case query time is $O(m^{1/2})$ and the index size is $O(nm^{1/2})$.

Table 3.1: This table shows in the 2nd and 3rd column a full 2-hop cover and in the 4th and 5th a minimum 2-hop cover.

vertex	full In	full Out	minimum In	minimum Out
1	$\emptyset$	$\{1,2,3\}$	$\emptyset$	$\{1,2\}$
2	$\{1\}$	$\{2,3\}$	$\{1\}$	$\{2,3\}$
3	$\{2,3\}$	$\{3\}$	$\{2\}$	$\emptyset$

Algorithm 1 shows this approximation algorithm. Let TC be the transitive closure of G. $(u,v) \in TC$ implies that u can reach v. Consider a node w. Let $A_w \subseteq ANCS(w)$ and $D_w \subseteq DESC(w)$. A cluster for w $C_w = S(A_w, w, D_w)$ indicates that every node $u \in A_w$ can reach any node $v \in D_w$. The storage of C_w is $|C_w| = |A_w| + |D_w|$. A cluster C_w indicates that any node in $A_w \subseteq V$ that can reach any node in $D_w \subseteq V$ with a path that goes through w.

Initially we have that for each $w \in V$ $A_w = ASCS(w)$ and $D_w = DESC(w)$ and that $TC' = TC$. As long as TC' is non-empty, i.e. there exists a 1 in the binary matrix of TC', we continue. We look for a cluster C_w that maximizes the equation on line **5**. This is computed by looking at the number of tuples (u,v) where $u \in A_w$ and $v \in D_w$ for which the matrix entry (u,v) in TC is 1. One particular C_w is chosen and for that cluster we remove all (u,v) from TC' and add w to $\mathsf{In}(v)$ and $\mathsf{Out}(u)$.

An interesting observation here is that a full transitive closure is computed, whereas one particular reason for using a 2-hop cover was that a transitive closure would use too much memory.

Algorithm 1 2Hop-Cover(G)

```
1: compute TC
2: TC' ← TC
3: In(v), Out(v) ← ∅ for all v ∈ V
4: while TC' ≠ ∅ do
5:    find max_{w∈} |C_w ∩ TC'|/(|A_w| + |D_w|)
6:    for u ∈ A_w ∧ v ∈ D_w do
7:       remove (u, v) from TC
8:       add w to In(v)
9:       add w to Out(u)
10:   end for
11: end while
```

2-hop cover maintenance

The graph can be updated in four different ways: adding or removing a new node or adding or removing an edge. For labelled graphs also updates of a label may be considered. In [9, Chapter 6], two methods are described to deal with the removal of a node v in a DAG when having a 2-hop code index.

The first method defines a set of nodes $V_{REL} = ANCS(v) \cup DESC(v)$ that consists of all ancestors v and all descendants of v. Let $G_{REL} = (V_{REL}, E_{REL})$ be the subgraph of G where E_{REL} is the maximal subset of E such that for any $(s,t) \in E_{REL}$ we have that $s,t \in V_{REL}$.

A 2-hop cover $L' = (\mathsf{In'}, \mathsf{Out'})$ is computed for G_{REL}. Next, we look at all connections $(a,d) \in E$ of which $a \in V_{REL} \vee d \in V_{REL}$. If $a \in V_{REL}$, then we have that $d \in V_{REL}$, $\mathsf{Out}(a) = \mathsf{Out'}(a)$ and $\mathsf{In}(d) = \mathsf{In'}(d)$. If $d \notin V_{REL}$, then $\mathsf{In}(d) = (\mathsf{In}(d) \setminus V_{REL}) \cup \mathsf{In'}(d)$ and $\mathsf{Out}(d) = (\mathsf{Out}(d) \setminus V_{REL}) \cup \mathsf{Out'}(d)$. The downside of this method is the large size of G_{REL} and the high cost of computation.

The second method trades computing time for storage. Suppose we have that $a \in ANCS(v)$ and $d \in DESC(v)$. Let $W \subseteq V$ be the set of all nodes on simple paths from a to d, excluding a and d. Obviously $v \in W$. The trade-off between computing time and storage can be found in storing some nodes $w \in W$ in $\mathsf{Out}(a)$ and $\mathsf{In}(d)$. If node $v \in W$ gets deleted, we can safely delete v in all $\mathsf{Out}(a)$ and $\mathsf{In}(d)$ because there is another route. Of course this method leads to a much lower compression rate.

3.2 Label-constrained reachability

3.2.1 Bonchi et al.

Bonchi et al. [1] tackle the problem of finding the distance of the shortest path P between two nodes v and w such that $L_P \subseteq L$ for a query (v, w, L). This is an extension of "LCR", because a distance is returned if v can reach w and the need to find a shortest distance implies we need to store any path P' with $L'_P \supseteq L_P$ if P' is a shorter path than P. Two approaches are cited that use landmarks in which the first method is more precise and in which the second is less precise but has a smaller index and needs less construction time.

Both approaches build on the definition of SP-minimality which is similar to Definition 2.4.7. A set of landmarks $X \subseteq V$ is defined. $d_X(s,t)$ denotes the distance between s and t using only edges with labels in X. For simplicity, each edge has a weight of 1. A vertex-pair $(u,x), u \in V \wedge x \in X$ is said to be SP-minimal w.r.t. a label-set T if and only if there is no $S \subset T$ s.t. $d_T(u,x) = d_S(u,x)$. One should note that the shortest distance can only strictly increase by taking a subset of the labels, because less edges can be used than before.

The first method runs single-shortest path (SSSP) between all landmarks $x \in X \subseteq V$ and all $v \in V$ for a set of candidate label sets $C \subseteq \mathcal{P}(\mathcal{L})$. The brute force method takes $O(2^{|C|}k(m + n|C|))$.

However, there are some ways to improve on this running time. Three methods to prune the candidate label sets are mentioned. The first is to look at the labels of edges incident on a landmark x. These labels must be present in a candidate set. The second is the observation that for a given label set L we have that $d_L(x,u) \geq |L|$ because otherwise at least one label $l \in L$ is not used in a path from u to x. The third and last observation uses a history of a set of vertices V_t that are at distance t from the landmark x using a SP-minimal path. The precise details of this method are not entirely clear to us.

The second method assigns a particular label $l \in \mathcal{L}$ to each landmark $x \in X$. Let l_x be that particular label. For each node $v \in V$ we store the "mono-chromatic" distance to each landmark, i.e. $d_{l_x}(v,x)$. The "mono-chromatic" distance is obtained by only using edges of a single type, e.g. only using edges in the label set $\{a\}$. The distances between landmarks are computed using a "bi-chromatic" distance metric, that is using two types of labels e.g. $\{a,b\}$. Given two landmarks $x_1, x_2 \in X$ the distance between x_1 and x_2 would be $d_{\{l_{x_1}, l_{x_2}\}}(x_1, x_2)$. Having k landmarks this can be obtained by doing k breadth-first searches for each node. Hence, we need $O(kn)$ space and $O(km)$ time. However this solution might return approximate or even wrong results. For instance, given two landmarks $x_1, x_2 \in X$ we can have that $d_{\{l_{x_1}, l_{x_2}\}}(x_1, x_2) = \infty$ as there is no $\{l_{x_1}, l_{x_2}\}$-path between the two, but there might be a $\{l_{x_1}, l_{x_2}, l\}$-path between x_1 and x_2 with $l \in \mathcal{L}$. A query $q = (x_1, x_2, \{l_{x_1}, l_{x_2}, l\})$ might incorrectly return ∞ in this case.

3.2.2 Zou et al.

In [10] a solution is proposed to solve LCR-queries. The results of the paper are convincing and it is one of the few paper about "LCR". Hence we have decided to treat it quite detailedly.

In the paper multiple methods for building an index for LCR-queries are being treated. We are only interested in the *transitive-closure method* as this method has the most promising results. Also, we only look at how the index is built and not look at the parts about maintenance on which the paper elaborates as well. From the paper we did not understand all the implementation details and some things remained ambiguous. Hence we will state what our assumptions are in these cases.

Definitions

- Let $G = (V, E, \Sigma, \lambda)$ be a graph where Σ is the set of labels and $\lambda : E \to \Sigma$.
- The set of labels over a path P is $L(P) = \bigcup_{e \in P} \lambda(e)$.
- A path P' covers path P if and only if P and P' have the same end-points and $L(P') \subseteq L(P)$.
- The distance of a path P is the number of distinct labels in P.

Finding single-source transitive closure

First an efficient algorithm to find the single-source transitive closure is sought, i.e. for a node $u \in V$ we basically want to have an index answering any query (u, w, L) with $w \in V$ and $L \subseteq \mathcal{L}$. This transitive closure is named $M_G(u, -)$.

A Dijkstra-like method is proposed. The distance metric is measured as the number of unique labels on a path P. The heap is filled with tuples of the form $(L(P), P, w)$. A tuple T_1 covers T_2 if and only if $w_1 = w_2$ and $L(p_1) \subseteq L(p_2)$. A list RS having all entries for u is filled during the course of the algorithm with these tuples, while $M_G(u, w)$ is filled with $L(p)$ if path p ends at w.

We start at a vertex $u \in V$. Initially, the heap H is filled with all neighbour-tuples of u, i.e. for each $(u, w, l) \in E$ we add an entry of the form $(\{l\}, \langle u \rangle, w)$. Then we iterate over the heap taking an entry (or tuple) per iteration. Let $T_1 = (L(P), P, w)$ be the entry taken from the heap. If T_1 is covered by some $T_2 \in RS$ then there is no need to process T_1 and we can continue. Otherwise, we add $L(P)$ to $M_G(u, w)$ and T_1 to RS. After this we generate neighbour-tuples for w. The paths of each of these new tuples must be simple and each new tuple should not already be covered by some tuple on the heap H. Only then, they are added to H.
In Figure 3.2 we see a graph for which we could run this algorithm. Let's say we wish to compute $M_G(u, -)$. First we would generate u's neighbour-tuples, which are $(\{b\}, \langle u \rangle, v)$ and $(\{a\}, \langle u \rangle, w)$. Next we could generate for instance $(\{a, b\}, \{u, w\}, v)$ because it is a neighbour tuple of w. However, $(\{b\}, \{u\}, v)$ in RS already covers this tuple and hence we do not add this to our index and process the next entry on the heap.

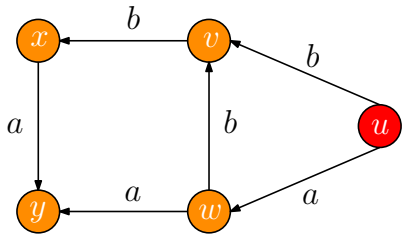


Figure 3.2: A labelled graph as an example. The neighbour-tuples of u are $(\{b\}, \langle u \rangle, v)$ and $(\{a\}, \langle u \rangle, w)$.

The running time of the algorithm is $O(D^d)$ where D is the maximal out-degree and d is the diameter of the graph. This is the theoretical maximum number of paths from any node $u \in V$.

Step 1: Creating a DAG

The first step of Zou's algorithm is to transform a graph G into a DAG D.

First a set of SCC's C is found for the graph G. We assume the classical definition of a SCC (see Definition 2.4.2) is used here and not some new definition entailed to labelled graphs. Although from Figure 3.3 another thing could be assumed, which is merging all vertices $v \in A$ s.t. there is a $\{l\}$-path between any $v, w \in A$ where $l \in \mathcal{L}$. Next we generate $M_{C_i}(u, -)$ using the Dijkstra-like method. This is the internal transitive closure of the i'th strongly connected component C_i.

After a set of SCC's C has been found using e.g. Tarjan's algorithm, we can use this to generate DAG D. Each vertex $v \in C_i$, where C_i is the i'th SCC, is labeled as either: in-portal and/or out-portal or internal. Looking at Figure 3.3, we can say 6 is an internal node and 3 is both an in- and out-portal. By classifying each vertex of a SCC C_i in this way we can generate a bipartite graph B_i.

B_i has a vertex for each in-portal $p_{i1} \in C_i$ and a vertex for each out-portal $p_{i2} \in C_i$. In case $p_{i1} = p_{i2}$ we add an edge to B_i of the form $(p_{i1}, p'_{i1}, \emptyset)$ where p'_{i1} is the second representative of p_{i1} in B_i. In case $p_{i1} \neq p_{i2}$ we look for all label sets L in $M_{C_i}(p_{i1}, p_{i2})$. For each such label set we add an edge of the form (p_{i1}, p_{i2}, L) (in case p_{i2} does not serve as an in-portal as well) and (p_{i1}, p'_{i2}, L) (in case it does serve as an in-portal as well). Figure 3.4 shows an example of a transformation of a SCC C_i to a bipartite graph B_i.

By concatenating all bipartite graphs and trivial SCC's (those consisting of one node) we get a new DAG D. Note that the edges in D are label sets and not labels.

One can be curious as to how effective this approach would be in graphs in which (almost) every node in the graph is an in- and/or out-portal. Then there needs to be an edge from every node in the SCC to every

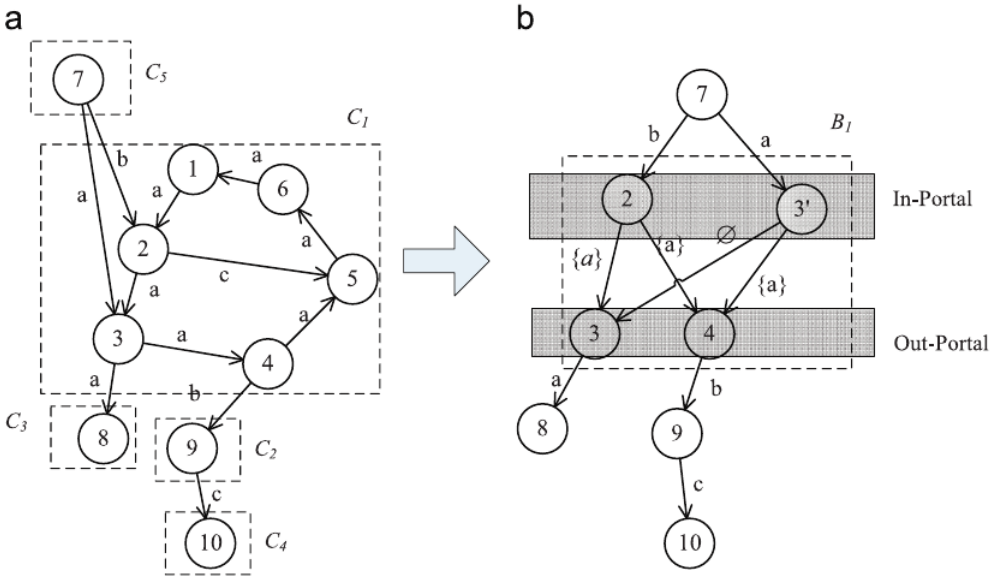


Fig. 4. Augmented DAG. (a) graph *G*. (b) Augmented DAG *D*.

Figure 3.3: Figure 4 from paper [10]. It shows a labeled graph G on the left and the 'classical' SCC's on the right.

other node, which is quadratic in the number of nodes in the SCC. There might be even more edges added then originally present. Thus, the effectiveness of this approach really depends on the percentage of nodes that are serving as either in- and/or out-portals.

Figure 3.4 illustrates this case. On the left a part of a graph eligible to become a SCC is displayed. On the right the result. We see that the number of edges has increased a lot.

Step 2: using D

The vertices in D are ordered according to a reverse topological order RT. Note that nodes that are internal to a SCC are not in D.

First we set for each node $M_G(u,-)$ to $M_{C_i}(u,-)$ if $u \in C_i$.

For $d \in RT$ and $u \in out(d)$ we do $\bigcup_{u \in out(d)} \mathsf{Prune}(\lambda(du) \odot M_G(u,-))$. In case $\lambda(du) = \emptyset$ and u is d's second representative, we do nothing as $M_G(u,-)$ and $M_G(d,-)$ are the same variable. $\mathsf{Prune}(S)$ is an operator that removes any path P' that is covered by another path P in the set S, essentially creating a minimal path set (see Definition 2.4.7). $\odot$ concatenates an edge label with a set of paths and their associated label sets. For example, given an edge (u,v,l) and a path $P = \langle v,w \rangle$ with $\mathsf{Labels}(P) = \{a\}$, we would have that $(u,v,c) \odot P$ results in a path $P' = \langle u,v,w \rangle$ and $\mathsf{Labels}(P') = \{a,c\}$. We repeat this procedure for all nodes $d \in RT$.

Figure 3.5 illustrates the idea. The figure displays the graph as it is, but you might as well see the set of bipartite graphs D as displayed on the right part of the figure. Changes are propagated from bottom to top in this graph.

We start at vertex $12 \in C_3$ after having computed its internal transitive closure. As 12 has no children, we go to C_2. For each out-portal in C_2 (which is $11'$) connected to an in-portal in C_3 (which is 12), we concatenate all entries of the local transitive closure of the in-portal with the edge label of the edge between that in- and out-portal. In the case of C_2 this would set $M_{C_2}(11',12)$ to $(\{b\} \odot M_{C_3}(12,-)$. For each in-portal of C_2 (which are vertices 10 and 11) we look at all the other out-portals in C_2 (which is $11'$). In the case of C_2 and vertices 10 and 11 this would set $M_{C_2}(10,11')$ to $M_{C_2}(10,11') \cup (M_{C_2}(11',-) \odot \{a\}) \cup (M_{C_2}(11',-) \odot \{b\})$, as there is an $\{a\}$ and a $\{b\}$-path connecting 10 and 11 in C_2. After we have completed this process for C_2 we do the same for C_1.

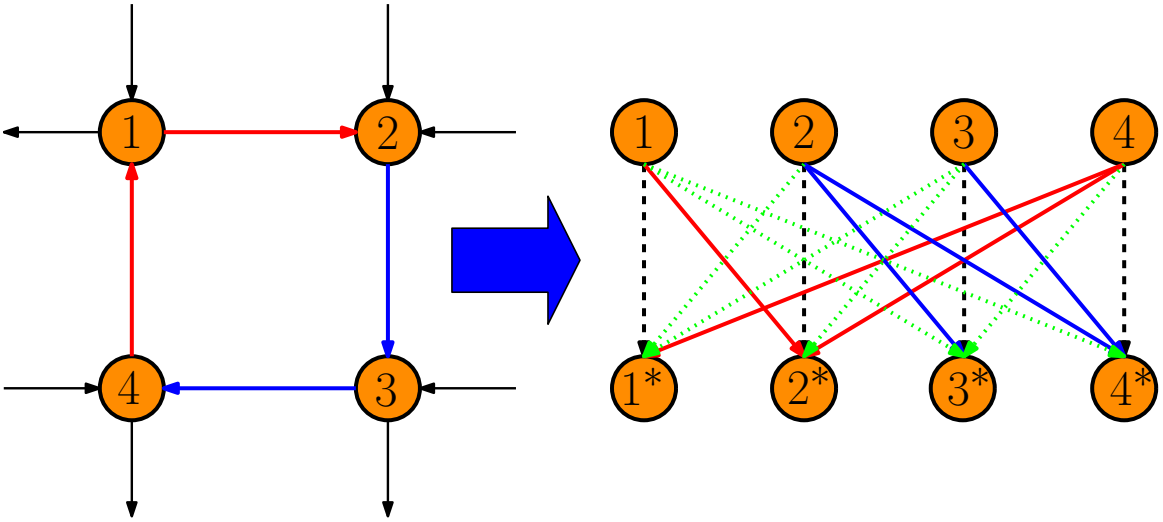


Figure 3.4: A part of G eligible to become a SCC C_i on the left. Nodes 1 to 4 form a SCC. On the right we see the resulting bipartite graph B_i. A dashed line indicates an edge with no label set, because the end points of that edge represent the same vertex in C_i. A colored edge indicates a connection between two portals in C_i that are connected through edges with only that color. A dotted green edge indicates the portals are connected through a combination of red and blue edges. We see that the new number of edges is much higher than in the original.

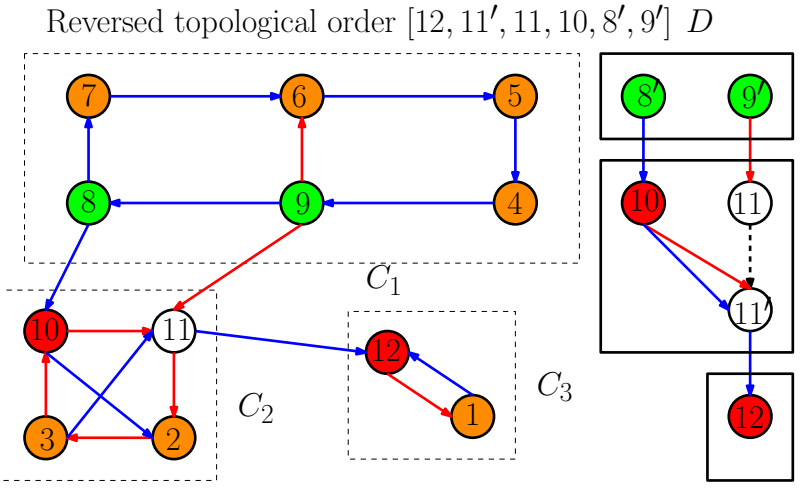


Figure 3.5: On the left a graph with SCC's C_1, C_2 and C_3, which could also be represented by their related bipartite graphs. The edge labels are indicated by the color. A dashed line indicates an edge with no label, which happens when two nodes in D represent the same vertex in G. Inner vertices are orange, out-portal are green and in-portals are red and nodes serving as both are white. The internal vertices are numbered. On the right the graph D holding all bipartite components. Changes are propagated upwards along all edges in the graph where the transitive closure of a node is concatenated with the edge label.

Step 3: Finalizing, inner to out

Now we have only covered the vertices $v \in D \subseteq V$, i.e we have not covered the internal vertices of each SCC. To get $M_G(u, -)$ for all nodes $u \in V$ we need to look to the in- and out-portals of each SCC respectively.

The inner vertices are processed in an order PT that adheres to RT, i.e. each internal vertex $v_i \in C_i$ is processed after all internal vertices $v_j \in C_j$ have been processed if the in-portals of B_j were processed before B_i in RT. For each inner vertex $v \in PT$ and out-portal $p_o ut$ in SCC C_i we set $M_G(v, -)$ to $\bigcup_{p_o ut \in outp_{C_i}} (\mathsf{Prune}(M_{C_i}(v, p_o ut) \odot M_G(p_o ut, -))$, where $outp_{C_i}$ is the set of out-portals of C_i and $M_{C_i}(v, p_o ut)$ is the minimal set of label sets that connects v and $p_o ut$.

The paper claims that the running time of the algorithm is $O(\max |V|^3, |V|^2 \cdot D^d)$.

Optimization

The paper has optimization to this algorithm which improves the results dramatically. For instance, for a 100k graph the optimization could reduce the index construction time from $5,000$ seconds to about 15 seconds. It tries to optimize the computation of $M_{C_i}(u, -)$. Some small adaptations to the algorithm to compute the transitive closure are made.

First, we create an ordering amongst the nodes. We look for the node in C_i with the highest in-degree u. Next, we look at u's in-neighbours. We iteratively expand by adding in-neighbours until we have hit all nodes.

Next, for each node in the ordering u we compute $M_{C_i}(u, -)$ by using the Dijkstra-like approach discussed in Section 3.2.2. But first we do the following: for each out-neighbour of u v we set $M_{C_i}(u, -)$ to $\{\lambda(u, v) \odot M_{C_i}(v, -)\}$. Next, we fill the heap H with the neighbour-tuples of u as usual. If during the course of the algorithm, a path $P = \langle u, \ldots, w \rangle$ is found and P is already covered by an entry in $M_{C_i}(u, -)$ then we do not continue. Basically, we use the already computed transitive closure to prune.

Zou results

The code was written in C++ and the experiments were conducted on a P4 3.0Ghz machine 2 GB RAM running Linux Ubuntu.

A number of synthetic datasets were generated using either the Erdos-Renyi (ER) or the Scale-Free (SF) model. ER is a classical random graph. $|E|$ edges are chosen randomly from $|V|(|V| - 1)$ possible edges. Each edge in this model is equally likely which may make it not comparable to real-life networks. SF graphs try to model preferential attachment', i.e. a node with a lot of connections (in and/or out) may be more likely to be connected to or from. Hence, we get a few nodes with a high number of nodes. The degree distribution of such a graph follows the shape of an exponential distribution and can be specified by the following parameters $|V|$, the minimal degree of a node, the maximal degree of a node and γ [1]. Typically, $2 \leq \gamma \leq 3$.

In the datasets of which we discuss the results here, $|\mathcal{L}| = 18$ and the query workload was around 30% of this. This means that each query (s, t, L) has $|L| = 6$. The label distribution is uniform.

In Figure 3.6, we can see the results for ER-graphs of different sizes. $|d| = 1.5$ meaning that the average degree per node was 1.5. We see that the optimized version of the *transitive-closure method* performs much better. From the paper we know the query times obtained by this method are roughly equal to the query times obtained by doing a double-sided BFS, i.e. a BFS with two threads. In Figure 3.7 we see the effects of increasing the density in ER graphs. The number of vertices has been set to $|V| = 10,000$.

In Figure 3.8 we see the results for SF-graphs. There is a major difference here compared to 3.6. The paper argues that most nodes in these graphs have a very low degree and hence much of the search space can be pruned early on.

One curious thing to note here is that the index size of all graphs is fairly low. Most do not exceed a megabyte. In our experiments (see Table 6.2) almost any index, even for 1k graphs, would exceed that much. In Figure 3.6 we see that a ER-graph with 10k vertices and a degree of 1.5 (i.e. 15k edges) does create an index of 33MB. The 10k ER-graphs in Figure 3.7 with a higher degree yield much smaller indices, e.g. for $d = 5$ we see a 186KB index. We find this very strange.

Looking at Figure 3.7 we also find it strange that the optimized versions time is that much lower as well. We see a 300 to 400 speed-up. We have implemented a similar optimization (see Section 5.2.2) strategy and have not seen this incredible speed-up.

[1] http://tinyurl.com/zzt8qgq

Table 4
Performance VS. $|V|$ in ER Graphs.

$\|V\|$	**Transitive closure method**			
$d=1.5$	IT (s)	IT-opt (s)	IS (KB)	QT (ms)
1K	15	3	415	0.01.
2K	23	5	1396	0.01
4K	217	7	4920	0.02
6K	623	10	9000	0.03
8K	964	12	13,200	0.03
10K	5065	15	33,000	0.04

Figure 3.6: Table 4 from paper [10]. For different sizes of ER-graphs it shows the time to build the graph using the *transitive-closure method* and the optimized version of it, the size in KB and the query time in ms.

Table 5
Performance VS. Density in ER Graphs.

Degree	**Transitive closure method**			
d	IT (s)	IT-opt (s)	IS (KB)	QT (ms)
2	6890	20	26.3	0.08.
3	11,112	23	102.3	0.09
4	25,347	35	160	0.12
5	33,169	80	186	0.23

Figure 3.7: Table 5 from paper [10]. For different densities of 10k ER-graphs it shows the time to build the graph using the *transitive-closure method* and the optimized version of it, the size in KB and the query time in ms. Interestingly the index size is very small.

Table 7
Performance VS. $|V|$ in SF graphs.

$\|V\|$	**Transitive closure method**				**Sampling-tree method**			Bi-directional search
$d=1.5$	IT (s)	IT-opt (s)	IS (KB)	QT (ms)	IT (s)	IS (KB)	QT (ms)	QT (ms)
1K	0.1	0.1	43	0.01.	0.6	1576	0.16	0.01
2K	0.2	0.1	94	0.01	2.6	2583	0.17	0.02
4K	0.2	0.1	140	0.01	9.5	4870	0.18	0.02
6K	0.4	0.2	289	0.01	27.8	8854	0.20	0.03
8K	0.4	0.2	390	0.01	45.5	11393	0.22	0.03
10K	1.1	0.3	492	0.01	80.4	23931	0.24	0.03

Figure 3.8: Table 7 from paper [10]. For different sizes of SF-graphs it shows the time to build the graph using the *transitive-closure method* and the optimized version of it, the size in KB and the query time in ms.

3.2.3 Fletcher and Yoshida

There are two possible algorithms taken from unpublished notes of Fletcher and Yoshida [4] for determining reachability in labelled graphs. One algorithm DoubleBFS performs two breadth-first searches (BFS'es) from each node $v \in V$. The other algorithm is an iteration-based algorithm in which nodes send messages to their neighbours.

DoubleBFS and NeighbourExchange

In this section we describe two methods (DoubleBFS and NeighbourExchange) that can be used for building a full exact index.

Both algorithms try to create two arrays $\mathsf{Out}(v)$, $\mathsf{In}(v)$ similar to 2-hop cover. We define $\mathsf{In}(v) = \{(s, L) \mid \exists[P \in P_{\min}(v, s) \mid \mathsf{Labels}(P) = L]\}$ and $\mathsf{Out}(v) = \{(s, L) \mid \exists[P \in P_{\min}(s, v) \mid \mathsf{Labels}(P) = L]\}$. A similar idea as in the 2-hop cover framework applies here. The formula to determine reachability query $q = (s, t, L)$ in the labelled-graph is:

$$Q(s, t, L) = \exists[v \in V \mid \exists[L_s, L_t \subseteq \mathcal{L} \mid (v, L_s) \in \mathsf{Out}(s) \wedge (v, L_t) \in \mathsf{In}(t) \wedge L_s \subseteq L \wedge \subseteq L_t \subseteq L]].$$

Some ordering of the vertices is created. Each node $v \in V$ gets a unique number in the range $v_1, \ldots, v_n$. This ordering has little or no effect on the algorithms discussed here, but it might have in more sophisticated versions of these algorithms later on.

The first algorithm loops over all n vertices in the ordering that has been created. From each v_i, it starts two Breadth-First searches: one using the direction of the edges and one using the opposite direction. Each time we hit a node (v, L) we try to add it to $\mathrm{Out}(v)$ or $\mathrm{In}(v)$ respectively.

The second algorithm NeighbourExchange keeps looping as long as there is a node v which had a change in its $\mathrm{In}(v)$ or $\mathrm{Out}(v)$. During each such iteration we loop over all v_i with $1 \leq i \leq n$. Each v_i first processes its updates and then propagates the update to all its out-neighbours w appending the edge label l between v_i and w.

Both algorithms use a form of pruning. As soon as an entry (u, L) is added to either $\mathsf{In}(v_i)$ or $\mathsf{Out}(v_i)$ and we have that $Q(s, t, L)$ is already true, we do the following. In the case of DoubleBFS we do not push any of u's neighbours on the queue. In the case of NeighbourExchange we do not push the update (u, L) to any of v_i's neighbours.

In Figure 3.9 and Table 3.2, we can see how algorithm NeighbourExchange works. In the table we see the messages that are exchanged during the first round and how that effects the In-value of each node.

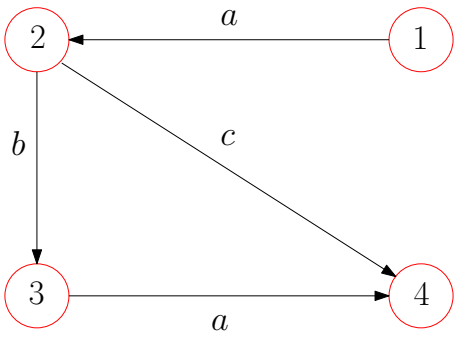


Figure 3.9: A graph with labels. In Table 3.2 we see the updates during the first round of the algorithm NeighbourExchange.

Both algorithms can prune by not storing or not continuing a search when we add a label (v_i, L) to $\mathrm{In}(v)$ or $\mathrm{Out}(v)$, when there exists a pair $(v_i, L') \in \mathrm{In}(v) \vee (v_i, L') \in \mathrm{Out}(v)$ s.t. $L' \subseteq L$.

While working on the code for DoubleBFS and NeighbourExchange and running experiments with it, we came along some issues.

Firstly, NeighbourExchange and DoubleBFS both build In and Out, but having one of these suffices to answer any query directly. This saves up memory and construction time. Secondly, NeighbourExchange and

Table 3.2: This table shows the first and only round of running NeighbourExchange on the graph in Figure 3.9. A node first processes its updates and then sends new updates to its out-neighbours for each change that was made.

Vertex-id	Receive	Send
1	$\emptyset$	$(1, 2, \{a\})$
2	$(1, \{a\})$	$(1, 3, \{a, b\}), (2, 3, \{b\}, (2, 1\{a, c\}), (2, 4\{c\})$
3	$(1, \{a, b\}), (2, \{b\})$	$(1, 4, \{a, b\}), (2, 4, \{a, b\})$
4	$(1, \{a, c\}), (2, \{c\}), (2, \{a, b\}), (3, \{a\})$	$\emptyset$

DoubleBFS do a lot of redundant computation. Suppose we have a graph like in Figure 3.10 in which the black-box in the middle could be any kind of graph which makes the red nodes reachable from the blue nodes. For each blue node we are redoing a BFS on the red nodes. We might as well compute $\mathsf{Out}(A) = \{B, C, D\}$ and then re-use this result any time we hit vertex A.

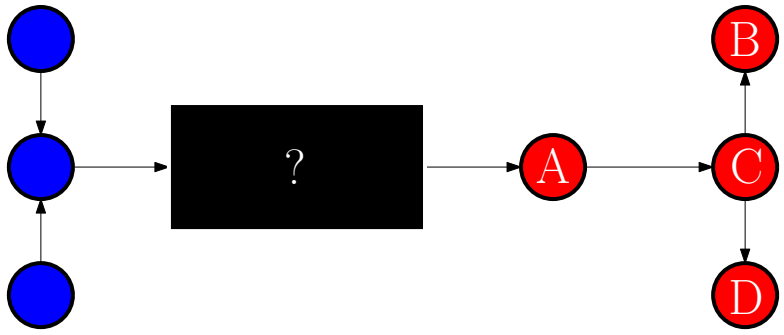


Figure 3.10: A graph. The black box could be any graph which makes the red nodes reachable from the blue nodes.

Managing the index size: union or intersection

The index size could become a real bottleneck for larger $|V|$ when building a full exact index. A solution suggested to remedy this by Fletcher and Yoshida [4] is to not add all entries to the index, but instead merge the label sets of some of the entries. Suppose $(w, \{\{a, b\}), (w, \{b, c\}), (w, \{c, d, e\}\}) \in \text{In}(v)$. We could try to reduce the size of the index by intersecting the label sets or taking the union of the label sets. We could do a union for w which would yield $(w, \{a, b, c, d, e\}$ or an intersection which is $(w, \{b\})$. A query $(v, w, \{b, d\}$ would yield false for a union and true for an intersection, whereas we know the query is false any way. On the other hand, a query $(v, w, \{b, c\}$ would be answered false by union and true by intersection, whereas the query is true any way. A union introduces false negatives and an intersection introduces false positives.

For both union and intersection we can delay merging entries until after a certain point. Let's call this point a budget b. Suppose $b = 2$ and we have $\text{Out}(v, s) = \{\{b, c\}, \{b, d\}\}$ for $v, s \in V$ and we were to add $\{b, e\}$ to $\text{Out}(v)$. This would exceed the budget and yield $\text{Out}(v, s) = \{\{b, c, d, e\}\}$. An index $I_{Union,l}$ for union consists of two lists $\text{In}(v)$ and $\text{Out}(v)$ for each $v \in V$ and a budget l s.t. $\forall[v \in V| \ \forall t \in V[|\text{In}(v, t)| \leq l \wedge |\text{Out}(v, t)| \leq l]]$.

A query (v, w, L) can be answered in the following way. We look for a node u between v and w and check whether $(v, u, L) \wedge (u, w, L)$ holds. If only (v, u, L) holds, we can recursively repeat the same process for (u, w, L)

ignoring v in any subsequent recursive call.

Chapter 4

Experimental design

In this chapter we describe the datasets and hardware that have been used in our experiments, as well as the way in which we generated queries for the experiments.

4.1 Datasets

For our experiments we have used both synthetic and real data. In Table 4.1 a full summary of all datasets can be found. In total 68 datasets have been used.

During our experiments we found that the connectivity of a graph was a key determinant of the construction time and index size. We define connectivity as the ratio between the number of nodes in the largest strongly connected component (SCC) and the total number of nodes and the number of triangles in the graph. One can imagine that if s and t are in a large strongly connected component with a lot of triangles that there are many paths which need to be explored by for example DoubleBFS .

4.1.1 Synthetic datasets

We generated the graphs using SNAP [7, 6] using either the preferential attachment model (pa) the Erdos-Renyi model (er), the forest fire model (ff) or the power law model (pl). In case a model (pa, er) would yield an undirected graph the direction would be uniformly randomly chosen. The edge labels can be either uniformly, normally ($\mu = |\mathcal{L}|/2, \sigma = |\mathcal{L}|/4$) or exponentially ($\lambda = \frac{|\mathcal{L}|}{1.7}$) distributed. In the case of an exponential label distribution with $|\mathcal{L}| = 8$ roughly 60% of the labels have the same value.

The datasets are distinguishable according to five measures: the number of vertices (1k, 5k, 125k or 625k), the average degree per node (2, 5 or 10), the number of labels (8 or 12), the model (pa/PA, ff/FF, pl/PL, er/ER) and the label distribution (or label set distribution).

4.1.2 Real datasets

The real datasets are taken from SNAP datasets [6] and KONECT [1]. There can be made two distinctions between real datasets: those which already had 'labels' (r2) or those to which labels were synthetically added (r1). In the latter case, this is always an exponential distribution with 8 labels. In Figure 4.1, we can see the label distribution of some r2-datasets. Datasets that contained self-loops had these removed.

- **Robots** has been taken from Trustlet [2] and was published on June 2014. It is a trust-network where user A can give a certain level of trust to user B by using a certain type of edge.
- **Advogato** has been taken from KONECT and was published on April 2016. Advogato is an online community platform for developers of free software launched in 1999. Nodes are users of Advogato and the directed edges represent trust relationships. A trust link is called a "certification" on Advogato, and three different levels of certifications are possible on Advogato, corresponding to three different edge labels.

[1] http://konect.uni-koblenz.de/networks/
[2] http://tinyurl.com/gnexfoy

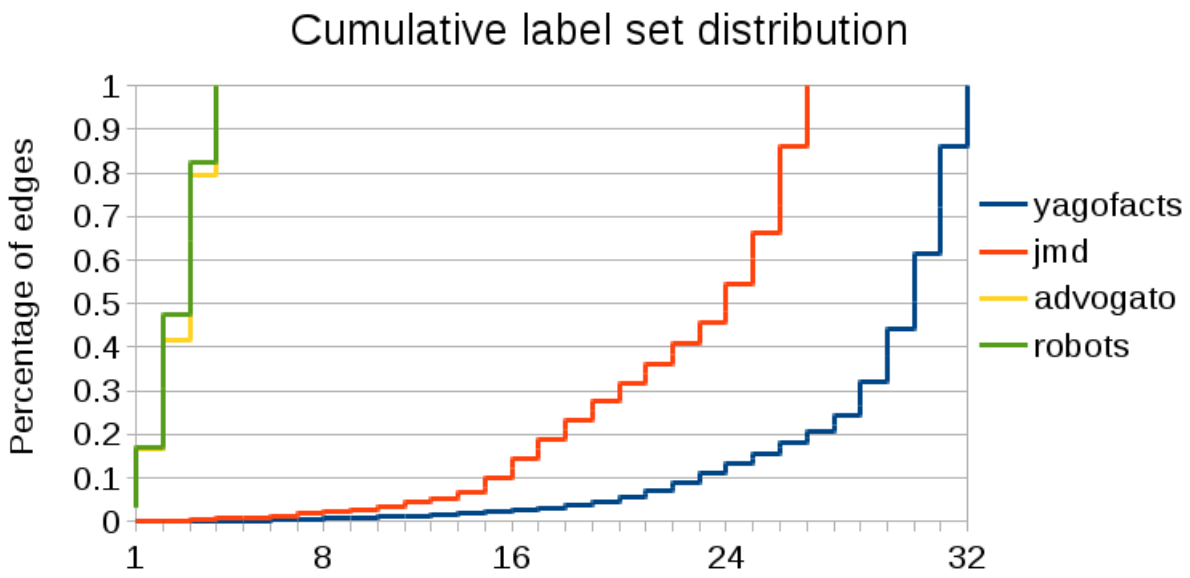


Figure 4.1: A cumulative distribution of the label set for the real datasets with an already existing label set distribution, i.e. the label set distribution was not synthetically added. **Robots** and **Advogato** have $|\mathcal{L}| = 4$, **jmd** has $|\mathcal{L}| = 25$ and **yagofacts** has $|\mathcal{L}| = 32$.

- **yagoFacts-small** is a sample taken from Max-Planck Institut [3]. YAGO is a huge semantic knowledge base, derived from Wikipedia WordNet and GeoNames. Currently, YAGO has knowledge of more than 10 million entities (like persons, organizations, cities, etc.) and contains more than 120 million facts about these entities.
- **jmd** is a sample taken from the RDF-dump of Jamendo. Jamendo is a large repository of Creative Commons licensed music, based in France.
- **subeljCoraL8** has been taken from KONECT and was put online on April 2016. Nodes represent scientific papers. An edge between two nodes indicates that the left node cites the right node.
- **arXivheppL8exp** has been taken from KONECT and was put online on April 2016. It is the network of publications in the arXiv's High Energy Physics Phenomenology (hep-ph) section.
- **p2p-GnutellaL8exp** has been taken from KONECT. It's a network of Gnutella hosts from 2002. The nodes represent Gnutella hosts, and the directed edges represent connections between them. The dataset is from August 31, 2002.
- **socSlahdot0902L8exp** has been taken from SNAP. The website features user-submitted and editor-evaluated current primarily technology oriented news. The network contains friend/foe links between the users of Slashdot. The network was obtained in February 2009.
- **soc-sign-epinionsL8** has been taken from SNAP. It is a who-trust-whom online social network of a a general consumer review site Epinions.com. Members of the site can decide whether to trust each other. The date it was obtained, is not given. We downloaded it on the 14th of April, 2016.
- **NotreDameL8exp** has been taken from KONECT. It is the directed network of hyperlinks between the web pages from the website of the University of Notre Dame. The date it was obtained, is not given. We downloaded it on the 14th of April, 2016.
- **webGoogle** has been taken from KONECT. This is a network of web pages connected by hyperlinks. The data was released in 2002 by Google as a part of the Google Programming Contest.

[3] http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/

- **webBerkStan** has been taken from KONECT. This is the hyperlink network of the websites of the Universities in Berkley and Stanford. Nodes represent web pages, and directed edges represent hyperlinks.
- **webStanford** has been taken from KONECT. This is the directed network of hyperlinks between the web pages from the website of the Stanford University.
- **socPokecRelationShipsL8exp** has been taken from KONECT. This is the friendship network from the Slovak social network Pokec. Nodes are users of Pokec and directed edges represent friendships.
- **zhishihudongL8exp** has been taken from KONECT. These are "related to" links between articles of the Chinese online encyclopedia Hudong (, `http://www.hudong.com/`).
- **usPatentsL8exp** has been taken from KONECT. This is the citation network of patents registered with the United States Patent and Trademark Office. Each node is a patent, and a directed edge represents a patent and an edge represents a citation.

4.1.3 Summary of datasets

Table 4.1 shows a full overview of all the datasets that were used in at least one experiment.

Table 4.1: A listing of all datasets with statistics. The model is one of the following: er (Erdos-Renyi), ff (Forest-Fire), pl (PowerLaw), pa (Preferential Attachment), r1 (real data, but synthetically added labels) and r2 (real data, real labels). max SCC % indicates the percentage of nodes that are in the largest strongly connected component (SCC). $\#\delta$ is the number of triangles in the graph, i.e. $u, v, w \in V$ s.t. $(u, v), (v, w), (w, u) \in E$. In the model column, pa stands for 'preferential attachment', pl for 'powerlaw' and ff for 'forest fire', whereas none indicates the dataset is a real one.

name	$\|V\|$	$\|E\|$	$\|L\|$	model	max SCC %	$\#\delta$	dia
ERV1kD2L8uni	1,000	2,000	8	er	0.63	0	18
ERV1kD5L8uni	1,000	5,000	8	er	0.98	35	9
ff1k-0.2-0.4	1,000	2,552	8	ff	0.48	405	15
pl-1kL8a2.0exp	1,000	2,306	8	pl	0.42	200	10
plV1kL8a2.0exp	1,000	2,306	8	pl	0.41	195	8
V1kD2L12exp	1,000	1,997	12	pa	0.56	12	13
V1kD2L8exp	1,000	1,997	8	pa	0.61	15	16
V1kD2L8norm	1,000	1,997	8	pa	0.54	13	13
V1kD2L8uni	1,000	1,997	8	pa	0.55	16	13
V1kD5L8exp	1,000	4,985	8	pa	0.97	184	8
V1kD5L8norm	1,000	4,985	8	pa	0.97	212	8
V1kD5L8uni	1,000	4,985	8	pa	0.97	198	7
robots	1,484	2,960	4	r1	0.14	37	10
ERV5kD2L8uni	5,000	10,000	8	er	0.63	1	27
ff5k-0.2-0.4	5,000	12,718	8	ff	0.43	2,004	17
pl-5kL8a2.0exp	5,000	15,634	8	pl	0.44	2,246	10
plV5kL8a2.0exp	5,000	15,634	8	pl	0.44	2,237	10
V5kD10L8exp	5,000	49,945	8	pa	0.99	2,161	7
V5kD2L12exp	5,000	9,997	12	pa	0.55	31	16
V5kD2L8exp	5,000	9,997	8	pa	0.56	30	19
V5kD2L8norm	5,000	9,997	8	pa	0.56	27	20
V5kD2L8uni	5,000	9,997	8	pa	0.55	28	18
V5kD5L8exp	5,000	24,985	8	pa	0.97	406	9
advogato	5,417	51,327	4	r1	0.59	4,175	10
yagoFacts-small	10,000	22,122	32	r1	0.00	34	13
subeljCoraL8exp	23,167	91,500	8	r2	0.17	793	41

Table 4.1: A listing of all datasets with statistics. The model is one of the following: er (Erdos-Renyi), ff (Forest-Fire), pl (PowerLaw), pa (Preferential Attachment), r1 (real data, but synthetically added labels) and r2 (real data, real labels). max SCC % indicates the percentage of nodes that are in the largest strongly connected component (SCC). $\#\delta$ is the number of triangles in the graph, i.e. $u, v, w \in V$ s.t. $(u, v), (v, w), (w, u) \in E$. In the model column, pa stands for 'preferential attachment', pl for 'powerlaw' and ff for 'forest fire', whereas none indicates the dataset is a real one.

name	$\|V\|$	$\|E\|$	$\|L\|$	model	max SCC %	$\#\delta$	dia
ERV25kD2L8uni	25,000	50,000	8	er	0.63	3	31
plV25L8ka2.0exp	25,000	90,449	8	pl	0.45	14,086	11
V25kD2L12exp	25,000	49,997	12	pa	0.55	43	23
V25kD2L8exp	25,000	49,997	8	pa	0.55	47	23
V25kD2L8norm	25,000	49,997	8	pa	0.55	52	24
V25kD2L8uni	25,000	49,997	8	pa	0.56	40	22
V25kD3L10exp	25,000	74,994	10	pa	0.84	154	15
V25kD3L12exp	25,000	74,994	12	pa	0.83	160	16
V25kD3L14exp	25,000	74,994	14	pa	0.83	155	16
V25kD3L16exp	25,000	74,994	16	pa	0.84	152	16
V25kD3L8exp	25,000	74,994	8	pa	0.83	159	17
V25kD4L10exp	25,000	99,990	10	pa	0.93	376	13
V25kD4L12exp	25,000	99,990	12	pa	0.93	421	14
V25kD4L14exp	25,000	99,990	14	pa	0.93	373	13
V25kD4L16exp	25,000	99,990	16	pa	0.93	388	13
V25kD4L8exp	25,000	99,990	8	pa	0.93	357	13
V25kD5L10exp	25,000	124,985	10	pa	0.97	666	11
V25kD5L12exp	25,000	124,985	12	pa	0.97	669	12
V25kD5L14exp	25,000	124,985	14	pa	0.97	653	12
V25kD5L16exp	25,000	124,985	16	pa	0.97	655	12
V25kD5L8exp	25,000	124,985	8	pa	0.97	645	11
arXivhepphL8exp	34,547	421,534	8	r2	0.36	248	45
p2p-GnutellaL8exp	62,587	147,892	8	r2	0.22	56	28
socSlashdot0902L8exp	82,168	870,161	8	r2	0.86	12,296	12
ERV125kD2L8uni	125,000	250,000	8	er	0.63	0	37
plV125L8ka2.0exp	125,000	518,207	8	pl	0.46	91,482	11
V125kD2L12exp	125,000	249,997	12	pa	0.56	71	24
V125kD2L8exp	125,000	249,997	8	pa	0.56	69	24
V125kD2L8norm	125,000	249,997	8	pa	0.56	81	27
V125kD2L8uni	125,000	249,997	8	pa	0.56	66	27
V125kD5L8exp	125,000	624,985	8	pa	0.97	994	14
soc-sign-epinionsL8exp	131,828	840,799	8	r2	0.31	73,928	17
webStanfordL8exp	281,904	2,312,497	8	r2	0.53	94,008	591
NotreDameL8exp	325,730	1,469,679	8	r2	0.16	52,100	72
citeseerL8exp	384,414	1,751,463	8	r2	0.04	1,566	71
twitterL8exp	465,018	834,797	8	r2	0.00	54	15
jmd	486,320	1,049,647	26	r1	0.00	0	5
V625kD5L8exp	625,000	3,124,985	8	pa	0.99	1894	16
plV625ka2.0L8exp	625,000	2,897,916	8	pa	0.97	350,125	14
webBerkstanL8exp	685,231	7,600,595	8	r2	0.48	318,370	670
webGoogleL8exp	875,713	5,105,039	8	r2	0.97	306,723	24
socPokecRelationshipsL8exp	1,632,803	30,622,534	8	r2	1	653,121	14
zhishihudongL8exp	2,452,715	18,854,882	8	r2	0.98	2,004,783	108
usPatentsL8exp	3,774,769	16,518,947	8	r2	0.00	0	23

4.2 Queries

We strived to generate a 'realistic query workload'. By this we meant two things. Queries $q = (v, w, L)$ should not have a L-path P from v to w where $\#P$ is very low, because this favours BFS over other approaches, e.g. LI . We do not wish that a large part of our queries has the same starting point, as this might favour for instance LI .

For each dataset, both synthetic and real, we generated three query sets. The number of labels in a query varies between the three query sets and is: $\lfloor |\mathcal{L}|/4 \rfloor$, $\lfloor |\mathcal{L}|/2 \rfloor$ and $|\mathcal{L}| - 2$ for the datasets when $8 \leq |\mathcal{L}|$ and 1, 2 and 3 otherwise. Each query set consists of 200 (for graphs with $|E| < 5,000$) or 2000 queries (otherwise). The first half of these queries should be answered with True and the second half should be answered with False.

For each query we took into account the difficulty. The difficulty of a query $q = (s, t, L)$ can be defined in the following way: the number of vertices visited during our implementation of BFS from s to t before reaching a conclusion, i.e. returning false or true. Our queue is a FIFO-queue and all edges for a given $v \in V$ of the form (v, w, l) are sorted in ascending order according to the vertex id of w. This difficulty can be influenced by five main factors: the distance between s and t in the graph, the number of labels in L, whether the result of the query is true or false, whether it is cyclic or whether it has a skewed out-degree distribution. False queries with a large label set may explore much larger sections of the graph than true queries with the same label set. Similarly a graph with a very skewed out-degree distribution has a low diameter and hence has more difficulty generating difficult queries. One can see that if we were to omit this requirement and the difficulty of all queries in a dataset would be for instance fairly low, it could benefit BFS's performance compared to that of any index.

Queries for a particular query set are generated in the following way over a set of rounds. Each round starts by choosing a random vertex $v \in V$ and a random minimal difficulty $d_{\min}$. The minimal difficulty is between $\lceil \log_2(N) \rceil$ and $N/10$ (for $N \leq 500,000$) and between $\lceil \log_2(N) + 10 \rceil$ and $N/100$ otherwise. For each other vertex w we generate up to $k' = nq/100$ random label sets L, where nq is the desired number of True- or False-queries. The labels in the label set are chosen uniformly random. Then it is tested whether (v, w, L) is True or False using BFS and how many vertices d where visited during BFS . If $d \geq d_{\min}$, we add the query to the True- or False-set depending on it's outcome until we have nq queries of both types. The parameters k' and $d_{\min}$ can respectively increase and decrease by 1 for any 100 rounds passed.

The parameters mentioned are chosen to in one sense ensure that the queries either have no or a L-path of sufficient length and that a small part of the queries have the same starting vertex $v \in V$ and in another sense to ensure that for any dataset we were able to generate queries. For some datasets, particularly acyclic datasets like **jmd**, it is hard to generate difficult queries. This is why we allowed the parameters k' and $d_{\min}$ to scale with the number of rounds. For a similar reason we chose to set the maximal difficulty to $N/100$ for $N \geq 500,000$, because the difficulty does not grow linearly with the number of nodes, at least for the True-queries.

4.3 Hardware

We used a server on the National Institute of Informatics (NII) in Tokyo, Japan. The server has 258GB of memory and a 2.9Ghz 32-core processor. However, we did not let any of our experiments exceed an index size of 128GB and we set a 6 hour time limit. The experiments were all single-threaded. The OS is Linux Ubuntu.

4.4 Methods

Table 4.2 describes all methods used in at least one experiment with an abbreviation.

Table 4.2: Summary of all used methods with an abbreviation, description and main parameters.

method	abbreviation	description	parameters
BFS	BFS	Regular BFS, no indexing.	none
Zou	Zou	Zou-method, exact index.	none
ClusteredExact	Clus	Clustered-method, exact index.	K: number of clusters
NeighbourExchange	Nei	Neighbour-exchange method, exact index.	none
PartialIndex	Par	Partial-index, a partial index.	b: budget per node
LI	L	Landmarked, exact index, no extension.	k: number of landmarks (always required)
LI+EXTv1	L2	Landmarked second extension.	b: budget per non-landmark node
LI+OTH+EXTv1	L12	Landmarked first and second extension.	b: budget per non-landmark node
LI+OTH+EXTv2	L23	Landmarked second and third extension.	b: budget per non-landmark node.
DoubleBFS	DBFS	Landmarked where $k = N$.	None.

Chapter 5

Methods

In this chapter we discuss the methods used in at least one experiment or that we have at least strongly considered using (JOINDEX). Also we discuss index maintenance and extensions to other queries besides LCR-queries. For our major contribution (LI) we give a proof and a worst-case running time and memory usage analysis.

5.1 Existing methods

This section discusses existing methods to answer LCR-queries.

5.1.1 BFS

BFS is the baseline idea. It has no index. The time to construct is the 'index' is equal the time to load the graph into memory. The same holds for the size of the 'index'. Updates are as simple as adding or removing a node or vertex from the graph. In many cases BFS is a good solution, as it is relatively fast, requires no construction and maintenance and has a low memory usage.

Algorithm 2 shows the precise implementation for a query (s, t, l). Just like normal BFS, this BFS has a running time complexity of $O(|V| + |E|)$.

Algorithm 2 BFS(s, t, L)

```
1:  Let marked : V ⟶ {0, 1} be a mapping
2:  Let q be a queue
3:  q.push(s)
4:  while q is not empty do
5:      v ← q.pop()
6:      marked[v] ← 1
7:      if v = t then
8:          return True
9:      end if
10:     for (v, w, l) ∈ E ∧ marked[w] = 0 do
11:         if l ∈ L then
12:             q.push(w)
13:         end if
14:     end for
15: end while
16: return False
```

The variable marked at line **1** has been implemented using a bitset, which makes it possible to modify and check for any value in constant time rather than in $O(\log(N))$ at the expense of $O(N)$ memory usage.

5.1.2 Best effort Zou

This method is a best effort approach to implement Zou's algorithm [10]. Let G be a labelled directed graph. Let $M_G(u,-) \subseteq V \times \mathcal{P}$. This is the same as $\mathsf{Ind}(u)$ in our implementation and hence one might as well replace $M_G(u,-)$ and $M_{C_i}(u,-)$ by $\mathsf{Ind}(u)$. The operation $\mathsf{Prune}(M_G(u,v) \odot M_G(v,-))$ is implemented by looking at any $(v,L) \in \mathsf{Ind}(u)$ and concatenating (v,L) to any $(w,L') \in \mathsf{Ind}(v)$ which results in $(w, L \cup L')$. $(w, L \cup L')$ is attempted to be added to $\mathsf{Ind}(u)$ by using tryInsert (Algorithm 11, Section 5.2.2).

The algorithm consists of a number of steps:

1. Generate a set of SCC's $C^* = \{C_1, \ldots, C_n\}$, e.g. by using Tarjan's algorithm for G.

2. For each $C_i \in C^*$ we build an index $M_{C_i} = \bigcup_{u \in C_i}(M_{C_i}(u,-))$ answering all queries local to C_i. In our code $M_{C_i}(u,-)$ is represented by $\mathsf{Ind}(u)$. After this step, for any $u, v \in C_i$ with a L-path from u to v we have that there exists an entry (v, L') in $M_{C_i}(u,-)$ with $L' \subseteq L$.

3. We find the in- and out-portals of each SCC C_i, i.e. we find those vertices $v \in C_i$ such that there exists an incoming edge $(w,v) \in E$ or an outgoing edge $(v,w) \in E$ where $w \notin C_i$. We define these sets as $IN(C_i)$ and $OUT(C_i)$ respectively.

4. We generate an acyclic graph $D = (V_D, E_D)$. The edges $e \in E_D$ have label sets rather than labels. The vertices $v \in V_D$ are in- and out-portals of a SCC C_i in G.

   For each out-portal $v \in OUT(C_i)$ we generate a replica node v'. In case v is also an in-portal, there is an edge: $(v, v', \{\emptyset\})$. v' serves as the out-portal if an edge $(v,w) \in E$ exists and v serves as an in-portal if an edge $(w,v) \in E$ exists. For each pair of in- and out-portals per C_i (v,w) we find all minimal label sets L that connect v and w. These are found using M_{C_i}. Each (v,w,L) is added to D.

5. D is acyclic and can be topologically sorted. This is done, after which we reverse the order to create an order RT.

6. The index is propagated upwards according to the topological order for all vertices in D. For $u \in V_D$ and all children $v \in V_D$ of u we set $M_G(u,-)$ to $\mathsf{Prune}(\bigcup_{(u,v) \in E_D}(\mathsf{Label}(u,v) \odot M_{C_i}(v,-)))$. This puts any entry $(w, L_1 \cup L_2)$ in $M_G(u,-)$, if there is an entry (v, L_1) in $M_G(u,-)$ and an entry (w, L_2) in $M_G(v,-)$.

7. For each out-portal u_i that is part of C_i we look for all inner vertices u_j of any C_j with $j \neq i$ and set $M_G(u_i,-) = \mathsf{Prune}(M_G(u_i,u_j) \odot M_G(u_j,-))$. In this way the out-portal 'knows' about the inner vertices of other SCC's as well. This step was not in the original paper, but necessary to get our implementation fully working in the end.

8. For each inner vertex u_i in C_i we look at all out-portals p_i of C_i and set $M_G(u_i,-)$ to the union of all out-portals, i.e. we set $M_G(u_i,-)$ to $\bigcup p_i \in OUT(C_i)(\mathsf{Prune}(M_G(u_i,p_i) \odot M_G(p_i,-)))$. This puts any entry $(u_j, L_1 \cup L_2)$ in $M_G(u_i,-)$, if there is an entry (p_i, L_1) in $M_G(u_i,p_i)$ and an entry (u_j, L_2) in $M_G(p_i,u_j)$.

9. For each inner vertex u_j with $j \neq i$ we look for an in-portal p_i such that $u_j \rightsquigarrow p_i$ and any other vertex $u_i \neq p_i$. We set $M_G(u_j,-)$ to $\mathsf{Prune}(M_G(u_j,p_i) \odot (M_G(p_i,u_i)) \cup \mathsf{Prune}(M_G(u_j,-))$.

Step 7 was not in the original paper, but we could not get the idea working without it. It seemed very logical to us that an out-portal needs to know about any vertex it can reach, after which any inner vertex can copy all the entries from that same out-portal. Apparently this had not been fully done after step 6.

We decided not to implement the optimization in the end or eliminate step 7 from the algorithm, because the results were already quite disappointing, i.e. they were not competitive to e.g. DoubleBFS and not in line with the results in the original paper. The difference between DoubleBFS and Zou was at least a factor 10. We did not believe this gap could be bridged. Steps 2, 7, 8 and 9 took the most time. Chapter 6 elaborates on Zou with experimental results as well.

We included the idea of step 2 of the algorithm into our implementations of DoubleBFS and LI , i.e. sort the entries of the heap according to the number of labels in the label set.

5.2 Our contribution

In this section we discuss our contributions for answering LCR-queries. The approach LI with its extensions is our major contribution.

5.2.1 General comments

Let $\mathsf{Ind} \subseteq V \times V \times \mathcal{P}(\mathcal{L})$ be an index for all nodes $v \in V$. Let $\mathsf{Ind}(v) \subseteq V \times \mathcal{P}(\mathcal{L})$ be the index for node v and let $\mathsf{Ind}(v, w) \subseteq \mathcal{P}(\mathcal{L})$ be a list of (minimal) label sets connecting v and w.

$\mathsf{Ind}(v)$ is said to be minimal if for any two different pairs (u, L) and (u, L') in $\mathsf{Ind}(v)$ we have that neither $L \subseteq L'$ or $L' \subseteq L$. An index Ind is said to be minimal if and only if for all $v \in V$ we have that $\mathsf{Ind}(v)$ is minimal.

5.2.2 LandmarkedIndex

During the experiments and tests we noticed that BFS had an already nice query evaluation time. We also saw that building the full index takes a lot of time and memory. Building a full index is often not needed to achieve a speed-up. For instance, if a sufficiently large subset of the nodes $V' \subseteq V$ is (fully) indexed, we might as well achieve a speed-up. Hence, we decided to opt for an approach that is similar to BFS in its baseline but uses an index.

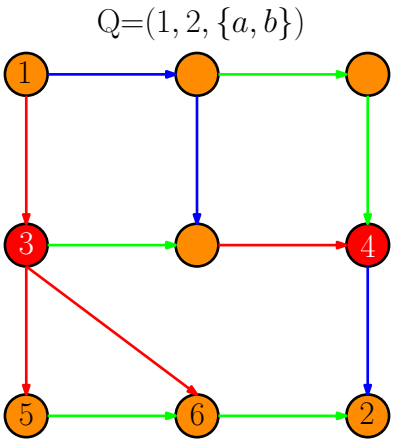


Figure 5.1: The query is $(1, 2, \{a, b\})$. The red edges are a, the blue edges b and the green edges c. Also let the red vertices 3 and 4 be landmarks. The query resolves to true as 1 can reach 4 and 4 can reach 2.

LI builds a complete index for a subset $V' \subseteq V$ of $|V'| = k$ nodes, also called landmarks. At query evaluation time, we run a normal BFS for a query (v, w, L). However, when we hit a node v' that is a landmark, we try to resolve the query directly. If the answer is True, we are done. If the answer is False, we do not need to process any of the out-neighbours of v' saving time compared to BFS.

We refer to LI as the Landmarked-index without extensions, LI+EXTv1 with only the first extension, LI+OTH with only the second extension and LI+OTH+EXTv1 with the first and second extension. We refer to LI with only the third extension as LI+EXTv2 and with both the first and the third extension as LI+OTH+EXTv2 . The third extension uses the same principle as the first extension and hence is a substitute for the first extension. Table 4.2 in Chapter 4 gives a summary.

Basic idea

First an ordering is created among the nodes, landmarks and non-landmarks. This is done by exhaustively looking at the node with the highest total degree (in- plus out-degree) from the set of all available nodes. The node $v \in V$ with the highest total degree among all available nodes is added to the ordering first. Next, all the nodes $w \in V$ that can reach v, i.e. $ASCS(v)$, are added by doing a reversed BFS. All these nodes are added to

the ordering one by one. For all nodes that have not been added to the ordering yet, we repeat this process. Let $O = \langle v_1, v_2, \ldots, v_n \rangle$ be the resulting ordering.

First we select k landmarks. This can be done in various ways, but the default way is picking the k vertices with the highest total degree. k has been specified before and is typically a fraction of N. After this we create a mapping $\mathsf{isL} : V \to \langle -1, 0, \ldots k-1$ that maps any vertex $v \in V$ to either -1 if it is not a landmark and a positive integer in the range $0 \ldots k-1$ if it is a landmark. We say $v \in V'$ or $v_i \in V$ for $i \geq 1$ if v is a landmark and $v \in V \setminus V'$ or $v_0 \in V$ if v is not a landmark.

The basic approach, i.e. without any extensions, runs Algorithm 3 for any landmark node first. The $i+1$'th call of the algorithm pushes a pair $(v_i, \{\})$ to the queue initially. Then, it iterates until the queue is empty. For each entry (u, L) obtained on line 4 we call TRYINSERT . It tries to insert (u, L) into $\mathsf{Ind}(v_i)$ and returns True if and only if this succeeded. TRYINSERT only succeeds whenever L is incomparable to any other L' in $\mathsf{Ind}(v_i, u)$ or whenever L is only a subset of some of the entries in L'. Moreover, it removes any superset L' of L in $\mathsf{Ind}(v_i, u)$. Hence, TRYINSERT preserves the minimality of $\mathsf{Ind}(v_i, u)$. Finally, we add all pairs $(w, L \cup \{l\})$ to q for each $(u, w, l) \in E$. The purpose of FORWARDPROP is explained in the following section.

Algorithm 3 LabelledBFSPerNode(v)

```
1: Let q be a queue
2: q.push(v, {})
3: while q is not empty do
4:    (u, L) ← q.pop()
5:    if tryInsert(u, v, L) = False then
6:       continue
7:    end if
8:    if hasBeenIndexed(u) = 1 then
9:       forwardProp(v, u, L)
10:      continue
11:   end if
12:   for (u, w, l) ∈ E do
13:      if l ∈ L then
14:         q.push(w, L ∪ {l})
15:      end if
16:   end for
17: end while
18: hasBeenIndexed(v) ← 1
```

Algorithm 4 tryInsert(u, v, L)

```
1: if v = u then
2:    return True
3: end if
4: if ∃[(u, L') ∈ Ind(v) | L' ⊆ L] then
5:    return False
6: end if
7: remove any (u, L') s.t. L ⊂ L' from Ind(v)
8: add (u, L) to Ind(v)
9: return True
```

Algorithm 5 forwardProp(v, u, L)

```
1: for (w, L') ∈ Ind(u) do
2:    tryInsert(v, w, L ∪ L')
3: end for
```

Propagation

There are a few variations/extensions of the basic idea. In our code we used by default 'forward propagation'. While building the index for a node v_j with $j \geq i$, we might visit a node v_i which has already been full indexed. When there is a L_1-path from v_j to v_i, we can say that any entry $(u, L_1 \cup L_2)$ can be appended to $\mathsf{Ind}(v_j)$ where $(u, L_2) \in \mathsf{Ind}(v_i)$. The idea came from studying the optimization of Zou's algorithm [10]. Forward propagation is used by the basic approach.

Again there are many ways to do propagation. We can do it for any node or only for a subset of the nodes. It can be done after constructing an index for a node v or during construction. Propagation can incur a lot of overhead for graphs with low connectivity, but might also provide a lot of reward for more well connected graphs. The best approach really depends on the type of the graph.

During construction we maintain a mapping that indicates which vertices $v \in V$ have already been (fully) indexed by using $\mathsf{hasBeenIndexed} : V \to \{0, 1\}$.

Heap

Algorithm 3 can explore parts of the graph multiple times. This can hurt the performance and be redundant in some cases. Inspired by Zou's [10] a possible extension of LI is changing the queue into a heap (or a min priority queue) where the heap entries are sorted on the number of labels in the label set. In this way we are assured that we never insert (u, L) before (u, L') to $\mathsf{Ind}(v_i)$ when $L \supset L'$. Hence tryInsert never has to remove any entries. The heap might incur some overhead, but on the other hand might save time when a large part of the graph is traversed multiple times. The heap is used by the basic approach.

Basic query algorithm

Algorithm 6 is the pseudocode for resolving a query for LI+OTH . To get the basic version without extensions you should ignore lines **5-13**.

We start by exploring the graph using a BFS at line **14**. Only when we hit a landmark $v' \in V'$ during this BFS, we try to resolve the query directly. This is done by calling Algorithm 7. This looks for an entry of the form $(w, L') \in \mathsf{Ind}(v')$. When this 'direct attempt' fails we do not need to look at any of the out-edges of v', because we know w cannot be reached from v'.

First and third extension

In figure 5.1 we see that vertex 1 can reach landmark 3, but 3 cannot reach vertex 2 over $\{a, b\}$. However, 3 can reach two other vertices: vertices 5 and 6. If one of these vertices were able to reach 2 then 3 could reach 2. Hence there is no point in evaluating any vertex that has a $\{a, b\}$-path from vertex 3.

This notion can be generalized. The basic idea for the first extension is the following. Given a query $q = (v, w, L)$ we might stumble upon a landmark v' one way or another. If a direct attempt (v', w, L) fails, we know that any vertex w' that has a L-path from v' cannot reach w. Hence it can be pruned.

When we use the first extension, we choose to store some extra redundant information for each landmark $v' \in V$. We store $l' = |\mathcal{L}|$ extra lists containing all vertices that can be reached using one particular label only. If a query (v', w, L) fails from a landmark v', we can prune all vertices that are in one of the in total l' lists if the corresponding label of that list is in L. We also call a query that prunes in this way an "extensive query".

Let LP be the variable holding these lists for each $v' \in V$. LP is filled during each iteration of Algorithm 3 after line **7** but before **8**. We obtain an entry (u, L) at line **4**. As long as $|L| = 1$ and $u \neq v$ we insert u to $\mathsf{LP}[v][l]$ if $L = \{l\}$.

One could think of maintaining more lists, i.e. also adding lists representing the all the label sets with two or three labels. However this can be quite expensive in cost of memory as there can be $\binom{|\mathcal{L}|}{3}$ such lists, which is $O(|\mathcal{L}|^3)$. There is also going to be a lot of redundancy in these lists.

As the first extension did not provide enough speed-up for False-queries for when $k \leq \frac{N}{50}$ we needed to tweak the idea of the first extension. This extension can only be used as a substitute for the first extension and it uses the same principle. It is either the first or the third extension, but never both.

For each landmark $v' \in V$ we maintain $l \leq 2^{MAXDIST}$ entries with $MAXDIST = |\mathcal{L}|/4 + 1$. The variable holding these entries is called seqE in our code. Each entry $e \in \mathsf{seqE}$ consists of a set $B \subseteq V$ and a label set L.

Algorithm 6 queryLM(v, w, L)

```
1: if isL(v) ≥ 0 then
2:   return queryDirect(v, w, L)
3: else
4:   Let marked : V → {0, 1} be a mapping
5:   for v' ∈ {v' | (v', L) ∈ Ind(v)} do
6:     // v' is always a landmark
7:     for L' ∈ Ind(v, v') ∧ L' ⊆ L do
8:       if queryDirect(v', w, L) = True then
9:         return True
10:      end if
11:      marked[v'] ← 1
12:    end for
13:  end for
14:  Let q be a queue
15:  q.push(v)
16:  while q is not empty do
17:    u ← q.pop()
18:    marked[u] ← 1
19:    if u = w then
20:      return True
21:    end if
22:    if isL(u) ≥ 0 then
23:      if queryDirect(u, w, L) = True then
24:        return True
25:      end if
26:      continue
27:    end if
28:    for (u, u', l) ∈ E ∧ marked[u'] = 0 do
29:      if l ∈ L then
30:        q.push(u')
31:      end if
32:    end for
33:  end while
34:  return False
35: end if
```

Algorithm 7 queryDirect(v, w, L)

```
1: // v is a landmark
2: for L' ∈ Ind(v, w) do
3:   if L' ⊆ L then
4:     return True
5:   end if
6: end for
7: return False
```

Each B is a 'bitset' of N bits in which a bit j is set if and only if we can reach the corresponding vertex w which has vertex id j. Hence reading or writing for a specific bit takes $O(1)$ time.

The third extension tweaks Algorithm 3 a little. Each time an entry (u, L) has been discovered for $Ind(v)$ on line **4**, we can do two things. Either, we create a new entry $e \in V \times \mathcal{L}$ where $e.B = \{u\}$ and $e.L = L$. Or, we find the already existing entry e for L and set $e.B$ to $e.B \cup \{u\}$.

At the end we merge the entries, i.e. given two entries $e, e' \in \mathsf{seqE}$ where $e'.L = e.L \cup \{l\}$ we join the sets of e with e' as e' can reach all entries of e.

The advantage of the third extension over the first is that we can use larger label sets, i.e. from 1 to $MAXDIST$. We chose to create a maximal distance, as we saw that medium-size label sets can often cover a large subset of the nodes in graphs with a high degree or an exponential label set distribution, and can be used a majority of the queries. The second advantage is that we can quickly join the bitset marked used by the query-algorithm with the bitset B used by any entry $e \in \mathsf{seqE}$. Joining can be done in $\frac{N}{64}$ time on a 64-bit machine.

Second extension

After all landmark vertices have been fully indexed, the second extension can add a number of entries (v_i, L) (where v_i is a landmark) to $\mathsf{Ind}(v_0)$ for all non-landmark vertices $v_0 \in V$.

We choose b to be a constant, typically a low value. Each v_0 adds at most b entries to $\mathsf{Ind}(v_0)$. After it has found this many paths or it can't find any more entries, it stops.

The algorithm for doing this is very similar to Algorithm 3. The differences are listed.

1. We only add entries (v_i, L) to $Ind(v_0)$ with $i \geq 1$.
2. We stop immediately after having successfully inserted b entries to $Ind(v_0)$.
3. We use both landmarked and non-landmarked nodes for FORWARDPROP , i.e. we copy the entries from each indexed node regardless of whether it is a landmark. hasBeenIndexed is set to 1 at the end for any $v_0 \in V$ as well.
4. We add a variable $\mathsf{marked} : V \to \{0, 1\}$ before line **3**. Each entry (u, L) found on line **4** sets $\mathsf{marked}[u] \leftarrow 1$. In case we have that $\mathsf{marked}[u] = 1$ we continue, i.e. we skip processing entry (u, L).

Query algorithm with extensions

Adding the second extension to LI enables lines **5-13** of Algorithm 6 to be actually used.

The difference between LI+OTH and LI+OTH+EXTV1 is that the first call of QUERYDIRECT has been replaced by a call to QUERYEXTENSIVEDIRECT (in case v is not a landmark). The difference between LI+OTH and LI+OTH+EXTV2 is that any call QUERYDIRECT has been replaced by a call to QUERYEXTENSIVEDIRECT .

Algorithm 8 consists of two versions. One is intended for the first extension and one is intended for the third extension. Both prune all w' that can be reached from a landmark v' with label set L that could not reach the query-target w with that same label set.

Memory usage

The worst-case memory usage analysis is for LI+OTH+EXTV1 with k landmarks and b budget per non-landmark node and for LI+OTH+EXTV2 with k landmarks and b budget per non-landmark node.

Considering only the landmarks, the memory usage can in the worst case be $O(|V| \cdot k \cdot 2^{|\mathcal{L}|})$. Each landmark v' inV' can index up to $|V| - 1$ other nodes and for each such node we can store $2^{|\mathcal{L}|})$ paths.

Each landmark v' can also have $|\mathcal{L}|$ lists in LP and each such list can be of length N at most. This results in $O(|\mathcal{L}| \cdot N)$ extra storage for v'.

Each landmark v' can also have $2^{MAXDIST} = 2^{|\mathcal{L}|/4+1}$ entries in seqE. Each consists of a set of at most $O(N)$ (B) and a constant $O(1)$ (L). This results in $O(2^{|\mathcal{L}|} \cdot N)$ extra storage for $v' \in V$.

Each remaining non-landmark vertex $v_0 \in V$ can store up to b entries, which takes $O(b)$.

Hence in total the memory usage is: $O(b + |V| \cdot k \cdot 2^{|\mathcal{L}|})$ (LI+OTH+EXTV1) and $O(b + |V| \cdot k \cdot 2^{|\mathcal{L}|})$ (LI+OTH+EXTV2).

Running time of index building

The worst-case running time analysis is for LI+OTH+EXTV1 with k landmarks and b budget per non-landmark node and for LI+OTH+EXTV2 with k landmarks and b budget per non-landmark node.

Considering just the landmarks the running time of the index construction algorithm is at most $O(|V| \cdot k \cdot 2^{2 \cdot |\mathcal{L}|})$. Each landmark $v' \in V'$ can index up to $|V| - 1$ other nodes and for each such node there can be at most $2^{|\mathcal{L}|}$ paths. Each path can invoke a call to TRYINSERT .

Algorithm 8 queryExtensiveDirect($v, w, L,$ marked)

```
1: // first extension
2: if queryDirect(v, w, L) = True then
3:    return True
4: end if
5: for l ∈ L do
6:    for v' ∈ LP[v][l] do
7:       marked[v'] ← 1
8:    end for
9: end for
1: // third extension
2: if queryDirect(v, w, L) = True then
3:    return True
4: end if
5: for (B, L') ∈ seqE do
6:    if L' ⊆ L then
7:       for v' ∈ B do
8:          marked[v'] ← 1
9:       end for
10:      break
11:   end if
12: end for
```

Any call to tryInsert can take at most $O(2^{|\mathcal{L}|})$ as there are at most $2^{|\mathcal{L}|}$ label sets.

Any call to forwardProp can take at most $O(|V| \cdot 2^{2\cdot|\mathcal{L}|})$ as there can be at most $|V|-2$ vertices (excluding u and v) in $\mathsf{Ind}(u)$ eligible for insertion and there can be at most $2^{|\mathcal{L}|}$ pairs between any such vertex v and w. For each pair we call tryInsert which takes $O(2^{|\mathcal{L}|})$. Hence we end up at $O(|V| \cdot 2^{2\cdot|\mathcal{L}|})$. As we do not push the out-edges of an indexed vertex u after a call to forwardProp on line **9**, we can claim that any call to tryInsert by forwardProp through u cannot be made from Algorithm 3 in the future. Hence forwardProp does not increase the total running time.

Filling LP for each landmark $v' \in V$ takes $O(|V| \cdot \log(|V|) \cdot 2^{|\mathcal{L}|})$. Inserting a certain u for a pair $(u, \{l\}$ to $\mathsf{LP}[v'][l]$ takes $O(\log(|V|))$. The number of times this happens cannot be more than the number of calls to tryInsert on line **5**.

Filling seqE takes $O(2^{|\mathcal{L}|})$ time as in the worst case we need to find the entry e to which label set L' of entry (u, L') belongs. Setting the bit of $e.B$ can be done in constant time, as we know u.

For the non-landmarked nodes $v_0 \in V$ we can make at most $2^{|\mathcal{L}|}$ calls to tryInsert per vertex and at most b entries can be inserted. In the worst case we make $2^{|\mathcal{L}|}$ for b different vertices. When building this part of the index we visit each vertex in the graph at most once. Hence the index construction time for non-landmarked nodes can take at most $O(b \cdot 2^{|\mathcal{L}|} + |V|)$.

In total this results to $O(b \cdot 2^{|\mathcal{L}|} + |V| \cdot ((k \cdot 2^{2\cdot|\mathcal{L}|}) + (\log(|V|) \cdot 2^{|\mathcal{L}|}))$ (LI+OTH+EXTv1) and $O(b \cdot 2^{|\mathcal{L}|} + |V| \cdot ((k \cdot 2^{2\cdot|\mathcal{L}|}) + (2^{|\mathcal{L}|}))$ (LI+OTH+EXTv2).

Running time of query algorithm

The running time of Algorithm 7, that is queryDirect , is $O(2^{|\mathcal{L}|} + \log(|V|))$ as there are at most $2^{|\mathcal{L}|}$ label sets between any two vertices and looking up w in $Ind(v)$ takes at most $\log(|V|)$ time.

The running time of Algorithm 8, that is queryExtensiveDirect , is that of queryDirect plus the worst case time of doing at most $2^{|\mathcal{L}|}$ comparisons and setting at most N bits in marked. Hence this takes $O(|V| + 2^{\mathcal{L}|}$.

In case of LI+OTH+EXTv1 or LI+OTH+EXTv2 , the total worst case running time of Algorithm 6 is $O(k \cdot (2^{|\mathcal{L}|} + \log(|V|)) + |V| + |E|)$. At most k direct attempts to resolve a query are made, either on lines **5-13** or on lines **23-25**. The graph exploration part of Algorithm 6 takes at most $O(|V| + |E|)$.

Proof of correctness

Let $v, w \in V$ and let $L \subseteq \mathcal{L}$. We say $query(v, w, L)$ = True if and only if there is a L-path from v to w. Otherwise we say $query(v, w, L)$ = False. We begin by proving that $query(v, w, L)$ = True $\Leftrightarrow \exists[(w, L') \in \mathsf{Ind}(v) \mid L' \subseteq L]$.

Theorem 5.2.1. *Let $V' \subset V$ be the set of landmark vertices. Let $\langle v_1, \ldots, v_k \rangle$ be an ordering of the k landmark vertices. Let $\mathsf{Ind}(v_j)$ be the index of the j'th landmark constructed by Algorithm 3. We wish to establish that for any $w \in V$ **(1)** $query(v_j, w, L)$ = True implies that $\exists[(w, L') \in \mathsf{Ind}(v_j) \mid L' \subseteq L]$ and **(2)** implies $query(v_j, w, L)$ = False that $\neg\exists[(w, L') \in \mathsf{Ind}(v_j) \mid L' \subseteq L]$.*

Proof. **Base case:** $j = 1$

(1): Assume $query(v_j, w, L)$ = True. Then there is a L-path P from v_j to w. Lines **12-16** find any label set (minimal or not) connecting v to some $u \in DESCS(v)$, where $DESCS(v)$ is the set of descendants of v. Only line **5** can block adding (w, L) to $Ind(v_j)$ in this case. Let (u, L') be the first vertex in path P where the if-condition on line **5** is true, where $L' \subseteq L$. This only happens if $\exists[(u, L^*) \in \mathsf{Ind}(v_j) \mid L^* \subseteq L']$. This argument can be repeated for any next vertex in path P and hence we have that $\exists[(w, L') \in \mathsf{Ind}(v_j) \mid L' \subseteq L]$.

(2): Assume $query(v_j, w, L)$ = False. Then there is no L-path P from v_j to w. If there is no such path lines **12-16** will never push an entry of the form (w, L') s.t. $L' \subseteq L$. Hence such an entry can never appear in $\mathsf{Ind}(v)$.

Step: $1 < j \leq k$. We assume that for $v_1, \ldots, v_{j-1}$ we have that **(1)** and **(2)** are true (IH).

(1): Assume $query(v_j, w, L)$ = True. Then there is a L-path P from v_j to w. Lines **12-16** find any label set (minimal or not) connecting v to some $u \in DESC(v)$ (see Definition 2.4.1), where $DESC(v)$ is the set of descendants of v. W.r.t. line **5** we have the same argument as in the base case. Line **8** can fire for any v_i with $i \leq j-1$. Let v_i (with label set L_1) be a vertex on path P s.t. this happens. In that case $\exists[(w, L_2) \in \mathsf{Ind}(v_i) \mid L_2 \subseteq L]$ and the entry $(w, L_1 \cup L_2)$ will be added to $\mathsf{Ind}(v_j)$.

(2): Assume $query(v_j, w, L)$ = False. Then there is no L-path P from v_j to w. If there is no such path lines **12-16** will never push an entry of the form (w, L') s.t. $L' \subseteq L$. As **(2)** holds for any v_i with $i \leq j - 1$ an incorrect entry is not added by any of the v_i to $\mathsf{Ind}(v_j)$. □

Lemma 5.2.2. *We wish to establish that $query(v_j, w, L)$ = True $\Leftrightarrow \exists[(w, L') \in \mathsf{Ind}(v_j) \mid L' \subseteq L]$ and $query(v_j, w, L)$ = False $\Leftrightarrow \neg\exists[(w, L') \in \mathsf{Ind}(v_j) \mid L' \subseteq L]$.*

Proof. As we either have that $query(v_j, w, L)$ = True or $query(v_j, w, L)$ = False, we get from Theorem 5.2.1 that $query(v_j, w, L)$ = True $\Leftrightarrow \exists[(w, L') \in \mathsf{Ind}(v_j) \mid L' \subseteq L]$ and $query(v_j, w, L)$ = False $\Leftrightarrow \neg\exists[(w, L') \in \mathsf{Ind}(v_j) \mid L' \subseteq L]$. ($P \Rightarrow Q \wedge \neg P \Rightarrow \neg Q$ is equivalent (contraposition) $P \Rightarrow Q \wedge Q \Rightarrow P$ which is equivalent to $P \Leftrightarrow Q$.) □

The entries (v', L') where v' is a landmark obtained for non-landmark nodes $v \in V \setminus V'$ are created by an algorithm with the same structure as Algorithm 3. However because the algorithm stops after inserting b entries we can only claim that if an entry exists than v can reach v' over L'.

Next we wish to verify the correctness of queryDirect and queryExtensiveDirect .

Theorem 5.2.3. queryDirect *returns true if and only if $query(v, w, L) = True$.* queryExtensiveDirect *only prunes a subset of the vertices $V^* \subseteq V$ s.t. for all $v^* \in V^*$ we have that $query(v^*, w, L) = False$.*

Proof. By Lemma 5.2.2 we know that queryDirect return true if and only if $query(v, w, L) =$ True.

Variables LP and seqE used by the first and third extension respectively are filled by using entries (u, L') obtained in the loop in lines **3-16** of Algorithm 3. As each such entry is in $\mathsf{Ind}(v)$ and $query(v, w, L)$ = True $\Leftrightarrow \exists[(w, L') \in \mathsf{Ind}(v) \mid L' \subseteq L]$ by Lemma 5.2.2, we have that any vertex u included in seqE and LP is reachable from v using L.

Lines **5-9** (first extension) or **5-10** (third extension) are only reached in case queryDirect returns false on line **2**. Any vertex $u \in V$ set marked on line **7** (both extensions) can be reached by some subset $L' \subseteq L$ from v. If such u were able to reach w over L then v could reach w as well. Hence such u cannot reach w and can correctly be pruned, i.e. included to marked. □

Finally we wish to prove the correctness of QUERYLM .

Theorem 5.2.4. *Let $v, w \in V$ and $L \subseteq \mathcal{L}$. We have that algorithm 6 (using either* LI *or extension* 1,2,12 *or* 13*) returns true if and only if query$(v, w, L) =$ True.*

Proof. This can be proven by a case distinction.

- **Case 1:** On line **2** we run QUERYDIRECT for (v, w, L). Because of Theorem 5.2.1 and 5.2.3, the statement holds.
- **Case 2:** On line **8** we run QUERYDIRECT (or QUERYEXTENSIVEDIRECT) for (v', w, L) where v' is a landmark that can be reached from v using L. Because Theorem 5.2.3 and Theorem 5.2.1, the statement holds and we prune a subset of the correct vertices, i.e. those vertices that have no L-path to w, in case of QUERYEXTENSIVEDIRECT .
- **Case 3:** On line **8** we run QUERYDIRECT (or QUERYEXTENSIVEDIRECT) for (v', w, L) where v' is a landmark that can be reached from v using L. The v to v' path has been discovered in the loop from lines **16-33**. Because of Theorem 5.2.1, the statement holds. Because of Theorem 5.2.3 we prune a subset of the correct vertices in case of QUERYEXTENSIVEDIRECT .
- **Case 4:** In this case the loop on lines **16-33** ended. There cannot be a L-path from v to w as no QUERYDIRECT call found such a path, only correct vertices were pruned according to Theorem 5.2.3 and this path was not find through graph exploration (Lines **16-33**). The algorithm returns false.

□

DoubleBFS

DOUBLEBFS is a special case of LI , i.e. $k = N$. Any extension is irrelevant in this case as any query is answered by QUERYDIRECT .

5.2.3 Partial

PARTIALINDEX also builds an index for a subset of k nodes like LI , but does not build a full index to cut down the index size. Rather it chooses a budget b which is a fraction of N, e.g. $\frac{N}{4}$. Each of the k nodes then index up to b nodes fully ignoring any other vertices. The difference with LI is that at query evaluation time we have to evaluate the out edges of a node v'. Otherwise, we might return wrong results.

PARTIALINDEX has a frequency. The frequency determines the number of nodes that needs to be hit before we try to resolve a query directly. Having to resolve a lot of direct queries can slow down the query evaluation and also often have no purpose if the sources of the direct queries are close to each other. Figure 5.2 shows an example. In the final implementation the frequency is roughly $O(\sqrt{(N)})$.

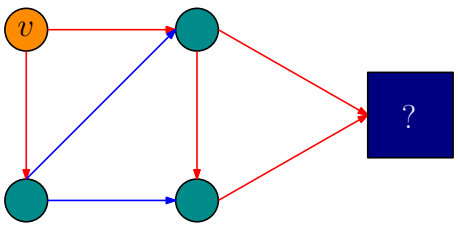


Figure 5.2: Suppose we start a query at orange vertex v. Let the box on the right be a large graph and the cyan-colored vertices be vertices for which there is a partial index. Each of these also has a partial index. Having to run a direct query for each of these will most likely yield the same result.

5.2.4 NeighbourExchange

This method is based on the discussion in Chapter 2 Section 3.2.3 with some differences applied. The algorithm builds a full index. Algorithm 9 shows a pseudocode.

Algorithm 9 NeighbourExchange()

```
1:  c ←True
2:  let Upd be a V × V × 2^L
3:  for v ∈ V do
4:    for (v, w, l) ∈ E do
5:      add (v, {l}) to Upd[w]
6:    end for
7:  end for
8:  while c =True do
9:    c ← False
10:   for v ∈ V do
11:     d ←False
12:     for (w, L) ∈ Upd[v] do
13:       if tryInsert(v, w, L) = True then
14:         c ← True
15:         d ←True
16:       end if
17:       if d = True then
18:         for (v, w′, l) ∈ E do
19:           if l ∈ L then
20:             add (v, {l} ∪ L) to Upd[w′]
21:           end if
22:         end for
23:       end if
24:     end for
25:     Upd[v] ← ∅
26:   end for
27: end while
```

On lines **3** to **7** we initialize Upd with the direct neighbour updates. This is necessary to kick off the entire algorithm. Next, we start a loop at line **8** that continues as long as there are updates to the index. We loop over any node $v \in V$ and check whether there are entries in Upd$[v]$. We add an entry (w, L) on line **16** to Ind(v) if and only if it does not violate the minimality of the index. Next, we add the neighbours of w.

One of the disadvantages of NeighbourExchange is that it has to use Upd and Ind. The memory usage of Upd can grow considerably depending on the order in which the updates are processed. Another disadvantage is that we see no way to use propagation for this index.

5.2.5 Joindex

This method is based on the discussion in Section 3.2.3 with some differences applied. Joindex can both be used with intersection or union.

The algorithm to construct Joindex is very similar to that of DoubleBFS or NeighbourExchange . We need two (exact) indices In and Out. First, we need to determine a budget b which is the maximum number of entries per pair (u, v) for some $u, v \in V$ and decide whether we wish to use intersection or union to merge the entries. For union we do the following. While adding an entry (u, L) to Ind(v), we verify whether this would result in exceeding b. If this is the case, we compare L to all b entries L' that are in the index. The entry (u, v, L') for which $|L' \cap L|$ is maximized, is merged with (u, v, L) yielding an entry $(u, v, L \cup L')$ for which a join flag is set indicating it has been merged (the most significant bit). For intersection, we do the same except the resulting entry is $(u, v, L \cap L')$. At query evaluation time, the join flag can be used.

Algorithm 10 shows the query algorithm in case JOINDEX uses union to merge entries. A query (v, w, L) is resolved by looking for a node s.t. $u \in \mathsf{Out}(v) \cap \mathsf{In}(w)$. If we have that $(u, L) \in Out(v)$ and $(u, L) \in In(w)$ and neither has a join flag on, we return True. Only in case the left or right part (u, L') has a join flag, indicated by $J(u, L')$, and the entry is a superset of (u, L), we need to recur on that part. If we have tried out all u, then we return False. The algorithm for answering JOINDEX queries using intersection is very similar.

Algorithm 10 JoindexQueryUnion(v, w, L, marked)

```
1: if |L| = 0 then
2:   return False
3: end if
4: if v = w then
5:   return True
6: end if
7: for u ∈ (Out(v) ∩ In(w) ∧ marked[u] = 0 do
8:   if ∃[(u, L') ∈ Out(v) | L' ⊆ L] then
9:     if ∃[(u, L') ∈ In(w) | L' ⊆ L] then
10:      return True
11:    else
12:      if ∃[(u, L') ∈ In(w) | L' ⊇ L ∧ J(u, L')] then
13:        marked[u] ← 1
14:        JoindexQueryUnion(u, w, L, marked)
15:        marked[u] ← 0
16:      end if
17:    end if
18:  else
19:    if ∃[(u, L') ∈ Out(v) | L' ⊇ L ∧ J(u, L')] then
20:      marked[u] ← 1
21:      JoindexQueryUnion(v, u, L, marked)
22:      marked[u] ← 0
23:    end if
24:  end if
25: end for
26: return False
```

In the end we decided not to fully implement JOINDEX , for the following reasons.

1. JOINDEX needs two times the index, In and Out, and hence has a double memory consumption for at least some time period.

2. JOINDEX only decreases the index size but not the construction time. LI and PARTIALINDEX decrease both the construction time and index size.

3. JOINDEX increases the query evaluation time. NEIGHBOUREXCHANGE or DOUBLEBFS can answer a query in $O(2^{|\mathcal{L}|})$, whereas JOINDEX needs $O(|V| + |E|)$.

5.2.6 ClusteredExact

This method is based on the observation that separating the graph into some subgraphs and building a full index for each of these subgraphs individually is more efficient than for the whole graph.

First, we create a clustering. Each of the N nodes is assigned to one of the K clusters. An exact index is generated, e.g. by using DOUBLEBFS , for each cluster.

The query evaluation algorithm is the most tricky part about this approach. Algorithm 11 gives a pseudocode. $\mathsf{cID} : V \to \langle 0, \ldots, K-1 \rangle$ maps each vertex to a certain cluster. outP gives the list of vertices that are out ports for that cluster. Line **6** checks whether the query can be resolved directly because x and w are in the same

Algorithm 11 ClusteredExactQuery(v, w, L)

```
1: Let marked : V → {0, 1}
2: Let q be a queue
3: while q is not empty do
4:    x ← q.pop()
5:    if cID[x] = cID[w] then
6:       if directQuery(x, w, L) = True then
7:          return True
8:       end if
9:    end if
10:   if marked[x] = 1 then
11:      continue
12:   end if
13:   marked[x] ← 1
14:   for p ∈ outP[cID[x]] do
15:      if directQuery(x,p) = True then
16:         for (p, s, l) ∈ E ∧ l ∈ L do
17:            push s onto q
18:         end for
19:      end if
20:   end for
21: end while
```

cluster. Otherwise, we loop over all out-ports p of the cluster x belongs to. It is checked whether x can reach p using L and whether the label l of an edge (p, s) is in L. If this is the case, we push s onto the stack.

5.3 Implementation details

Let $G = (V, E, \mathcal{L})$ be an arbitrary labelled directed graph.

There is a difference between labels and label sets. A label is an individual element in $\mathcal{L}$, e.g. $a \in \{a, b, c\}$. Labels are represented by a number in the range $0...|\mathcal{L}|-1$. Label sets are represented as bitsets which means that if $L \subseteq L'$ we mean that in our code L and L' have a non-empty bit-intersection, e.g. $L = \{a, b\} = 3 = (00000011)$ and $L' = \{a, b, c\} = 7 = (00000111)$ have a non-empty bit-intersection namely the first and second bit whereas $4 = (00000100) = \{c\}$ and $2 = (00000010) = \{b\}$ have none. 0 denotes the empty label set.

The code has the ability to scale down the number of bits used in a label set, e.g. if $|\mathcal{L}| = 8$ we can use just a byte, thereby cutting down the index size. However for all experiments we set the size of a label set to 4 bytes.

An index Ind can have two modes: blocked mode and non-blocked mode. In blocked mode we store the index as a triple-array with dimensions: $|V| \times |V| \times |\mathcal{P}(\mathcal{L})|$, in which the third dimension can scale depending on the actual number of label sets for a given pair (v, w). In non-blocked mode we store the index for each node $v \in V$ as a list of tuples of the form: (w, L^*) where L^* is a list of label sets. Blocked mode can cut down the index construction time compared to non-blocked mode, but takes much more memory and is not scalable. Hence we only used non-blocked mode in our experiments.

5.4 Index maintenance

In this section we stress the issue of index maintenance. Suppose that after running LI , PartialIndex , DoubleBFS , ClusteredExact or NeighbourExchange we have built an index $\mathsf{Ind}(v)$ for every $v \in V$. There can be 5 update events:

1. Addition of a node v
2. Removal of a node v
3. Addition of an edge (v, w, l)
4. Removal of an edge (v, w, l)
5. Changing edge label (v, w, l) to (v, w, l')

A choice we made for all approaches is to build a minimal index, i.e. for each pair of nodes (v, w) we have that all label sets L and L' are no super- or subset of each other. This can have implications for index maintenance, as the removal of one edge can necessitate rebuilding a part of the index. This part can become quite large. Like discussed in Section 3.1.1 storing some redundant information might help making maintenance more easy.

Figure 5.3 illustrates this problem which is posed by removing just a single edge. Suppose we would remove edge $(2, 3, \{a\})$, then we need to update 0 and add entries $\{a, b\}$ and $\{a, c\}$ to $(0, 4)$ while removing $\{a\}$. For graphs with a lot of cycles this problem can get even worse.

The maintenance algorithms below were built to work on an index with LI . As DoubleBFS is the same as LI when $k = N$, all these solutions also work for DoubleBFS .

5.4.1 Adding an edge

Suppose we add an edge of the form (v, w, l) to our graph. Then any entry of the form (v', w', L) that visited v during construction might be altered. Only for vertex w we can be sure that no entry is modified, as any entry in $Ind(w)$ that could use (v, w, l) is already covered by an entry in $Ind(w)$.

Algorithm 12 shows how to run an update. Vertex v is pushed onto a queue q. While the queue is not empty, we pop a vertex x (initially v). We run an adapted version of LabelledBFSPerNode, which will not attempt tryInsert if it can make a call to forwardProp. This will not do a full reconstruction. Rather it will tryInsert the entry (w, l) into $\mathsf{Ind}(v)$ or copy entries from a landmark v' that has been fully indexed.

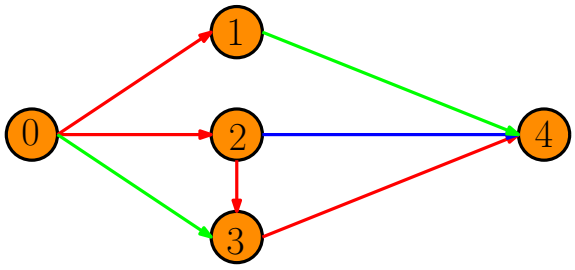


Figure 5.3: A simple graph, where edge color indicate the label. Let red be $\{a\}$, blue be $\{b\}$ and green be $\{c\}$. Vertex 0 can reach vertex 4 over several paths, but only has to store $\{a\}$, because all other paths are supersets of that path.

Algorithm 12 insert(v, w, l)

```
1: add (v, w, l) to G
2: reset hasBeenIndexed
3: hasBeenIndexed(w) ← True
4: let q be an empty queue
5: push v to q
6: while q is not empty do
7:    if hasBeenIndexed(w) = True then
8:       continue
9:    end if
10:   x ← q.pop()
11:   if x is a landmark then
12:      LabelledBFSPerNode'(x)
13:   end if
14:   for (v, x) ∈ E do
15:      q.push(v)
16:   end for
17:   hasBeenIndexed(x) ← True
18: end while
```

5.4.2 Removing an edge

Removing an edge (v, w, l) consists of two steps. First we need to remove any entry $(w', L) \in \mathsf{Ind}(v')$ for all $v' \in V$ which would not be in the graph if (v, w, l) would not be there. Let's call this set A. Next, we might need to rebuild parts of the index using LabelledBFSPerNode for every $v \in V$, because entries that were pruned previously could be part of the index after removal of the edge.

The first difficulty lies in a good approximation of A. So far it appears we can only find some superset A' of A. The second difficulty has to do with the fact that we end up with an index that has some but not all of the entries. A call to LabelledBFSPerNode started at node w assumes that if $(v, L) \in \mathsf{Ind}(w)$ we need not to look at any descendants of v, but this might be wrong if $(v, L) \in \mathsf{Ind}(w)$ but one of its descendants $v' \in DESC(v)$ does not have an entry in $\mathsf{Ind}(w)$ while it should.

Approximating A can be done by the following algorithm. After an edge (v, w, l) has been removed, we can (temporarily) promote v and w to a landmark and fully index these. Next, we iterate over all $w' \in V$. If there exists an entry $(v', L') \in Ind(w')$ s.t $L' \supseteq L \cup \{l\}$, we remove it. Existence of such an entry could mean that w' can no longer reach v' over L'. Let A' be the resulting approximation and let A'_V be $\{v \mid (v, L) \in A'\}$.

We did not manage to find a 'good' solution for the second part. Either the solution we found for this part

was too slow, i.e. not within 90% of the time of a full rebuild, or the solution would contain errors. For instance, we tried to start a LabelledBFSPerNode for any $y \in V$ with an adapted queue q that contained in the beginning all entries in the set $\{(w', L) \mid w' \in A'_V \wedge (v', w', l') \in E \wedge L \in \{\{l'\} \cup L' \mid (v', L') \in (\mathsf{Ind}(y)\}\}$. In this case the queue consists of all entries $(w', L' \cup \{l\})$ s.t. there is an in-neighbour v' with an entry $(v', L') \in \mathsf{Ind}(y)$ This solution would give errors.

5.4.3 Changing edge label

As this is quite similar to first removing and then adding an edge we chose not to implement this case.

5.4.4 Adding a node

Adding a node without any edges is trivial and can be done in constant time.

5.4.5 Removing a node

As adding an edge or removing an edge is already quite expensive we do not expect there to be an efficient solution for removing a node. Hence we think a full rebuild is a better solution here.

5.5 Extensions

In this section we discuss several extensions that could be possible based on the indices we have discussed so far.

5.5.1 Query for all nodes

One of the many possible extensions could be to add support for the following type of query: "find all v' s.t. given a label set L and $v \in V$ we have that query (v, w, L) is true". The result of the query could be a mapping $m : V \to \{0, 1\}$. m has the same role as the variable marked in the description of BFS (Section 5.1.1).

BFS could answer this query by simply continuing searching the graph rather than stopping at the target w. For each node $v \in V$ that is traversed, we set $m(v) \leftarrow 1$. This has a nice $O(|V| + |E|)$ running time. However this could be too much already.

LI could do the same as BFS , but make use of landmarked vertices. When a landmark v' is hit, we look at all (w, L') in $Ind(v')$ and test whether $L' \subseteq L$. If this is, we set $m(v) \leftarrow 1$.

5.5.2 Distance queries

Another extension could be to find the distance of a query as well, i.e. given a query $q = (v, w, L)$ find the shortest path P (w.r.t. the number of edges) that connects v and w s.t. $Labels(P) \subseteq L$. This kind of query is answered by Bonchi et al. [1].

There is an important difference with "LCR" here though. We wish to find a distance d s.t. $d = min_X(\#P(v, w))$ where $X = P \in P(v, w) \wedge \mathsf{Labels}(P) \subseteq L$. This implies we need to store supersets for a pair v and w as well, given a superset has a shorter distance from v to w. Given two entries (v, w, L_1) and (v, w, L_2) connecting v and w with $L_1 \subset L_2$, we need to store both entries if the distance associated to the second entry is less than the distance of the first entry. This will increase the index size.

There are two things we need to change to make e.g. LI capable of answering these kinds of queries.

tryInsert needs to be modified s.t. it only adds an entry (v, w, L_1, d_1) if for all other entries (v, w, L_2, d_2) we either have that $L_1 \not\subseteq L_2 \wedge L_2 \subseteq L_1$ or that $d_1 < d_2 wedge L_2 \subset L_1$. Moreover, it should remove all entries (v, w, L_2, d_2) when $L_1 \subseteq L_2$ and $d_2 \geq d_1$.

The entries for the heap in Algorithm3 need to include a distance metric w.r.t. the number of edges as well. Preferably the heap should order its entries by this distance.

Chapter 6

Experiments

In this chapter we describe the experimental results of the methods described in Chapter 5. The experiment has been divided into 2 parts. In each part we only experiment with datasets that have a number of edges between two values.

Th experiments have been divided into three parts, based on the maximum number of edges that are in the datasets of each of these parts: $0 < |E| \leq 5,000$ (small), $5,000 < |E| \leq 500,000$ (medium) and $|E| \geq 500,000$ (large).

The first part includes all methods discussed in Chapter 5 (PartialIndex , LI , Zou , NeighbourExchange , Joindex , ClusteredExact and DoubleBFS) and is meant to select the more promising methods out of the less promising ones.

The second part shows the effect of the out-degree and label set distribution and the size of the label set $|\mathcal{L}|$ on the index construction time and index size.

The third part shows the limits of our approach and the effects of having LI with $k \leq \frac{N}{50}$.

The goal of any index in the end is to speed up queries compared to BFS . Hence we discuss this for any experiment. A query condition is a pair of the form $(\{True, False\}, |L|)$ where $L \subseteq \mathcal{L}$.

When we discuss the speed-ups over BFS in this section, we mean "total speed-ups". A total speed-up over a query condition is equal to the sum of all query times of BFS over that query condition over the sum of all query times of a different method over that query condition. We do wish to note that the individual speed-ups can be very different from this. An "individual speed-up" is the time that a query q took using BFS over the time that it took using a different method. We may have that the "total speed-up" is about 2.0 but that about 30% of the queries has an "individual speed-up" of at least 100. An "average speed-up" is the average over all individual speed-ups for all queries within a certain query condition.

6.1 Part 1: small graphs $(0 < |E| \leq 5,000)$

6.1.1 Datasets and methods

We used the methods PartialIndex , LI+OTH ($k = N/10, b = k/10$), LI+OTH+EXTv1 ($k = N/10, b = k/10$) Zou , NeighbourExchange , ClusteredExact (5 clusters) and DoubleBFS . NeighbourExchange used blocked mode, which improved the index construction time at the cost of a larger index size.

6.1.2 Index construction time (s) and size (MB)

Tables 6.1 and 6.2 show the results for all datasets and methods. The abbreviations for the methods are listed in Table **??**.

We see that Zou needs a considerable amount of time to build the index. The index size of Zou is almost identical to that of LI+OTH+EXTv1 with $k = N$ (last column). The small difference has to do with the use of the second extension by LI+OTH+EXTv1 . Especially in the cases of a uniform distribution we see that the construction time of Zou explodes.

Table 6.1: Index construction time (s) for the 7 methods and all datasets.

dataset	BFS	Zou	Clus	Nei	Par	L2	L12	DBFS
ERV1kD2L8uni	0.01	1,099.13	0.03	13.56	0.13	0.86	0.86	3.18
ERV1kD5L8uni	0.01	1,219.76	0.89	51.94	1.18	3.91	3.92	17.61
ff1k-0.2-0.4	0.01	43.48	0.06	3.38	0.14	0.06	0.06	0.22
plV1kL8a2.0exp	0.01	39.60	0.06	2.19	0.15	0.13	0.13	0.39
robots	0.01	11.99	0.07	1.49	0.07	0.06	0.06	0.11
V1kD2L12exp	0.01	39.96	0.03	3.15	0.07	0.03	0.03	0.19
V1kD2L8exp	0.01	41.48	0.03	3.51	0.06	0.03	0.03	0.20
V1kD2L8norm	0.01	308.75	0.04	6.22	0.12	0.17	0.17	0.55
V1kD2L8uni	0.01	502.83	0.05	6.04	0.15	0.25	0.25	0.75
V1kD5L8exp	0.01	191.39	0.17	9.74	0.38	0.28	0.27	1.01
V1kD5L8norm	0.01	611.15	0.30	17.53	0.91	0.53	0.53	2.60
V1kD5L8uni	0.01	908.33	0.37	19.15	1.01	0.69	0.69	3.44

Table 6.2: Index size (MB) for the 7 methods and all datasets.

dataset	BFS	Zou	Clus	Nei	Par	L2	L12	DBFS
ERV1kD2L8uni	0.03	26.64	0.04	36.49	1.53	3.87	3.88	26.76
ERV1kD5L8uni	0.08	88.54	9.50	85.58	8.00	9.47	9.49	88.68
ff1k-0.2-0.4	0.04	8.61	1.03	15.52	1.70	1.52	1.54	8.73
plV1kL8a2.0exp	0.03	7.02	0.65	15.30	1.67	1.23	1.24	7.15
robots	0.04	4.84	0.90	28.26	1.15	1.76	1.78	4.95
V1kD2L12exp	0.03	10.70	0.37	15.78	1.41	1.36	1.39	10.87
V1kD2L8exp	0.03	11.66	0.38	15.87	1.48	1.47	1.48	11.77
V1kD2L8norm	0.03	17.35	0.47	23.74	1.90	2.61	2.63	17.47
V1kD2L8uni	0.03	21.36	0.77	27.96	2.26	3.33	3.35	21.48
V1kD5L8exp	0.07	29.26	4.23	26.10	4.65	3.84	3.86	29.39
V1kD5L8norm	0.07	49.21	4.82	46.40	7.69	5.86	5.89	49.35
V1kD5L8uni	0.07	57.28	5.75	54.40	8.88	6.73	6.76	57.42

If we look at the execution logs for Zou for each of the datasets, we see that Zou already needed seconds (e.g. 7 seconds **V1kD2L8uni**) for building the index of each SCC. We know that checking for each triple $(L(p), p, d)$ whether path p is simple is relatively computationally intensive (step 2, see Section 5.1.2). Steps 7 and 8 took minutes in some cases. In these steps a lot of costly concatenation $\odot$ and $Prune()$ operators are used. Moreover, there is no clever ordering of the vertices used by these steps, unlike in step 6 in which a reversed topological order was used.

We think that in these last steps and in general a more efficient implementation should be possible. However, as the steps before these steps, i.e. steps 1 to 6, already take more time than LI or PartialIndex , we do not believe it can compete with the other methods. The ratio between Zou and DoubleBFS is at least 80 and might be much higher. Hence, we decide to refrain from using Zou in the follow-up parts of this experiment.

We also see that the index construction time of NeighbourExchange is second after Zou . NeighbourExchange is also too slow, mostly due to the fact that it cannot use some form of pruning.

In general we can see a strong difference both in terms of construction time (s) and index size (MB) when we look at the difference between the exponential, uniform and normal distribution.

6.1.3 Speed-ups achieved

We chose to only analyse the speed-ups of ClusteredExact , PartialIndex and LI (both approaches) and DoubleBFS compared to BFS , because NeighbourExchange and Zou have an index construction time that

is not competitive with the other approaches to start with. Table 6.3 shows the total speed-up per condition (query set and True- or False-queries) for ClusteredExact and PartialIndex and Table 6.4 does the same for LI . The results for ClusteredExact are staggering. A 0.01 in the table means that the average query

Table 6.3: Total speed-up per condition (query set and True/False-queries) for ClusteredExact and PartialIndex .

dataset	qs_1,true $\lvert\mathcal{L}\rvert/4$	qs_1,false	qs_2,true $\lvert\mathcal{L}\rvert/2$	qs_2,false	qs_3,true $\lvert\mathcal{L}\rvert-2$	qs_3,false
ClusteredExact						
ff1k-0.2-0.4	0.11	0.01	0.08	0.01	0.33	0.01
ERV1kD2L8uni	0.01	0.01	0.01	0.01	0.01	0.01
plV1kL8a2.0exp	0.01	0.01	0.01	0.01	0.01	0.01
V1kD2L12exp	0.01	0.01	0.01	0.01	0.01	0.01
V1kD2L8exp	0.01	0.01	0.01	0.01	0.01	0.01
V1kD2L8norm	0.01	0.01	0.01	0.01	0.01	0.01
V1kD2L8uni	0.01	0.01	0.01	0.01	0.01	0.01
V1kD5L8exp	0.03	0.01	0.01	0.01	0.02	0.01
V1kD5L8norm	0.01	0.01	0.02	0.01	0.04	0.01
V1kD5L8uni	0.01	0.01	0.01	0.01	0.02	0.01
robots	0.01	0.01	0.01	0.01	0.01	0.01
	.02	.01	.01	.01	.04	.01
PartialIndex						
ff1k-0.2-0.4	16.45	0.89	25.38	0.92	20.04	0.93
ERV1kD2L8uni	0.81	0.40	1.04	0.83	2.33	0.92
plV1kL8a2.0exp	5.90	0.93	9.43	0.95	16.61	0.95
V1kD2L12exp	2.85	0.92	7.31	0.94	32.55	0.96
V1kD2L8exp	5.44	0.93	15.97	0.98	24.93	1.08
V1kD2L8norm	2.05	0.82	3.10	0.89	3.56	0.92
V1kD2L8uni	1.40	0.77	3.87	0.85	11.65	0.91
V1kD5L8exp	18.97	0.89	16.72	0.80	40.05	0.85
V1kD5L8norm	10.54	0.73	18.90	0.82	17.79	0.85
V1kD5L8uni	5.83	0.75	19.08	0.85	23.21	0.87
robots	2.30	0.95	4.04	0.93	7.49	0.94
	6.59	.81	11.34	.88	18.20	.92

execution time of ClusteredExact was at least 100 times as slow as that of BFS . The quality of the clustering seems to play a big role here. From the logs we can see that a lot of direct and failed attempts were made to reach an outgoing port or to reach the target from an incoming port. When there is a lot of interconnection between the clusters, this approach fails.

The results for PartialIndex were promising looking at Tables 6.1 and 6.2, but the total speed-ups achieved are bad. Only for True-queries there is an actual speed-up. This is due to the fact that when we hit a landmark v' when answering query (v, w, L) we can either have a direct hit or do not need to continue searching from v'. PartialIndex does not have these advantages. The speed-ups for True-queries are also fairly low compared tot he speed-ups achieved in Table 6.4. When we look at Table 6.4 we see that the speed-ups for LI+OTH+EXTv1 are slightly above those of LI+OTH in some cases and LI+OTH is clearly better in the remaining cases. There is not a clear winner here.

The speed-ups achieved by DoubleBFS , i.e. LI with $k = N$, are way higher than all the other speed-ups. The actual query answer times are very low for LI+OTH+EXTv1 (many in the range of 10^{-7}. There might be some inaccuracy here. LI+OTH+EXTv1 can answer any query immediately and does not have to initialize for instance a queue or bitset for any query. These might be explanations as to why the difference is so big.

Table 6.4: Total speed-up per condition (query set and True/False-queries) for LI+OTH ($k = N/10$) and ($k = N$) and DoubleBFS .

method name	qs_1,true $\lvert\mathcal{L}\rvert/4$	qs_1,false	qs_2,true $\lvert\mathcal{L}\rvert/2$	qs_2,false	qs_3,true $\lvert\mathcal{L}\rvert-2$	qs_3,false
LI+OTH	(k=$N/10$,b=$k/10$)					
ff1k-0.2-0.4	50.35	67.78	82.62	63.69	78.95	72.48
ERV1kD2L8uni	1.49	1.45	9.39	5.83	23.70	10.23
plV1kL8a2.0exp	134.04	158.41	143.64	134.72	170.08	162.39
V1kD2L12exp	51.34	97.17	42.83	102.12	88.46	101.84
V1kD2L8exp	66.42	122.55	66.05	138.35	73.42	172.11
V1kD2L8norm	15.59	10.48	30.23	33.30	50.58	85.88
V1kD2L8uni	12.34	6.43	24.31	34.37	43.75	91.44
V1kD5L8exp	31.20	2.91	37.13	28.90	75.18	16.54
V1kD5L8norm	22.02	31.86	28.73	11.47	30.88	4.23
V1kD5L8uni	23.01	34.33	31.13	24.05	38.09	3.57
robots	63.67	35.84	126.30	112.40	137.61	244.54
	42.86	51.74	56.57	62.65	73.70	87.75
LI+OTH+EXTv1	(k=$N/10$,b=$k/10$)					
ff1k-0.2-0.4	54.69	51.79	85.21	51.80	73.40	51.90
ERV1kD2L8uni	1.51	1.47	9.37	5.94	24.00	10.96
plV1kL8a2.0exp	139.65	134.77	144.60	111.14	168.89	118.95
V1kD2L12exp	63.32	85.99	76.97	90.78	92.60	86.21
V1kD2L8exp	67.47	85.15	65.09	130.52	71.35	82.27
V1kD2L8norm	14.82	9.65	30.20	30.23	47.05	66.67
V1kD2L8uni	11.45	6.33	23.59	33.85	48.85	79.25
V1kD5L8exp	31.14	3.68	36.84	28.86	73.24	20.78
V1kD5L8norm	23.50	30.62	32.05	18.38	32.72	6.94
V1kD5L8uni	23.25	33.88	31.18	28.62	39.71	4.91
robots	61.14	27.07	117.69	85.42	133.44	250.62
	44.72	42.76	59.34	55.95	73.20	70.86
DoubleBFS						
ff1k-0.2-0.4	155.91	2,844.84	178.56	3,220.82	199.93	3,537.62
ERV1kD2L8uni	7.17	64.35	41.55	416.49	211.83	1,953.72
plV1kL8a2.0exp	307.42	2,745.11	334.78	2,577.43	359.59	3,153.61
V1kD2L12exp	163.63	2,538.65	205.62	3,140.89	239.10	3,315.53
V1kD2L8exp	142.05	2,536.86	136.78	3,059.71	192.43	3,907.76
V1kD2L8norm	45.45	413.86	106.00	1,209.13	159.92	2,418.49
V1kD2L8uni	39.08	328.03	101.76	1,243.88	128.82	2,453.83
V1kD5L8exp	193.94	5,292.36	180.60	1,628.22	250.92	3,836.84
V1kD5L8norm	143.23	2,081.43	181.40	5,600.57	175.60	7,547.58
V1kD5L8uni	114.19	2,311.32	180.97	6,473.33	256.51	8,372.60
robots	97.79	695.46	260.54	1,889.16	304.76	2,379.20
	128.16	1,986.57	173.50	2,769.05	225.40	3,897.88

The total speed-ups for ER-datasets and uniformly distributed datasets are much higher. This is due to the fact that ER-datasets have a more evenly distributed out-degree distribution and that uniformly distributed datasets allow for more minimal label sets between any pair of nodes.

We do wish to note that the speed-ups are total speed-ups. The speed-up per query can still differ. In the next section we will look into the speed-up per individual query as well.

6.2 Part 2: medium graphs ($5,000 < |E| \leq 500,000$)

6.2.1 Datasets and methods

In this experiment we included all datasets that have more than $5,000$ and edges but at most $500,000$ edges. The methods we used are: LI+OTH ($k = N/20, b = 20$), LI+OTH ($k = N/20, b = 0$), LI+OTH+EXTv1 ($k = N/20, b = 20$) and LI+OTH+EXTv1 ($k = N/10, b = 20$). We fixed the budget either to 20 or to 0. In the case $b = 0$, one could consider LI+OTH to be the same as LI .

For the datasets of the type Preferential-Attachment that have $25,000$ vertices and with a degree ≥ 3 and $L \geq 8$ we only ran LI+OTH+EXTv1 ($k = N/10, b = 20$)

6.2.2 Index construction time (s) and size (MB)

Table 6.5: Index construction time (s) for the 4 methods and all datasets. For the dataset **ERV125kD2L8uni** no index was produced within the 6 hours time limit. The table is divided into a few sections: the top one contains the real datasets, the second one the ER-datasets, the third one the Forest-Fire datasets, the fourth one the Power-Law datasets and the last one the Preferential-Attachment datasets.

dataset	BFS	L12($\frac{N}{20}$, 20)	L1($\frac{N}{20}$, 20)	L2($\frac{N}{20}$, 0)	L12($\frac{N}{10}$, 20)
advogato	0.01	0.84	0.82	0.80	1.48
yagoFacts-small	27.87	28.04	28.04	28.03	28.04
subeljCoraL8exp	0.01	5.27	5.16	5.09	8.94
arXivhepphL8exp	0.06	291.75	292.83	293.22	417.95
p2p-GnutellaL8exp	0.02	103.05	102.19	101.50	173.48
ERV5kD2L8uni	0.01	26.98	27.03	26.98	41.33
ERV25kD2L8uni	0.02	978.67	977.01	1,069.33	1,448.87
ERV125kD2L8uni	-	-	-	-	-
ff5k-0.2-0.4	0.01	1.05	1.03	1.02	1.58
plV5kL8a2.0exp	0.01	1.48	1.46	1.45	2.40
plV25L8ka2.0exp	0.03	30.44	30.37	31.08	50.17
V5kD2L8exp	0.01	0.36	0.35	0.34	0.61
V5kD2L8norm	0.01	3.49	3.46	3.45	5.09
V5kD2L8uni	0.01	5.51	5.55	5.48	7.78
V25kD2L8exp	0.01	9.81	9.65	9.48	16.54
V25kD2L8norm	0.02	110.03	110.09	109.99	156.99
V25kD2L8uni	0.01	157.78	157.49	157.38	216.81
V125kD2L8exp	0.03	262.30	267.49	254.91	445.85
V125kD2L8norm	0.03	2,944.69	2,956.99	2,953.46	4,182.90
V125kD2L8uni	0.03	4,341.67	4,380.90	4,372.79	6,080.10

The top section of Table 6.5 contains the real datasets. There is quite some difference when we look at the index construction times of these datasets, e.g. **subeljCoraL8exp** (23k edges) took roughly 5 seconds whereas **arXivheppL8exp** (34k edges) took way longer. A clear difference between these two is the out-degree distribution. **subeljCoraL8exp** has a more skewed out-degree distribution, i.e. a few nodes have a very high degree, whereas **arXivheppL8exp** has a more balanced out-degree distribution. We have seen this effect multiple times, e.g. the ER-datasets have an almost uniform out-degree distribution due to the random selection of the edges. The number of vertices is not a big determinant of the index size and construction time in this case.

The graph construction time of **yagoFacts-small** is quite high: 27.87 seconds. The only thing that makes this dataset very different from other datasets is the fact that it has $|\mathcal{L}| = 32$. However this does not explain why the graph construction time is this high.

The ER-datasets all have a uniform label distribution and a very evenly distributed out-degree. These datasets are known to be very difficult to process. This explains why we were unable to build an index within the

time limit of 6 hours (21,600 s) for dataset **ERV125kD2L8uni**. From its logs we could discover that the construction process was around 50% when the limit was reached and that the index had reached a size of 20GB. In general we see that the constructing the first 20% landmarks takes about 80% of the time. With a little extra bit of time it might have completed. However, this would have only been the $\frac{N}{20}$-variant. Looking at the fourth and fifth column for ER we can see that the $\frac{N}{10}$ can take up to 1.5 times more time.

Looking at the PA-datasets and the label set distribution in Table 6.5 we see a strong difference between the datasets with a uniform and normal label set distribution and the exponential distribution. There is a ratio of about 13 between the uniform and exponential counterparts for $|V| = 5$k, $|V| = 25$k and $|V| = 125$k.

Table 6.6: Index size (MB) for the 4 methods and all datasets. The sections in the table are the in Table 6.5.

dataset	BFS	L12($\frac{N}{20}$, 20)	L1($\frac{N}{20}$, 20)	L2($\frac{N}{20}$, 0)	L12($\frac{N}{10}$, 20)
advogato	0.82	27.57	27.46	27.44	54.60
yagoFacts-small	0.28	0.51	0.30	0.30	0.74
subeljCoraL8exp	1.46	54.94	54.67	54.67	109.86
arXivhepphL8exp	6.74	613.21	611.96	611.92	1,281.71
p2p-GnutellaL8exp	2.36	3,668.12	3,667.25	3,667.24	7,298.90
ERV5kD2L8uni	0.16	62.00	61.97	61.97	119.71
ERV25kD2L8uni	0.80	1,645.72	1,645.56	1,645.56	3,151.02
ERV125kD2L8uni	-	-	-	-	-
ff5k-0.2-0.4	0.20	17.08	17.03	16.93	32.22
plV5kL8a2.0exp	0.25	17.38	17.29	17.29	32.71
plV25L8ka2.0exp	1.44	421.25	420.68	420.68	811.87
V5kD2L8exp	0.15	18.12	18.08	17.99	35.22
V5kD2L8norm	0.15	39.09	39.05	38.94	71.66
V5kD2L8uni	0.15	46.39	46.35	46.25	84.70
V25kD2L8exp	0.79	437.62	437.38	437.16	857.19
V25kD2L8norm	0.79	992.47	992.25	992.00	1,830.24
V25kD2L8uni	0.79	1,169.63	1,169.43	1,169.17	2,121.92
V125kD2L8exp	3.99	10,787.30	10,786.07	10,785.39	21,291.71
V125kD2L8norm	3.99	24,381.91	24,380.83	24,380.22	45,072.80
V125kD2L8uni	3.99	29,044.04	29,043.01	29,042.46	53,146.00

The top section of Table 6.6 shows the real datasets. An interesting observation here is that construction time is not directly related to index size. For example **p2p-GnutellaL8exp** has the largest index size in the last column 7.3GB roughly, which took less than 2 minutes to build. In contrast, **arXivhepphL8exp** took almost 5 minutes and ended up with a smaller index. An explanation for this could be in the fact that **p2p-GnutellaL8exp** has more skewed degree distributions than **arXivhepphL8exp** . Landmarks are set to be those vertices with a high total degree. When $K < k$ landmarks have already been constructed and a landmark $v' \in V'$ needs to be constructed, the chances are higher in a graph like **p2p-GnutellaL8exp** (with a skewed out-degree distribution) that it will stumble onto one of those K landmarks and can use forward pruning than in a graph like **arXivhepphL8exp** . This is because the landmarks are constructed according to an ordering in which the landmarks with the highest total degree are built first.

We can see that the index size grows very steeply for the ER-datasets like the construction time. Dataset **ERV25kD2L8uni** has a larger index size (3.1GB) compared to its PA-counterpart (2.1GB) looking at the last column.

Looking at the PA-datasets and the label set distribution in Table 6.6 we see a strong difference between the datasets with a uniform and normal label set distribution and the exponential distribution. There is a ratio of about 2.4 between the uniform and exponential counterparts for $|V| = 5$k, $|V| = 25$k and $|V| = 125$k.

The rate of growth of the PA-datasets is also concerning. The index size of a PA-dataset from 1k to 125k

with an exponentially distributed label set grows from: $11.77 \rightarrow 35.22 \rightarrow 857.19 \rightarrow 21,291.92$. This makes for a rate of growth of roughly 24, except for the first step. In case of a normal distribution the numbers are: $17.47 \rightarrow 71.66 \rightarrow 1,830.24 \rightarrow 45,072.80$. This makes for a rate of growth of roughly 25, except for the first step. In case of a uniform label set distribution we get: $21.48 \rightarrow 84.7 \rightarrow 1,830.24 \rightarrow 53,146$. This makes for an average rate of growth of 27 for the last two steps. This shows that a dataset X for which the labels stem from a uniform distribution has a slightly higher rate of growth than when they would stem from an exponential distribution. The rate of growth regardless of the label set distribution exposes that we cannot build a full $k = N$-index for larger N, as we expected.

Table 6.7: Index construction time (s) for the datasets of the type 'Preferential-Attachment' having $25,000$ edges and a degree of either $3, 4$ or 5 and at least 8 labels using LI+OTH+EXTv1 ($k = N/20, b = 20$).

Degree	$\|\mathcal{L}\| = 8$	$\|\mathcal{L}\| = 10$	$\|\mathcal{L}\| = 12$	$\|\mathcal{L}\| = 14$	$\|\mathcal{L}\| = 16$
3	33.75	70.92	223.31	1,058.74	4,219.61
4	72.44	215.71	721.78	3,554.97	-
5	113.46	353.74	1,414.50	6,527.27	-

Table 6.8: Index size (MB) for the datasets of the type 'Preferential-Attachment' having $25,000$ edges and a degree of either $3, 4$ or 5 and at least 8 labels using LI+OTH+EXTv1 ($k = N/20, b = 20$).

Degree	$\|\mathcal{L}\| = 8$	$\|\mathcal{L}\| = 10$	$\|\mathcal{L}\| = 12$	$\|\mathcal{L}\| = 14$	$\|\mathcal{L}\| = 16$
3	736.01	953.55	1,496.42	2,783.56	4,219.61
4	1,022.86	1,566.40	2,677.61	5,404.49	-
5	1,250.53	2,004.95	3,720.64	7,238.79	-

Index construction time (s) and size (MB) for $D \geq 3$ and $|\mathcal{L}| \geq 8$

For the datasets of the type 'Preferential-Attachment' (PA) having $25,000$ vertices and a degree of either $3, 4$ or 5 and at least 8 labels and an exponential label distribution, we only ran LI+OTH+EXTv1 ($k = N/20, b = 20$). Table 6.7 and 6.8 show the index construction time (s) and index size (MB) for these datasets. In two cases, we were again unable to get an index built within the 6 hours time limit. An increase in terms of the degree or the label set size can have a huge impact on the construction time and to a lesser extent the index size. Especially the label set size, $|\mathcal{L}|$, has an impact. There is an exponential growth in the index construction time when we look at the label set size. This exponential growth is also visible in the index size, but is much less steep, i.e. below 2.

An explanation for the effect of the label set size is in the fact that the growth of the index size and construction time can both be expressed in a form exponential in the label set size, i.e. $O(2^{|\mathcal{L}|})$. The results though are similar to the ones in Bonchi et al. [1]. Only their approximate method, ChromLand was able to deal with $|\mathcal{L}| \geq 10$.

6.2.3 Speed-ups achieved

First we discuss the total speed-up per dataset and per query condition: (True/False, $\{|\mathcal{L}|/4, |\mathcal{L}|/2, |\mathcal{L}| - 2\}$). Table 6.9 shows the mean query execution time (ms) of BFS for these approaches. We can see that the query execution times are typically low, often in the order of magnitude of 10^{-4}. There can be random differences in the query answer time, because the query generation process by itself is random.

In Tables 6.10, 6.9 and 6.11 we see the total speed-up for queries of the first, second and third query sets respectively. We can compare the different methods against each other by looking at the respective columns.

When we look at the first two columns against the third and fourth column in the three tables, we can't observe any advantage of using the first extension. Like in the first part of the experiment the results for not using the first extension are in general better. In a few cases there is a small improvement, e.g. **advogato** in Table 6.10. In the experiments leading up to this final experiment we found some gain in using the first extension,

Table 6.9: Mean query execution time (ms) of BFS for all datasets per query condition.

method name	qs_1,true $\lvert\mathcal{L}\rvert/4$	qs_2,true $\lvert\mathcal{L}\rvert/2$	qs_3,true $\lvert\mathcal{L}\rvert-2$	qs_1,false	qs_2,false	qs_3,false
ff5k-0.2-0.4	0.067	0.043	0.044	0.183	0.125	0.169
ERV5kD2L8uni	0.000	0.008	0.079	0.000	0.013	0.121
ERV25kD2L8uni	0.001	0.052	0.389	0.002	0.058	0.674
plV5kL8a2.0exp	0.146	0.155	0.160	0.134	0.173	0.220
plV25L8ka2.0exp	0.841	0.862	0.917	0.823	0.929	1.341
V5kD2L8exp	0.059	0.048	0.049	0.154	0.177	0.191
V5kD2L8norm	0.018	0.042	0.037	0.025	0.076	0.139
V5kD2L8uni	0.007	0.029	0.033	0.007	0.063	0.135
V25kD2L8exp	0.207	0.170	0.147	0.807	0.893	0.877
V25kD2L8norm	0.303	0.143	0.139	0.195	0.385	0.738
V25kD2L8uni	0.041	0.141	0.115	0.049	0.367	0.749
V125kD2L8exp	0.304	0.260	0.279	4.243	3.901	5.066
V125kD2L8norm	0.235	0.267	0.253	0.402	1.856	3.786
V125kD2L8uni	0.101	0.264	0.163	0.251	1.662	4.008
advogato	0.203	0.243	0.246	0.297	0.383	0.370
yagoFacts-small	0.001	0.001	0.002	0.000	0.004	0.004
subeljCoraL8exp	0.225	0.240	0.261	0.337	0.378	0.420
arXivhepphL8exp	1.305	1.077	0.937	1.758	2.139	2.606
p2p-GnutellaL8exp	0.427	0.297	0.302	1.965	2.185	2.241

but this was for some specific large datasets with few landmarks. Therefore we will keep using the first extension in future experiments and keep using it. Moreover because we already see asymmetry between the True- and False-queries and we wish to see this asymmetry reduced.

The difference between LI+OTH with a budget b of 20 and LI+OTH with $b = 0$ is larger though. One could consider the latter one simply as LI . Looking at the third and fifth column of Table 6.10, Table 6.9 and Table 6.11, i.e. the True-queries, there is a significant difference in favour of LI+OTH with $b = 20$ in all cases. Setting the budget from $b = 0$ to $b = 20$ can double the speed-up for True-queries in some cases. For the False-queries, i.e. columns 4 and 6, we the opposite effect. The total speed-ups are higher for the $b = 0$ condition in this case, although the differences are in most cases less than 5%.

The question on the ideal budget size remains to be answered still. We can imagine that a very large budget might even slow down queries, e.g. when $|L|$ is low, but might speed up considerably when $|L|$ is high and one of the b entries gives a direct hit. From the experiments leading up to this experiment we discovered that a relatively low budget works best.

Adding more landmarks, i.e. going from $\frac{N}{20}$ to $\frac{N}{10}$, improves the results significantly. In many cases the speed-up of True-queries is roughly doubled. The smaller datasets exhibit this effect to a lesser extent. For the False-queries the results can vary strongly. In the case of **p2p-GnutellaL8exp** in Table 6.10 we see an increase of 400 in the speed-up, whereas we only see a very small effect for **V25D2L8uni**.

When we look at the speed-ups for datasets that have an exponential label set distribution, e.g. **V5kD2L8exp**, and compare these against the speed-ups for their normal or uniform counterparts, e.g. **V5kD2L8uni** and **V5kD2L8norm**, we see that the speed-ups in the first case are higher. In the datasets with an exponential label distribution we have that one or two labels can dominate a majority of the edges. The presence of these labels in a True-query (v, w, L) can speed-up resolving the query as from a vertex $v \in V$ we can find with a high probability a landmark v' which in turn can with a high probability find w. The absence of these labels in a False-query on the contrary can eliminate a large part of the edges making the speed-up over BFS , which is label set distribution agnostic, higher. The speed-ups for PL-datasets and **subeljCoraL8exp** and **p2p-GnutellaL8exp** are relatively high. Again these are datasets with a skewed out-degree distribution which favours the landmarked

Table 6.10: Total speed-up for query conditions (True, $|\mathcal{L}|/4$) and (False, $|\mathcal{L}|/4$).

dataset	LI+OTH+EXTv1		LI+OTH		LI+OTH+EXTv1		LI+OTH+EXTv1	
	$(20, N/20)$		$(20, N/20)$		$(0, N/20)$		$(20, N/10)$	
ff5k-0.2-0.4	307.78	185.77	311.90	288.89	134.57	293.73	286.96	345.31
ERV5kD2L8uni	0.99	1.06	1.00	1.11	1.01	1.26	1.09	1.16
ERV25kD2L8uni	2.87	3.14	2.98	3.11	3.04	3.56	3.44	3.39
plV5kL8a2.0exp	651.29	319.53	661.20	375.38	346.68	423.30	701.33	662.25
plV25L8ka2.0exp	2,309.31	1,279.65	2,425.09	1,441.89	1,253.13	1,549.01	2,691.57	2,127.51
V5kD2L8exp	273.93	215.95	278.27	294.13	155.54	319.45	315.30	503.51
V5kD2L8norm	73.07	73.16	71.44	76.42	32.25	76.42	72.68	78.02
V5kD2L8uni	23.78	21.94	25.13	24.35	15.00	25.31	27.66	24.64
V5kD5L8exp	259.50	66.00	250.83	78.27	126.80	85.98	224.72	80.78
V25kD2L8exp	699.84	978.21	706.24	1,223.47	347.02	1,240.32	741.49	1,392.55
V25kD2L8norm	783.50	314.64	784.16	356.63	392.31	311.33	811.60	342.90
V25kD2L8uni	112.71	93.06	111.74	94.05	63.07	92.97	122.59	97.22
V125kD2L8exp	416.95	3,104.94	417.12	4,100.90	286.31	4,150.62	434.50	4,679.52
V125kD2L8norm	323.78	451.99	338.66	464.71	206.92	423.22	341.30	474.97
V125kD2L8uni	148.44	263.33	143.24	277.81	93.24	255.58	165.21	288.98
advogato	586.82	4.74	584.11	4.27	375.09	4.27	719.30	28.33
yagoFacts-small	4.55	1.83	4.52	2.08	3.32	2.13	5.27	1.95
subeljCoraL8exp	615.57	234.99	631.11	238.86	311.46	247.32	681.96	386.32
arXivhepphL8exp	687.32	4.97	706.53	4.84	627.12	4.83	1,183.89	20.56
p2p-GnutellaL8exp	1,027.91	5.25	1,044.45	5.49	576.22	5.47	1,346.45	2,020.55
	452.63	466.02	460.16	586.53	260.78	595.12	525.91	771.16

approach. In general the speed-ups increase with the size of the graph and are a reflection on the time and memory invested in building the index. Hence one should be able to lower the ratio $\frac{k}{N}$ for larger graphs and still acquire a decent speed-up.

Speed-ups for $D \geq 3$ and $|\mathcal{L}| \geq 8$

Table 6.13 shows the speed-ups for 'Preferential Attachment' datasets with $25,000$ edges, a degree ≥ 3 and $|\mathcal{L}| \geq 8$ and an exponential label distribution. The speed-ups are high.

Speed-up per query

The speed-ups discussed in the previous section were "total speed-ups". One should be careful here. The actual speed-up per query could be very different.

In Figure 6.1 we can see the speed-up per query for **p2p-GnutellaL8exp** and the third query set, i.e. with $|L| = |\mathcal{L}| - 2$, using LI $12(k = \frac{N}{10}, b = 20)$. The total speed-ups (Table 6.12) are 771.22 and 646.22. About 55% of the True-queries and 93% of the False-queries are faster than this. In the figure we can see a steep increase for the False-queries around 50%. This could have to do with the way the queries are generated, having some queries that are in the relative proximity of landmarks.

Table 6.11: Total speed-up for query conditions (True, $|\mathcal{L}|/2$) and (False, $|\mathcal{L}|/2$).

dataset	LI+OTH+EXTv1		LI+OTH		LI+OTH+EXTv1		LI+OTH+EXTv1	
	$(20, N/20)$		$(20, N/20)$		$(0, N/20)$		$(20, N/10)$	
ff5k-0.2-0.4	164.73	150.02	176.95	216.47	76.16	232.80	196.63	370.89
ERV5kD2L8uni	11.23	12.74	11.47	13.16	10.38	13.83	17.66	24.97
ERV25kD2L8uni	55.11	37.42	55.14	37.46	47.56	39.47	84.21	72.92
plV5kL8a2.0exp	757.26	349.79	756.66	558.84	395.94	591.28	818.06	516.40
plV25L8ka2.0exp	2,491.84	1,239.38	2,598.33	1,501.20	1,342.06	1,586.26	3,074.60	1,989.90
V5kD2L8exp	217.68	239.92	226.79	360.22	109.30	376.70	225.93	342.46
V5kD2L8norm	156.69	166.32	164.23	172.24	63.92	183.34	173.58	190.85
V5kD2L8uni	131.64	160.49	127.38	167.17	76.06	190.87	123.10	179.01
V5kD5L8exp	150.14	22.46	157.36	21.59	81.10	22.73	192.73	182.00
V25kD2L8exp	534.99	660.86	532.77	1,005.91	256.72	1,030.47	589.99	1,378.69
V25kD2L8norm	553.15	626.96	541.12	666.18	300.78	545.65	553.74	661.91
V25kD2L8uni	353.07	554.60	349.54	598.65	183.86	590.92	389.71	646.90
V125kD2L8exp	395.02	1,753.97	397.41	3,519.16	265.31	3,585.02	410.49	3,129.41
V125kD2L8norm	352.67	1,851.65	358.91	1,909.63	225.81	1,755.74	362.25	2,004.46
V125kD2L8uni	324.63	1,488.86	328.73	1,584.35	213.34	1,439.30	409.34	1,616.89
advogato	696.33	3.39	689.43	2.89	483.19	2.89	976.80	14.66
yagoFacts-small	3.86	9.23	4.08	10.84	2.52	10.65	4.73	9.59
subeljCoraL8exp	703.14	238.04	728.82	249.90	312.11	249.98	825.56	453.70
arXivhepphL8exp	959.57	5.45	981.89	5.01	610.50	5.00	1,662.81	21.73
p2p-GnutellaL8exp	668.38	2.73	672.63	2.94	401.28	2.95	771.22	646.25
	459.59	578.57	468.49	707.27	255.99	703.59	552.39	832.52

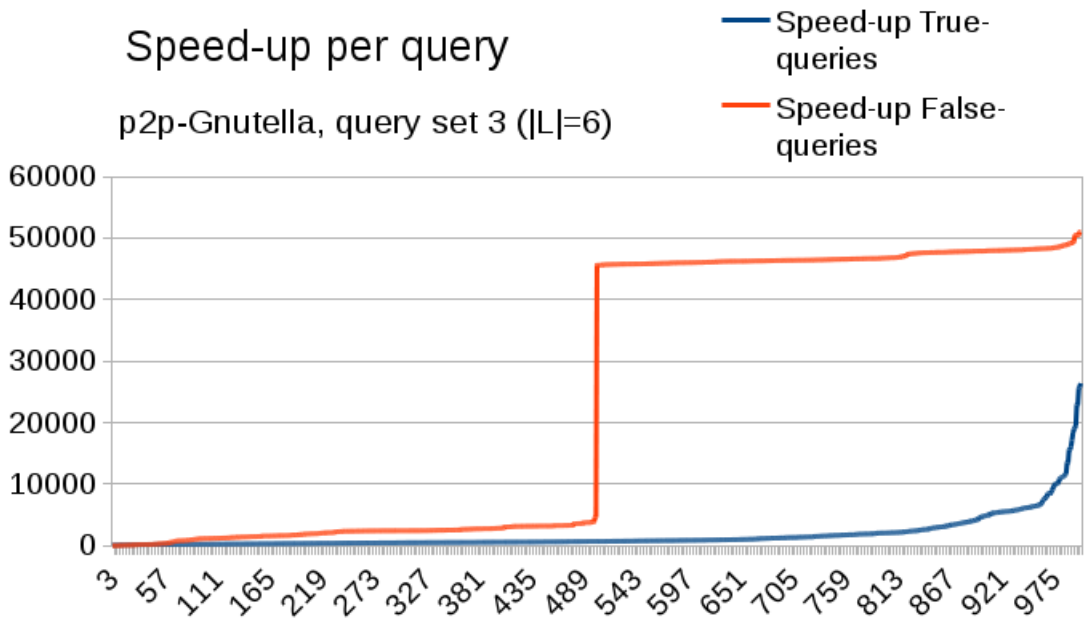


Figure 6.1: Speed-up per query (sorted in ascending order) for **p2p-GnutellaL8exp** and the third query set using LI+OTH+EXTv1 ($k = N/10, b = 20$).

Table 6.12: Total speed-up for query conditions (True, $|\mathcal{L}|-2$) and (False, $|\mathcal{L}|-2$).

dataset	LI+OTH+EXTv1		LI+OTH		LI+OTH+EXTv1		LI+OTH+EXTv1	
	$(20, N/20)$		$(20, N/20)$		$(0, N/20)$		$(20, N/10)$	
ff5k-0.2-0.4	217.50	158.84	202.46	277.44	104.16	292.53	194.98	390.77
ERV5kD2L8uni	163.49	5.46	166.75	5.29	106.43	5.34	225.74	76.79
ERV25kD2L8uni	610.40	4.72	604.85	4.72	393.05	4.73	932.80	389.21
plV5kL8a2.0exp	781.53	308.26	786.45	464.32	413.91	498.58	779.24	470.15
plV25L8ka2.0exp	2,694.00	1,137.77	2,832.51	1,649.38	1,361.76	1,673.05	2,846.10	2,020.66
V5kD2L8exp	232.45	225.10	246.24	234.68	113.71	247.78	252.81	414.74
V5kD2L8norm	164.18	192.42	169.53	212.75	62.33	210.08	167.39	302.03
V5kD2L8uni	139.82	326.93	147.65	324.28	67.04	370.80	142.46	361.71
V5kD5L8exp	180.94	278.20	169.91	273.02	74.39	314.76	200.15	399.34
V25kD2L8exp	603.90	212.05	616.39	210.77	307.41	211.70	642.46	1,384.24
V25kD2L8norm	370.65	953.96	389.75	1,020.88	168.51	942.32	405.34	1,151.69
V25kD2L8uni	315.86	897.93	311.52	950.25	160.47	1,003.63	335.32	1,823.13
V125kD2L8exp	381.12	3,119.93	386.20	3,233.09	231.98	3,282.21	475.99	5,386.13
V125kD2L8norm	334.58	4,533.95	340.76	4,828.57	221.42	4,793.20	342.29	5,515.67
V125kD2L8uni	189.58	3,179.83	187.02	3,308.09	125.31	3,181.34	213.12	3,863.40
advogato	1,032.94	5.01	1,027.80	3.69	625.76	3.69	1,207.44	21.32
yagoFacts-small	8.75	10.67	9.11	11.40	4.44	12.67	9.00	10.05
subeljCoraL8exp	788.96	142.83	812.33	140.80	379.78	141.93	947.71	409.45
arXivhepphL8exp	865.54	4.52	902.63	4.31	563.99	4.30	1,597.88	15.14
p2p-GnutellaL8exp	759.17	4.09	794.94	4.36	439.87	4.37	846.21	770.44
	510.51	784.25	522.49	1,035.80	525.91	771.16	591.83	1,401.42

Table 6.13: Total speed-ups for all 6 query conditions and 'Preferential-Attachment' datasets using LI+OTH+EXTv1 ($k = N/20, b = 20$).

degree,$\lvert\mathcal{L}\rvert$	qs_0,true	qs_0,false	qs_1,true	qs_1,false	qs_2,true	qs_2,false
	$\lvert\mathcal{L}\rvert/4$		$\lvert\mathcal{L}\rvert/2$		$\lvert\mathcal{L}\rvert-2$	
$3, 8$	575.65	34.72	491.32	295.45	565.15	711.61
$3, 10$	629.00	2.38	540.14	307.73	571.57	816.15
$3, 12$	666.71	21.72	724.43	393.65	585.70	957.84
$3, 14$	710.23	6.94	682.77	254.04	630.65	846.47
$3, 16$	977.86	3.00	535.27	10.50	780.25	40.88
$4, 8$	550.42	113.30	628.94	8.30	649.86	129.08
$4, 10$	590.96	16.92	508.76	50.65	647.62	485.68
$4, 12$	595.97	3.82	605.84	514.64	445.91	119.14
$4, 14$	552.87	62.47	497.39	518.51	493.54	28.38
$4, 16$	-	-	-	-	-	-
$5, 8$	778.94	297.72	614.06	699.38	795.42	64.36
$5, 10$	682.26	147.58	417.01	827.39	682.71	16.51
$5, 12$	550.57	5.20	525.59	772.23	436.84	14.68
$5, 14$	664.51	3.22	790.71	640.55	613.85	11.40
$5, 16$	-	-	-	-	-	-

Table 6.14: The datasets included in this part of the experiment with k expressed as the fraction of the number of vertices N.

Dataset	k	N
jmd	$N/50$	486,320
citeSeerL8exp	$N/50$	384,414
NotreDameL8exp	$N/50$	325,730
soc-sign-epinionsL8exp	$N/50$	131,828
socSlashdotL8exp	$N/50$	82,168
TwitterL8exp	$N/100$	465,018
webBerkstanL8exp	$N/100$	685,231
webStanfordL8exp	$N/100$	281,904
V125kD5L8exp	$N/100$	125,000
pl125ka2.0L8exp	$N/100$	125,000
webGoogleL8exp	$N/250$	875,713
zhishihudongL8exp	$N/250$	2,452,715
usPatentsL8exp	$N/250$	3,774,769
V625kD5L8exp	$N/250$	625,000
pl625ka2.0L8exp	$N/250$	625,000
socPokec	$N/500$	1,632,803

6.3 Part 3: large graphs ($|E| > 500,000$)

Table 6.15 shows the datasets divided into four categories where k is either $\frac{N}{50}$, $\frac{N}{100}$, $\frac{N}{250}$ or $\frac{N}{500}$. These distinctions were made to not let exceed any of the datasets the 6 hours time limit and the 128GB memory limit while still being able to provide a reasonable speed-up.

We used LI+OTH with $b = 20$ and LI+OTH+EXTv1 for $N/50$ both with ($b = 20$ and $b = 40$) only to demonstrate the effect of the first extension here. For the remainder we only used LI+OTH+EXTv1 with ($b = 20$ and $b = 40$). This is to see whether a larger budget works better or not.

6.3.1 LI+OTH+EXTv1 : Index construction time and size

In Table 6.15 we see the index construction time (s) and size (MB) for several datasets and LI+OTH+EXTv1 ($b = 20$) and ($b = 40$). We excluded the data for LI+OTH from the top section of the table as it is very similar to LI+OTH+EXTv1 .

There is very little to no difference between $b = 20$ and $b = 40$. This surprising as any non-landmarked vertex needs to do a double amount of work in the second case. However when $b = 40$ any non-landmark node that has been indexed can be used in forward propagation as can any landmark node. Hence with $b = 40$ there are more pruning opportunities cutting down construction time slightly in some cases.

socPokec did not complete because it exceeded the maximum memory size 128GB. By then it had built around 70% of the landmarks within around 16,000 seconds, which is at roughly 75% of the maximal time.

jmd has a very small index, because it is acyclic. The same holds for **usPatentsL8exp** . A large chunk of the index size for both is due to the graph size as well. It is difficult to generate difficult queries for these kinds of datasets, i.e. queries for which BFS needs to discover let's say at least 10% of the vertices, as we choose a random starting point for each query which might turn out to be the root of a very small subtree.

The datasets in the upper section of Table 6.15 have their indices built within 800 seconds each. This is in strong contrast with the datasets in the second and third section of the table. Each of these datasets has a more skewed out-degree distribution than in the other parts. The datasets in the third section have a large number of edges as well.

Figures 6.2 and 6.3 show the evolution of the part of the index construction in which we build all landmarks of datasets **plV625ka2.0L8exp** and **webGoogleL8exp** . For the size we see a roughly linear growth for both datasets. For the time there is a difference between the two datasets. **plV625ka2.0L8exp** grows linearly

Table 6.15: Index construction time (s) and index size (MB) for all the datasets involved.

Dataset	Time (s)	Size (MB)	Time (s)	Size (MB)
	LI+OTH+EXTv1 $b = 20$		LI+OTH+EXTv1 $b = 40$	
jmd	468.91	22.55	538.28	21.74
NotreDameL8exp	743.47	7,390.52	740.82	7,387.86
citeSeerL8exp	179.4	2,120.12	180.25	2,154.19
soc-sign-epinionsL8exp	359.05	3,426.55	358.73	3,425.45
socSlashdotL8exp	258.01	3,686.17	257.68	3,684.49
plV125ka2.0L8exp	158.31	1,988.32	157.83	1,987.27
V125kD5L8exp	1,483.89	12,395.29	1,483.46	12,394.51
TwitterL8exp	1,526.82	17,597.71	1,527.14	17,588.68
webBerkstanL8exp	8,555.18	54,040.7	8,549.91	54,034.57
webStanfordL8exp	3,929.01	24,818.4	3,926.55	24,816.83
zhishihudongL8exp	9,971.12	29,738.07	9,958.60	29,736.57
webGoogleL8exp	8,351.06	53,923.01	8,347.77	53,919.79
V625kD5L8exp	12,400.35	66,290.43	12,404.22	66,644.64
plV625ka2.0L8exp	1,955.14	19,446.65	1,960.67	19,441.35
usPatentsL8exp	7,539.27	705.02	7,503.74	563.98
socPokec	-	-	-	-
Average	3,858.6	19,839.3	3,859.71	19,853.46

whereas **webGoogleL8exp** has a stronger growth in the beginning which grows nearly flat for the last 5% of the landmarks. We must note that the index construction time of **webGoogleL8exp** is much higher than that of **plV625ka2.0L8exp**.

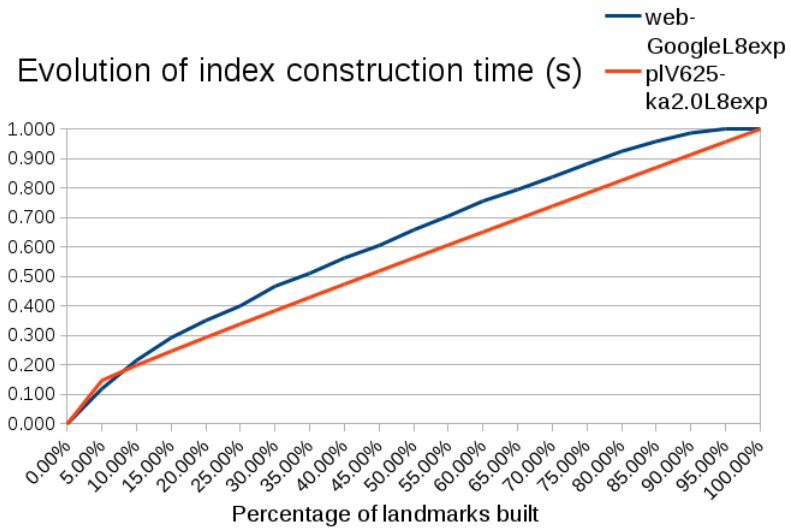


Figure 6.2: On the y-axis we see the percentage of the total time needed to build all landmarks against the percentage of landmarks (LI+OTH+EXTv1) that have been built on the x-axis. This shows the index growth over time of datasets **plV625ka2.0L8exp** and **webGoogleL8exp** .

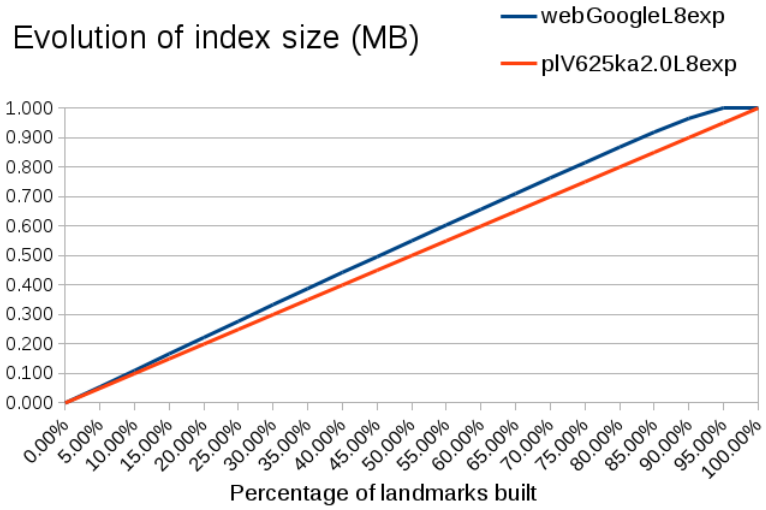


Figure 6.3: On the y-axis we see the percentage of the total size needed to build all landmarks against the percentage of landmarks (LI+OTH+EXTv1) that have been built on the x-axis. This shows the index growth over time of datasets **plV625ka2.0L8exp** and **webGoogleL8exp** .

6.3.2 LI+OTH+EXTv1 : Speed-ups

Table 6.16: Mean query execution time (ms) of BFS for all datasets per query condition.

dataset	qs_1,true $\lvert\mathcal{L}\rvert/4$	qs_2,true $\lvert\mathcal{L}\rvert/2$	qs_3,true $\lvert\mathcal{L}\rvert-2$	qs_1,false	qs_2,false	qs_3,false
jmd	0.003	0.003	0.003	0.003	0.003	0.003
NotreDameL8exp	0.237	0.183	0.194	0.931	1.826	1.070
soc-sign-epinionsL8exp	3.818	5.166	4.786	6.930	8.610	9.846
socSlashdotL8exp	3.355	5.345	3.547	7.666	10.641	9.047
citeSeerL8exp	0.861	1.030	1.254	1.582	2.122	2.619
plV125ka2.0L8exp	4.909	4.780	5.171	4.720	5.773	7.466
TwitterL8exp	5.608	5.049	5.826	7.873	7.166	1.785
V125kD5L8exp	0.281	0.165	0.221	6.764	5.380	5.745
webBerkstanL8exp	5.649	5.051	3.464	48.107	30.779	48.158
webStanfordL8exp	4.723	3.891	4.694	16.000	12.346	20.791
V625kD5L8exp	1.065	1.240	1.424	7.304	34.741	75.612
plV625ka2.0L8exp	45.528	47.592	47.831	52.401	63.567	70.526
usPatentsL8exp	0.324	0.407	0.422	0.332	0.416	0.391
webGoogleL8exp	31.727	26.746	26.052	90.562	50.733	89.902
zhishihudongL8exp	10.149	13.326	11.869	36.621	34.458	34.125
Average	7.882	7.998	7.784	19.186	17.904	25.139

Table 6.16 shows the query times of BFS . Table 6.17 shows the total speed-ups for the datasets for which we set $k = \frac{N}{50}$ and the methods LI+OTH+EXTv1 with a budget $b = 20$ (first row), LI+OTH+EXTv1 with $b = 40$ (second row) and LI+OTH with $b = 20$ (third row).

jmd is a special case. It is acyclic, like **usPatentsL8exp** . We expected the construction time of this dataset to be much lower, because each node can only reach a fraction of the nodes. This also makes the query times much lower. This can be seen in Table 6.16.

Table 6.17: Speed-ups for the methods LI+OTH+EXTv1 with a budget $b = 20$ (first row), LI+OTH+EXTv1 with $b = 40$ (second row) and LI+OTH with $b = 20$ (third row).

dataset	qs_1,true $\lvert\mathcal{L}\rvert/4$	qs_1,false	qs_2,true $\lvert\mathcal{L}\rvert/2$	qs_2,false	qs_3,true $\lvert\mathcal{L}\rvert-2$	qs_3,false
jmd	7.18	1.27	2.63	1.21	1.16	0.98
	6.58	1.08	2.39	1.08	1.14	0.92
	7.22	1.28	2.63	1.22	1.18	0.98
NotreDameL8exp	142.24	43.92	107.23	31.07	113.35	16.86
	135.8	17.54	105.9	8.61	110.29	12.81
	143.42	17.81	108.08	8.74	113.29	13.03
citeSeerL8exp	382.66	123.29	440.74	98.4	552.45	84.3
	254.44	86.68	394.09	94.84	571.69	81.23
	386.32	123.75	443.33	98.34	563.44	84.02
soc-sign-epinionsL8exp	5,372.24	4.77	6,973.25	4.36	6,801.43	3.4
	5,149.79	4.48	6,417.96	2.94	6,132.68	2.9
	5,364.51	4.57	7,019.03	3.01	6,780.85	2.99
socSlashdotL8exp	5,302.07	2.42	9,942.41	2.46	6,666.50	3.98
	4,215.56	2.09	9,016.06	1.95	6,137.54	2.58
	5,289.85	2.15	10,110.35	2.01	6,647.29	2.67
Average	2,241.28	35.13	3,493.25	27.5	2,826.98	21.9
	1,952.43	22.37	3,187.28	21.88	2,590.67	20.09
	2,238.26	29.91	3,536.68	22.66	2,821.21	20.74

NotreDameL8exp has decent speed-ups in all query conditions. There is a clear difference between the first row (LI+OTH+EXTv1 $b = 20$) and the third row here (LI+OTH $b = 20$). The first extension pays off here clearly. However adding more budget to each non-landmark node has a negative effect. Similar results can be seen for **soc-sign-epinionsL8exp** and **socSlashdotL8exp** .

Table 6.18: Speed-ups for the methods LI+OTH+EXTv1 with a budget $b = 20$ (first row), LI+OTH+EXTv1 with $b = 40$ (second row) and LI+OTH with $b = 20$ (third row) and the datasets for which we set $k = \frac{N}{100}$.

dataset	qs_1,true $\lvert\mathcal{L}\rvert/4$	qs_1,false	qs_2,true $\lvert\mathcal{L}\rvert/2$	qs_2,false	qs_3,true $\lvert\mathcal{L}\rvert-2$	qs_3,false
plV125ka2.0L8exp	6,853.96	6.43	7,296.94	7.94	8,102.48	9.42
	6,382.26	6.29	7,260.51	7.85	7,818.16	9.26
TwitterL8exp	56,020.11	27,872.16	50,487.78	71,657.92	58,264.06	17,852.12
	56,079.89	22,972.38	50,326.67	71,657.92	58,264.06	17,852.12
V125kD5L8exp	336.24	1.22	244.2	1.91	291.17	10.55
	263.19	1.17	222.46	1.29	256.7	9.34
webBerkstanL8exp	1,437.66	5.98	1,217.78	4.41	905.13	4.19
	1,267.31	5.83	1,135.14	4.3	870	4.07
webStanfordL8exp	3,299.61	25.45	2,566.25	22.88	3,099.08	14.56
	3,266.36	21.83	2,700.26	19.45	3,188.28	12.02
Average	13,589.52	5,582.25	12,362.59	14,339.01	14,132.38	3,578.17
	13,451.8	4,601.5	12,329.01	14,338.16	14,079.44	3,577.36

Table 6.18 shows the speed-ups for the datasets for which we had set $k = \frac{N}{100}$. The speed-ups for **TwitterL8exp** are very high, compared to the other speed-ups and even the speed-ups before. The reason for this is

that **TwitterL8exp** has a very skewed out-degree distribution and that we select the landmarks based on their total degree (in- plus out-degree). Hence any query (v, w, L) will hit a landmark v' within a few steps. As the landmarks have a lot of out-edges as well, a relative large space is pruned whenever direct attempt (v', w, L) is false. Also note that the mean BFS query times for **TwitterL8exp** are also on the high side looking at Table 6.15. **TwitterL8exp** is an exception though if we look at the speed-ups.

The asymmetry between the speed-ups for the True- and False-queries is growing when we compare Table 6.17 with Table 6.18 and Table 6.18 with Table 6.19. This is due to the ratio between the number of landmarks to the number of vertices is shrinking. For a True-query the gain from resolving a query (v, w, L) directly through a landmark v' increases as the graph (and with it the query difficulty) grows. For a False-query the relative low number of landmarks does not help that much. The first extension of LI only works for the first direct attempt. A better version of the first extension is necessary to make this difference smaller, even if this goes at the expense of the True-query speed-up.

Table 6.19: Speed-ups for the methods LI+OTH+EXTv1 with a budget $b = 20$ (first row), LI+OTH+EXTv1 with $b = 40$ (second row) and LI+OTH with $b = 20$ (third row) and the datasets for which we set $k = \frac{N}{250}$.

dataset	qs_1,true $\lvert\mathcal{L}\rvert/4$	qs_1,false	qs_2,true $\lvert\mathcal{L}\rvert/2$	qs_2,false	qs_3,true $\lvert\mathcal{L}\rvert - 2$	qs_3,false
zhishihudongL8exp	775.96 749.67	1 1	999.37 1,026.45	0.89 0.9	891.01 884.79	0.99 1
V625kD5L8exp	320.04 322.75	1,572.78 1,611.81	361.58 371.02	3.07 3.05	402.29 408.75	1.22 1.21
plV625ka2.0L8exp	14,862.82 14,808.65	5.09 5.09	15,426.74 15,512.66	5.25 5.24	15,565.15 15,475.5	2.07 2.07
usPatentsL8exp	1.73 1.73	1.31 1.31	3.03 3.05	1.38 1.38	1.56 1.59	1.1 1.1
webGoogleL8exp	6,379.07 6,442.69	1.42 1.4	3,970.09 3,741.13	2.48 2.44	5,904.38 5724.36	1.98 1.95
Average	4,467.92 4,465.1	316.32 324.12	4,152.16 4,130.86	2.61 2.6	4,552.88 4,499	1.47 1.47

6.3.3 LI+OTH+EXTv2 : Index construction time and size

The disappointing results for the False-queries in Table 6.19 lead us to re-examine the first extension as this extension was particularly aimed at speeding up false queries. We found a different way of using the idea behind the first extension which ended in the third extension.

Table 6.20 shows the same datasets as Table 6.15 but with a different often lower number of landmarks. The experiments were conducted with the methods LI+OTH+EXTv2 $b = 15$ or $b = 30$ and LI+OTH+EXTv1 $b = 15$.

Looking at Table 6.21 we see there is a small difference between the LI+OTH+EXTv2 and LI+OTH+EXTv1 methods when we look at the index size and index construction time. Often there is only a few hundreds of MB difference between LI+OTH+EXTv2 and LI+OTH+EXTv1 . The difference in the construction time is often larger but manageable, i.e. no more than 15% increase. Only for **jmd** the difference is quite large, i.e. 7 times larger.

All datasets have an index construction time below $(6,500$ seconds). The datasets in the upper section of the table all have construction times below $1,000$ seconds. The index size relative to the number of vertices is also low. Considering the worst-case index size for k landmarks, i.e. $O(b + |V| \cdot k \cdot 2^{|\mathcal{L}|})$ (see Section 5.2.2) we have a very small index size.

Table 6.20: The datasets included in this part of the experiment with k expressed as the fraction of the number of vertices N.

Dataset	k	N
jmd	$N/100$	486,320
citeSeerL8exp	$N/100$	384,414
NotreDameL8exp	$N/100$	325,730
soc-sign-epinionsL8exp	$N/100$	131,828
socSlashdotL8exp	$N/100$	82,168
V125kD5L8exp	$N/100$	125,000
pl125ka2.0L8exp	$N/100$	125,000
TwitterL8exp	$N/100$	465,018
webBerkstanL8exp	$N/500$	685,231
webStanfordL8exp	$N/500$	281,904
webGoogleL8exp	$N/500$	875,713
zhishihudongL8exp	$N/500$	2,452,715
usPatentsL8exp	$N/500$	3,774,769
V625kD5L8exp	$N/500$	625,000
pl625ka2.0L8exp	$N/500$	625,000

Table 6.21: Index construction time (s) and size (MB) for LI+OTH+EXTv2 $b = 15$ or $b = 30$ (L23) and LI+OTH+EXTv1 $b = 15$ (L12).

dataset	LI+OTH+EXTv2 $b = 15$		LI+OTH+EXTv2 $b = 30$		LI+OTH+EXTv1 $b = 15$	
citeSeerL8exp	114.03	1,158.75	115.15	1,181.46	97.63	1,000.99
jmd	23.72	21	23.47	21.05	3.92	18.4
NotreDameL8exp	395.12	4,777.49	394.96	4,782.76	344.13	4,112.74
plV125kL8a2.0L8exp	183.99	2,538.03	184.7	2,538.97	158.85	1,987.15
V125kD5L8exp	957.97	6,467.02	958.84	6,523.72	950.78	6,353.11
soc-sign-epinionsL8exp	205.51	2,390.25	205.9	2,395.20	171.64	1,713.02
socSlashdotL8exp	138.25	2,137.47	138.64	2,146.08	126	1,853.14
TwitterL8exp	889.16	17,632.80	889.17	17,632.80	866.23	17,427.31
zhishihudongL8exp	6,418.79	16,198.93	6,435.23	16,572.12	5,623.60	14,802.09
webBerkstanL8exp	1,604.28	9,879.21	1,603.63	9,888.58	1,580.64	9,476.28
webGoogleL8exp	4,690.88	27,117.16	4,694.70	27,147.40	4,736.21	26,813.72
webStanfordL8exp	295.57	2,557.07	295.98	2,557.07	292.12	2,499.10
usPatentsL8exp	6,463.42	21,915.31	6,399.90	21,915.31	5,561.24	475.39
plV625kL8a2.0L8exp	1,233.28	12,982.43	1,240.02	12,982.43	1,063.14	9,726.56
V625kD5L8exp	6,433.90	34,300.96	6,437.99	34,568.91	6,897.07	32,534.70
Average	2,003.19	10,804.93	2,001.22	10,856.92	1,898.21	8,719.58

6.3.4 LI+OTH+EXTv2 : Speed-ups

Table 6.22 shows the speed-ups for the datasets with $k = \frac{N}{100}$. Note that the query times of BFS are still the same (see Table 6.16). In the first two rows of each dataset we see LI+OTH+EXTv2 with $b = 15$ and $b = 30$ and in the third row we see LI+OTH+EXTv1 with $b = 15$. The speed-ups for the True-queries are roughly comparable for the first two rows and the third row, but for the False-queries there is a clear improvement. The magnitude of the improvement differs per dataset. This has to do with the out-degree distribution.

When we compare the $b = 15$-version against the $b = 30$-version there is no clear winner here. In some occasions one variant works better and in some occasions the other works better.

Table 6.22: Speed-ups for LI+OTH+EXTv2 $b = 15$ or $b = 30$ and LI+OTH+EXTv1 $b = 15$ with $k = \frac{N}{100}$.

dataset	qs_1,true $\lvert\mathcal{L}\rvert/4$	qs_1,false	qs_2,true $\lvert\mathcal{L}\rvert/2$	qs_2,false	qs_3,true $\lvert\mathcal{L}\rvert - 2$	qs_3,false
citeSeerL8exp	208.53	82.34	279.52	76.05	389.55	55.04
	229.93	81.37	308.25	73.83	443.08	54.91
	211.47	2.23	294.43	1.71	407.37	1.33
jmd	7.22	0.98	1.75	0.95	1.1	1
	6.78	0.93	1.72	0.96	1.11	1.01
	7.14	0.99	1.74	0.98	1.12	0.99
NotreDameL8exp	14.55	91.55	16.51	102.99	65.22	215.55
	14.49	87.92	16.29	165.59	65.66	235.31
	14.54	4.02	16.24	2.26	65.56	1.77
plV125kL8a2.0L8exp	1,498.19	418.39	1,875.59	671.07	2,561.84	445.03
	1,655.61	415.73	1,973.44	689.51	3,101.36	472.22
	1,489.66	1.7	1,915.09	2.17	2,559.05	2.76
soc-sign-epinionsL8exp	1,733.57	1,894.27	3,930.49	2,755.08	4,213.19	2,958.56
	1,769.20	1,926.03	4,128.76	2,758.99	4,117.01	2,927.49
	1,736.41	1.52	3,893.73	1.27	4,145.18	1.49
socSlashdotL8exp	1,065.00	1,274.13	5,971.36	1,992.09	1,503.16	656.8
	1,049.51	1,258.57	5,794.21	1,936.08	1,514.89	650.81
	1,051.95	1.24	5,961.80	1.26	1,476.11	1.37
TwitterL8exp	54,335.96	27,718.68	48,976.11	69,439.02	56,912.68	17,487.74
	54,335.96	27,289.48	48,976.11	69,439.02	56,912.68	17,487.74
	54,335.96	27,211.45	48,976.11	69,439.02	56,912.68	17,487.74
Average	8,409.0	4,497.19	8,721.62	10,719.61	9,378.11	3,117.1
	8,437.35	4,437.15	8,742.68	10,723.43	9,450.83	3,118.5
	8,406.73	3,889.02	8,722.73	9,921.24	9,366.72	2,499.64

Table 6.23 shows the speed-ups but then for $k = \frac{N}{500}$. Again we see a strong performance improvement for LI+OTH+EXTv2 with respect to the False-queries. The improvement is the highest for $|\mathcal{L}|/4$ which has to do with the following. When we prune a subset of the graph by using the third extension, the effect of this is larger of queries with a smaller label set because a smaller number of the edges in the graph can be used. A way to get higher numbers for the False-queries in case $|L| = |\mathcal{L}| - 2$ would be to increase the $MAXDIST$-parameter , currently equal to $|sL|/4 + 1$, of the third extension. This would include more entries into $seqE$ and give queries with a larger label set the opportunity to prune even more vertices.

The speed-ups for **usPatentsL8exp** are bad. We saw the same thing with **jmd**. Part of this has to do with the good performance of BFS on this which is caused by the difficulty to generate 'difficult' queries for tree-like graphs. Another part may have to do with our approach. For trees we would prefer a different construction algorithm. Build a landmark for every root of a subtree T' of a size of at least K.

Table 6.23: Speed-ups for LI+OTH+EXTv2 $b = 15$ or $b = 30$ and LI+OTH+EXTv1 $b = 15$ with $k = \frac{N}{500}$.

dataset	qs_1,true $\lvert\mathcal{L}\rvert/4$	qs_1,false	qs_2,true $\lvert\mathcal{L}\rvert/2$	qs_2,false	qs_3,true $\lvert\mathcal{L}\rvert - 2$	qs_3,false
zhishihudongL8exp	803.99	911.96	1,057.82	106.22	954.23	20.77
	804.83	911.62	1,062.50	106.35	954.31	20.71
	803.63	0.93	1,056.27	0.98	953.79	0.93
webBerkstanL8exp	517.56	342.29	228.37	31.49	894.44	40.62
	513.55	341.16	278.72	31.63	877.37	40.69
	518.5	1.21	228.94	1.4	895.1	1.32
webGoogleL8exp	4,181.61	5,908.13	2,246.65	17.12	4,385.16	20
	4,310.40	6,060.13	2,447.64	17.59	4,638.08	20.24
	4,146.34	1.08	2,252.71	1.22	4,388.31	1.25
webStanfordL8exp	1,414.59	239.56	295.47	15.82	2,380.15	13.06
	1,381.55	241.09	820.96	15.85	2,446.68	14.4
	1,408.43	1.24	293.87	1.13	2,354.91	1.19
usPatentsL8exp	1.12	1.04	1.4	1.16	1.11	0.98
	1.11	1.02	1.42	1.15	1.13	0.97
	1.09	1	1.34	1.11	1.09	0.98
plV625kL8a2.0L8exp	3,913.11	1,257.15	2,416.38	709.51	2,304.48	246.42
	3,924.00	1,233.11	2,981.31	700.4	2,303.75	245.42
	3,921.50	1.9	2,417.68	1.63	2,305.84	1.16
V625kD5L8exp	183.49	56.56	218.79	10.93	270.66	3.09
	188.2	58.1	254.12	11.24	272.59	3.26
	183.64	1.59	217.75	0.72	268.83	0.67
Average	1,573.64	1,245.24	923.55	127.46	1,598.6	49.28
	1,589.09	1,263.75	1,120.95	126.32	1,641.99	49.38
	1,569.02	1.28	924.08	1.17	1,595.41	1.07

6.4 Maintenance

We tested the correctness and speed of our maintenance methods by doing the following. First we built a LI lI for a graph G. Then we applied $K = 30$ random operations (adding an edge, removing an edge) to G and to lI. Afterwards, we built a LI for G lI'. We compared lI and lI' over $L = 200$ random queries.

6.4.1 Adding an edge

Table 6.24 shows for 4 datasets the percentage of the construction time of lI (before the update) that it took to add a random edge (v, w, l) to the graph. We can see that for label sets with an exponential distribution inserting a new edge takes relatively more time (often roughly 22%) than for datasets with a normal or uniform distribution. The data in the table shows that frequent insertion of edges on a large index is not desirable. A more efficient method is thus desired.

Table 6.24: Ratio of the initial construction time (s) that adding a random edge took for 4 datasets, e.g. 0.05 means 5% of the construction time. 30 random edges were inserted.

V5kD2L8uni	**V5kD2L8norm**	**V5kD2L8exp**	**V25kD2L8exp**
0.06	0.07	0.2	0.23
0.06	< 0.01	< 0.01	0.23
0.06	0.07	0.19	0.24
0.06	0.07	0.19	< 0.01
< 0.01	0.08	< 0.01	0.24
0.06	0.08	0.2	0.23
0.06	0.07	0.19	0.25
0.06	0.07	0.18	0.25
< 0.01	0.07	0.21	< 0.01
0.06	0.07	0.2	< 0.01
0.06	0.06	< 0.01	0.48
0.06	0.06	< 0.01	0.24
0.05	0.07	0.2	0.24
0.06	< 0.01	0.2	0.23
0.07	0.08	0.19	0.22
< 0.01	0.07	0.19	0.23
0.07	< 0.01	0.2	0.23
0.06	0.08	0.2	0.23
< 0.01	< 0.01	0.19	0.22
< 0.01	0.07	< 0.01	0.22
< 0.01	0.07	0.2	0.24
0.06	< 0.01	0.2	0.26
0.06	0.07	< 0.01	0.48
0.06	0.07	0.21	0.24
0.06	0.07	0.19	0.22
0.06	0.08	< 0.01	0.23
< 0.01	0.07	0.3	< 0.01
0.06	0.08	0.4	0.23
< 0.01	0.07	0.38	0.24
< 0.01	0.08	0.4	0.23
0.04	0.06	0.17	0.22

6.5 Extensions

6.5.1 Query for all nodes

Figures 6.4 and 6.5 show the speed-up per query for QueryAll-queries in two different settings. In the first setting the label set distribution was altered. In the second setting, the degree was altered. From the first figure we can see that the speed-ups per query are higher for a dataset with an exponential distribution than for one with a normal or uniform distribution. The degree is not that influential. For both figures, we ran 200 queries (half of which $|L| = |\mathcal{L}|/2$, other half $|L| = |\mathcal{L}| - 2$). Also we demanded from a query that it would hit at least 10% of the vertices. For uniform or normal datasets this requirement would not be met if we set $|L| = |\mathcal{L}|/4$. Hence we omitted that one.

6.5.2 Distance queries

Table 6.26 shows the index size and construction time for two versions of LI+OTH+EXTv1 ($k = N/10, b = 0$) and some datasets. The first version is the normal "LCR"-version. The second version is the version that can give the distance of the shortest path P for a query (v, w, L) s.t. $Labels(P) \subseteq L$. The results show that the index size is always at least 2 times larger. The index construction time can even grow much stronger. We do

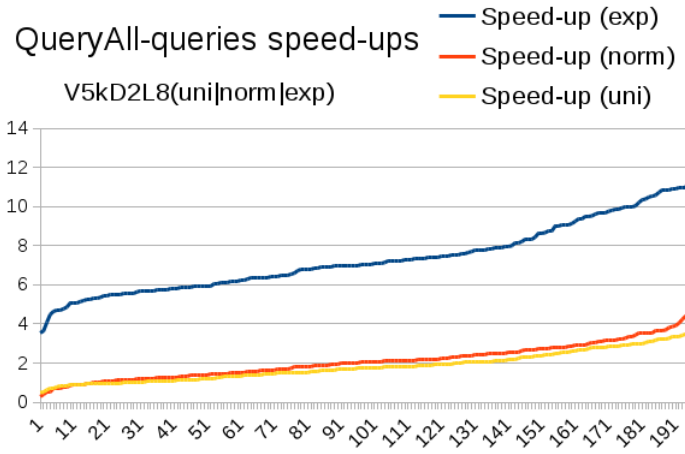


Figure 6.4: Speed-ups per query for QueryAll-queries for PA-datasets with $5,000$ vertices, a degree of 2, 8 labels and either a uniform, normal or exponential label set distribution. We used LI+OTH+EXTv1 ($k = N/10, b = 0$).

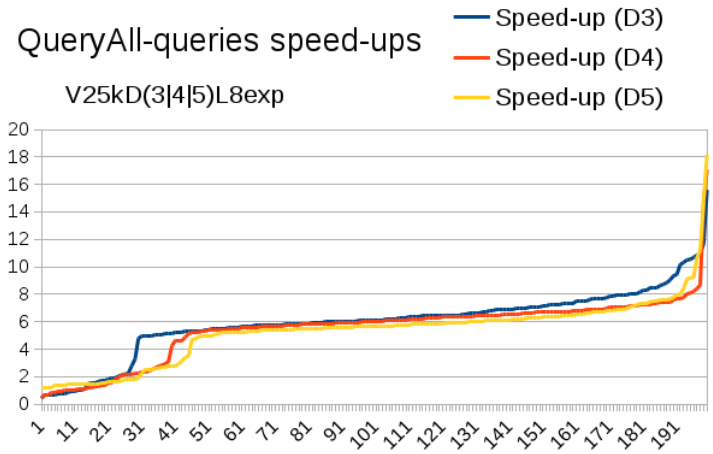


Figure 6.5: Speed-ups per query for QueryAll-queries for PA-datasets with $25,000$ vertices, a degree of 3, 4 or 5, 8 labels and an exponential label set distribution. We used LI+OTH+EXTv1 ($k = N/10, b = 0$).

Table 6.25: Average speed-up for QueryAllQueries.

dataset	speed-up $\lvert L\rvert = \lvert\mathcal{L}\rvert/2$	speed-up $\lvert L\rvert = \lvert\mathcal{L}\rvert - 2$
V5kD2L8uni	1.24	2.49
V5kD2L8norm	1.40	2.90
V5kD2L8exp	5.97	8.86
V25kD3L8exp	4.40	7.55
V25kD4L8exp	4.04	6.95
V25kD5L8exp	3.81	6.80

wish to note that this solution returns the exact distance, i.e. no approximation of the distance.

Table 6.26: Index size (MB) and index construction time (s) for LI+OTH+EXTv1 and LI+OTH+EXTv1 with distance ($k = N/10, b = 0$).

dataset	size (MB)	time (s)	size (MB) WD	time (s) WD	ratio (size)	ratio (time)
V5kD2L8uni	84.7	8.4	380.6	65.4	4.5	7.7
V5kD2L8norm	71.6	5.4	307.7	39.4	4.3	7.3
V5kD2L8exp	35.2	102.3	0.77	4.5	2.9	5.9
V25kD2L8exp	857.1	20.4	2,871.9	217.6	3.3	10.6
advogato	54.5	1.7	110.1	58.1	2.0	33.9

Chapter 7

Conclusion

In this paper we introduced a new reachability problem with a label-constraint. This problem has a lot of potential in real world applications (RDF, social networks and biological networks).

Several methods (LI , PartialIndex , DoubleBFS , NeighbourExchange , ClusteredExact and Zou) were experimentally compared against each other. The results clearly were in favour of LI . This method proved to be most scalable in terms of index construction time and size when the graph size increased. In combination with the second and third extension, i.e. LI+OTH+EXTv2 , we were able to achieve decent speed-ups even for large datasets and a low number of landmarks. However datasets with a close to uniform out-degree distribution (like ER-graphs) or datasets with a large label set ($|\mathcal{L}| \geq 16$) proved to be difficult still. Using just BFS in these cases might be a good solution as this method has an acceptable running time.

The index is also not capable of dealing with very frequent updates, e.g. multiple times in a minute.

7.1 Future work

One possible extension could be to use our main contribution, i.e. LI , to also answer distance queries. In one way the current index could be used to just return ∞ to queries (v, w, L) for which there is no L-path. In another way we could extend on the current idea to answer these queries as well. Though we have seen in Chapter 6 (see Section 6.5.2) that the index construction time grows if we wish to answer these queries.

The index construction time is still relatively high for the large datasets. It might not be possible to reduce it further without paying some price in terms of memory or speed-up. One way could be only using the third extension of LI . In this case a landmark would just hold a list of pairs where each pair is a label set L' of size at most $|\mathcal{L}|/4$ and a bitset b of size N. A query (v, w, L) would then hit a landmark v' at some point. The direct attempt for v' would then try to find a pair for which $L' \subseteq L$ and in which the bit for w in b has been set. In that case v can reach w. Otherwise we can prune all vertices which have their bit set in b.

A good maintenance algorithm is still on the list. Addition of edges is still very costly. Removal of edges might require adding some redundancy, e.g. storing some pairs $(u, L'), (u, L) \in Ind(v)$ for $v \in V$ s.t. $L' \subseteq L$.

One could still look at the potential of making a multi-threaded version of LI . We did not have the time to make it.

Bibliography

[1] Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Antti Ukkonen. Distance oracles in edge-labeled graphs. In *a*, EDBT, 2014. 2, 3, 11, 42, 49

[2] Minghan Chen, Yu Gu, Yubin Bao, and Ge Yu. Label and distance-constraint reachability queries in uncertain graphs. In SouravS. Bhowmick, CurtisE. Dyreson, ChristianS. Jensen, MongLi Lee, Agus Muliantara, and Bernhard Thalheim, editors, *Database Systems for Advanced Applications*, volume 8421 of *Lecture Notes in Computer Science*, pages 188–202. Springer International Publishing, 2014. 5

[3] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. 9

[4] George Fletcher and Yuichi Yoshida. Notes on landmark labeling label-constrained reachability queries. 17, 18

[5] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. Computing label-constraint reachability in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 123–134, New York, NY, USA, 2010. ACM. 5

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

[7] Jure Leskovec and Rok Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, June 2014. 21

[8] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, CIKM '13, pages 1601–1606, New York, NY, USA, 2013. ACM. 3

[9] JeffreyXu Yu and Jiefeng Cheng. Graph reachability queries: A survey. In Charu C. Aggarwal and Haixun Wang, editors, *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 181–215. Springer US, 2010. 3, 9, 10

[10] Lei Zou, Kun Xu, Jeffrey Xu Yu, Lei Chen, Yanghua Xiao, and Dongyan Zhao. Efficient processing of label-constraint reachability queries in large graphs. *Inf. Syst.*, 40:47–66, March 2014. 1, 5, 11, 13, 16, 28, 31